VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

DATABASE SYSTEMS (CO2013)

Assignment 2

FABRIC AGENCY DATABASE

Advisor:	PhD. Phan Trọng Nhân
Class:	CC02
Group:	5
Students:	Đinh Việt Thành - 2152966
	Trần Nhật Tân - 2112259
	Vũ Châu Duy Quang - 2153730
	Trần Bảo Nguyên - 2153637
	Dương Trọng Phúc - 2152237

HO CHI MINH CITY, OCTOBER 2023

Contents

1 Member list & Workload

No.	Fullname	Student ID	Percentage of work
1	Đinh Việt Thành	2152966	100%
2	Vũ Châu Duy Quang	2153730	100%
3	Trần Nhật Tân	2112259	100%
4	Trần Bảo Nguyên	2153637	100%
5	Dương Trọng Phúc	2152237	100%

2 Introduction

This is a design of Fabric Agency Database, which has been given by advisor Phan Trọng Nhân. In this report, the implementation of database designation at and above physical level, as well as performing operation will be explained. Further, the application progress is also demonstrated.

3 Physical Database Design

In the first part, we will implement our database, based on assigned topic, using MySQL.

3.1 A - Implementing the database

Our database design in MySQL bases on the previous ERD mapping in Assignment 1, the source of mapping image will be included in report submission.

3.1.1 Table structure design

Here is the description of MySQL tables created:

Table: `employee`

Columns:

- `employee_code`: Unique code for each employee (e.g., 'EM0001').
- `employee_type`: Type of employee (manager, ops_staff, ofc_staff, partner_staff).
- `first_name`, `last_name`: Employee's first and last name.
- `gender`: Employee's gender (male or female).
- `address`: Employee's address.

Primary Key: `employee_code`

Table: employee_phone_number

Columns:

- employee_code: Employee code associated with the phone number.
- phone_num: Employee's phone number.

Foreign Key: employee_code references employee(employee_code).
Primary Key: Composite key of (employee_code, phone_num)

Table: customer

Columns:

- customer_code: Unique code for each customer (e.g., 'CU0001').
- office_staff_code: Code of the office staff associated with the customer.
- first_name, last_name: Customer's first and last name.
- address: Customer's address.
- mode: Customer mode ('normal' by default).
- arrearage: Customer's arrearage amount.
- debt_date: Date when the debt was incurred (default 0).

Foreign Key: office_staff_code references employee(employee_code).
Primary Key: customer_code

Table: customer_phone_number

Columns:

- customer_code: Customer code associated with the phone number.
- phone_num: Customer's phone number.

Foreign Key: customer_code references customer(customer_code).
Primary Key: Composite key of (customer_code, phone_num)

Table: fab_order

Columns:

- order_code: Unique code for each order (e.g., 'OR0001').
- customer_code: Code of the customer associated with the order.
- total_price: Total price of the order.
- res_price: Reserved price of the order.
- or_status: Order status ('new', 'ordered', 'partial paid', 'full paid', 'cancelled').
- date_time: Date and time of the order.

Foreign Key: customer_code references customer(customer_code).
Primary Key: order_code

Table: processed_order

Columns:

- order_code: Order code associated with the processing.
- ops_staff_code: Code of the operations staff processing the order.
- processed_datetime: Date and time of order processing.

Foreign Keys:

- order_code references fab_order(order_code).
- ops_staff_code references employee(employee_code).

Primary Key: order_code

Table: cancelled_order

Columns:

- order_code: Order code associated with the cancellation.
- ops_staff_code: Code of the operations staff cancelling the order.
- cancelled_reason: Reason for order cancellation.

Foreign Keys:

- order_code references fab_order(order_code).
- ops_staff_code references employee(employee_code).

Primary Key: order_code

Table: order_partial_payment

Columns:

- order_code: Order code associated with the partial payment.
- customer_code: Code of the customer making the payment.
- pay_date, pay_time: Date and time of the payment.
- amount: Payment amount.

Foreign Keys:

- order_code references fab_order(order_code).
- customer_code references customer(customer_code).

Primary Key: Composite key of (order_code, pay_date, pay_time)

Table: `supplier`

Columns:

- `supplier_code`: Unique code for each supplier (e.g., 'SU0001').
- `partner_staff_code`: Code of the partner staff associated with the supplier.
- `name`: Supplier's name.
- `address`: Supplier's address.
- `bank_account`: Supplier's bank account.
- `tax_code`: Supplier's tax code.

Foreign Key: `partner_staff_code` references `employee(employee_code)`.
Primary Key: `supplier_code`

Table: `supplier_phone_number`

Columns:

- `supplier_code`: Supplier code associated with the phone number.
- `phone_num`: Supplier's phone number.

Foreign Key: `supplier_code` references `supplier(supplier_code)`.
Primary Key: Composite key of (`supplier_code`, `phone_num`)

Table: `fabric_cat`

Columns:

- `fabcat_code`: Unique code for each fabric category (e.g., 'FA0001').
- `supplier_code`: Code of the supplier associated with the fabric category.
- `name`, `color`: Fabric category name and color.
- `quantity`: Quantity of fabric available.

Foreign Key: `supplier_code` references `supplier(supplier_code)`.
Primary Key: `fabcat_code`

Table: `fabcat_current_price`

Columns:

- `fabcat_code`: Fabric category code associated with the current price.
- `valid_date`: Date when the price is valid.
- `price`: Current price of the fabric category.

Foreign Key: `fabcat_code` references `fabric_cat(fabcat_code)`.
Primary Key: Composite key of (`fabcat_code`, `valid_date`, `price`)

Table: `import_info`

Columns:

- `fabcat_code`: Fabric category code associated with the import information.
- `supplier_code`: Code of the supplier associated with the import.
- `import_date`, `import_time`: Date and time of the import.
- `quantity`: Quantity of fabric imported.
- `price`: Price of the imported fabric.

Foreign Keys:

- `fabcat_code` references `fabric_cat(fabcat_code)`.
- `supplier_code` references `supplier(supplier_code)`.

Primary Key: Composite key of (`fabcat_code`, `supplier_code`, `import_date`, `import_time`)

Table: `bolt`

Columns:

- `bolt_code`: Unique code for each bolt in a category(e.g., 'BO0001').
- `fabcat_code`: Code of the fabric category associated with the bolt.
- `length`: Length of the bolt.

Foreign Key: `fabcat_code` references `fabric_cat(fabcat_code)`.
Primary Key: Composite key of (`bolt_code`, `fabcat_code`)

Table: `bolt_and_order`

Columns:

- `bolt_code`: Bolt code associated with the order.
- `order_code`: Order code associated with the bolt.
- `fabcat_code`: Fabric category code associated with the bolt.

Foreign Keys:

- `bolt_code` references `bolt(bolt_code)`.
- `order_code` references `fab_order(order_code)`.
- `fabcat_code` references `bolt(fabcat_code)`.

Primary Key: Composite key of (`bolt_code`, `order_code`, `fabcat_code`)

3.1.2 Additional functions

We also implement some functions and triggers to remain the consistency of our database, as well as ensure security.

Function: `get_length`

This function retrieves the length of a bolt specified by its category code (`catCode`) and bolt code (`boltCode`) from the `bolt` table.
Parameters:

- `catCode`: `VARCHAR(6)` - Fabric category code.
- `boltCode`: `VARCHAR(6)` - Bolt code.
- `RETURN`: `INT` - The length of the specified bolt.

Function: `get_selling_price`

This function retrieves the selling price of a fabric category specified by its code (`catCode`) from the `fabric_cat` and `fabcat_current_price` tables.
Parameters:

- `catCode`: `VARCHAR(6)` - Fabric category code.
- `RETURN`: `INT` - The selling price of the specified fabric category.

Function: `order_quantity`

This function calculates the quantity of bolts associated with a given order code (`order_code`) from the `fab_order` and `bolt_and_order` tables.
Parameters:

- `order_code`: `INT` - Order code.
- `RETURN`: `INT` - The quantity of bolts in the specified order.

Function: `get_job`

This function retrieves the job type (employee type) for a given employee code (`n_employee_code`) from the `employee` table.
Parameters:

- `n_employee_code`: `VARCHAR(6)` - Employee code.
- `RETURN`: `VARCHAR(50)` - The job type of the specified employee.

