CCPS 610 - Database Systems II

Assignment 2

Karel Tutsu, 501145868
Andrew Parsons, 500992021
Roberto Mariani, 500747921

The Project

Our final product is a **web store-app**, hosted by DigitalOcean, that can be reached by any browser at https://brewbean.store

Project repository: https://github.com/KarelOmab/CCPS610-Ass02-Brewery
*will be made public on the day of presentation

We have implemented all of the required tasks, some with minor adjustments but the core functionality is there. In addition, created some additional features that were not explicitly required but were nice to have features:

- Session Control & Usergroups
    - Guest users (anonymous) can only interact with:
        - Home page
        - Products page (view products and add to shopping cart)
        - Login page
    - Shoppers can interact with:
        - Login page
        - Home page
        - Update profile and password
        - Products page -> add items to shopping card -> order checkout -> order placing
        - Baskets -> Basket Items and status (only of their own baskets data)
        - Reports -> ONLY Check Stock of Basket Items (only for their own baskets)
    - Employees and Admins have ‘god view’ where they can interact with:
        - Home page
        - Login page
        - Update password (not profile)
        - Products page -> edit products (name, description, active, sale etc)
        - Baskets -> Basket Items and status (of all users)
        - Update Order Status -> Add shipping notes etc
        - Reports -> Purchase Total and Check Stock of Basket Items for any order
        - Audits -> Account Logon and Employee Actions
- Register new accounts
- Customer testimonials
- Data hashing
    - The stored passwords are hashed in the bb_account table
    - The stored card number is hashed in the bb_basket table
- Auditing
    - bb_audit_logon table contains events whenever accounts are successfully logged in through the frontend
    - bb_audit_employee table contains event data where employees insert or update data such as products or basket status (shipping data)

System Configuration

The following list of software and dependencies are the necessary requirements in order to reproduce the results as shown in the following Task documentation.

Front-end:

- HTML
- CSS (Bootstrap 5)
- JavaScript

Back-end:

- MySQL Server 8.0.33
- Flask 2.3.2 (Python3 lightweight web framework)
    - We utilized Flask to extend the python language to allow us to build html templates

Project Configuration

Tools used:

- Navicat for MySQL (GUI)
    - For creating stored procedures and other data table structures
- VIM and VSCODE text editors
- Transmit5 (SFTP client)

Misc Notes:

- See the github repo for the latest MySQL build script for the project
    - Converted the original Oracle SQL dataset to MySQL
- Domain name bought from gandi.net for $1 (annual)
- DigitalOcean Ubuntu VM $3 (monthly)
- LetsEncrypt SSL certificate $0

Remote Server Configuration

1. Launch ubuntu server
2. Configure linux users and groups
    a. adduser yourusername
    b. usermod -aG sudo yourusername
    c. su yourusername
    d. sudo groupadd workgroup
    e. sudo usermod -aG workgroup yourusername
3. sudo apt update
4. Install and configure MySQL server
    5. sudo apt install mysql-server
    6. sudo mysql_secure_installation
    7. sudo nano /etc/mysql/mysql.conf.d/mysqld.cnf
       comment out **bind-address** and **mysqlx-bind-address** (for remote access)
    8. sudo systemctl restart mysql
    9. sudo ufw allow 3306 (open port)
    10. Login to mysql and create remote access accounts
        a. mysql -u root -p
        b. CREATE USER 'breweryapp'@'%' IDENTIFIED BY 'secret';
        c. CREATE DATABASE brewery;
        d. GRANT ALL PRIVILEGES ON brewery.* TO 'breweryapp'@'%';
        e. FLUSH PRIVILEGES;
        f. EXIT;

       At this point the database is remotely accessible

11. Install and configure Web Server
    a. sudo apt install nginx python3-pip python3-dev python3.10-venv build-essential libssl-dev libffi-dev python3-setuptools
    b. Create project directory and set group
        i. sudo mkdir /brewery
        ii. sudo chgrp -R workgroup /brewery
        iii. sudo chmod -R 2775 /brewery
    c. Create and activate python venv
        i. python3 -m venv .venv
        ii. . .venv/bin/activate
        iii. pip install flask wheel uwsgi python-dotenv mysql-connector-python flask_login
12. Done

Note, in our application code we have a custom written sql wrapper class that handles all sql transactions by calling stored procedures. This function is called within each task so we are highlighting it here for once and for all. **Nowhere in our application code are we making explicit SQL string queries (except test.py which is a unit test module because we need to retrieve cursor last_id to delete dummy data by id that stored procedures do not return).** Furthermore, every interaction with the database opens a new connection, performs required tasks and closes the connection. There are no hanging connections in our application design.

File : **sql.py**

```
def call_stored_procedure(proc_name, params=(), fetchone=False, update=False):
    connection = create_connection()
    cursor = connection.cursor(dictionary=True)

    try:
        cursor.callproc(proc_name, params)
        connection.commit()  # Commit the transaction
        if fetchone:
            for result in cursor.stored_results():
                return result.fetchone()
        elif update:
            return True
        else:
            results = []
            for result in cursor.stored_results():
                results.extend(result.fetchall())
            return results

    except mysql.connector.Error as err:
        print(f"Error: {err}")
        connection.rollback()  # Rollback the transaction
        return None

    finally:
        cursor.close()
        connection.close()
    return True

# MySQL connection parameters
config = {
    "host": os.getenv("MYSQL_HOST"),
    "user": os.getenv("MYSQL_USER"),
    "password": os.getenv("MYSQL_PASSWORD"),
    "database": os.getenv("MYSQL_DB")
}

def create_connection():
    return mysql.connector.connect(**config)
```

Task 1 : Product Description change ability.

CapressoBar Model #351

Product Id: 1
Active: True

A fully programmable pump espresso machine and 10-cup coffee maker complete with GoldTone filter

Form: N/A
Size: N/A
Price: 99.99
Sale Start: None
Sale End: None

Edit Product

Edit Product

Name:

CapressoBar Model #351

Description:

A fully programmable pump espresso machine and 10-cup coffee maker complete with GoldTone filter

Active:

Date Start:

yyyy-mm-dd

Date End:

yyyy-mm-dd

Sale Price:

Sale Price

Cancel Update

brewbean.store says

Product updated successfully.

OK

Edit Product

Name:

CapressoBar Model #351

Description:

A fully programmable pump espresso machine and 10-cup coffee maker complete with GoldTone filter upd

CapressoBar Model #351

Product Id: 1
Active: True

A fully programmable pump espresso machine and 10-cup coffee maker complete with GoldTone filter upd

Form: N/A
Size: N/A
Price: 99.99
Sale Start: None
Sale End: None

Edit Product

Front-end implementation:

