DATABASE MANAGEMENT SYSTEMS - CS6106

PROJECT

ONLINE CLOTHING STORE MANAGEMENT SYSTEM

Prepared and submitted by
Nithin K - 2019103550
Safiya Fathima - 2019103052

INDEX

PROJECT ABSTRACT

The project 'Online Clothing Store Management System' is developed to maintain a record of clothes, users, carts and bills of a particular Ecommerce clothing website.

Languages, frameworks and tools used:

- Front-end: HTML, CSS, JavaScript, Bootstrap
- Back-end: NodeJS
- Database Server: PostgreSQL Server 13.2
- Database Tools: psql (PostgreSQL) Shell 13.2, pgAdmin 4

FUNCTIONAL AND NON FUNCTIONAL REQUIREMENTS

Functional Requirements:

The system maintains the data of:

- Clothing details and available quantity
- User details
- Items currently in every user's cart
- Items ordered by every user
- Billing details of every user

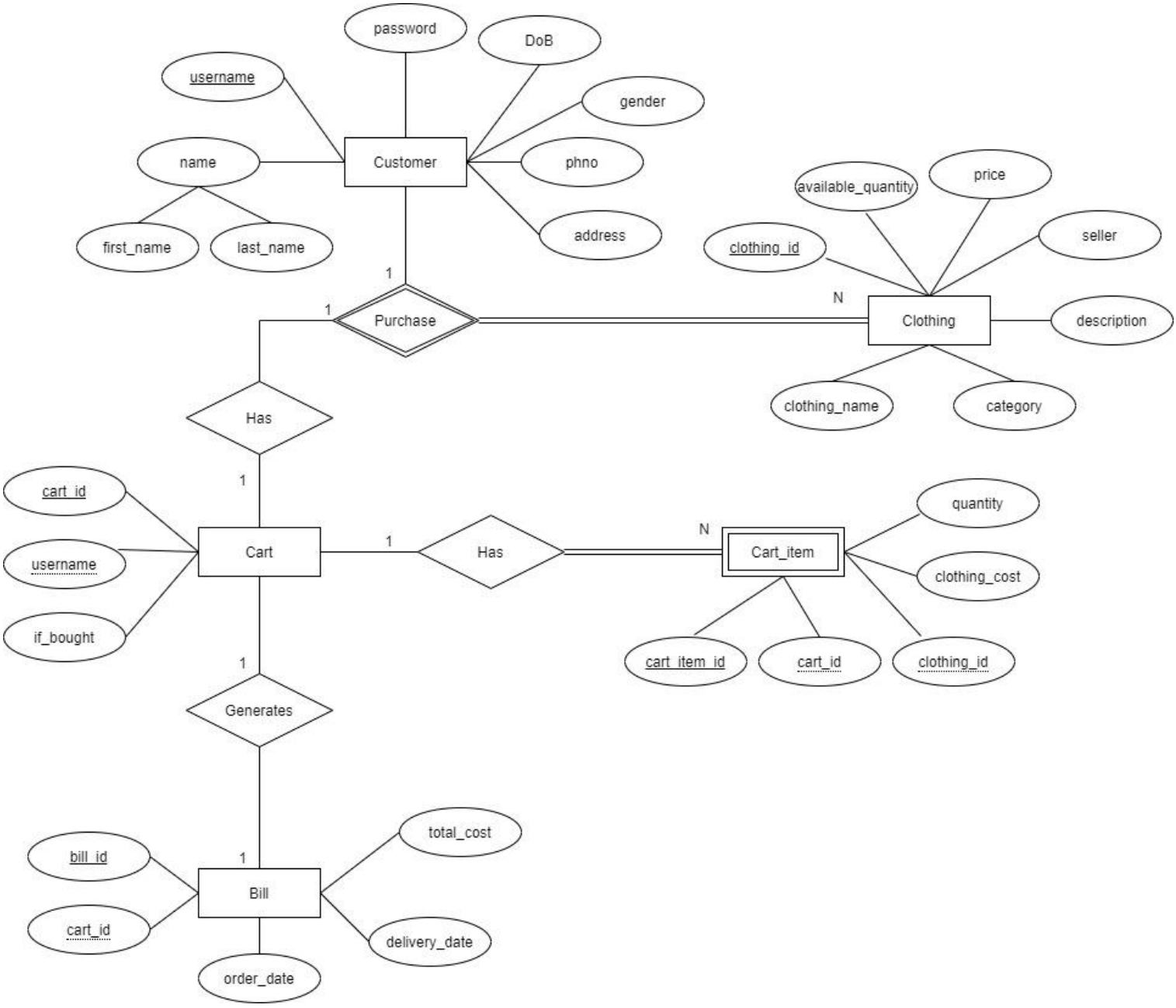
Non Functional Requirements:

The system provides the following:

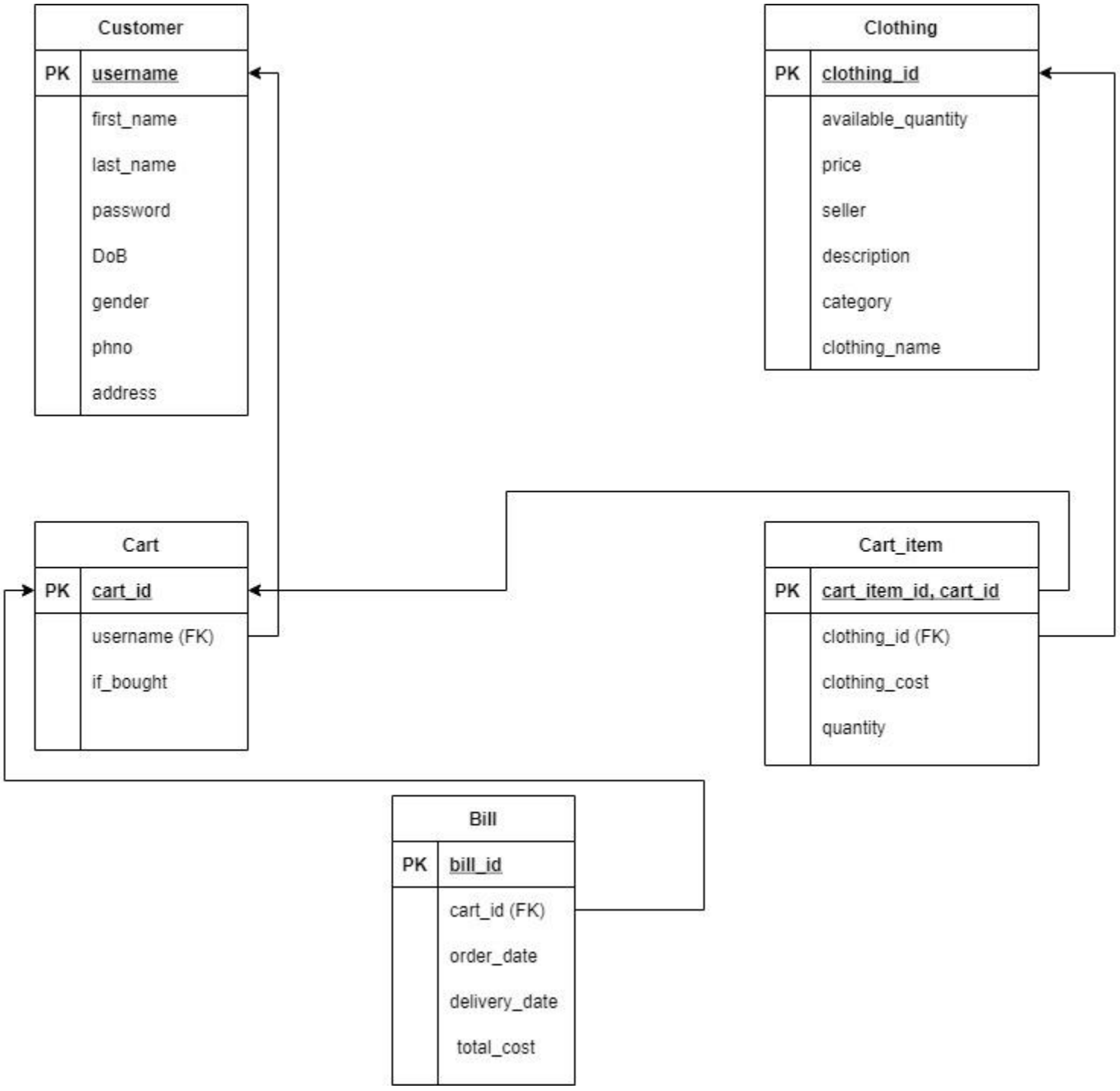
- Security
  - ➔ Hashing of passwords stored in the database.

- User Interface
  - ➔ A friendly and intuitive user interface has been developed for better user experience.

ENTITY RELATIONSHIP DIAGRAM



RELATIONAL SCHEMA



USER INTERFACE DESIGN

The user interface has been constructed using HTML, CSS and JavaScript, with help from frameworks like Bootstrap to get a pleasant user interface.

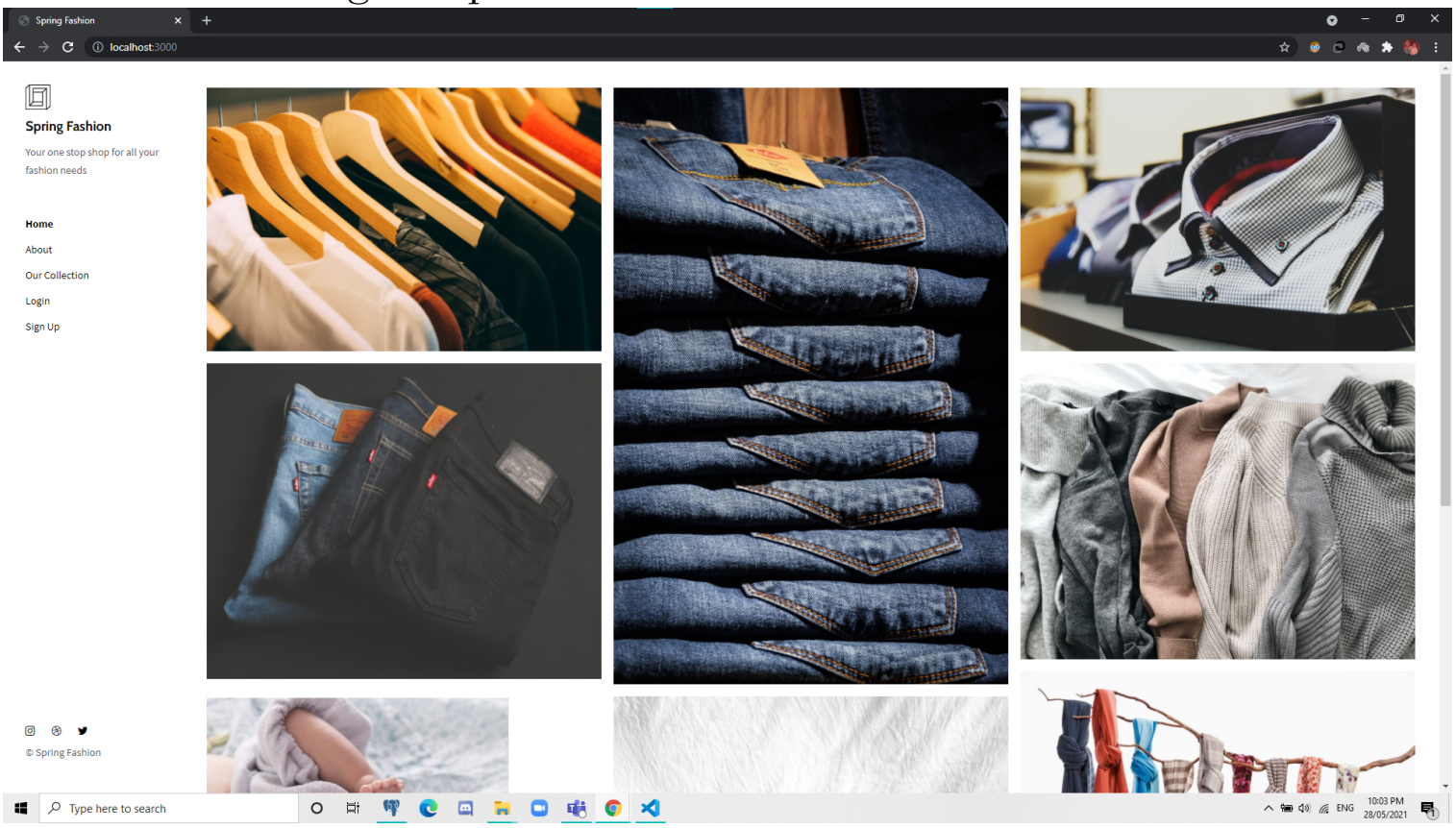
Here is a sample code snippet for the landing page of the application:

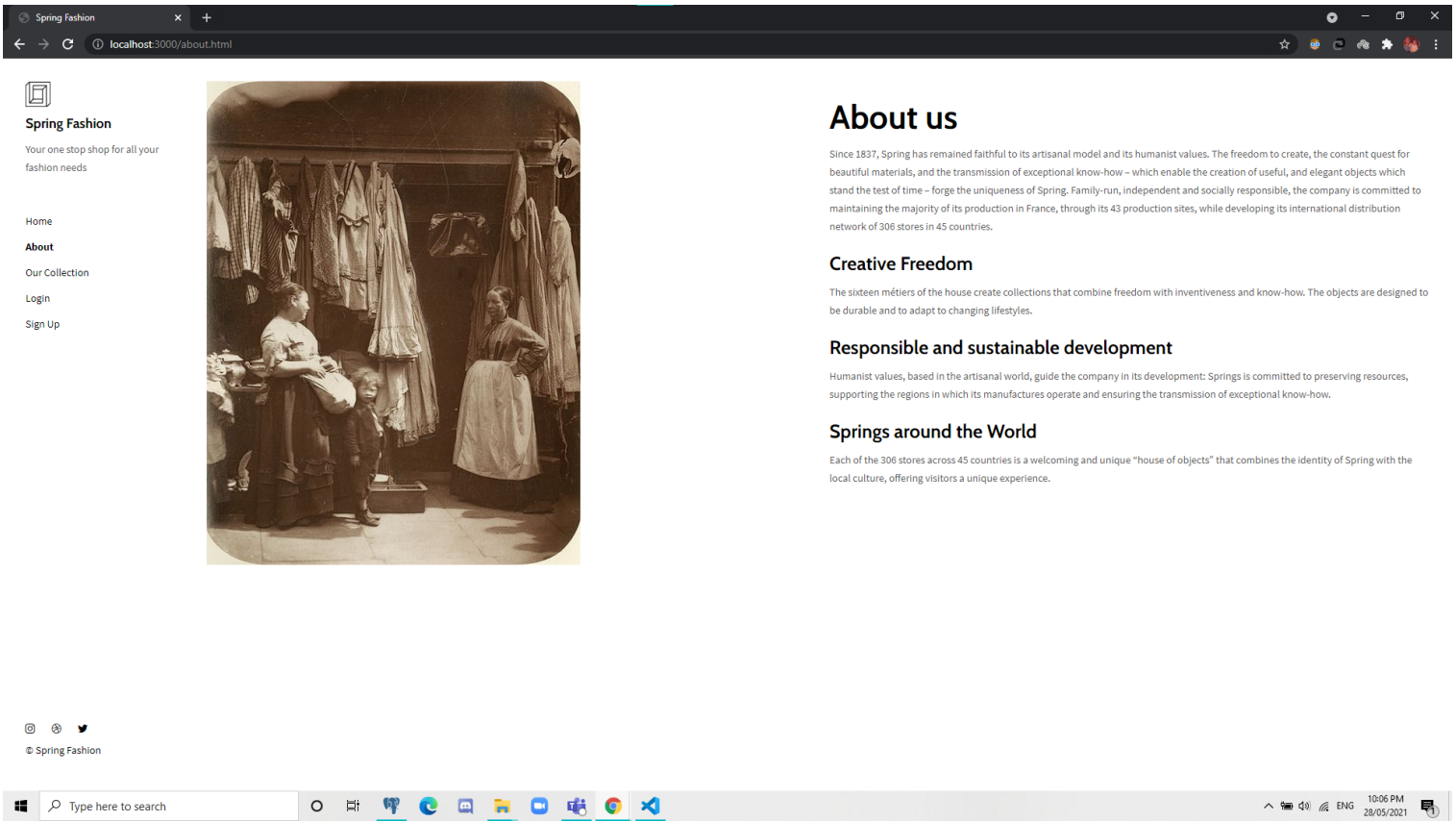
index.html

main-app > public > index.html > html > head

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta content="IE=edge" http-equiv="X-UA-Compatible">
  <meta content="width=device-width,initial-scale=1" name="viewport">
  <meta content="Page description" name="description">
  <meta name="google" content="notranslate" />

  <!-- Disable tap highlight on IE -->
  <meta name="msapplication-tap-highlight" content="no">

  <link href="./assets/apple-icon-180x180.png" rel="apple-touch-icon">
  <link href="./assets/favicon.ico" rel="icon">



  <title>Spring Fashion</title>

<link href="./main.82cfd66e.css" rel="stylesheet"></head>

<body>

<!-- Add your content of header -->
<header class="">
  <div class="navbar navbar-default visible-xs">
    <button type="button" class="navbar-toggle collapsed">
      <span class="sr-only">More</span>
      <span class="icon-bar"></span>
      <span class="icon-bar"></span>
      <span class="icon-bar"></span>
    </button>
    <a href="./index.html" class="navbar-brand">Spring Fashion</a>
  </div>
```

With the following end products:

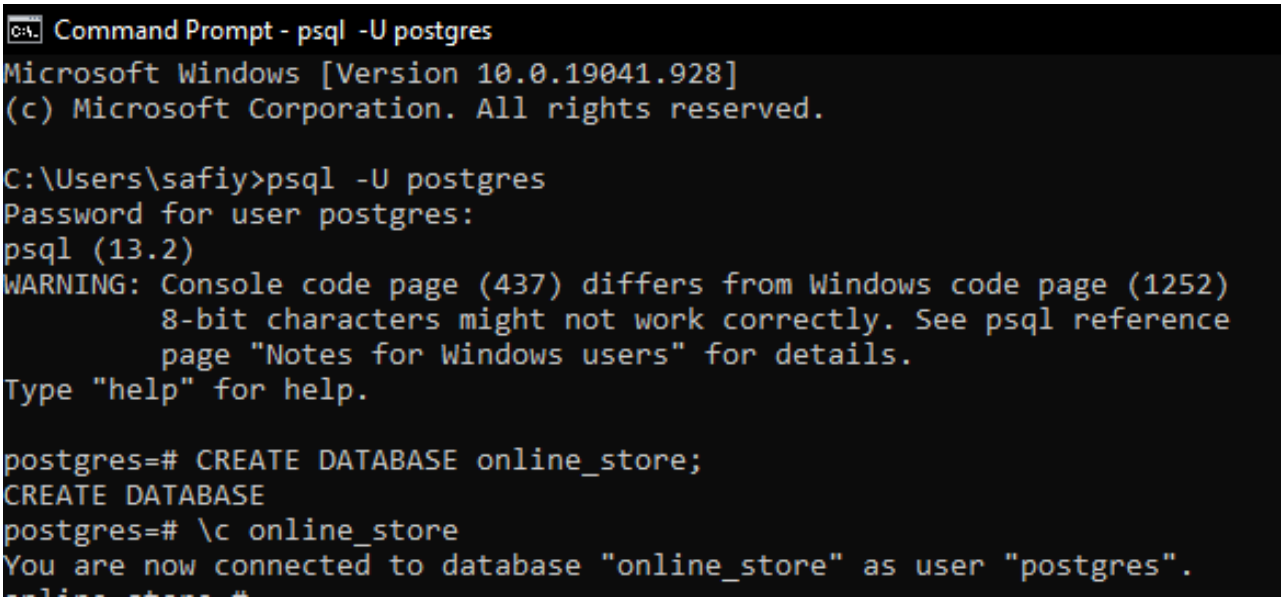




CONNECTING TO THE DATABASE

PostgreSQL database has been used for the implementation of the system.

An online_store database is created using the psql Shell which is accessed through the Windows command prompt.



In NodeJS:

```
const connectionString = 'postgressql://postgres:060669@localhost:5432/online_store';
const { Client } = require('pg');

const client = new Client({
    connectionString
})
client.connect();
```

SETTING UP RELATIONS

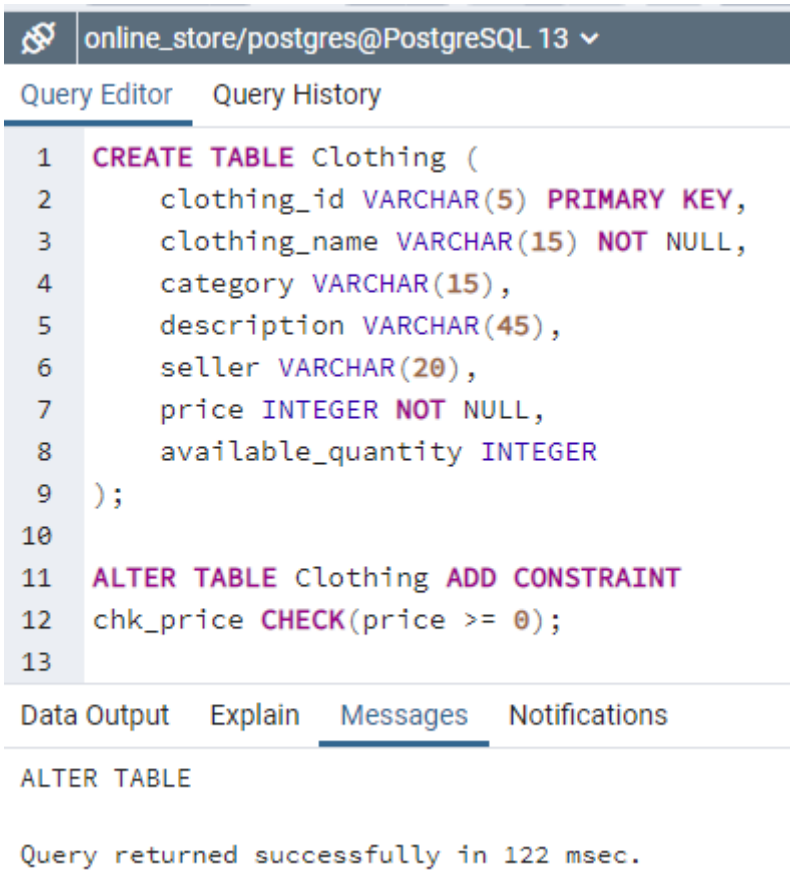
The following relations have been created using the pgAdmin GUI interface:

1. Clothing

   Attributes:

   - **Clothing_id** (primary key, uniquely identifies a clothing)
   - Clothing_name
   - Category
   - Description
   - Seller
   - Price
   - Available_quantity

All attributes are non-transitively determined by the primary key - clothing_id. Hence the relation is in **Boyce-Codd Normal Form.**



```
online_store=# \d+ clothing
                                                Table "public.clothing"
      Column       |          Type          | Collation | Nullable | Default | Storage  | Stats target | Description
-------------------+------------------------+-----------+----------+---------+----------+--------------+-------------
 clothing_id       | integer                |           | not null |         | plain    |              |
 clothing_name     | character varying(15)  |           | not null |         | extended |              |
 category          | character varying(15)  |           |          |         | extended |              |
 description       | character varying(200) |           |          |         | extended |              |
 seller            | character varying(20)  |           |          |         | extended |              |
 price             | integer                |           | not null |         | plain    |              |
 available_quantity | integer                |           |          |         | plain    |              |
Indexes:
    "clothing_pkey" PRIMARY KEY, btree (clothing_id)
Check constraints:
    "chk_price" CHECK (price >= 0)
Referenced by:
    TABLE "cart_item" CONSTRAINT "clothingid_fk" FOREIGN KEY (clothing_id) REFERENCES clothing(clothing_id)
Access method: heap
```

2. Customer

Attributes:

- **Username** (primary key, uniquely identifies a customer)
- Password
- Name - First_name and Last_name
- DoB
- Gender
- Phno
- Email
- address

All attributes are non-transitively determined by the primary key - username. Hence the relation is in **Boyce-Codd Normal Form.**

online_store/postgres@PostgreSQL 13

Query Editor | Query History