Function: `get_or_status`

This function retrieves the order status for a given order code (`n_order_code`) from the `fab_order` table.
Parameters:

- `n_order_code`: `VARCHAR(6)` - Order code.
- `RETURN`: `VARCHAR(15)` - The order status of the specified order.

3.1.3 Additional triggers

Trigger: `fill_customer_code`

This trigger is executed before inserting a record into the `order_partial_payment` table. It fills the `customer_code` field with the existing customer code associated with the provided order code.

Trigger: `fill_supplier_code`

This trigger is executed before inserting a record into the `import_info` table. It fills the `supplier_code` field with the existing supplier code associated with the provided fabric category code.

Trigger: `import_fabric`

This trigger is executed after inserting a record into the `import_info` table. It updates the quantity of the fabric category in the `fabric_cat` table by adding the newly imported quantity.

Trigger: `delete_Bolt`

This trigger is executed after deleting a record from the `bolt` table. It updates the quantity of the fabric category in the `fabric_cat` table by decrementing it.

Trigger: `insert_bolt_in_order`

This trigger is executed after inserting a record into the `bolt_and_order` table. It updates the total and reserved price of the associated order by calculating the price based on the length and selling price.

Trigger: `delete_bolt_in_order`

This trigger is executed before deleting a record from the `bolt_and_order` table. It updates the total price of the associated order by subtracting the price based on the length and selling price.

Trigger: `price_change`

This trigger is executed after updating a record in the `fab_order` table. It checks if the total price of the order has changed and updates the customer's arrearage accordingly.

Trigger: `check_mode`

This trigger is an event scheduled to run every second. It updates the mode and debt date of customers based on their arrearage.

Trigger: `delete_part_payment`

This trigger is executed after deleting a record from the `order_partial_payment` table. It updates the customer's arrearage by adding back the deleted partial payment amount.

Trigger: `insert_part_payment`

This trigger is executed after inserting a record into the `order_partial_payment` table. It updates the customer's arrearage and the reserved price of the associated order.

Trigger: `partial_payment_status`

This trigger is executed after inserting a record into the `order_partial_payment` table. It checks and updates the order status to "partial paid" if it was in a "new" status.

Trigger: `update_part_payment`

This trigger is executed after updating a record in the `order_partial_payment` table. It updates the customer's arrearage if the partial payment amount has changed.

Trigger: `insert_customer`

This trigger is executed before inserting a record into the `customer` table. It checks if the job position of the associated office staff is appropriate.

Trigger: `insert_supplier`

This trigger is executed before inserting a record into the `supplier` table. It checks if the job position of the associated partner staff is appropriate.

Trigger: `insert_processed_order`

This trigger is executed before inserting a record into the `processed_order` table. It checks if the job position of the associated operations staff is appropriate and updates the order status.

Trigger: `insert_cancelled_order`

This trigger is executed before inserting a record into the `cancelled_order` table. It checks if the job position of the associated operations staff is appropriate and updates the order status, removing the order from processed orders.

Trigger: `update_customer_arrearage_after_insert`

This trigger is executed after inserting a record into the `fab_order` table. It updates the customer's arrearage based on the sum of reserved prices.

Trigger: `update_customer_arrearage_after_update`

This trigger is executed after updating a record in the `fab_order` table. It updates the customer's arrearage based on the sum of reserved prices.

3.2 B - Insert data

Here is the result of some table after we implement the data insertion, use appropriate environment to inspect all the tables if necessary:

Employee Code	Employee Type	First Name	Last Name	Gender	Address
EM0001	manager	Nguyen	Tran	male	720A Dien Bien Phu
EM0002	ops_staff	Thanh	Dinh Viet	male	15 To Hien Thanh
EM0003	ofc_staff	Linh	Tran Khanh	female	72 To Huu
EM0004	partner_staff	Ngoc	Nguyen Bao	female	260 Ly Thuong Kiet
EM0005	partner_staff	Minh	Nguyen Nhat	male	24 Dong Nai
EM0006	ops_staff	Huyen	Phan Khanh	female	26 Thanh Thai
EM0007	ofc_staff	Nhi	Phan Uyen	female	150 Ly Thai To
EM0008	ops_staff	Thien	Ton Nu Y	male	135 To Hien Thanh
EM0009	ops_staff	Thanh	Nguyen Thanh	male	15 Thanh Thai

Table 1: Employee Data

Employee Code	Phone Number
EM0001	0956054654
EM0001	0985054654
EM0002	0541355654
EM0003	0989866654
EM0004	0985058086
EM0005	0985054459
EM0005	0985059854
EM0006	0354151335
EM0007	0935115050
EM0007	0985065450
EM0007	0985990535

Table 2: Employee Phone Number Data

Supplier Code	Partner Staff Code	Name	Address	Bank Account	Tax Code
SU0001	EM0004	Silk Agency	15 Le Thanh Ton	00129300312	FA1234
SU0002	EM0005	MSoft	24 CMT8	00131351353	FA3514
SU0003	EM0004	Amaron	155 Ly Thai To	00988453213	FA9803
SU0004	EM0005	Mate	213 Truong Dinh	00153684352	FA6512
SU0005	EM0005	Appel	25 Ly Thai To	00135121351	FA1351

Table 3: Supplier Data

4 Store Procedure / Function / SQL

In this chapter, our mission is to working with created data on the physical part. In assignment 2, there are 4 tasks expected to be done:

- Increase Silk selling price to 10% of those provided by all suppliers from 01/09/2020.
- Select all orders containing bolt from the supplier named 'Silk Agency'.
- Write a function to calculate the total purchase price the agency has to pay for each supplier
- Write a procedure to sort the suppliers in increasing number of categories they provide in a period of time

4.1 Question 1

Additional Information

Increase Silk selling price to 10% of those provided by all suppliers from 01/09/2020.

```sql
-- Exercise 2.2 a ---------------------------------------------------
SELECT * FROM fabcat_current_price
WHERE valid_date >= '2020-09-01';

-- Operation
SET SQL_SAFE_UPDATES = 0;
UPDATE fabcat_current_price
SET price = price * 1.1
WHERE valid_date >= '2020-09-01';

-- Check result
SELECT * FROM fabcat_current_price
WHERE valid_date >= '2020-09-01';
```

Listing 1: Exercise 2.2 a: SQL Code

Table 4: Original Fabcat Current Price Table

fabcat_code	valid_date	price
'FA0001'	2023-11-01	146
'FA0001'	2023-11-02	193
'FA0001'	2023-12-05	219
'FA0002'	2023-10-31	330
'FA0002'	2023-12-01	220
'FA0003'	2023-11-01	293
'FA0004'	2023-12-01	250
'FA0005'	2023-12-01	300
'FA0006'	2023-11-01	350
'FA0006'	2023-11-02	605
'FA0007'	2023-10-01	400
'FA0008'	2023-11-01	450
'FA0009'	2023-11-01	500

Table 5: Updated Fabcat Current Price Table

fabcat_code	valid_date	price
'FA0001'	2023-11-01	177
'FA0001'	2023-11-02	233
'FA0001'	2023-12-05	265
'FA0002'	2023-10-31	399
'FA0002'	2023-12-01	266
'FA0003'	2023-11-01	354
'FA0004'	2023-12-01	303
'FA0005'	2023-12-01	363
'FA0006'	2023-11-01	424
'FA0006'	2023-11-02	733
'FA0007'	2023-10-01	484
'FA0008'	2023-11-01	545
'FA0009'	2023-11-01	605

4.2 Question 2

Additional Information

Select all orders containing bolt from the supplier named 'Silk Agency'.

```sql
-- Exercise 2.2 b ---------------------------------------------------
SELECT DISTINCT fo.*, s.supplier_code, s.name as supplier_name
FROM fab_order fo
JOIN bolt_and_order bao ON fo.order_code = bao.order_code
JOIN bolt b ON bao.bolt_code = b.bolt_code
JOIN fabric_cat fc ON bao.fabcat_code = fc.fabcat_code
JOIN supplier s ON fc.supplier_code = s.supplier_code
WHERE s.name = 'Silk Agency';
```