Source file: /templates/products.html

```
<!-- Edit Product Modal -->
   <div class="modal fade" id="editProductModal" tabindex="-1" aria-labelledby="editProductModalLabel"
aria-hidden="true">
      <div class="modal-dialog modal-dialog-centered">
         <div class="modal-content">
            <div class="modal-header">
               <h5 class="modal-title" id="editProductModalLabel">Edit Product</h5>
               <button type="button" class="btn-close" data-bs-dismiss="modal" aria-label="Close"></button>
            </div>
            <div class="modal-body">
               <form>
                  <input type="hidden" id="idProduct" readonly>
                  <div class="mb-3">
                     <label for="ProductName" class="form-label">Name:</label>
                     <input type="text" id="ProductName" class="form-control" maxlength="25">
                  </div>
                  <div class="mb-3">
                     <label for="Description" class="form-label">Description:</label>
                     <textarea id="Description" rows="4" class="form-control" maxlength="100"></textarea>
                  </div>
                  <div class="mb-3 form-check">
                     <input type="checkbox" id="Active" class="form-check-input" value="0">
                     <label for="Active" class="form-check-label">Active:</label>
                  </div>
                  <div class="mb-3">
                     <label for="DateStart" class="form-label">Date Start:</label>
                     <input type="date" id="DateStart" class="form-control" placeholder="Start Date">
                  </div>
                  <div class="mb-3">
                     <label for="DateEnd" class="form-label">Date End:</label>
                     <input type="date" id="DateEnd" class="form-control" placeholder="End Date">
                  </div>
                  <div class="mb-3">
                     <label for="SalePrice" class="form-label">Sale Price:</label>
                     <input type="number" id="SalePrice" class="form-control" placeholder="Sale Price">
                  </div>
               </form>
            </div>
            <div class="modal-footer">
               <button type="button" class="btn btn-secondary" data-bs-dismiss="modal">Cancel</button>
               <button type="button" onclick="updateProduct()" class="btn btn-primary">Update</button>
            </div>
         </div>
      </div>
   </div>
```

```
function updateProduct() {
        var idProduct = document.getElementById("idProduct").value;
        var name = document.getElementById("ProductName").value;
        var description = document.getElementById("Description").value;
        var active = document.getElementById("Active").checked;
        var saleStart = document.getElementById("DateStart").value;
        var saleEnd = document.getElementById("DateEnd").value;
        var salePrice = document.getElementById("SalePrice").value;

        var xhr = new XMLHttpRequest();
        xhr.open("POST", "/update_product", true);
        xhr.setRequestHeader("Content-Type", "application/json");
        xhr.send(JSON.stringify({
            idProduct: idProduct,
            name: name,
            description: description,
            active: active,
            salestart: saleStart,
            saleend: saleEnd,
            saleprice: salePrice
        }));

        xhr.onload = function () {
            if (xhr.status == 200) {
                alert("Product updated successfully.");
                location.reload(); // This will reload the page
            } else {
                alert("Failed to update the product.");
            }
        }
    }
```

Source file: app.py:

```
@app.route('/update_product', methods=['POST'])
@login_required
def update_product():
    data = request.json
    product_id = data['idProduct']
    name = data['name']
    description = data['description']
    active = data['active']
    sale_start = data['salestart']
    sale_end = data['saleend']
    sale_price = data['saleprice']

    if not sale_start:
        sale_start = None
    if not sale_end:
        sale_end = None
    if not sale_price:
        sale_price = None


    res = sql.update_product(product_id, name, description, active, sale_start, sale_end, sale_price)

    if res:
        action_string = "product_name={}, description={}, active={}, sale_start={}, sale_end={},
sale_price={}".format(name, description, active, sale_start, sale_end, sale_price)
        sql.insert_audit_employee(current_user.id, "UPDATE", None, action_string)
        return jsonify({'success': True})
    else:
        return jsonify({'failed to update': True})
```

Source file: sql.py:

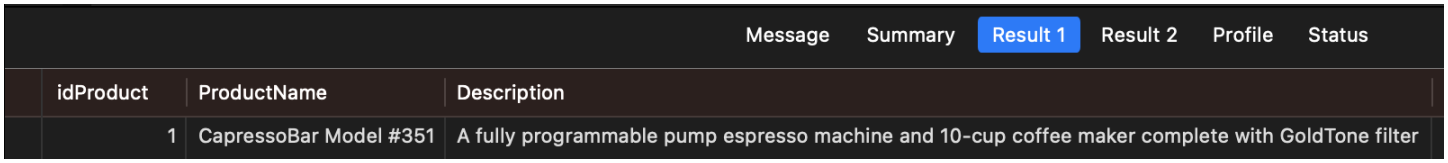
```
def update_product(product_id, name, description, active, sale_start, sale_end, sale_price):
    return call_stored_procedure('update_product', (product_id, name, description, active, sale_start, sale_end,
sale_price), fetchone=False, update=True)
```

Backend implementation:

```
CREATE DEFINER=`breweryapp`@`%` PROCEDURE `update_product`(
  IN inProductId int,
  IN inProductName VARCHAR(25),
  IN inDescription VARCHAR(100),
  IN inActive tinyint,
  IN inSaleStart VARCHAR(10),
  IN inSaleEnd VARCHAR(10),
  IN inSalePrice DECIMAL(6,2)
)
BEGIN
  DECLARE convertedSaleStart DATE;
  DECLARE convertedSaleEnd DATE;

  IF inSaleStart IS NULL OR inSaleStart = '' THEN
    SET convertedSaleStart = NULL;
  ELSE
    SET convertedSaleStart = STR_TO_DATE(inSaleStart, '%Y-%m-%d');
  END IF;

  IF inSaleEnd IS NULL OR inSaleEnd = '' THEN
    SET convertedSaleEnd = NULL;
  ELSE
    SET convertedSaleEnd = STR_TO_DATE(inSaleEnd, '%Y-%m-%d');
  END IF;

  IF inSalePrice IS NULL THEN
    SET inSalePrice = NULL;
  END IF;

  UPDATE bb_product SET
    ProductName=inProductName,
    Description=inDescription,
    Active=inActive,
    SaleStart=convertedSaleStart,
    SaleEnd=convertedSaleEnd,
    SalePrice=inSalePrice
  WHERE idProduct=inProductId;

END
```

Backend testing:

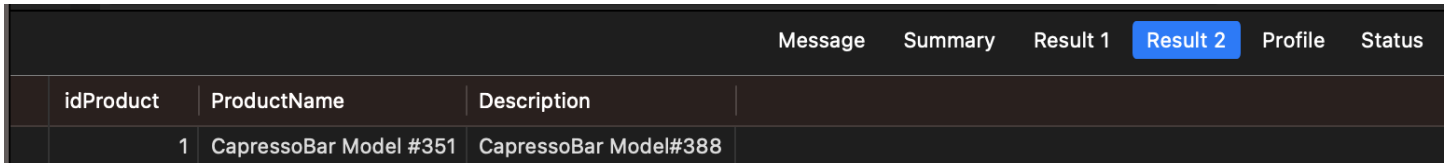
```
SELECT idProduct, ProductName, Description from bb_product WHERE idProduct=1;
```



Message | Summary | Result 1 | Result 2 | Profile | Status

idProduct	ProductName	Description
1	CapressoBar Model #351	A fully programmable pump espresso machine and 10-cup coffee maker complete with GoldTone filter