```
1   CREATE TABLE Customer (
2       username VARCHAR(10) PRIMARY KEY NOT NULL,
3       pass VARCHAR(15) NOT NULL,
4       first_name VARCHAR(20),
5       last_name VARCHAR(20),
6       DoB DATE,
7       gender CHAR,
8       phno INTEGER,
9       address VARCHAR(50)
10  );
11
12  ALTER TABLE Customer ADD CONSTRAINT
13  chk_gender CHECK (gender IN('M', 'F', 'O'));
14
```

Data Output | Explain | Messages | Notifications

```
ALTER TABLE

Query returned successfully in 166 msec.
```

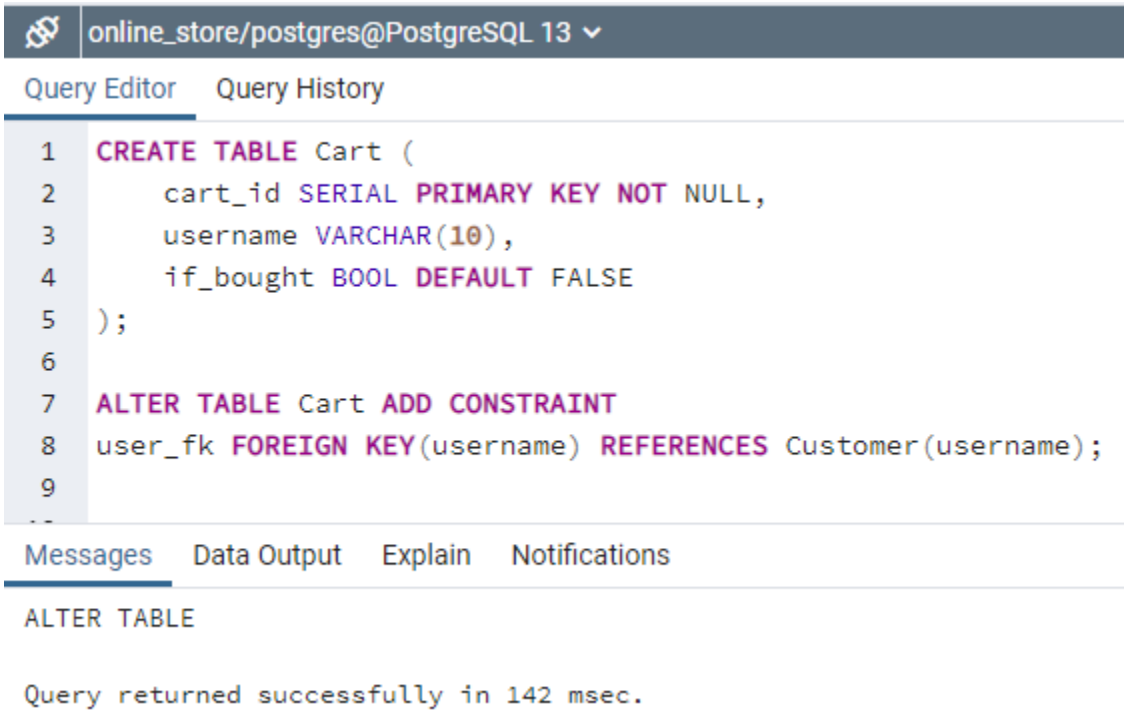
```
online_store=# \d+ customer
                                        Table "public.customer"
   Column   |          Type          | Collation | Nullable | Default | Storage  | Stats target | Description
------------+------------------------+-----------+----------+---------+----------+--------------+-------------
 username   | character varying(20)  |           | not null |         | extended |              |
 pass       | character varying(65)  |           | not null |         | extended |              |
 first_name | character varying(100) |           |          |         | extended |              |
 last_name  | character varying(100) |           |          |         | extended |              |
 dob        | date                   |           |          |         | plain    |              |
 gender     | character(1)           |           |          |         | extended |              |
 phno       | bigint                 |           |          |         | plain    |              |
 email      | text                   |           | not null |         | extended |              |
 address    | character varying(500) |           |          |         | extended |              |
Indexes:
    "customer_pkey" PRIMARY KEY, btree (username)
    "customer_email_key" UNIQUE CONSTRAINT, btree (email)
Check constraints:
    "chk_gender" CHECK (gender = ANY (ARRAY['M'::bpchar, 'F'::bpchar, 'O'::bpchar]))
Referenced by:
    TABLE "cart" CONSTRAINT "user_fk" FOREIGN KEY (username) REFERENCES customer(username)
Access method: heap
```

3. Cart

Attributes:

- **Cart_id** (primary key, uniquely identifies a cart)
- Username - Foreign Key derived from customer table
- If_bought - A variable to indicate if a cart is open or closed, default value is FALSE

All attributes are non-transitively determined by the primary key - cart_id. Hence the relation is in **Boyce-Codd Normal Form**.




```
online_store=# \d+ cart
                                                  Table "public.cart"
  Column   |         Type          | Collation | Nullable |                Default                | Storage  | Stats target | Description
-----------+-----------------------+-----------+----------+---------------------------------------+----------+--------------+-------------
 cart_id   | integer               |           | not null | nextval('cart_cart_id_seq'::regclass) | plain    |              |
 username  | character varying(10) |           |          |                                       | extended |              |
 if_bought | boolean               |           |          | false                                 | plain    |              |
Indexes:
    "cart_pkey" PRIMARY KEY, btree (cart_id)
Foreign-key constraints:
    "user_fk" FOREIGN KEY (username) REFERENCES customer(username)
Referenced by:
    TABLE "bill" CONSTRAINT "cartid_fk" FOREIGN KEY (cart_id) REFERENCES cart(cart_id)
    TABLE "cart_item" CONSTRAINT "cid_fk" FOREIGN KEY (cart_id) REFERENCES cart(cart_id)
Access method: heap
```

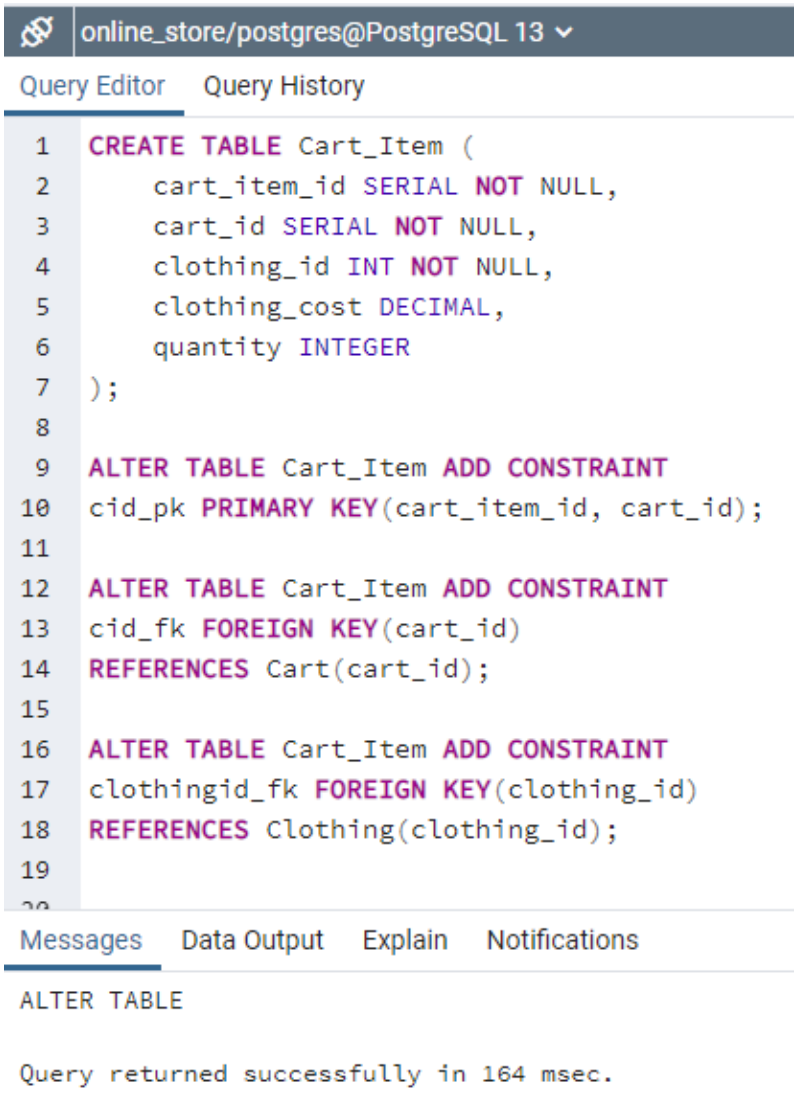
4. Cart_item

Attributes:

- **<u>Cart_item_id</u>**
- Cart_id - Foreign Key derived from Cart table
- Clothing_id - Foreign Key derived from Clothing table
- Clothing_cost
- Quantity

Here, a composite primary key of cart_item_id and cart_id is implemented to ensure no duplication.

All attributes are non-transitively determined by the primary key - cart_item_id. Hence the relation is in **Boyce-Codd Normal Form.**



```
online_store=# \d+ cart_item
                                                    Table "public.cart_item"
    Column     |  Type   | Collation | Nullable |                    Default                    | Storage | Stats target | Description
---------------+---------+-----------+----------+-----------------------------------------------+---------+--------------+-------------
 cart_item_id  | integer |           | not null | nextval('cart_item_cart_item_id_seq'::regclass) | plain   |              |
 cart_id       | integer |           | not null | nextval('cart_item_cart_id_seq'::regclass)      | plain   |              |
 clothing_id   | integer |           | not null |                                               | plain   |              |
 clothing_cost | numeric |           |          |                                               | main    |              |
 quantity      | integer |           |          |                                               | plain   |              |
Indexes:
    "cid_pk" PRIMARY KEY, btree (cart_item_id, cart_id)
Foreign-key constraints:
    "cid_fk" FOREIGN KEY (cart_id) REFERENCES cart(cart_id)
    "clothingid_fk" FOREIGN KEY (clothing_id) REFERENCES clothing(clothing_id)
Access method: heap
```

5. Bill

Attributes:

- **Bill_id** (primary key, uniquely identifies a bill)
- Cart_id - Foreign Key derived from Cart table
- Order_date
- Delivery_date
- Total_cost

All attributes are non-transitively determined by the primary key - cart_item_id. Hence the relation is in **Boyce-Codd Normal Form.**

online_store/postgres@PostgreSQL 13

Query Editor | Query History

```
1  CREATE TABLE Bill (
2      bill_id SERIAL PRIMARY KEY NOT NULL,
3      cart_id BIGINT UNIQUE NOT NULL,
4      order_date DATE,
5      delivery_date DATE,
6      total_cost BIGINT
7  );
8
9  ALTER TABLE Bill ADD CONSTRAINT
10 cartid_fk FOREIGN KEY(cart_id)
11 REFERENCES Cart(cart_id);
```

Messages | Data Output | Explain | Notifications

```
ALTER TABLE

Query returned successfully in 52 msec.
```

```
online_store=# \d+ bill
                                                  Table "public.bill"
    Column     |  Type   | Collation | Nullable |                Default                | Storage | Stats target | Description
---------------+---------+-----------+----------+---------------------------------------+---------+--------------+-------------
 bill_id       | integer |           | not null | nextval('bill_bill_id_seq'::regclass) | plain   |              |
 cart_id       | bigint  |           | not null |                                       | plain   |              |
 order_date    | date    |           |          |                                       | plain   |              |
 delivery_date | date    |           |          |                                       | plain   |              |
 total_cost    | bigint  |           |          |                                       | plain   |              |
Indexes:
    "bill_pkey" PRIMARY KEY, btree (bill_id)
    "bill_cart_id_key" UNIQUE CONSTRAINT, btree (cart_id)
Foreign-key constraints:
    "cartid_fk" FOREIGN KEY (cart_id) REFERENCES cart(cart_id)
Access method: heap
```

All relations in the database are Boyce-Codd Normalized.

POPULATING THE TABLES

The Clothing table is populated with different clothing items. The remaining tables are populated in real time- as and when users interact with the web app.

online_store/postgres@PostgreSQL 13

Query Editor | Query History