Listing 2: Exercise 2.2 b: SQL Code

Table 6: Order Information

Order Code	Customer Code	Total Price	Reserved Price	Order Status	Date/Time	Supplier Code	Supplier Name
OR0008	CU0001	8237	8237	ordered	2023-09-27 11:18:33	SU0001	Silk Agency
OR0002	CU0002	7477	7077	ordered	2023-07-22 04:12:03	SU0001	Silk Agency
OR0004	CU0004	18277	11277	ordered	2022-09-06 20:15:33	SU0001	Silk Agency
OR0009	CU0002	8686	8686	cancelled	2022-06-26 09:36:37	SU0001	Silk Agency

4.3 Question 3

Additional Information

Write a function to calculate the total purchase price the agency has to pay for each supplier.

```sql
-- Exercise 2.2 c ---------------------------------------------------
DELIMITER $$
```

```sql
DROP PROCEDURE IF EXISTS total_purchase_price;
CREATE PROCEDURE total_purchase_price (IN sup_code varchar(6))
BEGIN
    SELECT  i.import_date, i.import_time, i.fabcat_code, (i.price * i.quantity) AS 'Total purchase price'
    FROM import_info AS i
    WHERE i.supplier_code = sup_code;

END; $$
DELIMITER ;
-- example call
CALL total_purchase_price('SU0005');
```

Listing 3: Exercise 2.2 c: SQL Code

Table 7: Example call

Import Date	Import Time	Fabcat Code	Total Purchase Price
2023-09-06	14:00:00	FA0001	1200
2023-09-12	05:00:00	FA0001	312
2023-09-18	07:00:00	FA0001	1000
2023-09-09	19:00:00	FA0004	1320

4.4 Question 4

Additional Information

Write a procedure to sort the suppliers in increasing number of categories they provide in a period of time.

```sql
-- Exercise 2.2 d ---------------------------------------------------
DELIMITER $$
DROP PROCEDURE IF EXISTS sort_supplier_by_categories;
CREATE PROCEDURE sort_supplier_by_categories(IN start_date date,
                      IN end_date date)
BEGIN
  SELECT t2.supplier_code, t2.supplier_name, COUNT(t2.fabcat_code)
  FROM
    (SELECT t1.supplier_name, t1.supplier_code, t1.fabcat_name, t1.fabcat_code
    FROM (SELECT S.supplier_code, S.name AS supplier_name, F.fabcat_code, F.name AS fabcat_name FROM fabric_cat AS F JOIN supplier AS S
        ON F.supplier_code = S.supplier_code) AS t1 JOIN import_info ON import_info.fabcat_code = t1.fabcat_code
    WHERE import_info.import_date >= start_date and import_info.import_date <= end_date
    ) AS t2
    GROUP BY  t2.supplier_code, t2.supplier_name
    ORDER BY COUNT(t2.fabcat_code) ASC;
END; $$
DELIMITER ;
CALL sort_supplier_by_categories('2023-8-8', '2023-11-11');
```

Listing 4: Exercise 2.2 d: SQL Code

Table 8: Supplier Information

Supplier Code	Supplier Name	Count of Fabcat Codes
SU0002	MSoft	1
SU0003	Amaron	1
SU0004	Mate	1
SU0005	Appel	4

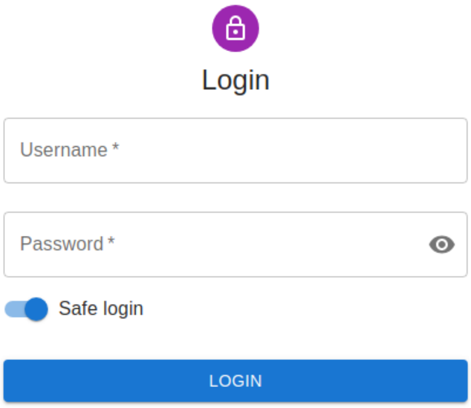
5 Building Application

In this section, we will demonstrate the progress building an application that takes over the physical database established in previous section. Group choose Javascript (mainly Reactsj and Nodejs) to write the application as well as backend API. There are 4 operations at which users can interact with:

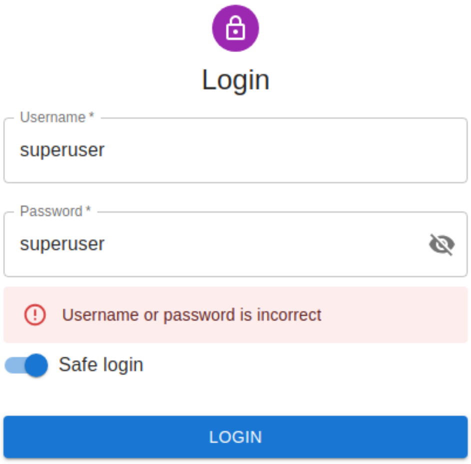
- Search material purchasing information: Search results include the name, phone number of the suppliers and information about the supply.
- Add information for a new supplier.
- List details of all categories which are provided by a supplier.
- Make a report that provides full information about the order for each category of a customer.

5.1 Login/logout authentication

For the authentication part, we implement a simple login page with user name and password for the administrator.



If the username and password submitted are correct, the user will be redirect to the home page. If not, there will be a error message displayed.



Login

Username *
superuser

Password *
superuser

Username or password is incorrect

Safe login

The "Safe login" switch will be reserved for discussion in the Security section.

5.2 API implementation

5.2.1 Search material purchasing information

In this operation, the request and the response between frontend and backend has following information:

Request

```
{
    categoryId: string,
    categoryName: string,
    searchByID: bool (if true, query = categoryID, false then query = categoryName),
    enableDateTimeRange: bool
    dateFrom: string (format yyyy:mm:dd),
    timeFrom: string (format 24h hh:mm),
    dateTo: string,
    timeTo: string
}
```

Response

```
[
    {
        categoryName: string,
        categoryID: string,
        supplierID: string,
        supplierName: string,
        supplierPhoneNumbers: string[],
        importInfos: [
            {
                date: string,
                time: string,
                quantity: int,
                price: float
            },
            {
                date: string,
                time: string,
                quantity: int,
                price: float
            },
            ...
        ]
    },
    ...
]
```

Additionally, there are some extend functions to support the operation:

- import_info: get all import information of a category
- get_fabric: get all information of a fabric category
- get_supplier: get all information of a supplier
- get_phone: get all phone numbers of a supplier

Detail implementations of each function are revealed in the code file.

5.2.2 Add information for a new supplier

This subsection has following request and response format:
Request

```
{
    name: string,
    address: string,
    bankAccount: string,
    taxCode: string,
    phoneNumbers: string[]
}
```

Response

```
{
    success: bool,
    statusMessage: string,
    supplierCode: string
    staffID: string,
    staffFName: string,
    staffLName: string
}
```

Also, there are some addition functions implemented to support this operation. Detail implementation can be experienced in the source code file.

5.2.3 List details of all categories which are provided by a supplier

API format for this section: Request

```
{
    supplierID: string,
    searchByID: bool,
    supplierName: string,
    searchByName: bool,
    supplierPhoneNumber: string,
    searchByPhoneNumber: bool
}
```

Response

```
[
    {
        supplierID: string,
        supplierName: string,
        categories: [
            {
                ID: string,
                name: string,
                color: string,
                quantity: string,
                priceHistory: [
                    {
                        date: string,
                        price: float
                    },
                    ... other price changes
                ]
            },
            ... other categories
        ]
    },
    ... other suppliers with same name.
]
```

Supplementary functions are already explained in source code.

5.2.4 Make a report that provides full information about the order for each category of a customer

API following Request

```
{
    customerID: string,
    customerPhoneNumber: string,
    searchByID: bool
}
```

Response

```
{
    ID: string,
    fName: string,
    lName: string,
    address: string,
    arrearage: float,
    deptStartDate: string,
    phoneNumbers: string[],
    orders: [
        {
            ID: string,
            dateTimeMade: string,
            totalPrice: float,
            status: string,
            dateTimeProcessed: string,
            cancelReason: string,
            staffID: string,
            staffFName: string,
            staffLName: string,
            paymentHistory: [
                {
                    date: string,
                    time: string,
                    amount: float
                },
                ... other payments
            ],
            categories: [
                {
                    categoryID: string,
                    categoryName: string,
                    boltNumber: int
                    bolts: [
                        {
                            boltID: string,
                            boltLength: float
                        },
                        ... other bolts
                    ]
                },
                ... other categories
            ]
        },
        ... other orders
    ]
}
```

5.3 Application implementation

For our front-end website, the Axios library was used to establish communication with the back-end application. In general, the `requestMapper()` function maps the parameters inputted from the website to the correct format of the API, and the `responseMapper()` function maps the response from the API to the desirable format for our website implementation. The `apiqx()` function (with x from 1-4 corresponds to each question) takes the mapped parameters and makes a request to the API. This function returns the response data from the API depending on which type of request the question needs. For question 1, 3 and 4, a GET request was made, while for question 2, a POST request was made. Detail implementations are revealed in the source code file.