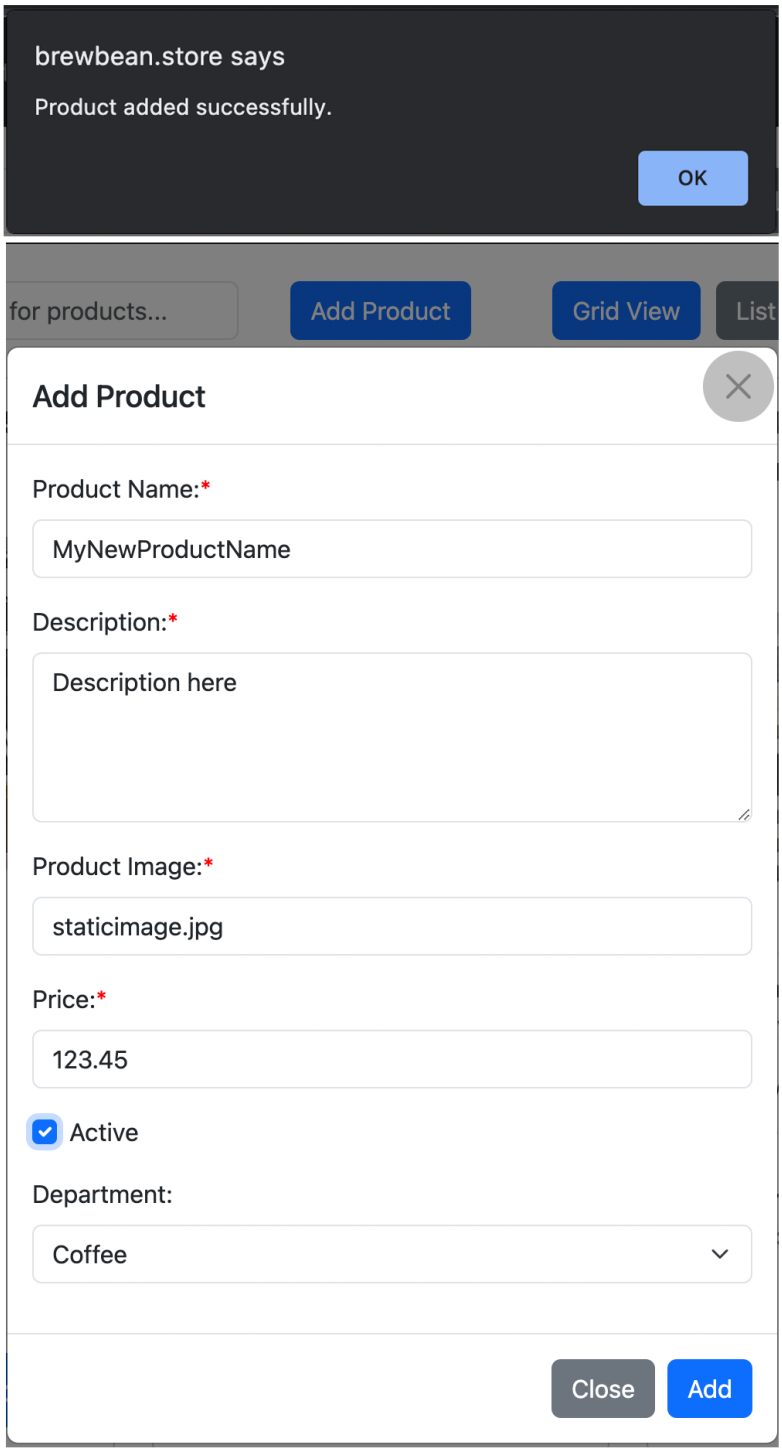
```
CALL update_product(1, "CapressoBar Model #351", "CapressoBar Model#388", 1, NULL, NULL, NULL);
SELECT idProduct, ProductName, Description from bb_product WHERE idProduct=1;
```

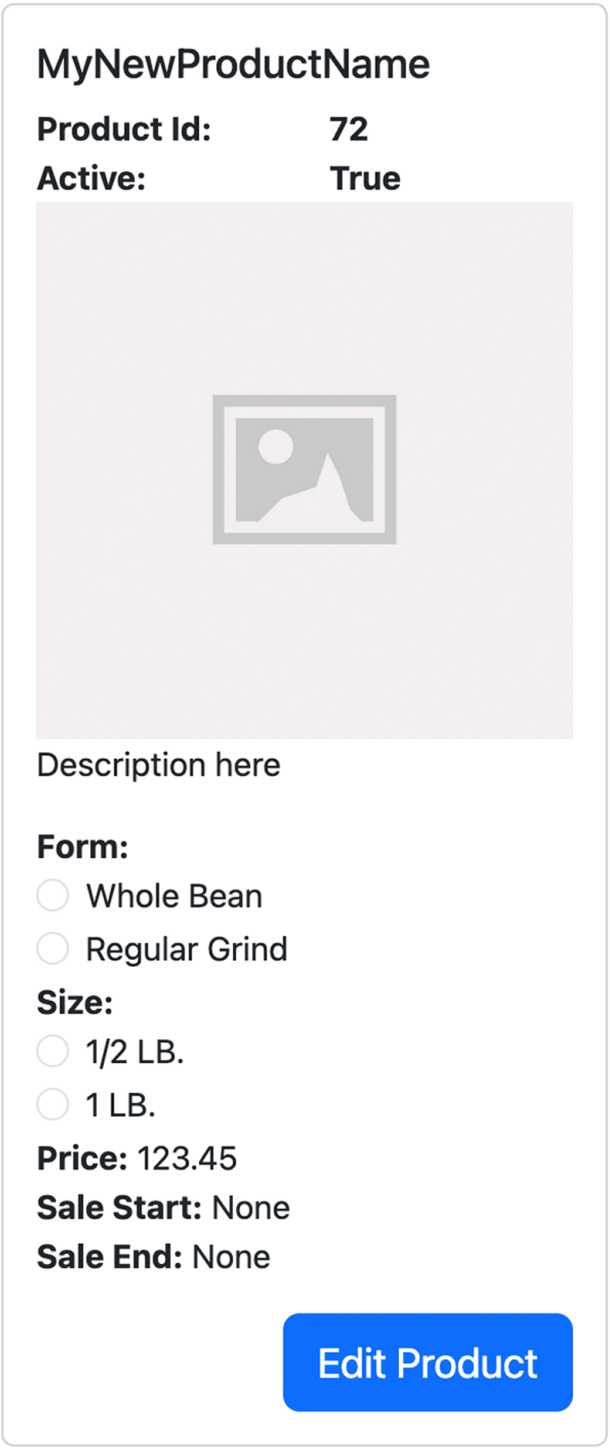


Message | Summary | Result 1 | Result 2 | Profile | Status

idProduct	ProductName	Description
1	CapressoBar Model #351	CapressoBar Model#388

Task 2 : Enter Product





Front-end implementation:

```
<!-- Add Product Modal -->
  <div class="modal fade" id="addProductModal" tabindex="-1" aria-labelledby="addProductModalLabel"
aria-hidden="true">
    <div class="modal-dialog modal-dialog-centered">
      <div class="modal-content">
        <div class="modal-header">
          <h5 class="modal-title" id="addProductModalLabel">Add Product</h5>
          <button type="button" class="btn-close" data-bs-dismiss="modal" aria-label="Close"></button>
        </div>
        <div class="modal-body">
          <form>
            <div class="mb-3">
              <label for="newProductName" class="form-label">Product Name:<span style="color:
red;">*</span></label>
              <input type="text" id="newProductName" class="form-control" required>
            </div>
            <div class="mb-3">
              <label for="newDescription" class="form-label">Description:<span style="color:
red;">*</span></label>
              <textarea id="newDescription" class="form-control" rows="4" required></textarea>
            </div>
            <div class="mb-3">
              <label for="newProductImage" class="form-label">Product Image:<span style="color:
red;">*</span></label>
              <input type="text" id="newProductImage" class="form-control" required>
            </div>
            <div class="mb-3">
              <label for="newPrice" class="form-label">Price:<span style="color: red;">*</span></label>
              <input type="number" id="newPrice" class="form-control" step="0.01" required>
            </div>
            <div class="mb-3 form-check">
              <input type="checkbox" id="newActive" class="form-check-input" value="1">
              <label class="form-check-label" for="newActive">Active</label>
            </div>
            <div class="mb-3">
              <label for="department" class="form-label">Department:</label>
              <select id="department" class="form-select" required></select>
            </div>
          </form>
        </div>
        <div class="modal-footer">
          <button type="button" class="btn btn-secondary" data-bs-dismiss="modal">Close</button>
          <button type="button" class="btn btn-primary" onclick="addProduct()">Add</button>
        </div>
      </div>
    </div>
  </div>
```

```
function addProduct() {
        if (validateForm()){
            var productName = document.getElementById("newProductName").value;
            var description = document.getElementById("newDescription").value;
            var productImage = document.getElementById("newProductImage").value;
            var price = document.getElementById("newPrice").value;
            var isActive = document.getElementById("newActive").checked ? 1 : 0;
            var departmentId = document.getElementById("department").value;
            var xhr = new XMLHttpRequest();
            xhr.open("POST", "/add_product", true);
            xhr.setRequestHeader("Content-Type", "application/json");
            xhr.send(JSON.stringify({
                product_name: productName,
                description: description,
                product_image: productImage,
                price: price,
                is_active: isActive,
        id_department: departmentId
            }));

            xhr.onload = function () {
                if (xhr.status == 200) {
                    alert("Product added successfully.");
                    location.reload();
                } else {
                    alert("Failed to add the product.");
                }
            }
        }
    }
```