```
INSERT INTO Clothing VALUES
(
    nextval('cloth_sequence'),
    'Jeans', 'Men',
    'Navy Blue; Skinny Fit; Stretchable',
    'Levis', 1339, 105
),
(
    nextval('cloth_sequence'),
    'T Shirt', 'Men',
    'Yellow; Half Sleeve; Polyester',
    'Reebok', 349, 242
),
(
    nextval('cloth_sequence'),
    'Formal Shirt', 'Men',
    'Black; Slim Fit; Cotton Blennd',
    'English Navy', 599, 135
),
```

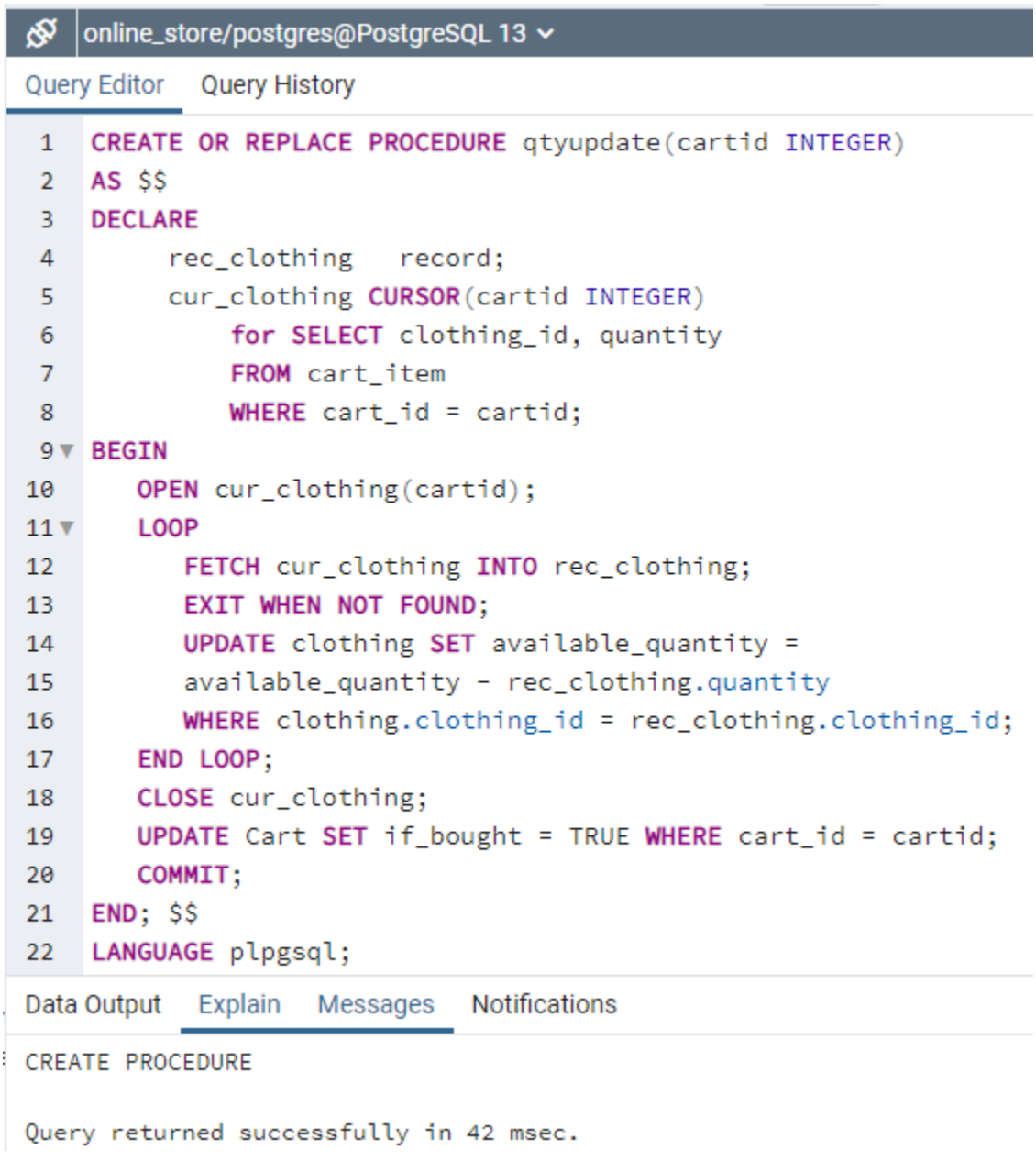
Messages | Data Output | Explain | Notifications

```
INSERT 0 12

Query returned successfully in 39 msec.
```

SETTING UP PROCEDURES

A procedure update_qty() which takes in cartid as argument, has been implemented. This procedure is executed after the bill is generated so as to reduce the amount of clothing items ordered from the clothing inventory.

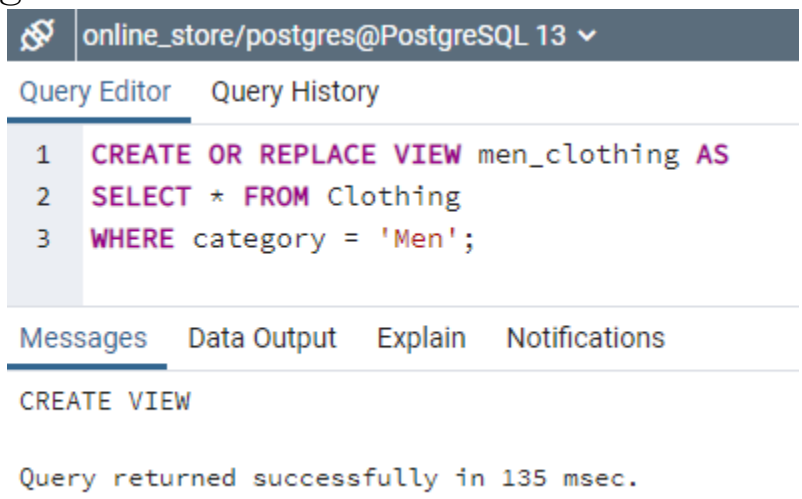
The procedure also closes the particular open cart of the user, after the bill is generated.



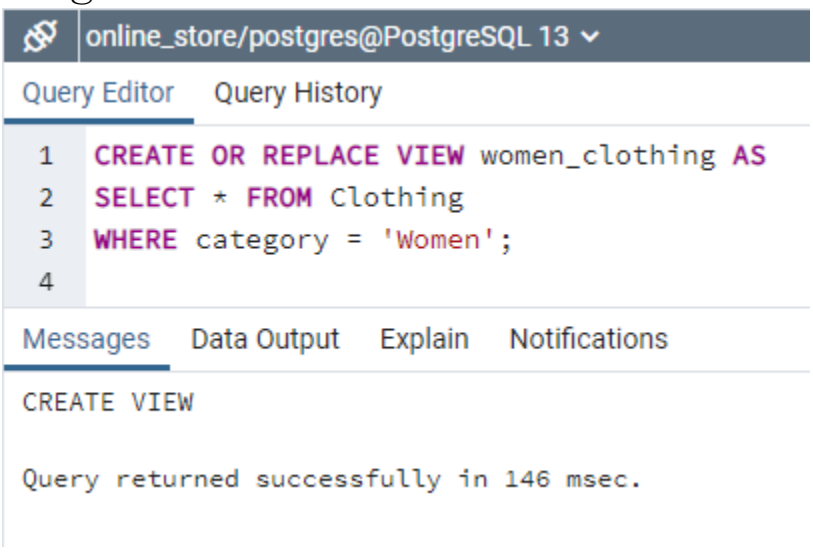
SETTING UP VIEWS

Views have been implemented in order to display the clothing collection as per the user choice - Men, Women or Kids

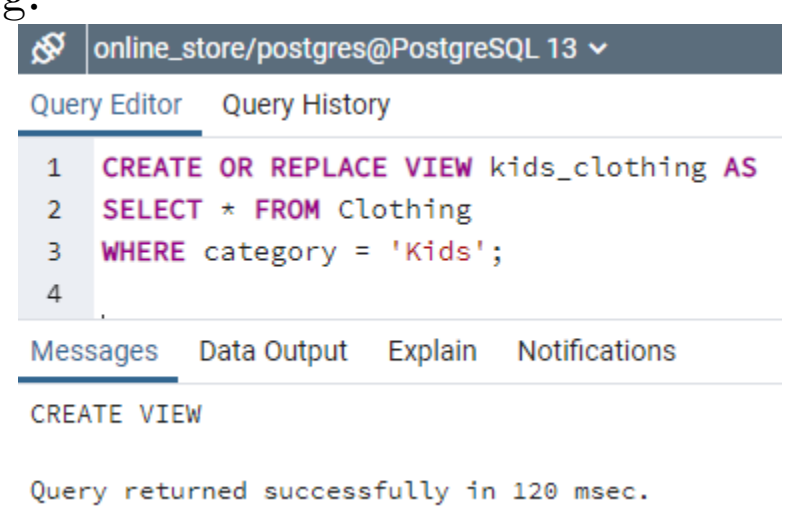
1. Men_clothing:



2. Women_clothing:



3. Kids_clothing:



SETTING UP TRIGGERS

A trigger has been set up so as to automatically restock the available quantity, once it goes below the minimum threshold.

online_store/postgres@PostgreSQL 13

Query Editor | Query History

```
1  CREATE OR REPLACE FUNCTION restock()
2  RETURNS TRIGGER
3  LANGUAGE PLPGSQL
4  AS
5  $$
6  BEGIN
7      UPDATE Clothing SET
8      available_quantity = available_quantity + 30
9      WHERE available_quantity < 20;
10     RETURN NEW;
11 END;
12 $$
```

Data Output | Explain | Messages | Notifications

```
CREATE FUNCTION

Query returned successfully in 206 msec.
```

online_store/postgres@PostgreSQL 13

Query Editor | Query History