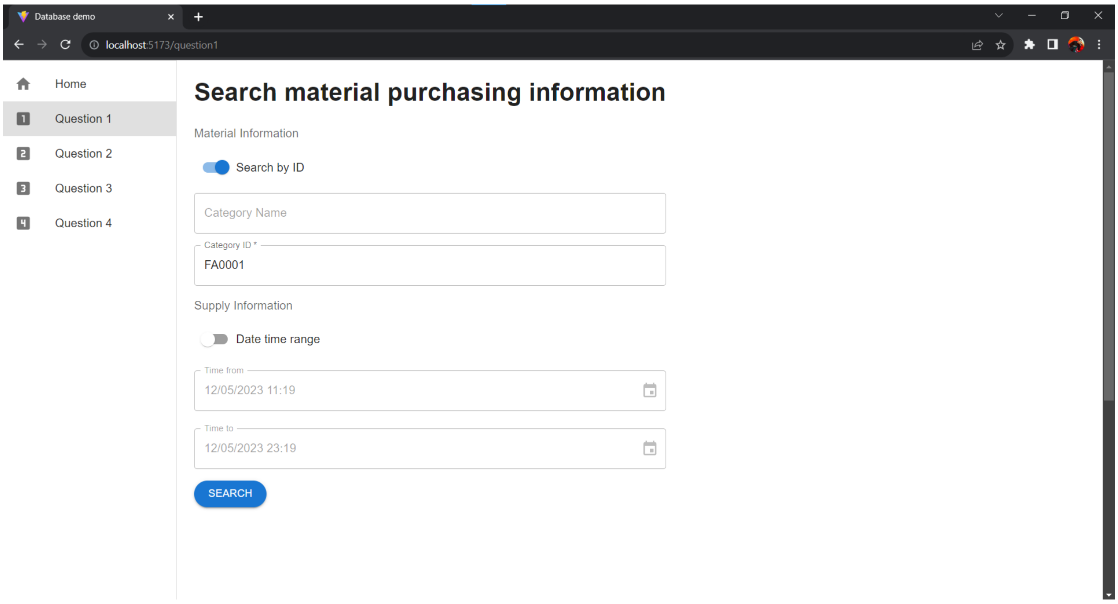
Using ReactJS with the help of the Material UI library, we are able to create a user-friendly interface which can take the user input to make a request to the API and display the response data. Detail implementations are revealed in the source code file. Below are the demos of this website.

5.3.1 Search material purchasing information

We can search the material purchasing information with either the fabric category name or id. In this example, we will search by the fabric category id of ”FA0001”

The API returns the following information:

```
{
  categoryName: ’Tasar Silk’,
  categoryID: ’FA0001’,
  supplierID: ’SU0005’,
  supplierName: ’Appel’,
  supplierPhoneNumbers: [ ’0656506568’, ’0905560655’ ],
  importInfos: [
    RowDataPacket {
      fabcat_code: ’FA0001’,
      supplier_code: ’SU0005’,
```



Search material purchasing information

Material Information

Search by ID

Category Name

Category ID *: FA0001

Supply Information

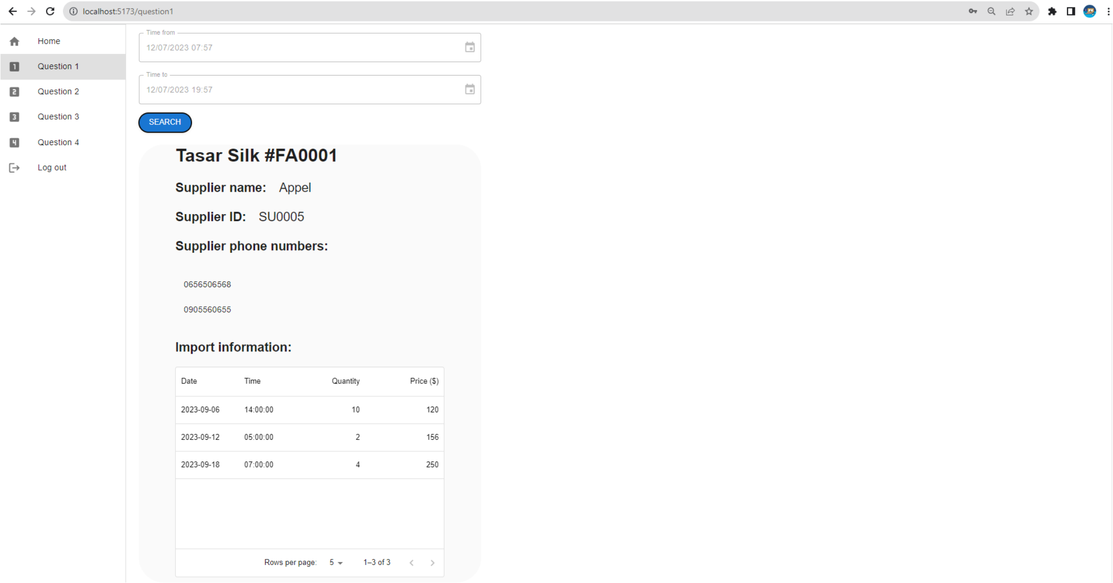
Date time range

Time from: 12/05/2023 11:19

Time to: 12/05/2023 23:19

```
   import_date: ’2023-09-06’,
   import_time: ’14:00:00’,
   quantity: 10,
   price: 120
  },
  RowDataPacket {
   fabcat_code: ’FA0001’,
   supplier_code: ’SU0005’,
   import_date: ’2023-09-12’,
   import_time: ’05:00:00’,
   quantity: 2,
   price: 156
  },
  RowDataPacket {
   fabcat_code: ’FA0001’,
   supplier_code: ’SU0005’,
   import_date: ’2023-09-18’,
   import_time: ’07:00:00’,
   quantity: 4,
   price: 250
  }
 ]
}
```

Which is displayed in our website as



Time from: 12/07/2023 07:57

Time to: 12/07/2023 19:57

Tasar Silk #FA0001

Supplier name: Appel

Supplier ID: SU0005

Supplier phone numbers:

0656506568

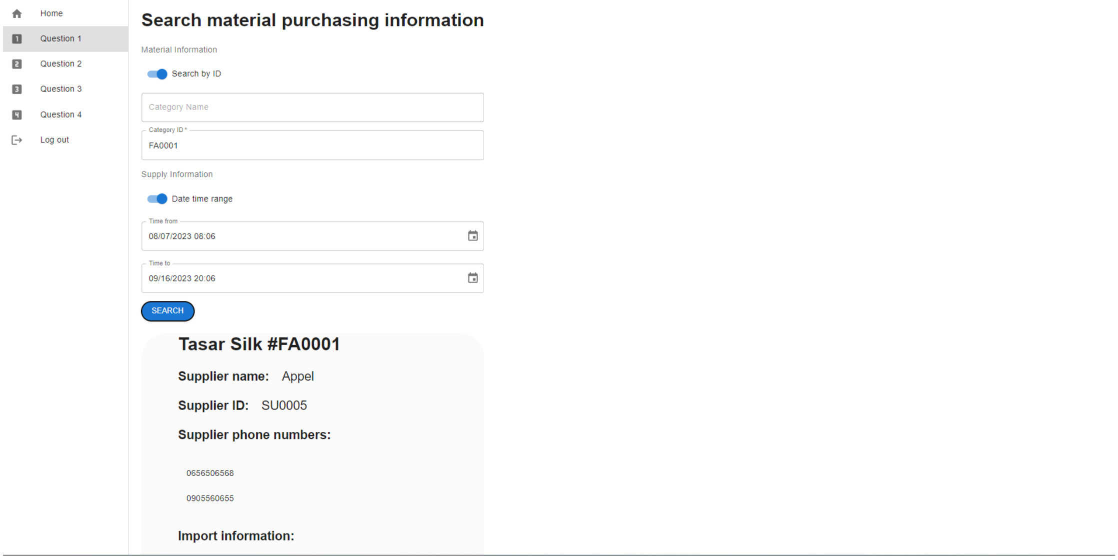
0905560655

Import information:

Date	Time	Quantity	Price ($)
2023-09-06	14:00:00	10	120
2023-09-12	05:00:00	2	156
2023-09-18	07:00:00	4	250

Rows per page: 5 1–3 of 3

If we want to search for supply information within a certain date/time range, we can enable this by pressing the switch. Note that date range and time range are separate.