Source file: app.py:

```
@app.route('/add_product', methods=['POST'])
@login_required
def add_product():
    product_details = request.json
    product_name = product_details.get('product_name')
    description = product_details.get('description')
    product_image = product_details.get('product_image')
    price = product_details.get('price')
    active = product_details.get('is_active')
    id_department = product_details.get('id_department')

    result = sql.insert_product(product_name, description, product_image, price, active, id_department)

    if result:
        action_string = "product_name={}, description={}, product_image={}, price={}, active={},
id_department={}".format(product_name, description, product_image, price, is_active, id_department)
        sql.insert_audit_employee(current_user.id, "INSERT", None, action_string)

    return jsonify({'result': result})
```

Source file: sql.py:

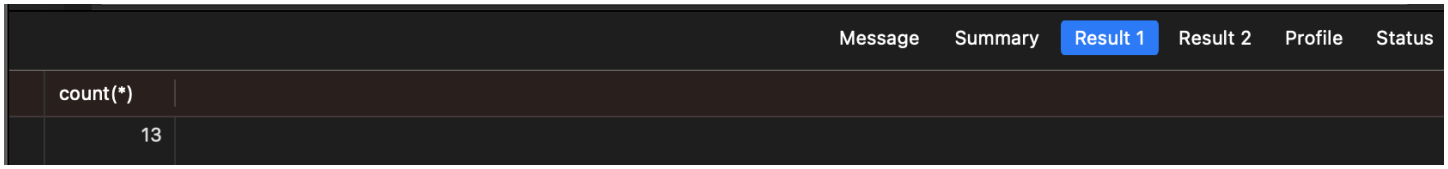
```
def insert_product(product_name, description, product_image, price, is_active, id_department):
    return call_stored_procedure('insert_product', (product_name, description, product_image, price, is_active,
id_department), fetchone=False, update=False)
```

Backend implementation:

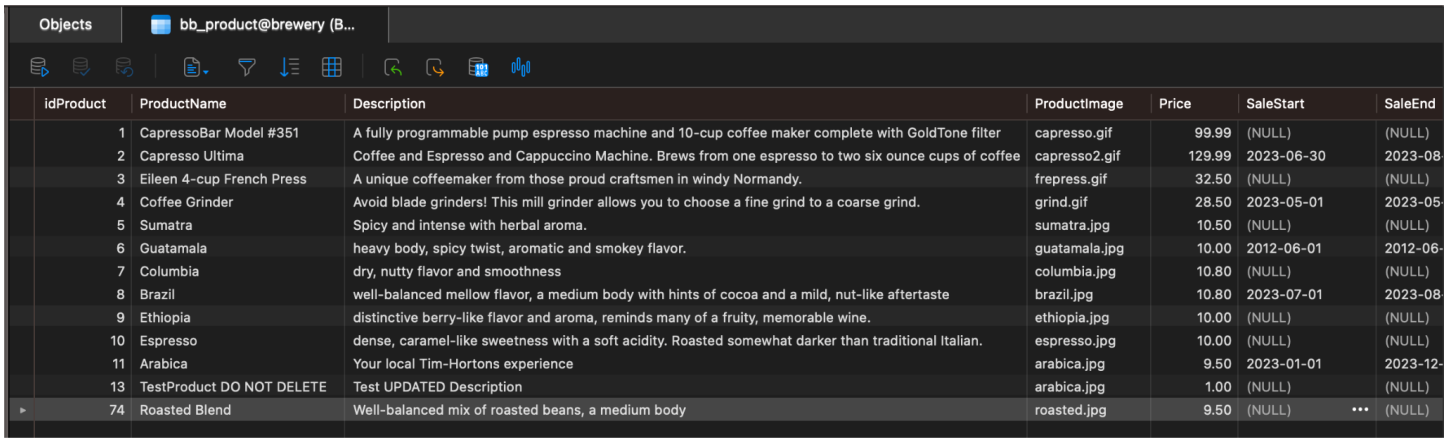
```
CREATE DEFINER=`breweryapp`@`%` PROCEDURE `insert_product`(
IN inProductName varchar(25),
IN inDescription varchar(100),
IN inProductImage varchar(25),
IN inPrice decimal(6, 2),
IN inActive tinyint(1),
IN inIdDepartment int(11))
BEGIN
  INSERT INTO bb_product (ProductName, Description, ProductImage, Price, Active, idDepartment)
         VALUES (inProductName, inDescription, inProductImage, inPrice, inActive, inIdDepartment);
END
```

Backend testing:

```
SELECT COUNT(*) FROM bb_product;
```



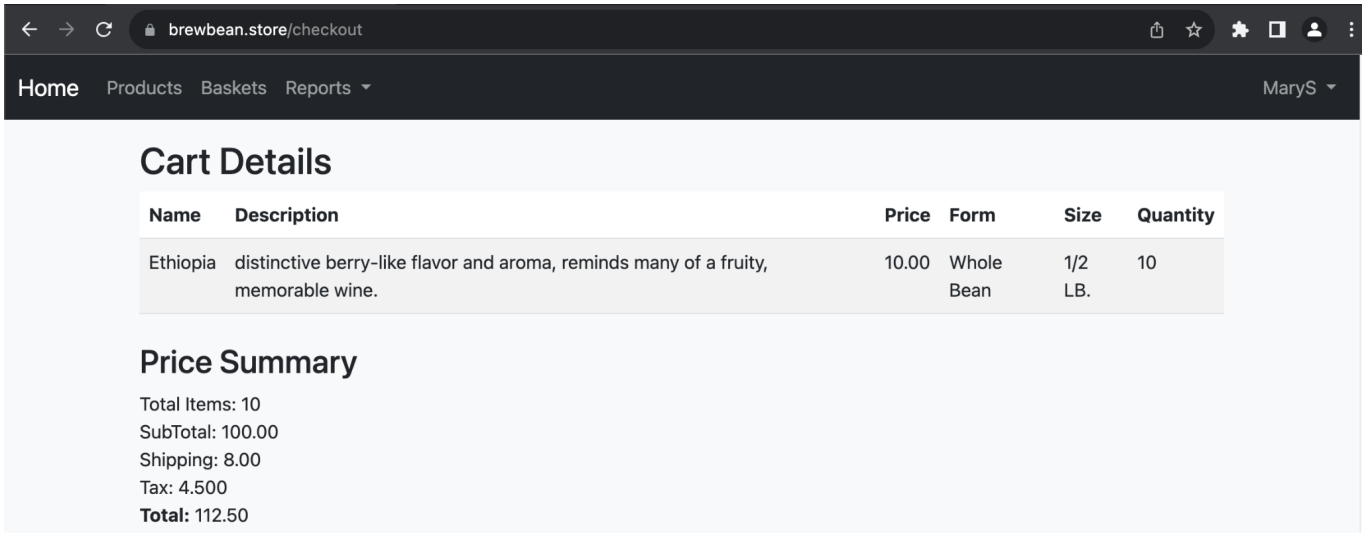

```
CALL insert_product('Roasted Blend', 'Well-balanced mix of roasted beans, a medium body', 'roasted.jpg',
9.50, 1, 1);
SELECT COUNT(*) FROM bb_product;
```