```
1  CREATE TRIGGER clothing_restock
2  AFTER INSERT
3  ON Bill
4  FOR EACH STATEMENT
5  EXECUTE PROCEDURE restock();
```

Data Output | Explain | Messages | Notifications

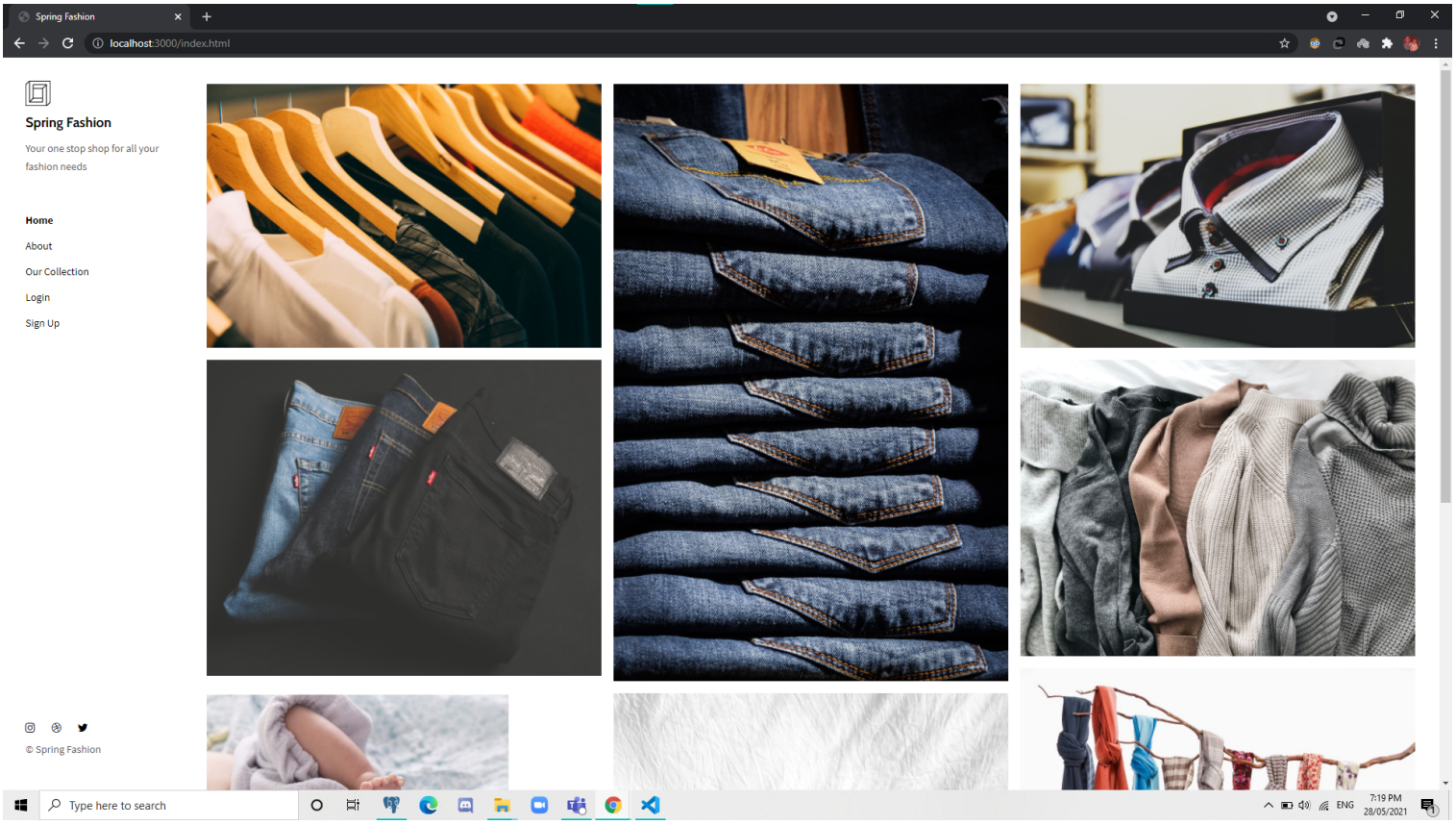
```
CREATE TRIGGER

Query returned successfully in 44 msec.
```

RESULTS

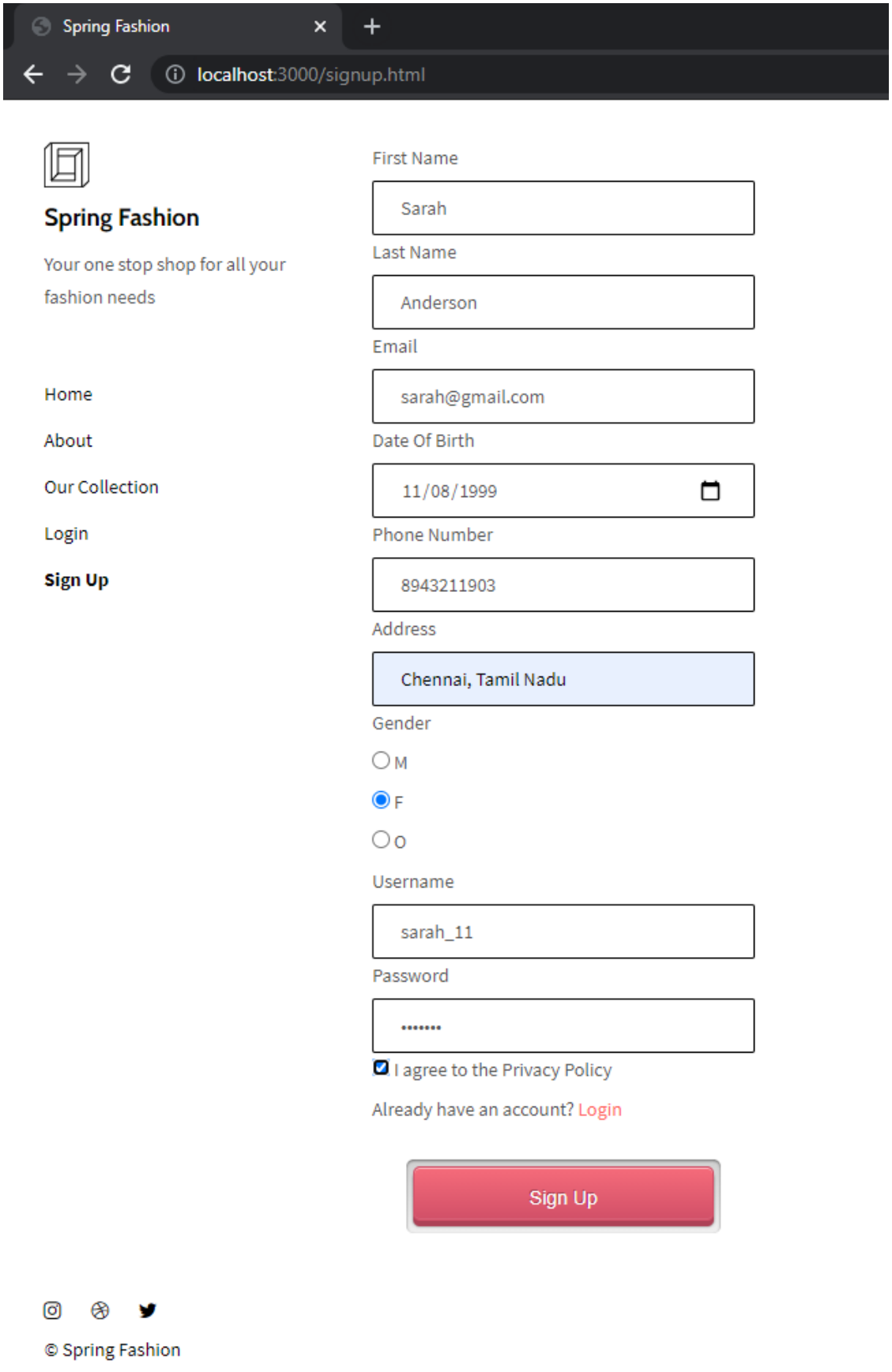
1. First, we start the server so as to run the application.

```
safiy@LAPTOP-AQ9CTGNG MINGW64 ~/Documents/Safiya/textbooks/YEAR 2/SEM4/DBMS/PROJECT/DBMS-Project/main-app (main)
$ node server.js
App is running on port 3000
```

2. Open http://localhost:3000 to access the application



3.Create a new user

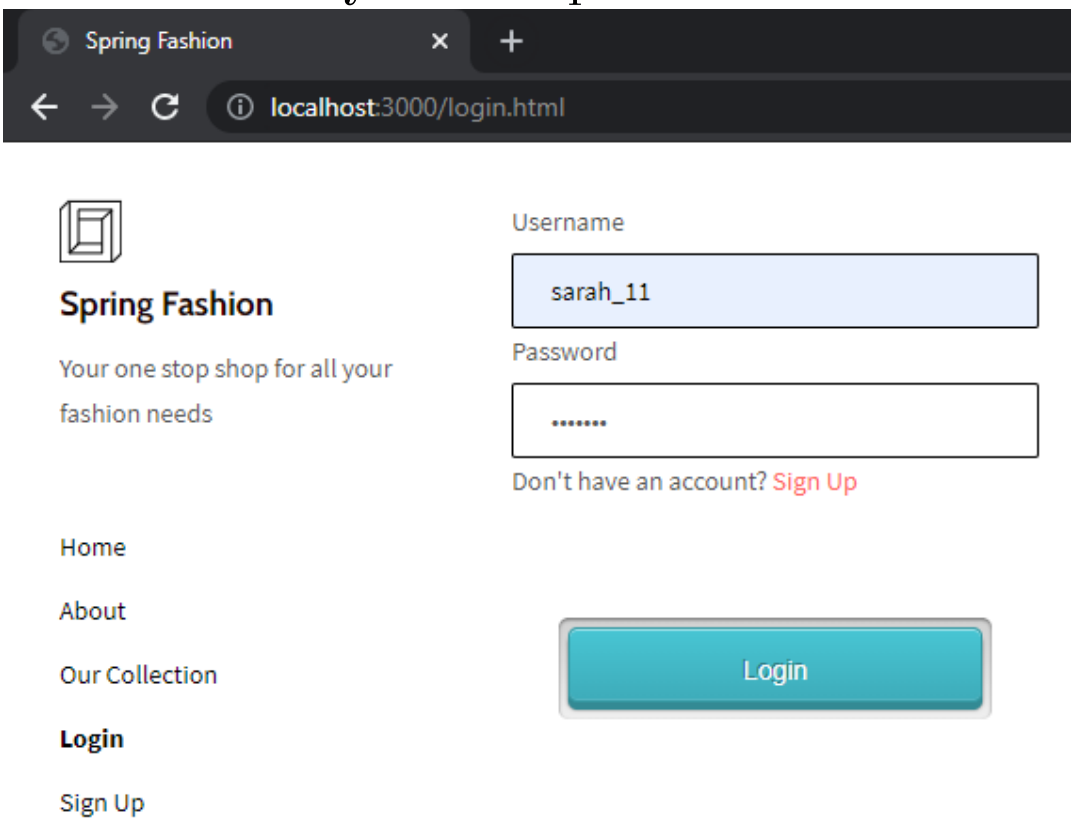


On the server side, the following query is written to add the user information to the Customer table, where the password is hashed, after checking whether the username or email already exists:

```
app.post('/signup', (req, res) => {
    const {
        firstname,
        lastname,
        dob,
        phno,
        address,
        gender,
        email,
        username,
        password
    } = req.body;

    client
        .query('SELECT * FROM Customer WHERE email = $1', [email],
        (err, result) => {
            if(result.rows.length !== 0)
            {
                res.status(400).sendFile(path.join(__dirname, 'public', 'emailexists.html'));
                return;
            }
           //  res.end();
        });

    client
        .query('SELECT * FROM Customer WHERE username = $1', [username],
        (err, result) => {
            if(result.rows.length !== 0)
            {
                res.status(400).sendFile(path.join(__dirname, 'public', 'usernameexists.html'));
                return;
            }
         //   res.end();
        });
```

```
bcrypt.hash(password, saltRounds, (err, hash) => {
    client
    .query('INSERT INTO Customer VALUES($1, $2, $3, $4, $5, $6, $7, $8, $9)',
    [username, hash, firstname, lastname, dob, gender, phno, email, address],
    (err, result) => {
        if(err) {
            console.log(err);
            res.sendStatus(500);
            return;
        }
        res.status(201).sendFile(path.join(__dirname, 'public', 'created.html'));