Search material purchasing information

Material Information

Search by ID

Category Name

Category ID *: FA0001

Supply Information

Date time range

Time from: 08/07/2023 08:06

Time to: 09/16/2023 20:06

Tasar Silk #FA0001

Supplier name: Appel

Supplier ID: SU0005

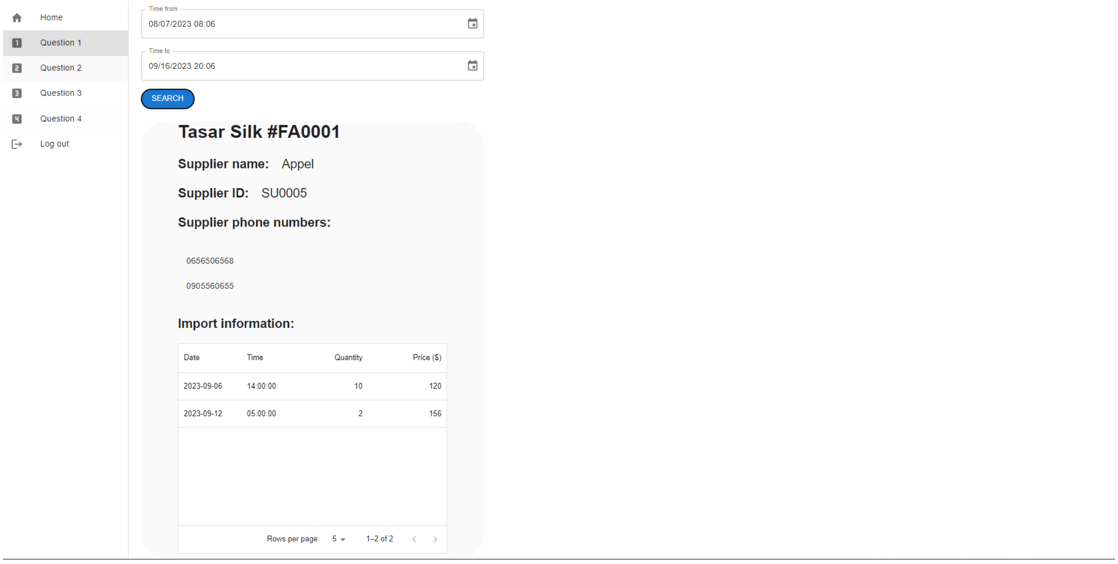
Supplier phone numbers:

0656506568

0905560655

Import information:

The last import does not fit into this date-time range, and this time they don't show up in our result



Time from: 08/07/2023 08:06

Time to: 09/16/2023 20:06

Tasar Silk #FA0001

Supplier name: Appel

Supplier ID: SU0005

Supplier phone numbers:

0656506568

0905560655

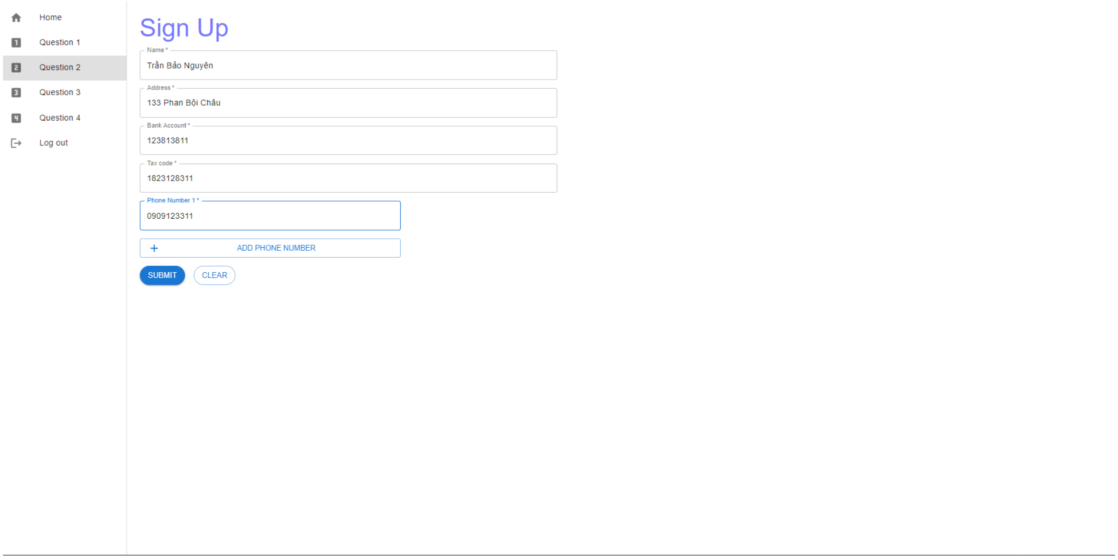
Import information:

Date	Time	Quantity	Price ($)
2023-09-06	14:00:00	10	120
2023-09-12	05:00:00	2	156

Rows per page: 5 1–2 of 2

5.3.2 Add information for a new supplier

In this question, we add a new supplier with the information discussed in the previous section. Since a supplier can have multiple phone numbers, there is a button to include an extra phone number. In this example, we input two phone numbers. Regular expressions are used to check if the input of some of these values like Bank Account, Tax Code or Phone Number are in the right format.



Sign Up

Name *: Trần Bảo Nguyên

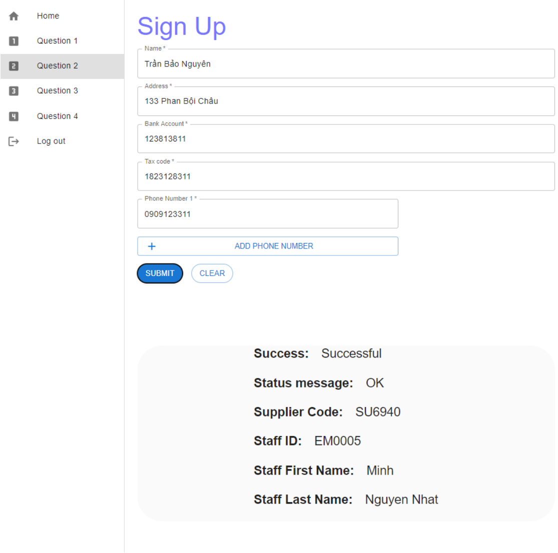
Address *: 133 Phan Bội Châu

Bank Account *: 123813811

Tax code *: 1823128311

Phone Number 1 *: 0909123311

Since this is a POST request, no response data is needed except for the response status. In our implementation, however, extra information is shown for a partner staff which is assigned randomly to the supplier. The supplier code is also assigned randomly by the back-end application. Response status can be either "Successful", error code 400 for trying to add data that already existed or error code 500 for internal server error. Detail implementations can be seen in the back-end code file.