idProduct	ProductName	Description	ProductImage	Price	SaleStart	SaleEnd
1	CapressoBar Model #351	A fully programmable pump espresso machine and 10-cup coffee maker complete with GoldTone filter	capresso.gif	99.99	(NULL)	(NULL)
2	Capresso Ultima	Coffee and Espresso and Cappuccino Machine. Brews from one espresso to two six ounce cups of coffee	capresso2.gif	129.99	2023-06-30	2023-08
3	Eileen 4-cup French Press	A unique coffeemaker from those proud craftsmen in windy Normandy.	frepress.gif	32.50	(NULL)	(NULL)
4	Coffee Grinder	Avoid blade grinders! This mill grinder allows you to choose a fine grind to a coarse grind.	grind.gif	28.50	2023-05-01	2023-05
5	Sumatra	Spicy and intense with herbal aroma.	sumatra.jpg	10.50	(NULL)	(NULL)
6	Guatamala	heavy body, spicy twist, aromatic and smokey flavor.	guatamala.jpg	10.00	2012-06-01	2012-06
7	Columbia	dry, nutty flavor and smoothness	columbia.jpg	10.80	(NULL)	(NULL)
8	Brazil	well-balanced mellow flavor, a medium body with hints of cocoa and a mild, nut-like aftertaste	brazil.jpg	10.80	2023-07-01	2023-08
9	Ethiopia	distinctive berry-like flavor and aroma, reminds many of a fruity, memorable wine.	ethiopia.jpg	10.00	(NULL)	(NULL)
10	Espresso	dense, caramel-like sweetness with a soft acidity. Roasted somewhat darker than traditional Italian.	espresso.jpg	10.00	(NULL)	(NULL)
11	Arabica	Your local Tim-Hortons experience	arabica.jpg	9.50	2023-01-01	2023-12
13	TestProduct DO NOT DELETE	Test UPDATED Description	arabica.jpg	1.00	(NULL)	(NULL)
74	Roasted Blend	Well-balanced mix of roasted beans, a medium body	roasted.jpg	9.50	(NULL)	(NULL)

Task 3 : Calculating the Tax on an Order.

Note: In our application there is no separate tax calculator tool, instead we can simulate the exact same behavior by much more realistic means:



- Here we have an application user, MaryS, about to place an order for 10x products at $10 each which adds up to a subtotal of $100.
- The application shows a tax line item for the order on a separate line which is $4.50 (the expected result in this case).
- The html table in the checkout page renders a result from a custom table, bb_checkout, that contains a single row of data for each applicable shopper:

DESCRIBE bb_checkout;

Message | Summary | Result 1 | Pro

Field	Type	Null	Key	Default	Extra
idCheckout	int	NO	PRI	(NULL)	auto_increment
idShopper	int	NO	UNI	(NULL)	
Quantity	int	NO		(NULL)	
CartDetails	longtext	NO		(NULL)	
SubTotal	decimal(10,2)	YES		(NULL)	
Shipping	decimal(10,2)	YES		(NULL)	
Tax	decimal(10,3)	YES		(NULL)	
Total	decimal(10,2)	YES		(NULL)	
dtCreated	timestamp	YES		(NULL)	on update CURRENT_TIMESTAMP

idCheckout	idShopper	Quantity	CartDetails	SubTotal	Tax	Total	dtCreated
4	22	1	[{"productId": "9", "name": "Ethiopia", "description": "disti	100.00	4.500	104.50	2023-07-25 00:53:45

The reason we have checkout data generated and stored on the backend is, of course, for security reasons because if it was simply on the DOM (html) then anyone could go and edit the data before making a POST request and thus undermining the entire system. In this case, upon placing an order, the system uses product data on the backend, NOT the values in the front end upon creating a basket object.

The stored procedure for insert_checkout is lengthy and complex – full details are available in the build script hosted on the project repository. In simple terms as far as the Tax field is concerned, the insert_checkout stored procedure computes the order subtotal, fetches the tax factor by shopperId and multiplies subtotal against tax factor to get the tax amount.

```
CREATE DEFINER=`breweryapp`@`%` PROCEDURE `insert_checkout`(IN inIdShopper INT, IN inCartDetails
LONGTEXT)
BEGIN
    -- NOTE: SQL CODE IS OMITTED TO PRESERVE PAGE SPACE; SEE REPO FOR FULL ROUTINE
    -- Declare variables
    -- Determine the length of the JSON array
    -- Loop through each element in the JSON array
        -- Extract product ID and quantity from the JSON, remove quotes and cast them to the appropriate types
        -- Increment index counter
        -- Get price for the current product
        -- If the current date is within the product's sale period, fetch the SalePrice, otherwise fetch the regular
price.
        -- Add to subtotal
        SET subtotal = subtotal + (actual_price * quantity);
                                SET totalQuantity = totalQuantity + 1;
    END WHILE;

   -- Get the tax rate
    SELECT IFNULL(t.TaxRate, 0) INTO taxRate
                                FROM bb_shopper s
                                LEFT JOIN bb_tax t ON t.State = s.State
                                WHERE s.idShopper = inIdShopper;
-- Calculate tax
    SET tax = subtotal * taxRate;

    -- Calculate total
    SET total = subtotal + tax;
   -- Get shipping fee
    -- Insert data into bb_checkout
    INSERT INTO bb_checkout (idShopper, Quantity, CartDetails, SubTotal, Tax, Shipping, Total, dtCreated)
                VALUES (inIdShopper, totalQuantity, inCartDetails, subtotal, tax, total, NOW())
                ON DUPLICATE KEY UPDATE
                                Quantity = totalQuantity,CartDetails = inCartDetails,
                                SubTotal = subtotal,
                                Tax = tax,
                                Shipping = @shipping,
                                Total = total,
                                dtCreated = NOW();
END
```

Front-end implementation:

Source file: /templates/checkout.html:

```
<!-- Price Summary -->
      <div id="priceSummary" class="my-4">
         <h3>Price Summary</h3>
         <div class="form-group">
            <label for="totalItems">Total Items:</label>
            <span id="totalItems">{{ checkout['Quantity'] }}</span>
         </div>
         <div class="form-group">
            <label for="subTotal">SubTotal:</label>
            <span id="subTotal">{{ checkout['SubTotal'] }}</span>
         </div>
         <div class="form-group">
            <label for="shipping">Shipping:</label>
            <span id="shipping">{{ checkout['Shipping'] }}</span>
         </div>
         <div class="form-group">
            <label for="taxRate">Tax:</label>
            <span id="taxRate">{{ checkout['Tax'] }}</span>
         </div>
         <div class="form-group">
            <label for="total"><b>Total:</b></label>
            <span id="total">{{ checkout['Total'] }}</span>
         </div>
      </div>
```

Source file: app.py:

```
@app.route('/checkout')
def checkout():
   if not current_user.is_authenticated:
      # Retrieve the 'next' parameter
      next_url = request.args.get('next')
      if next_url is None:
         next_url = request.url
      return redirect(url_for('login', next=next_url))

   shopper_id = current_user.id_shopper
   shopper = sql.get_shopper_by_shopperid(shopper_id)
   checkout = sql.get_checkout_by_shopperid(shopper_id)
   return render_template('checkout.html', shopper_details=shopper, checkout=checkout)
```

Source file: sql.py:

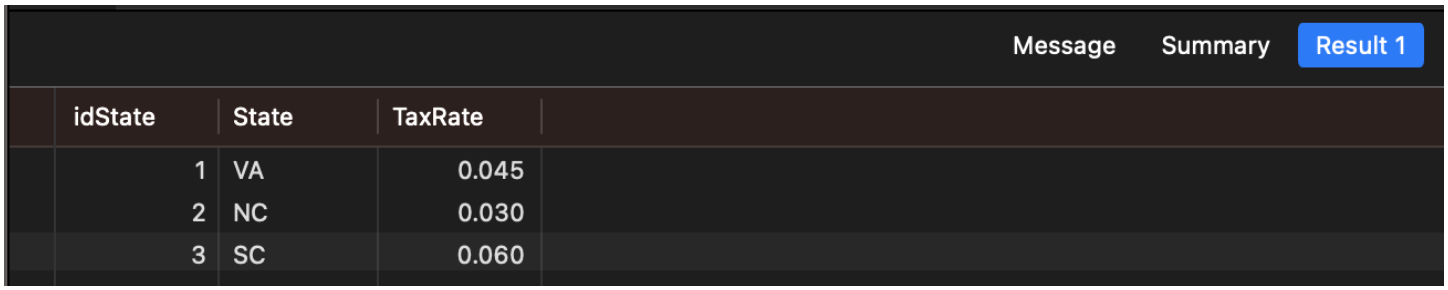
```
def get_checkout_by_shopperid(shopper_id):
   return call_stored_procedure('get_checkout_by_shopperid', (shopper_id,), fetchone=True, update=False)
```