    });
});
```

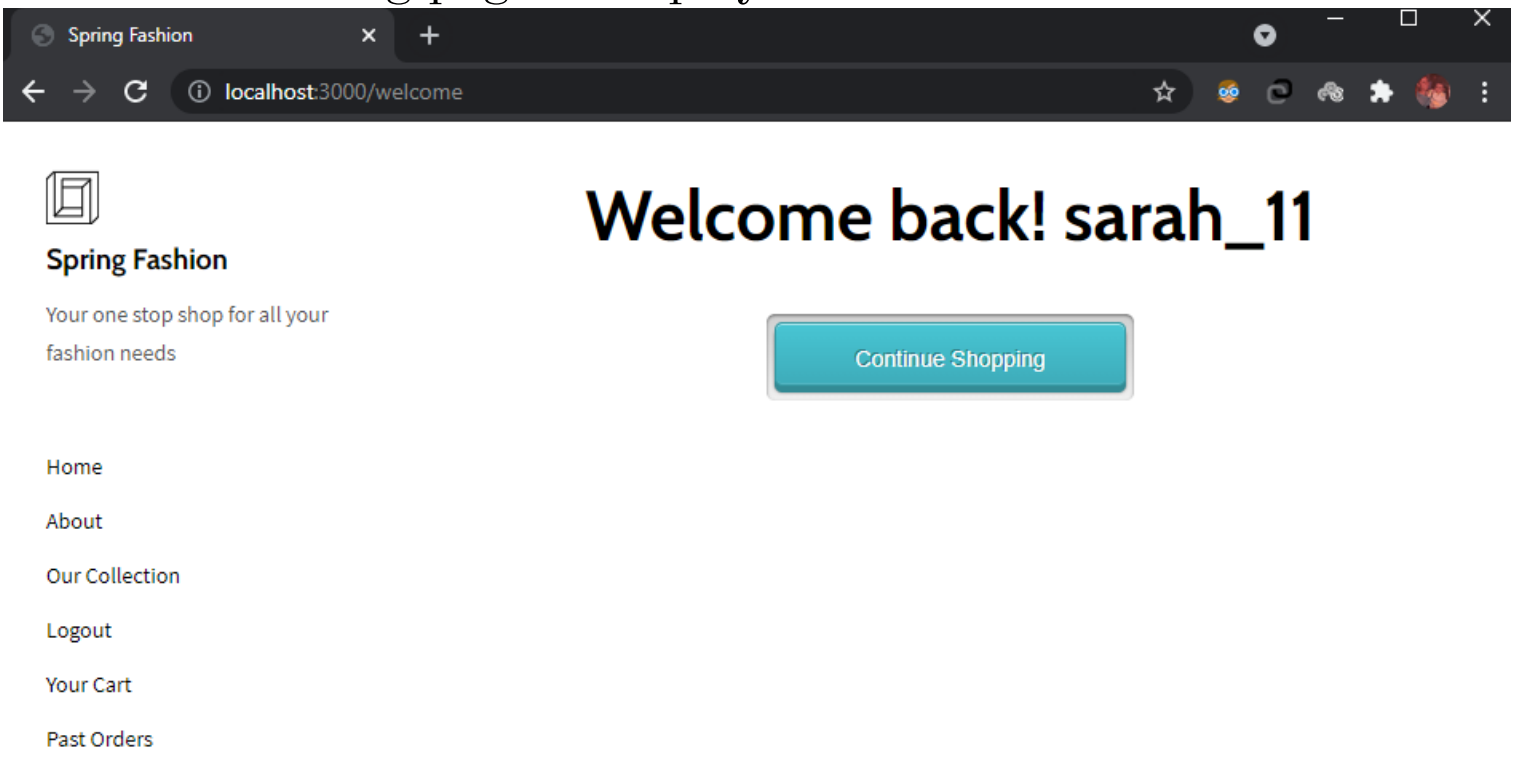
4. Login with the newly created profile



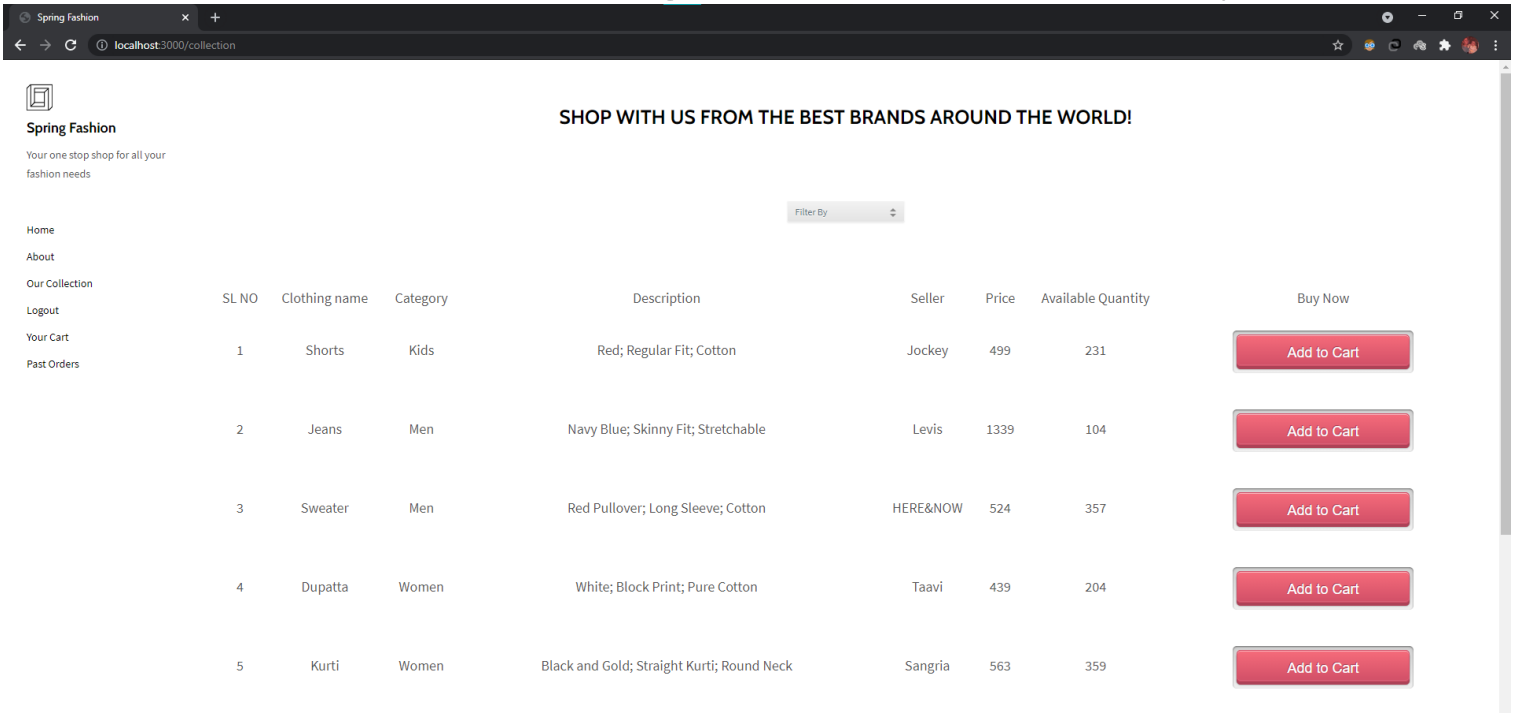
The entered username and password is checked with the existing users from the Customer table using the following query:

```
app.post('/login', (req, res) => {
    const { username, password } = req.body;
    if(username && password) {

        client
            .query('SELECT * FROM Customer WHERE username = $1', [username],
            (err, result) => {
                if(err) {
                    console.log(err);
                    res.sendStatus(500);
                    return;
                }
                if(result.rows.length !== 0) {
                    bcrypt.compare(password, result.rows[0]['pass'], (error, result1) => {
                        if(result1 === true)
                        {   req.session.loggedin = true;
                            req.session.username = username;
                            res.redirect('/welcome');
                        }

                        else
                            res.redirect('/loginerror');
                    });
                }  else res.sendFile(path.join(__dirname, 'public', 'invaliduser.html'));

            });
    } else {
        res.send('<script>alert("INVALID")</script>');
    }
});
```

5.The following page is displayed



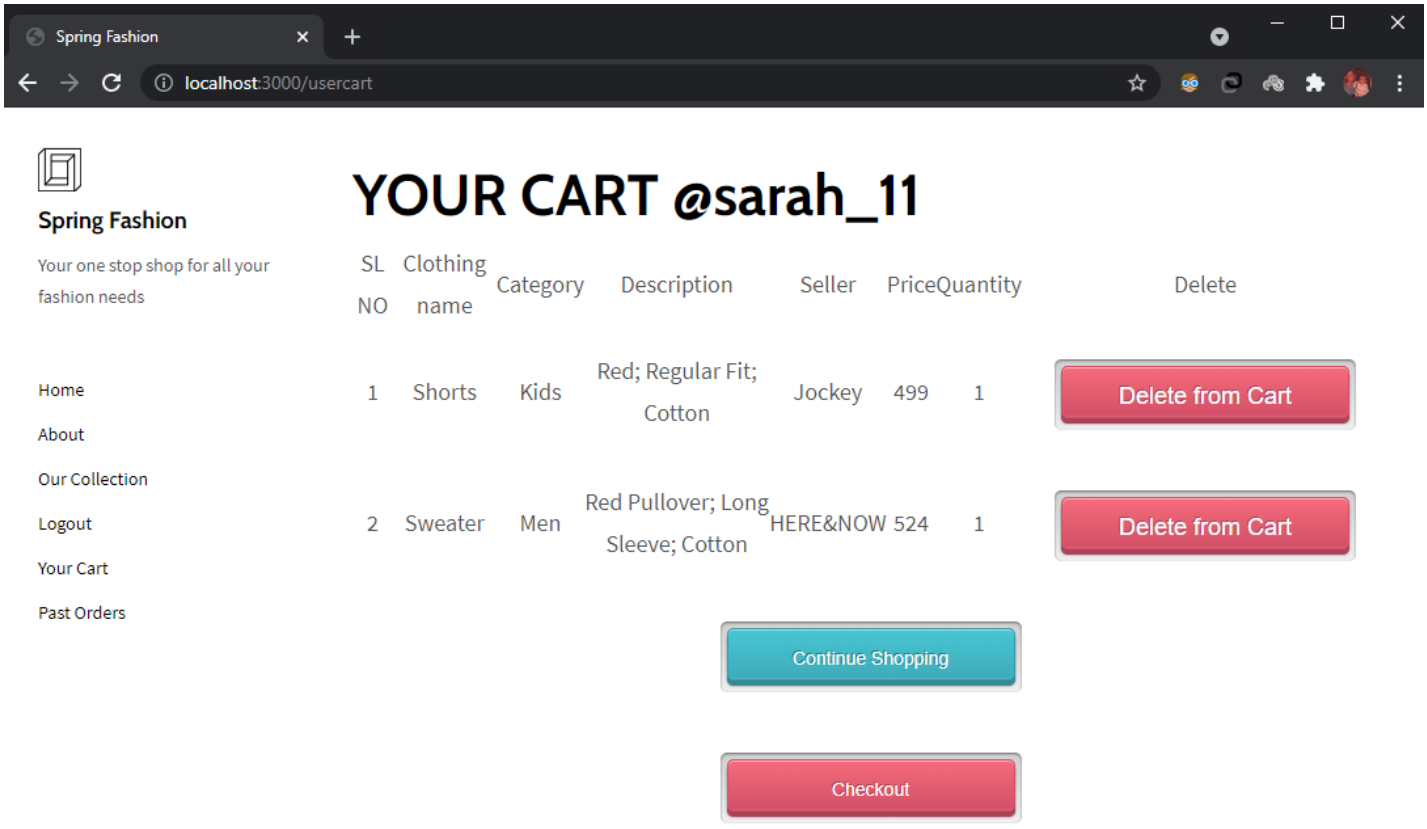
6.Select Continue Shopping, the collection is displayed.



The collection is displayed from the Clothing table by executing the following query:

```
//To display clothing table
app.get('/collection', (req, res) => {
    client
    .query('SELECT * FROM CLOTHING',
    (err, result) => {
        if(err) {
            console.log(err);
            res.sendStatus(500);
            return;
        }
        if(result.rows.length > 0) {
            if(req.session.loggedin === true)
                res.render('collection_user.ejs', {result: result});
            else
                res.render('collection_guest.ejs', {result: result});
        }