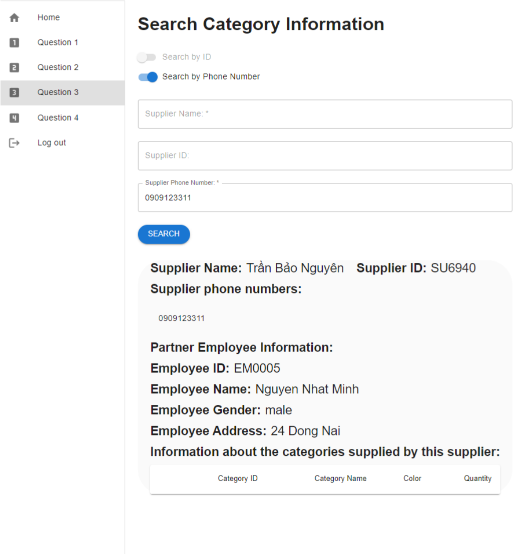
Sign Up

Name *: Trần Bảo Nguyên
Address *: 133 Phan Bội Châu
Bank Account *: 123813811
Tax code *: 1823128311
Phone Number 1 *: 0909123311

Success: Successful
Status message: OK
Supplier Code: SU6940
Staff ID: EM0005
Staff First Name: Minh
Staff Last Name: Nguyen Nhat

5.3.3 List details of all categories which are provided by a supplier

To check the information of the last supplier added into the database we can use this question as an example. There are two switches, one for searching with supplier ID and one for searching with supplier's phone number. The default option is to search with supplier's name; turning on the first switch lets you search by ID and turning on the other switch overrules the last switch and lets you search by phone number. In the first example for this section, we will use the phone number that we have just inputted in the previous example.



Search Category Information

○ Search by ID
● Search by Phone Number

Supplier Name: *
Supplier ID:
Supplier Phone Number: *: 0909123311

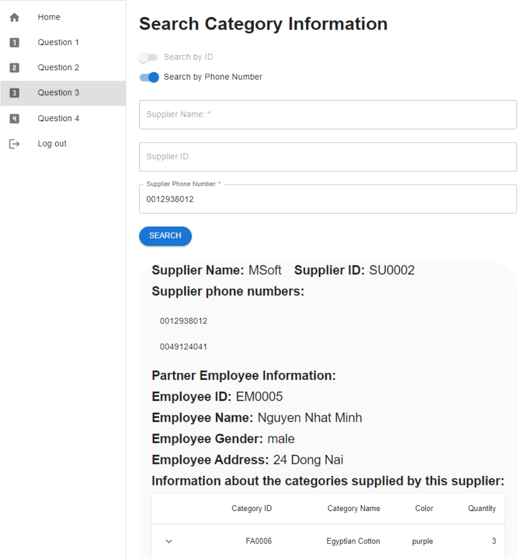
Supplier Name: Trần Bảo Nguyên **Supplier ID:** SU6940
Supplier phone numbers:

0909123311

Partner Employee Information:
Employee ID: EM0005
Employee Name: Nguyen Nhat Minh
Employee Gender: male
Employee Address: 24 Dong Nai
Information about the categories supplied by this supplier:

Category ID	Category Name	Color	Quantity

This supplier doesn't currently supply any fabric category however, because no information about them was inputted. To have the program display the full information list shown in the API, this second example will use a pre-existing supplier.



Search Category Information

○ Search by ID
● Search by Phone Number

Supplier Name: *

Supplier ID:

Supplier Phone Number: * 0012938012

Supplier Name: MSoft **Supplier ID:** SU0002
Supplier phone numbers:

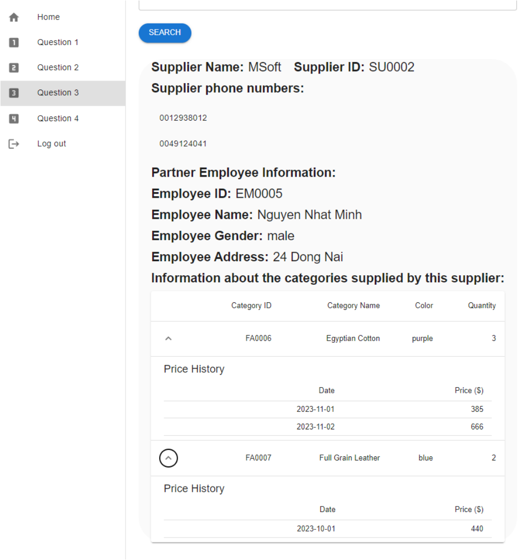
0012938012

0049124041

Partner Employee Information:
Employee ID: EM0005
Employee Name: Nguyen Nhat Minh
Employee Gender: male
Employee Address: 24 Dong Nai
Information about the categories supplied by this supplier:

	Category ID	Category Name	Color	Quantity
⌄	FA0006	Egyptian Cotton	purple	3

We use a collapsible table to provide full information about the prices of each category that this supplier supplies.



Supplier Name: MSoft **Supplier ID:** SU0002
Supplier phone numbers:

0012938012

0049124041

Partner Employee Information:
Employee ID: EM0005
Employee Name: Nguyen Nhat Minh
Employee Gender: male
Employee Address: 24 Dong Nai
Information about the categories supplied by this supplier:

	Category ID	Category Name	Color	Quantity
^	FA0006	Egyptian Cotton	purple	3

Price History

Date	Price ($)
2023-11-01	385
2023-11-02	666

	Category ID	Category Name	Color	Quantity
^	FA0007	Full Grain Leather	blue	2

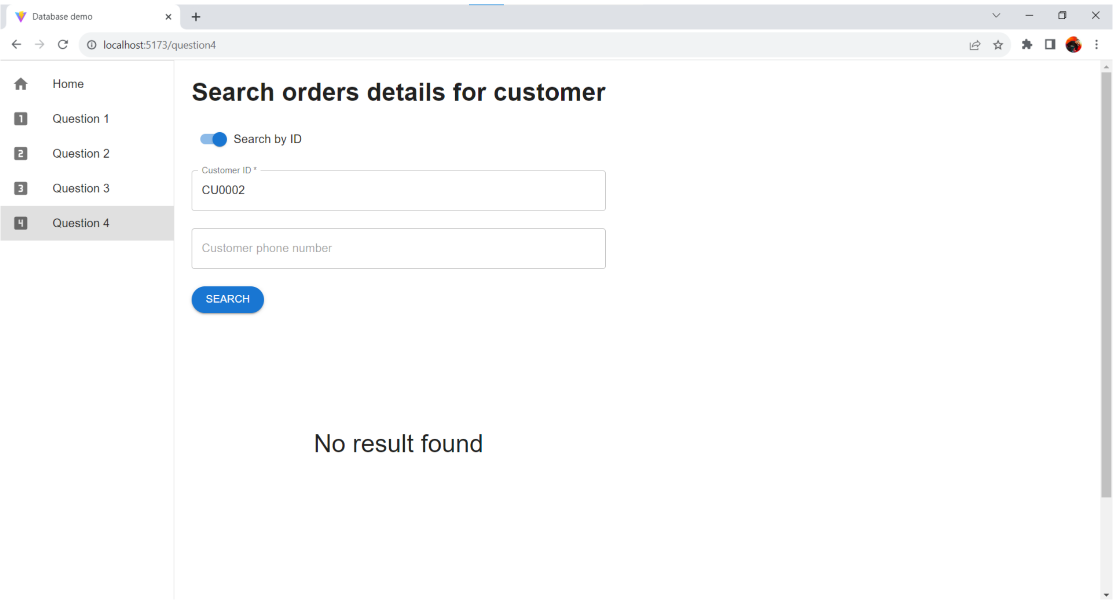
Price History

Date	Price ($)
2023-10-01	440

5.3.4 Make a report that provides full information about the order for each category of a customer

In question 4, we are required to make a report that provides full information about the order for each category of a customer. There are options to search for this customer, either by their

ID in the system or by their registered phone number. In this example, we will find the report of the customer with ID "CU0002"