Backend implementation:

```
CREATE DEFINER=`breweryapp`@`%` PROCEDURE `get_checkout_by_shopperid`(IN inShopperId int)
BEGIN
  SELECT * FROM bb_checkout WHERE idShopper=inShopperId;
END
```

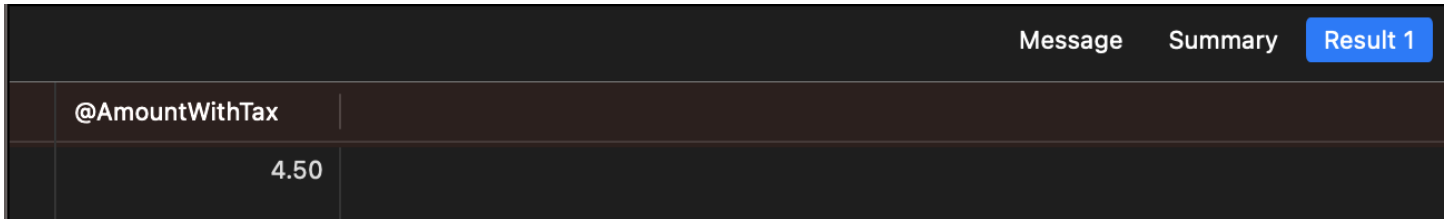
Backend Testing (dummy tests from the problem spec):

```
SELECT * FROM bb_tax
```



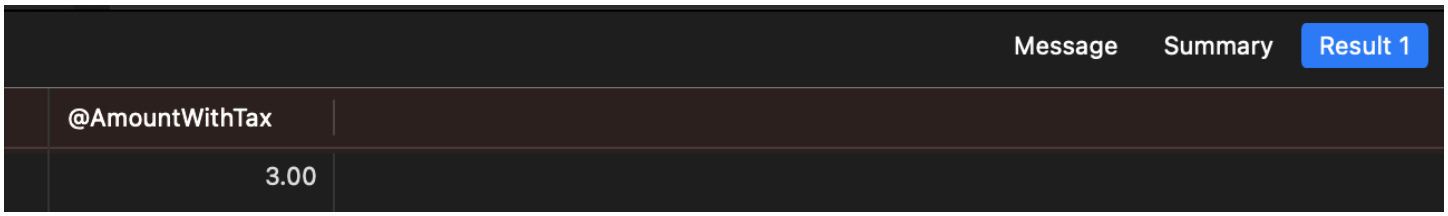
idState	State	TaxRate
1	VA	0.045
2	NC	0.030
3	SC	0.060

```
CALL get_tax_by_state_and_amount('VA', 100);
```



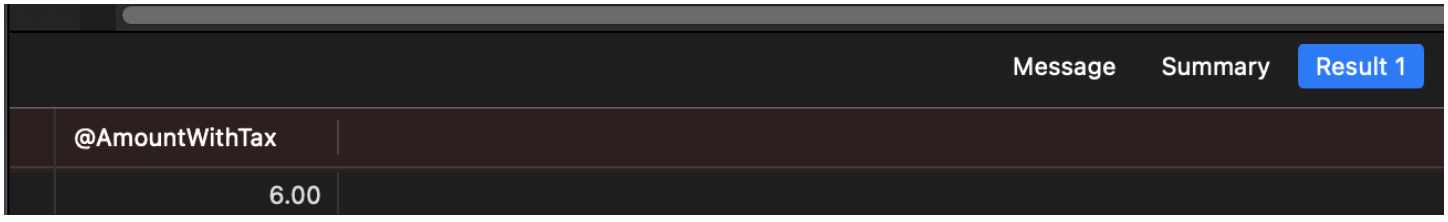
@AmountWithTax
4.50

```
CALL get_tax_by_state_and_amount('NC', 100);
```



@AmountWithTax
3.00

```
CALL get_tax_by_state_and_amount('SC', 100);
```



@AmountWithTax
6.00

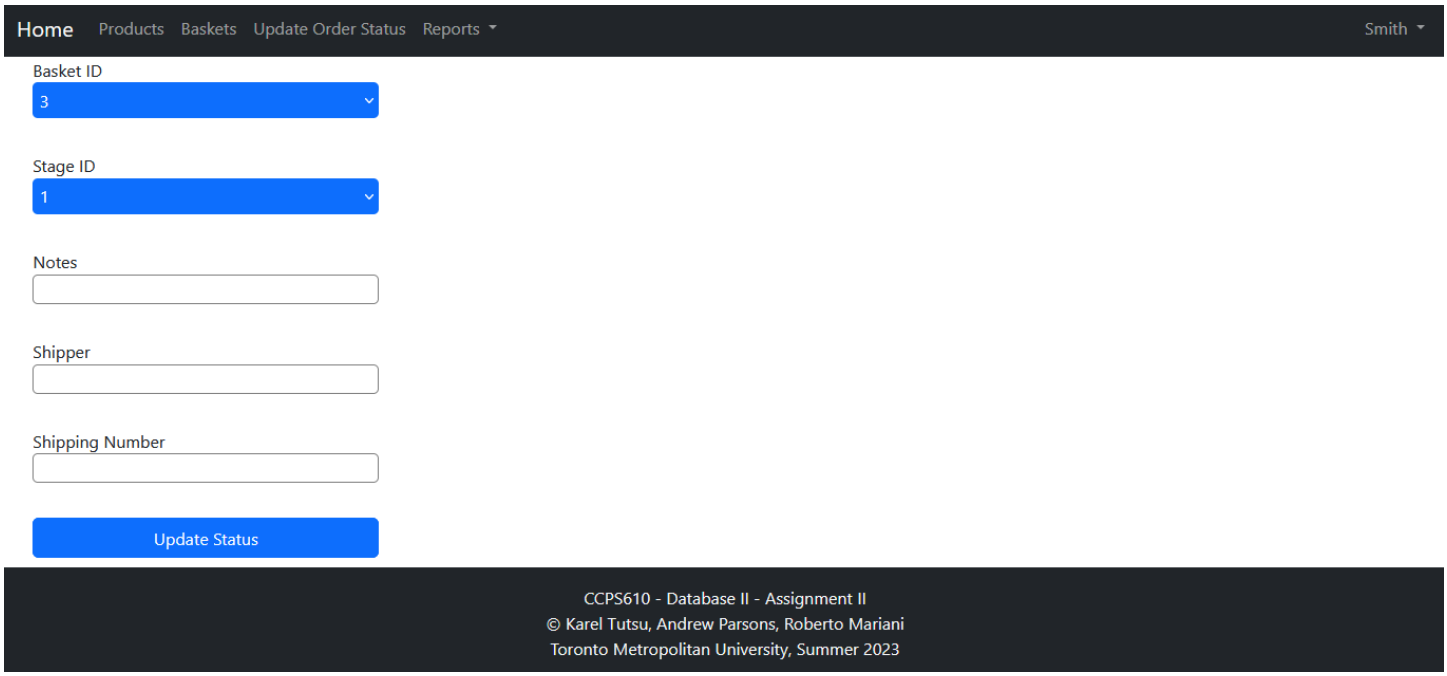
```
CALL get_tax_by_state_and_amount('DC', 100);
```

@AmountWithTax
0.00

Task 4 - Updating Order Status.

When a user navigates to the update order status page all the baskets id's are retrieved from the database and sent to the front end with the html template. The front end contains a form where the user can select the id of the basket and the stage id. The user can also choose to add notes, the shipper and shipping number. Clicking the update status button results in a post request being sent to the server. At the server the form data is passed to the stored procedure and the stored procedure then inserts the data to the bb_basket_status table in the database. If the data was inserted successfully then a success message will be rendered on the page letting the user know that the basket status was successfully updated.