    })
})
```

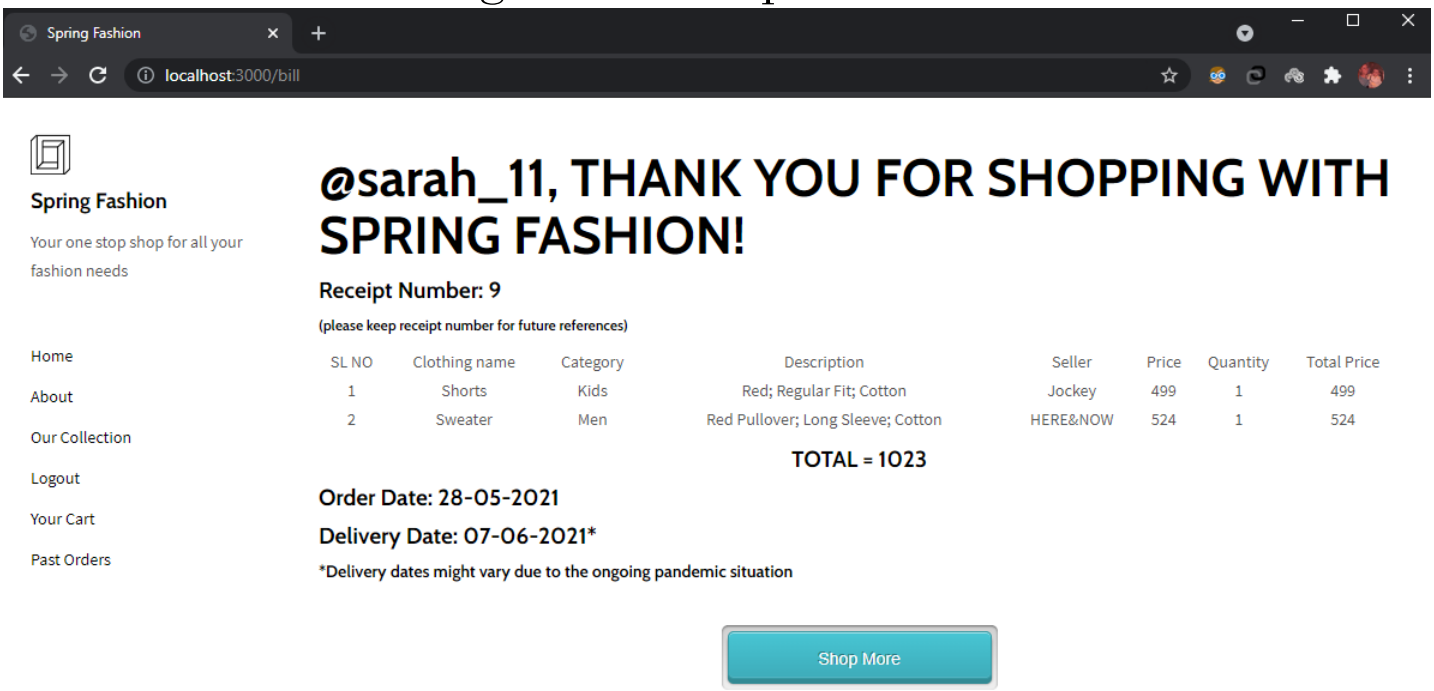
7. Add the desired items to the cart.



The user's cart is displayed from the Cart_item table by executing the following query:

```
    var qry =   "SELECT cart_item.clothing_id, clothing_name, category, description, seller, price, quantity, price*quantity AS total \
    FROM Clothing, Cart_item \
    WHERE Clothing.clothing_id = Cart_item.clothing_id \
    AND Cart_item.cart_id = (SELECT cart_id FROM Cart \
    WHERE if_bought = FALSE AND username = $1)";
    client
        .query(qry,
        [req.session.username],
        (err, result) => {
            if(err) {
                console.log(err);
                res.sendStatus(500);
                return;
            }
            if(result.rows.length === 0)
                res.render('emptycart.ejs', {res: req.session.username});
            else {
                let qry1 = "SELECT SUM(price*quantity) AS sum FROM Clothing, Cart_item WHERE Clothing.clothing_id = Cart_item.clothing_id " +
                           "AND Cart_item.cart_id = (SELECT cart_id FROM Cart WHERE if_bought = FALSE AND username = $1)"
                client
                    .query(qry1,
                    [req.session.username],
                    (err, total) => {
                        if(err) {
                            console.log(err);
                            res.sendStatus(500);
                            return;
                        }
                        res.render('usercart.ejs', {result: result, res: req.session.username, total: total.rows[0]['sum']});
                    })
            }
        })
    }
})
```

8. Click Checkout to get the receipt.



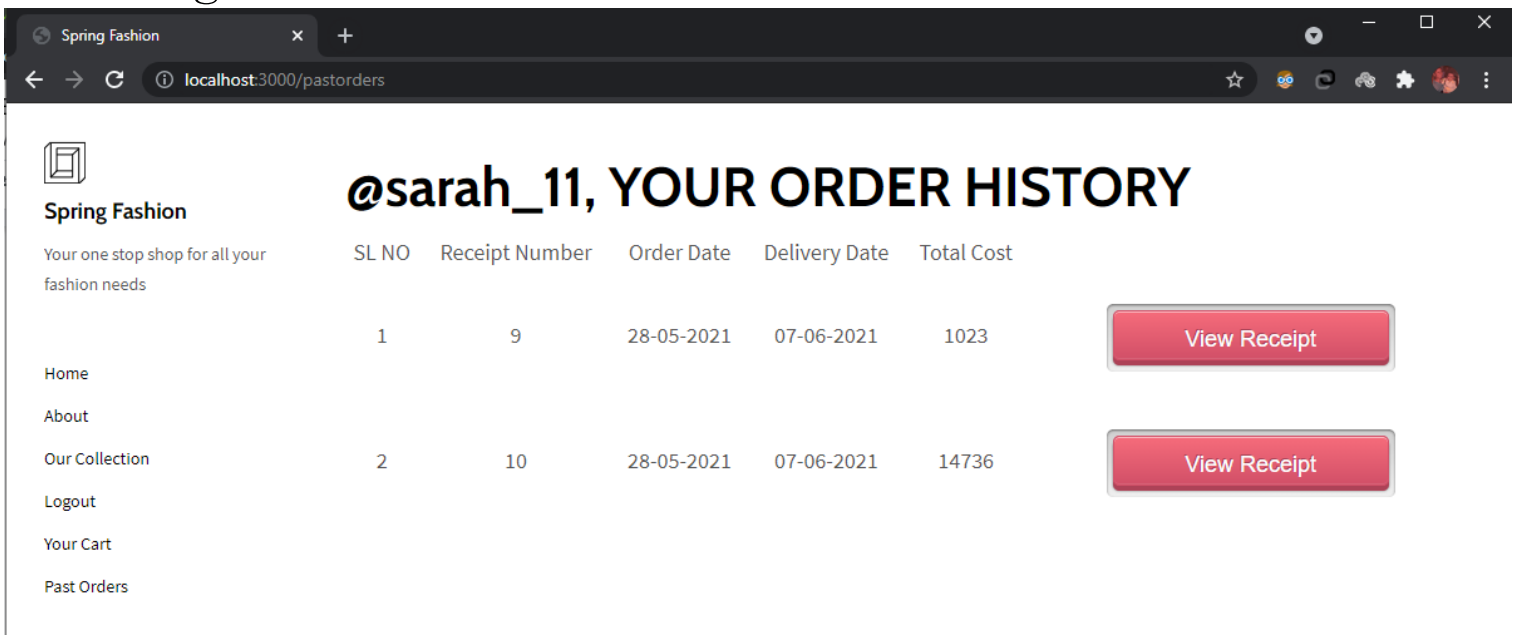
The bill is calculated by executing the following queries:

```
client
    .query('SELECT cart_id FROM Cart WHERE if_bought = FALSE and username = $1',
    [req.session.username],
    (err, result) => {
        if(err) {
            console.log(err);
            res.sendStatus(500);
            return;
        }
        cartid = result.rows[0]['cart_id'];
        client
            .query('SELECT SUM(price*quantity) AS total FROM clothing, cart_item WHERE clothing.clothing_id = cart_item.clothing_id and cart_item.cart_id = $1',
            [cartid],
            (err, result1) => {
                if(err) {
                    console.log(err);
                    res.sendStatus(500);
                    return;
                }
                client
                    .query('INSERT INTO Bill(cart_id, order_date, delivery_date, total_cost) VALUES($1, $2, $3, $4)',
                    [cartid, moment().format('DD-MM-YYYY'), moment().add(10, 'days').format('DD-MM-YYYY'), result1.rows[0]['total']],
                    (err, result) => {
                        if(err) {
                            console.log(err);
                            res.sendStatus(500);
                            return;
                        }
                        client
                            .query('CALL qtyupdate($1)',
                            [cartid],
                            (err, result) => {
                                if(err) {
                                    console.log(err);
                                    res.sendStatus(500);
                                    return;
                                }
                            })
                    })
                res.redirect('/bill');
            })
    })
```

```
app.get('/bill', (req, res) => {
    if(req.session.loggedin !== true)
        res.redirect('/logintocontinue');
    let qry =   "SELECT bill_id, TO_CHAR(order_date, 'DD-MM-YYYY') AS order_date, TO_CHAR(delivery_date, 'DD-MM-YYYY') AS delivery_date, " +
                'cart_item.clothing_id, clothing_name, category, description, ' +
                'seller, price, quantity, price*quantity AS total, total_cost ' +
                'FROM Bill, Clothing, Cart_item ' +
                'WHERE Clothing.clothing_id = Cart_item.clothing_id ' +
                'AND Cart_item.cart_id = $1  AND Bill.cart_id = $1';
    client
        .query(qry,
        [cartid],
        (err, results) => {
        if(err) {
            console.log(err);
            res.sendStatus(500);
            return;
        }
        res.render('bill.ejs', {result: results, res: req.session.username});
    })
})
```

Additional functionalities:

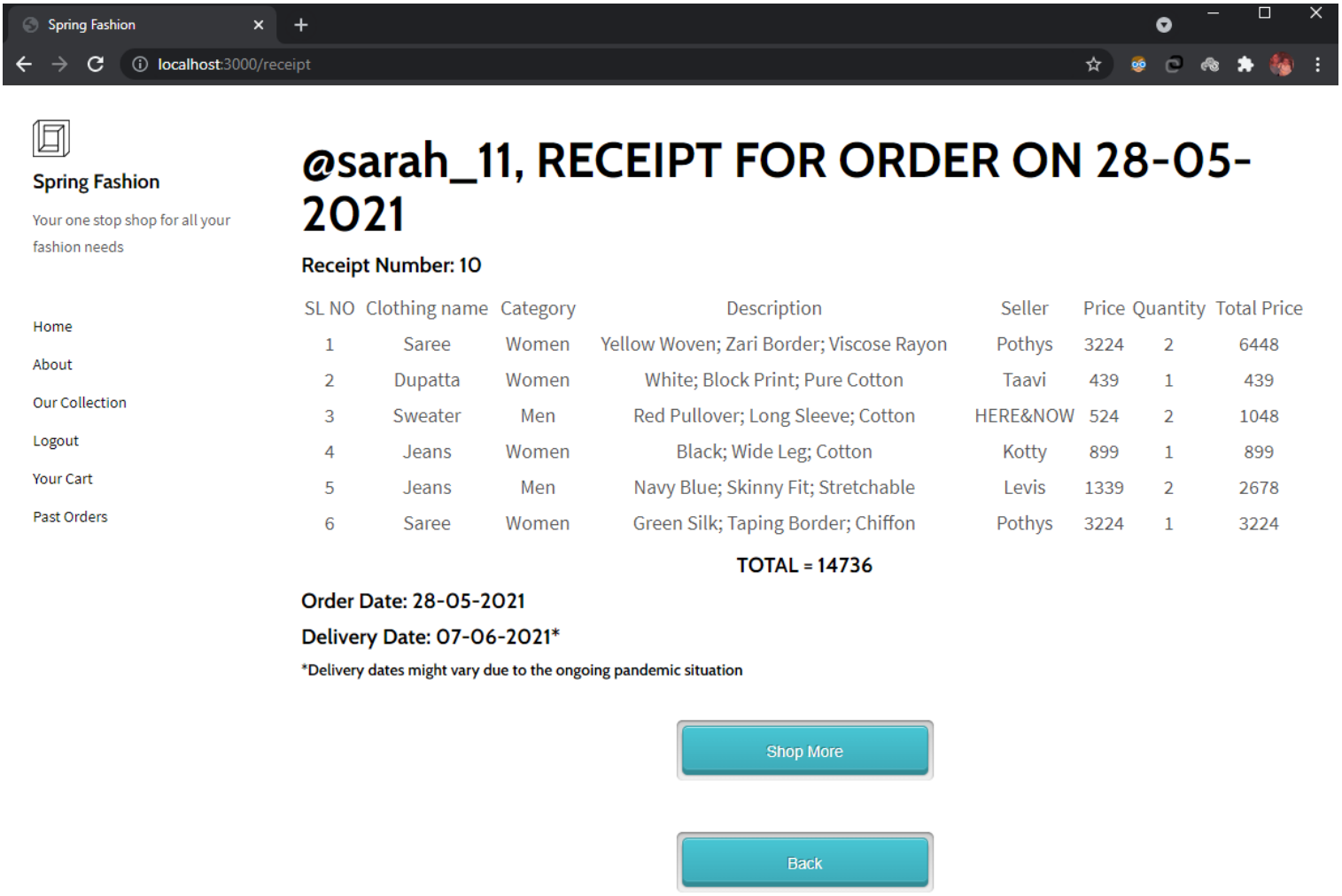
1. View past orders by selecting the past orders option in the navigation bar:



The past orders are displayed by executing the following query:

```
//To get past orders of user
app.get('/pastorders', (req, res) => {
    if(req.session.loggedin !== true)
        res.redirect('/logintocontinue');
    let qry =  "SELECT bill_id, TO_CHAR(order_date, 'DD-MM-YYYY') AS order_date, TO_CHAR(delivery_date, 'DD-MM-YYYY') AS delivery_date, " +
               "total_cost, cart_id FROM Bill WHERE cart_id IN (SELECT cart_id FROM Cart WHERE if_bought = TRUE AND username = $1)";
    client
        .query(qry,
        [req.session.username],
        (err, result) => {
            if(err) {
                console.log(err);
                res.sendStatus(500);
                return;
            }
            res.render('pastorders.ejs', {result: result, res: req.session.username});
        })
})
```

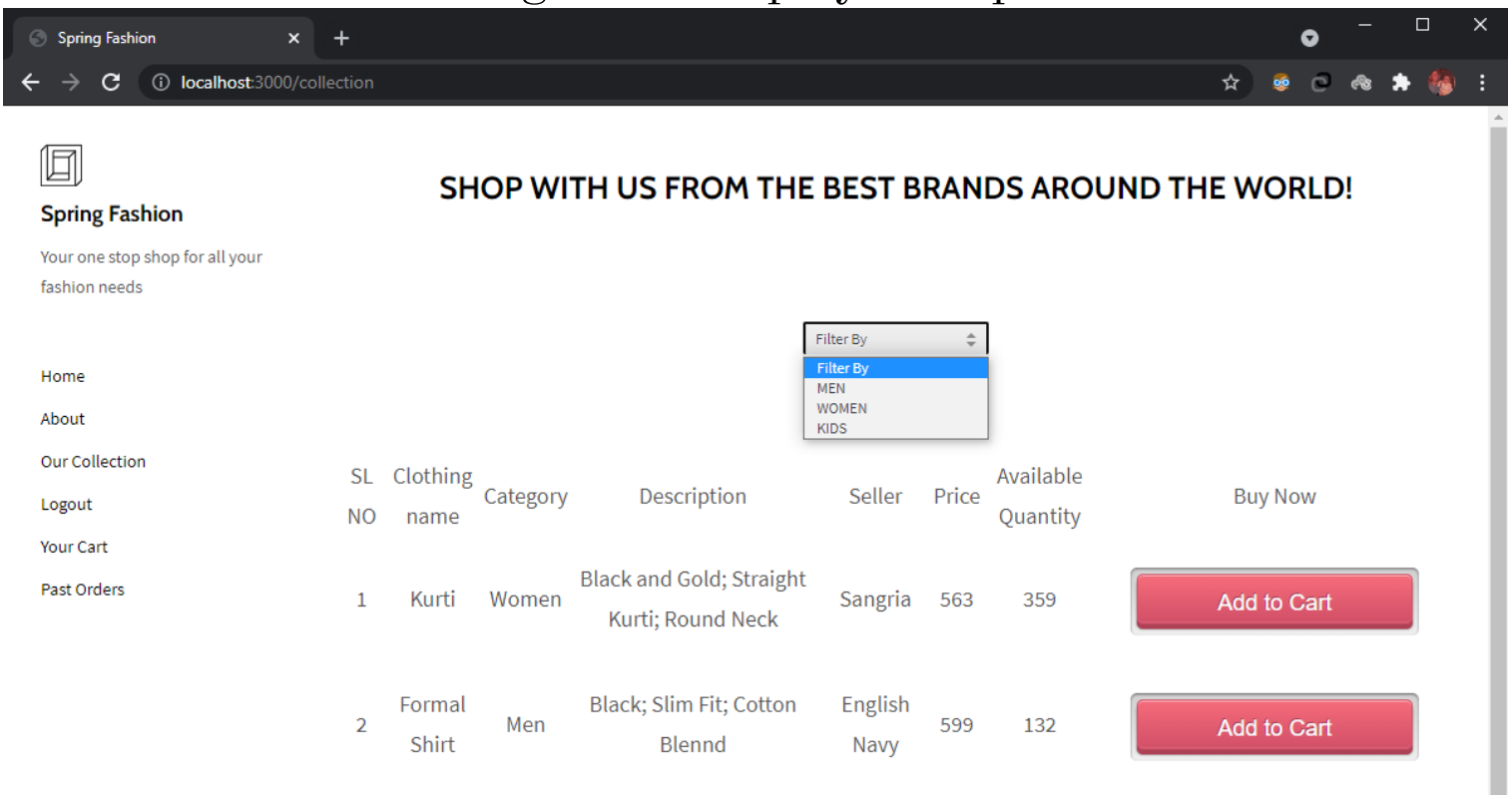
a) Select View Receipt to get the Receipt of the desired order

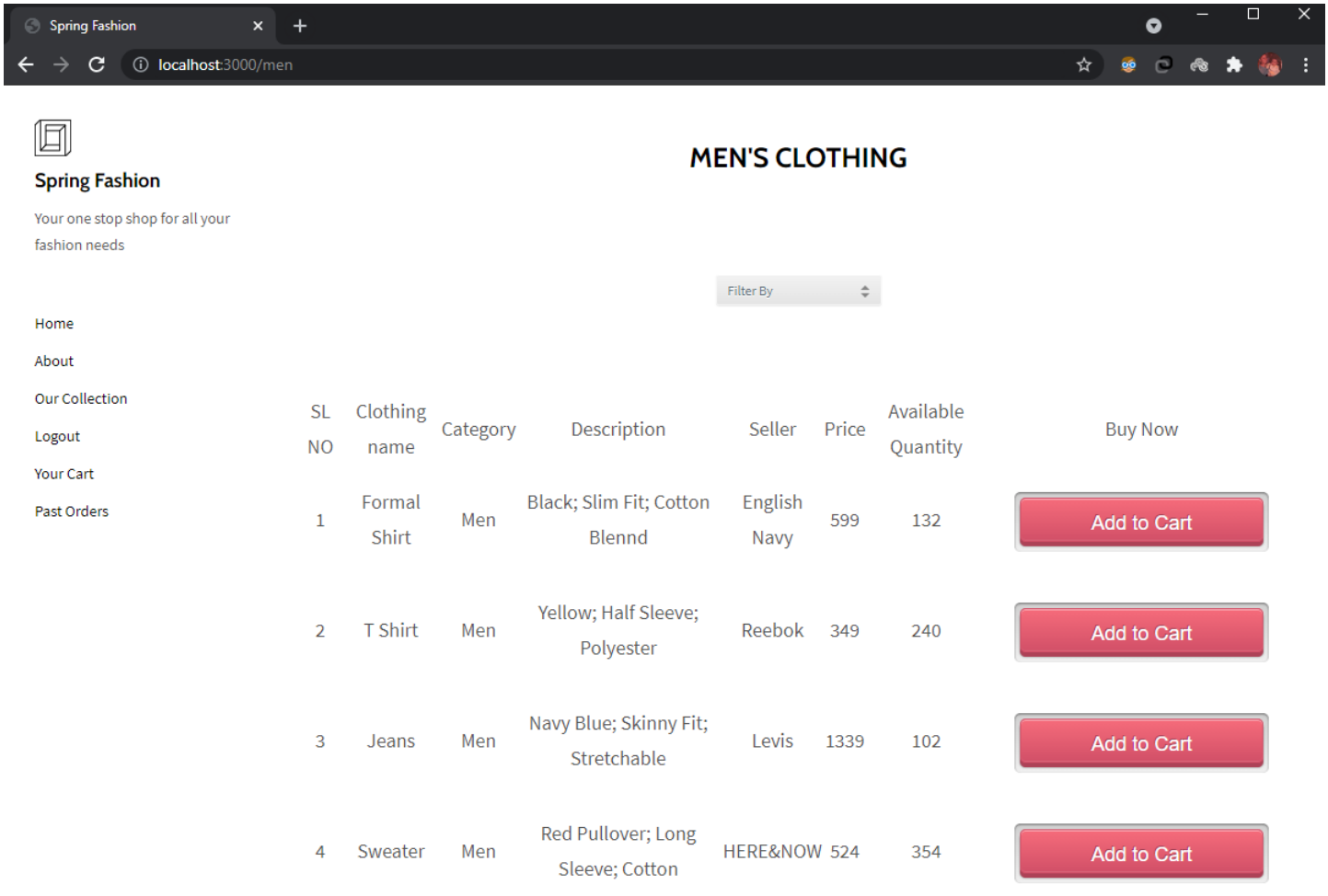


The particular receipt is displayed by executing the following query:

```
app.get('/receipt', (req, res) => {
    if(req.session.loggedin !== true)
        res.redirect('/logintocontinue');
    let qry =   "SELECT bill_id, TO_CHAR(order_date, 'DD-MM-YYYY') AS order_date, TO_CHAR(delivery_date, 'DD-MM-YYYY') AS delivery_date, " +
                'cart_item.clothing_id, clothing_name, category, description, ' +
                'seller, price, quantity, price*quantity AS total, total_cost ' +
                'FROM Bill, Clothing, Cart_item ' +
                'WHERE Clothing.clothing_id = Cart_item.clothing_id ' +
                'AND Cart_item.cart_id = $1  AND Bill.cart_id = $1';
    client
        .query(qry,
        [cartid],
        (err, results) => {
            if(err) {
                console.log(err);
                res.sendStatus(500);
                return;
            }
            res.render('receipt.ejs', {result: results, res: req.session.username});
        });
})
```

2. Filter the clothing items displayed as per user choice:





The filters are implemented by using the following query:

```
//to display men's clothing - view
app.get('/men', (req, res) => {
    client
        .query('SELECT * FROM men_clothing',
        (err, result) => {
            if(err) {
                console.log(err);
                res.sendStatus(500);
                return;
            }
            if(req.session.loggedin !== true)
                res.render('men_guest.ejs', {result: result});
            else
                res.render('men_user.ejs', {result: result});
        })
});
```

```
//to display women's clothing - view
app.get('/women', (req, res) => {
    client
        .query('SELECT * FROM women_clothing',
        (err, result) => {
            if(err) {
                console.log(err);
                res.sendStatus(500);
                return;
            }
            if(req.session.loggedin !== true)
                res.render('women_guest.ejs', {result: result});
            else
                res.render('women_user.ejs', {result: result});
        })
});
```

```
//to display kids' clothing - view
app.get('/kids', (req, res) => {
    client
        .query('SELECT * FROM kids_clothing',
        (err, result) => {
            if(err) {
                console.log(err);
                res.sendStatus(500);
                return;
            }
            if(req.session.loggedin !== true)
                res.render('kids_guest.ejs', {result: result});
            else
                res.render('kids_user.ejs', {result: result});
        })
});
```

CONCLUSION

The ‘Online Clothing Store Management System’ application has been created and implemented successfully.

Source code for the same can be found at:
https://github.com/sxfiy-a/DBMS-Project

REFERENCES

- Introduction · Bootstrap v5.0
- What is a good reason to use SQL views?
- PostgreSQL with Nodejs. Most of the applications need at some... | by Laurent Renard | DailyJS
- Postgres Stored Procedures with Input and Output Parameters SQL | ObjectRocket
- 10 Examples of PostgreSQL Stored Procedures | EDB
- PostgreSQL Create View with Example
- Date in mmm yyyy format in postgresql
- Next pg.Client
- Geshan's Blog Node.js Postgresql tutorial: Build a simple REST API with Express step-by-step
- bcrypt
- Documentation: 9.5: Trigger Functions
- PostgreSQL Trigger: Create, Drop Example