Search orders details for customer

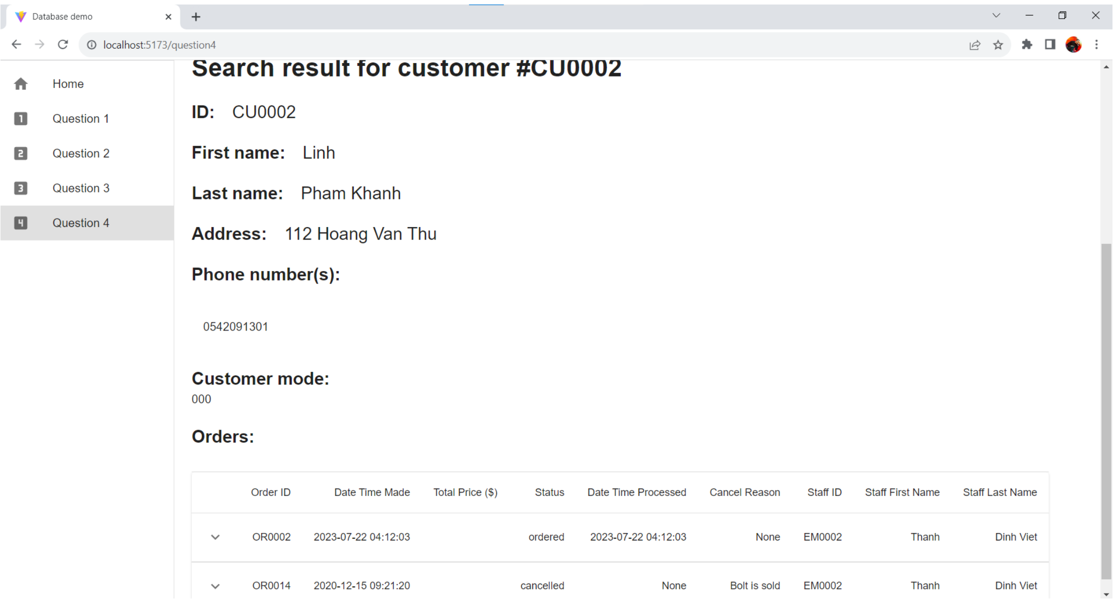
Search by ID

Customer ID *
CU0002

Customer phone number

No result found

The returned result has the following information.



Search result for customer #CU0002

ID: CU0002

First name: Linh

Last name: Pham Khanh

Address: 112 Hoang Van Thu

Phone number(s):

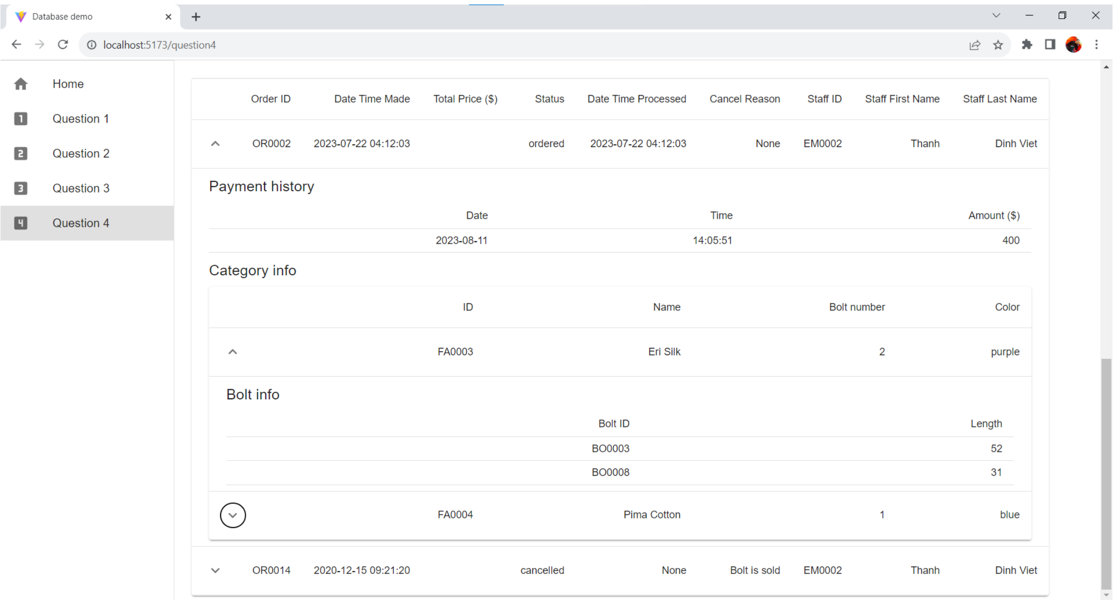
0542091301

Customer mode:
000

Orders:

	Order ID	Date Time Made	Total Price ($)	Status	Date Time Processed	Cancel Reason	Staff ID	Staff First Name	Staff Last Name
⌄	OR0002	2023-07-22 04:12:03		ordered	2023-07-22 04:12:03	None	EM0002	Thanh	Dinh Viet
⌄	OR0014	2020-12-15 09:21:20		cancelled	None	Bolt is sold	EM0002	Thanh	Dinh Viet

We can see that apart from the usual customer information, the details about each order is nicely wrapped inside a double-collapsible table. These information include the date and time the order was made and process, the order status and cancelled reason (if the order was cancelled), the payment history and the detail of each category that is in the order.



	Order ID	Date Time Made	Total Price ($)	Status	Date Time Processed	Cancel Reason	Staff ID	Staff First Name	Staff Last Name
^	OR0002	2023-07-22 04:12:03		ordered	2023-07-22 04:12:03	None	EM0002	Thanh	Dinh Viet

Payment history

Date	Time	Amount ($)
2023-08-11	14:05:51	400

Category info

	ID	Name	Bolt number	Color
^	FA0003	Eri Silk	2	purple

Bolt info

Bolt ID	Length
BO0003	52
BO0008	31

	ID	Name	Bolt number	Color
⌄	FA0004	Pima Cotton	1	blue

	Order ID	Date Time Made	Total Price ($)	Status	Date Time Processed	Cancel Reason	Staff ID	Staff First Name	Staff Last Name
⌄	OR0014	2020-12-15 09:21:20		cancelled	None	Bolt is sold	EM0002	Thanh	Dinh Viet

6 Database Management

6.1 A - Proving one use-case of indexing efficiency in your scenarios

In table import_info, we used the created fabcat_codes to generate more data. In other words, we create more import information for each given fabric category. Specifically, after generating data, we have totally 100,000 points of data for this table. We select this table for inserting because it is much more conveniently add more data into table with the primary key containing date and time which can be easily generated. We create query to create index on the target column (import_date)

```sql
CREATE INDEX idx_date ON fabric_agency.import_info (import_date);
```

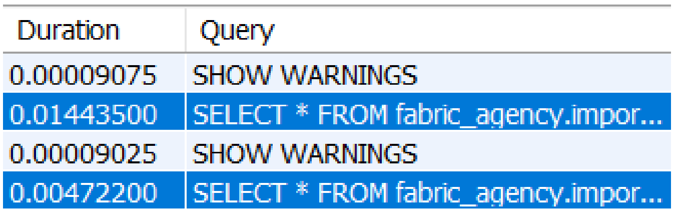
Listing 5: Exercise 2.2 a: SQL Code

The execution time of the query with and without indexing are recorded in order to compare the efficiency by the below SQL:

```sql
-- Enable profiling
SET PROFILING = 1;
-- Our query
SELECT * FROM fabric_agency.import_info
WHERE (import_date between ’2020-01-01’ and ’2020-3-31’) OR
    (import_date between ’2022-01-01’ and ’2022-3-31’) OR
    (import_date between ’2023-01-01’ and ’2022-3-31’);
-- Disable profiling
SET PROFILING = 0;
-- Show profiles
SHOW PROFILES;
```

Listing 6: Exercise 2.2 a: SQL Code

Conclusion: It is clearly seen that with indexing the execution time is faster about 3 times. With a simple indexing on the importing date column, we still observe an acceptable efficiency.



Duration	Query
0.00009075	SHOW WARNINGS
0.01443500	SELECT * FROM fabric_agency.impor...
0.00009025	SHOW WARNINGS
0.00472200	SELECT * FROM fabric_agency.impor...

Figure 1: Execution times of query without indexing and with indexing, respectively

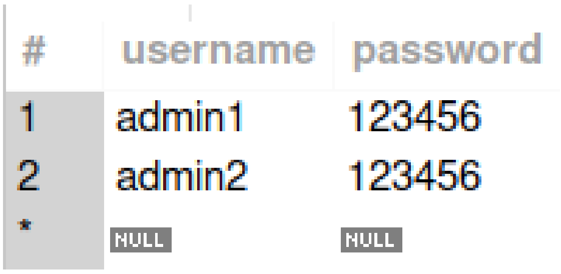
6.2 B - Solving one use-case of database security in your scenarios

Because the application requires administrative access to be able to access to the internal information such fabric categories, suppliers, customers, etc, so it is required that the authentication protocol has high level of reliability and resistance. Not only that, when we scale up the system, there may be authentication features for regular or organizational users (such as customers, suppliers) as well. Regarding the security risks of the the authentication protocol of our database, they can be generally classified into 2 categories: **External threats** and **Internal threats**.

6.2.1 External security threats

For **External threats**, it encapsulates all the ways that an attacker from outside the company organization can exploit the system's weaknesses. There are many threats for this one, but in the scope of our assignment, we only consider about the *SQL Injection Attack*. It is a type of attack when an attacker tries to insert a malicious into input fields of a web application. In our case, without "Safe login" and *SQL inject* correctly, an attacker can completely bypass the authentication without having to know password or even username of 1 of the admins.

In our "Unsafe login" case, we store the authentication data in the database with 2 columns: "username" and "password", with "password" is in the plain text form. Each time user submit their `username` and `password`, those 2 data will be substitute directly into a SQL code to extract a record that matches the information.



#	username	password
1	admin1	123456
2	admin2	123456
*	NULL	NULL

Figure 2: Unsafe password storing

```
const unsafeLogin = (req) => {
  return new Promise((resolve, reject) => {
    const query = ‘SELECT * FROM admin_account WHERE username = '${req["username"]}' AND password = '${req["password"]}'‘;
    db.query(query, (err, result) => {
      if (err) {
        reject(err);
      } else {
        resolve(result);
      }
    });
  });
}
```

Listing 7: Unsafe login

This poses a problem because if the inputs contains quotes ” or ’, the query field can be ended early and after that, we can put any SQL command we want and it can be executed accordingly. Then if we put something in the ¡username¿ field `admin1` in the ¡password¿ field like `’ OR 1=1; #`, the resulting query will be like this

```
SELECT * FROM admin_account WHERE username = ’admin1’ AND password = ’’ OR 1=1; #’
```

Listing 8: Unsafe query

The bit after # will be ignored and thus, leave us with the query to find the records which satisfies the condition: `username = ’admin1’ AND password = ’’ OR 1=1`. That condition is always `TRUE`, thus given that there is 1 or more records in our admin_account table, we will always be able to login, without needing to know username and password.

Solution: The solution is instead of substituting directly the user's input into the query, we use **parameterized query** to execute the query. Under the surface, **parameterized query** extract the logic of the query from the actual data that are passed to it, thus the data will always be data, not executable SQL code. And thus it is safer to use **parameterized query** in our case.

```
SELECT * FROM admin_account WHERE username = ? AND password = ?;
```

Listing 9: Parameterized query

But this is not the final solution to our authentication problem, because we also have to deal with **Internal security threats**

6.2.2 Internal security threats

Internal threats refer to security risks that originate from within an organization or involve individuals with legitimate access to the database. In our case, if a person with privileged access to the database has malicious intends, they can leak and sell out the username and password of all admins or potentially regular users (if we scale up the system in the future). And there are different ways we can store the passwords of our users, which includes:

1. **Plain text:** This is the worse way we can store our users' passwords, because if a insider can sneak out those information, everyone's password will be visible to everybody.

2. **Encrypted:** This may seems like a good way to store passwords because in order to know the password in plain text, we have to have not only the encrypted password but also the key to decrypt that password. But if a insider has the access to the key, all passwords are like plain text again.

3. **Hashed:** This is the better way to store password the password, as hash algorithms are irreversible, meaning we can only check if a password is correct by hashing that password using the same hash algorithm and compare the result with the one stored in the database. If the result matches, it is indeed the password stored, and if not, it is not the correct

password. While this may sounds excellent, it is still susceptible to attacks like "rainbow table" or "dictionary" attack.

A **rainbow table attack** is a type of precomputed attack used to crack password hashes. It's an attempt to reverse-engineer password hashes stored in a database by comparing them to precomputed tables of possible hash values. Attackers generate rainbow tables, which are essentially large databases of precomputed hash chains. A hash chain is a sequence of hashes generated by repeatedly applying a hash function to an initial input. The tables store pairs of plain text passwords and their corresponding hash values. If an attacker gains unauthorized access to a database containing hashed passwords, they can compare the hashed values to entries in the rainbow table. If a match is found, the corresponding plain text password is known.

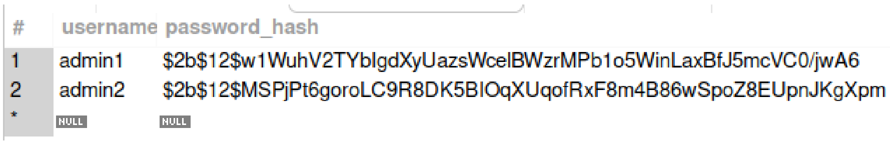
A **dictionary attack** is a type of attack where an attacker systematically tries to guess passwords by using a precompiled list of likely passwords, known as a "dictionary". Unlike brute-force attacks, which systematically try every possible combination of characters, dictionary attacks are more focused and efficient because they leverage a list of commonly used passwords, words from the dictionary, or other easily guessable combinations.
Given increasingly more powerful hardwares being manufactured, common hashing algorithm like MD-5, SHA family (SHA-128, SHA-256, or SHA-512) can be executed very fast, up to millions or billions of computational results per second. Given that, using **dictionary attack** combined with **rainbow table attack**, attackers can compute the hashes very quickly and in the process, many passwords may be known.

Solution: In essence, the strength of **rainbow table attack** is that common passwords can be pre-hashed, and the strength of **dictionary attack** is that common passwords can be brute forced.

For **rainbow table attack**, we want to make our stored hashed passwords cannot be pre-computed. The solution is to utilize a technique called **salting**. It is a technique used to enhance the security of password storage by introducing a random and unique value, called a "salt," before hashing a password. That way, **rainbow table attack** are not a threat anymore, as suppose there are 2 passwords that are identical, because of the randomly generated salt obtained just before hashing, the 2 hashed values can be completely different.

For **dictionary attack**, we might want to make the process of computing the hashes slow and computationally expensive. Slow hashing, also known as key stretching or intentionally slow hashing, is a security practice that involves deliberately slowing down the password hashing process to make it computationally expensive and time-consuming for attackers. The goal is to increase the difficulty and resources required to perform brute-force attacks and other password-cracking techniques. If we configure correctly, the computational throughput can scale down from millions or billions result per second to only thousands per second. Obviously, this solution is not perfect against **dictionary attack**, and there may be a few passwords can be found, but the damage is much lower and perhaps the attacker can shift their attention to other weaker database system, leaving our system alone.

So in our application, we decided to use a hashing algorithm called **bcrypt**. This hashing algorithm combines both of the solutions: salting and slow hashing. Using this, we can compute and stores the hashed password in our database. If the users create password with responsibility (long, involves many characters, number), the authentication data in our database are very hard to crack.



#	username	password_hash
1	admin1	$2b$12$w1WuhV2TYblgdXyUazsWcelBWzrMPb1o5WinLaxBfJ5mcVC0/jwA6
2	admin2	$2b$12$MSPjPt6goroLC9R8DK5BIOqXUqofRxF8m4B86wSpoZ8EUpnJKgXpm
*	NULL	NULL

Figure 3: Safer password storing (both account have the same password ”123456”, but completely different hash)

```
const get_username_and_hashed_password = (username) => {
    return new Promise((resolve, reject) => {
        const query = 'SELECT * FROM admin_account_hash WHERE username = ?';
        db.query(query, [username], (err, result) => {
            if (err) {
                reject(err);
            } else {
                resolve(result);
            }
        })
  })}

const safeLogin = async (req) => {
  try {
    const username = req["username"];
    const enteredPassword = req["password"];

    const result = await get_username_and_hashed_password(username);

    if (result.length === 0) {
      return [];
    }

    const storedHashedPassword = result[0]['password_hash'];

    const passwordMatch = await bcrypt.compare(enteredPassword,
    storedHashedPassword);

    if (passwordMatch) {
      return result;
    } else {
      return [];
    }
  } catch (error) {
    console.log(error);
    throw error;
  }
};
```

Listing 10: Safer login query

Coming back to the login page, the ”Safe login” switch is simply a switch that toggle between the safe and unsafe login protocol. If toggle off, we can bypass the login by simple SQL injection, and if toggle on, **bcrypt** algorithm is applied, and we cannot login without using the right username and password.