Front end screenshot:



Screenshot of backend code, form data is passed to a function that calls the stored procedure in the database.

```python
@app.route('/update_status', methods=['POST', 'GET'])
@login_required
def update_status():
    success = False
    baskets = []
    result = sql.get_baskets(current_user)

    for item in result:
        baskets.append(item)

    if request.method == 'POST':
        basketId = request.form['basketId']
        stageId = request.form['stageId']
        notes = request.form['notes']
        shipper = request.form['shipper']
        shippingnum = request.form['shippingnum']

        result = sql.insert_orderstatus(basketId, stageId, notes, shipper, shippingnum)

        if result:
            action_string = "SET stageId={}, notes={}, shipper={}, shippingnum={}".format(stageId, notes, shipper, shippingnum)
            sql.insert_audit_employee(current_user.id, "UPDATE", basketId, action_string)
            success = True

    return render_template('updatestatus.html', success=success, baskets=baskets)
```

Python code to call the stored procedure:

```python
def insert_orderstatus(basketId, stageId, notes, shipper, shippingnum):
    return call_stored_procedure('insert_orderstatus', (basketId, stageId, notes, shipper, shippingnum), fetchone=False, update=True)
```

Screenshot of stored procedure:

```sql
CREATE DEFINER=`breweryapp`@`%` PROCEDURE `insert_orderstatus`(
    IN p_idbasket INT,
    IN p_idstage TINYINT,
    IN p_notes VARCHAR(50),
    IN p_shipper VARCHAR(5),
    IN p_shippingnum VARCHAR(20)
)
BEGIN
    DECLARE lv_status_id INT;
    DECLARE lv_dtstage TIMESTAMP;

    SELECT MAX(idStatus) INTO lv_status_id FROM bb_basket_status;

    SET lv_dtstage = CURRENT_TIMESTAMP;
    SET lv_status_id = lv_status_id + 1;

    INSERT INTO bb_basket_status(idstatus, idbasket, idstage, dtstage, notes, shipper, shippingnum)
      VALUES (lv_status_id, p_idbasket, p_idstage, lv_dtstage, p_notes, p_shipper, p_shippingnum);

    COMMIT;
END
```

Test case for inserting order status:

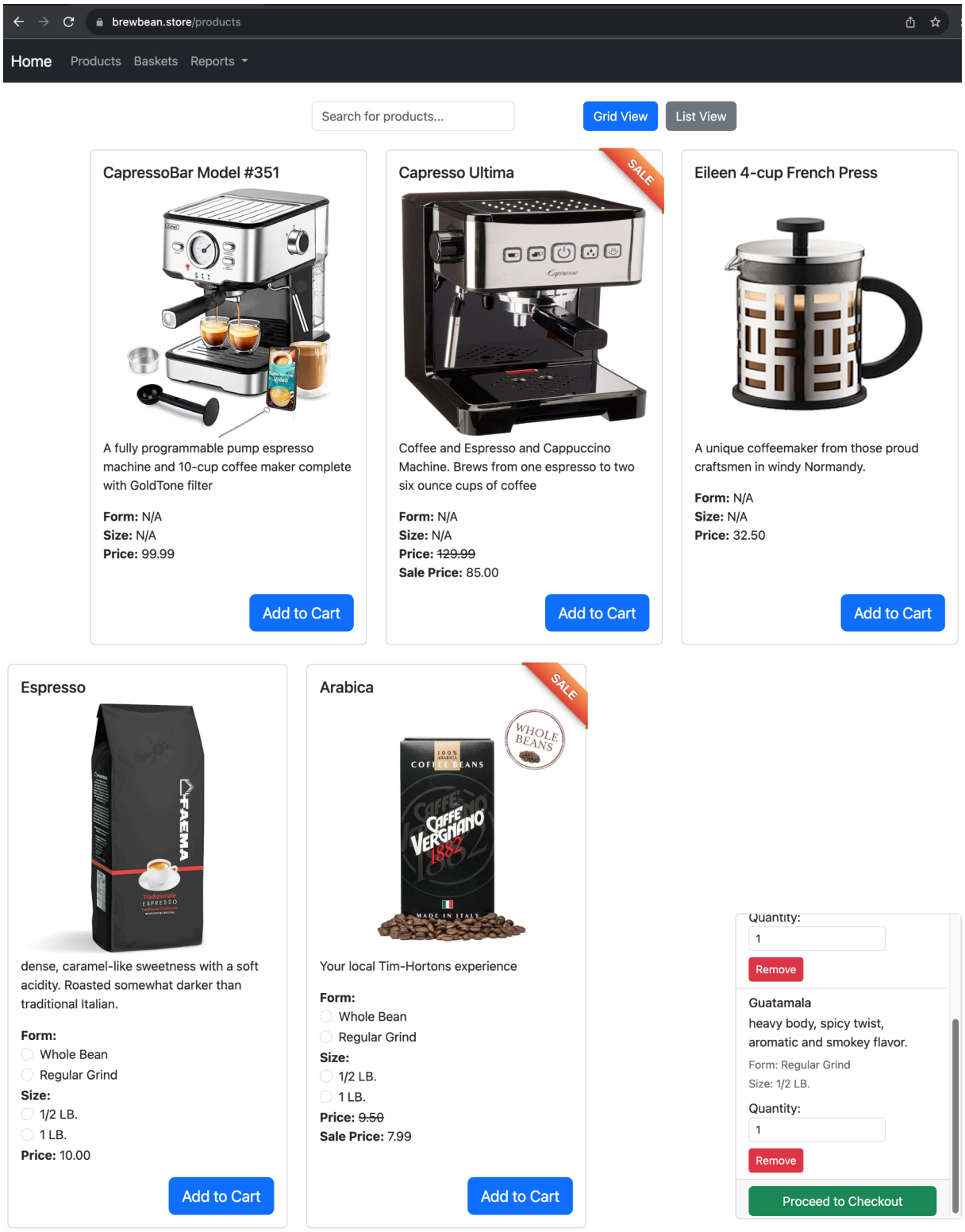
```python
def test_insert_order_status(self):
    result = self.call_stored_procedure('insert_orderstatus', ('3', '3', 'this is a test', 'UPS', '1234'), fetchone=False, update=True)
    self.assertEqual(result, None)
```

Screenshot showing test case passing:

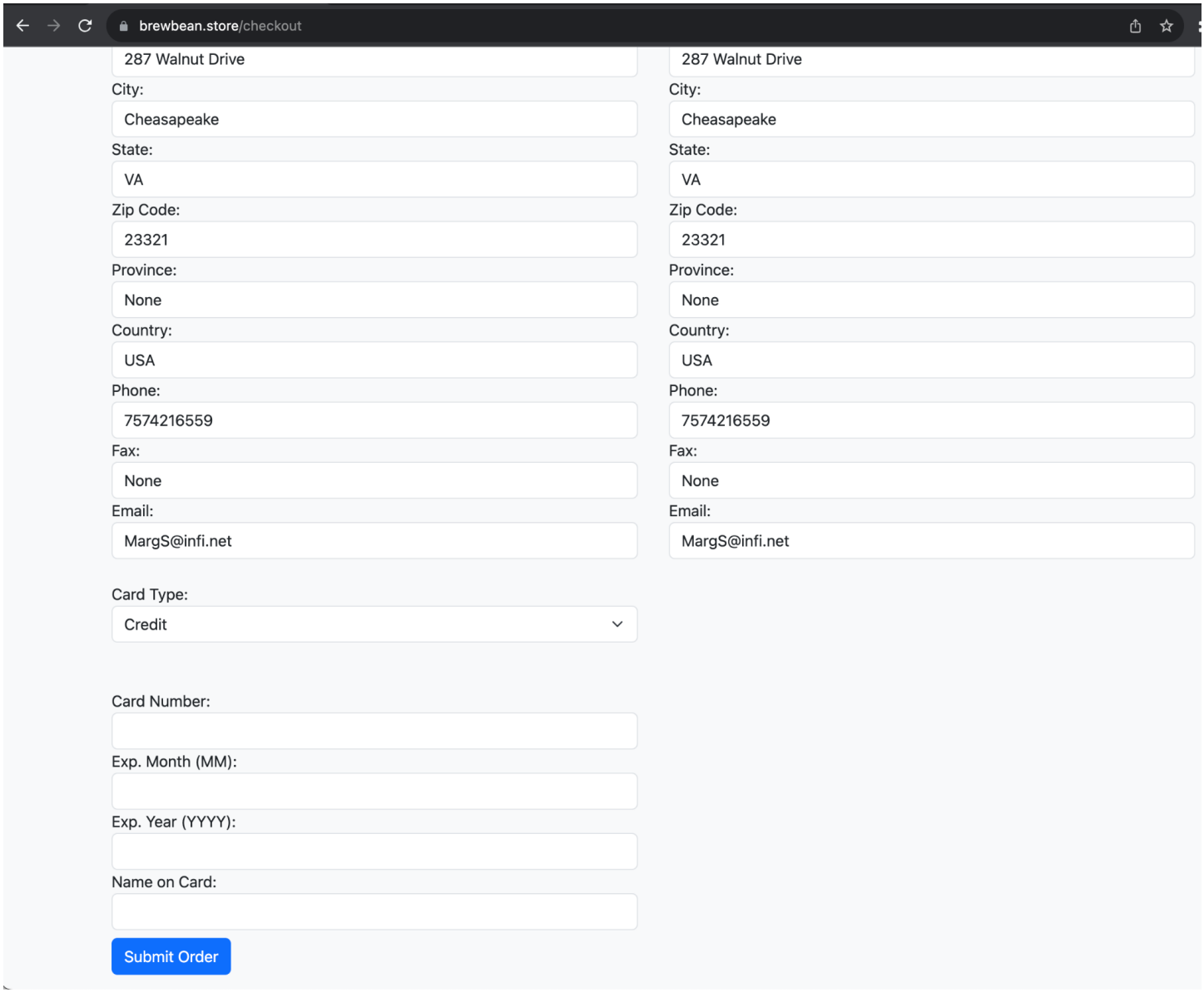
```
test_insert_order_status (__main__.TestDatabase.test_insert_order_status) ... ok
```

Task 5 : Adding Items to a Basket.

The system is designed much like any other modern shopping platform, we have a concept of a shopping cart where end users can add items, set the respective quantities, remove items and proceed to checkout:



On the checkout page the users are expected to fill in their billing and shipping details:



Once the form is successfully completed - this will create:

1. A new record in the bb_basket table that will contain relevant basket data
2. For each product that was in the basket, a separate bb_basket_item record is created

Front-end implementation:

Please see **/templates/products.html** and **/templates/checkout.html** for more details on the project repository. Since our application is not really “mickey-mouse level” then the source code is lengthy in this section, but here is a short descriptive summary:

1. Upon the user landing on the products.html page the system will call the stored procedure get_products_by_usergroup (where admin=1, employee=2, shopper=3, guest=4)
    - This will in return call the respective view created for each separate user group
2. The product page is dynamically generated, based on the returned view results (note Active=0 products are not visible to customers and guest while they are to employees and admins)
3. Guests and shoppers are able to add products to their shopping cards while employees and admins cannot (instead they have edit product option)
4. When the users add products to their cart then JavaScript code executes on the front end to facilitate that functionality
5. Upon clicking the ‘Proceed to Checkout’ button:
    - If its a ‘guest’ user then they are prompted to login as a shopper since our system does not support anonymous shoppers at the moment
    - Once the guest successfully logs on or if the shopper was already logged in, then they are simply taken to the checkout page
6. The checkout page will show the order overview and will ask the end user to enter their billing and shipping details.
7. Upon completing the html form details, the user has to confirm the order and that will create a new bb_basket record representing the order and for each product within that particular basket. A new bb_basket_item record is created.
8. The shopper can always review their basket details and items under the ‘Baskets’ page

```
<form id="checkoutForm" action="{{ url_for('submit_checkout') }}" method="post" class="my-3">
```

Source file: app.py:

```
@app.route('/submit_checkout', methods=['POST'])
def submit_checkout():
    # Get data from form
    data = request.form

    # Call the stored procedure
    sql.insert_basket([
        current_user.id_shopper,
        0,
        data['shippingFirstName'],
        data['shippingLastName'],
        data['shippingAddress'],
        data['shippingCity'],
        data['shippingState'],
        data['shippingZipCode'],
        data['shippingPhone'],
        data['shippingFax'],
        data['shippingEmail'],
        data['shippingProvince'],
        data['shippingCountry'],
        data['billingFirstName'],
        data['billingLastName'],
        data['billingAddress'],
        data['billingCity'],
        data['billingState'],
        data['billingZipCode'],
        data['billingPhone'],
        data['billingFax'],
        data['billingEmail'],
        data['billingProvince'],
        data['billingCountry'],
        data['billingCardType'],
        data['billingCardNumber'],
        data['billingExpMonth'],
        data['billingExpYear'],
        data['billingCardName']
    ])

    # Redirect to a success page or wherever you want
    return redirect(url_for('baskets'))
```

Source file: sql.py:

```
def insert_basket(basket_data):
    return call_stored_procedure('insert_basket', basket_data)
```

Back-end implementation:

```
CREATE DEFINER=`breweryapp`@`%` PROCEDURE `insert_basket`(IN inIdShopper INT(5),
   IN inPromo INT(2), IN inShipFirstName VARCHAR(10),IN inShipLastName VARCHAR(20),
   IN inShipAddress VARCHAR(40),IN inShipCity VARCHAR(20),IN inShipState VARCHAR(2),
   IN inShipZipCode VARCHAR(15),IN inShipPhone VARCHAR(15),IN inShipFax VARCHAR(10),
   IN inShipEmail VARCHAR(25),IN inShipProvince VARCHAR(20), IN inShipCountry VARCHAR(20),
   IN inBillFirstName VARCHAR(10),IN inBillLastName VARCHAR(20),IN inBillAddress VARCHAR(40),
   IN inBillCity VARCHAR(20),IN inBillState VARCHAR(2),IN inBillZipCode VARCHAR(15),
   IN inBillPhone VARCHAR(15),IN inBillFax VARCHAR(10),IN inBillEmail VARCHAR(25),
   IN inBillProvince VARCHAR(20), IN inBillCountry VARCHAR(20),IN inCardType CHAR(1),
   IN inCardNumber VARCHAR(20), IN inExpMonth CHAR(2),IN inExpYear CHAR(4),
   IN inCardName VARCHAR(25))
BEGIN
   DECLARE done INT DEFAULT 0;
   DECLARE checkout_subtotal DECIMAL(10, 2);
   DECLARE checkout_tax DECIMAL(10, 3);
   DECLARE checkout_total DECIMAL(10, 2);
   DECLARE checkout_quantity INT;
   DECLARE cart_details JSON DEFAULT NULL;
   DECLARE cart_item JSON DEFAULT NULL;
   DECLARE idx INT DEFAULT 0;
   DECLARE num_items INT DEFAULT 0;

   START TRANSACTION;
   BEGIN

      INSERT INTO bb_basket (
         IdShopper,dtCreated,Promo,

ShipFirstName,ShipLastName,ShipAddress,ShipCity,ShipState,ShipZipCode,ShipPhone,ShipFax,ShipEmail,S
hipProvince,ShipCountry,
         BillFirstName,BillLastName,BillAddress,BillCity,BillState,BillZipCode,BillPhone,BillFax,BillEmail,
         dtOrdered,BillProvince,BillCountry,CardType,CardNumber,ExpMonth,ExpYear,CardName
      ) VALUES (
         inIdShopper,NOW(),inPromo,

inShipFirstName,inShipLastName,inShipAddress,inShipCity,inShipState,inShipZipCode,inShipPhone,inShipFa
x,inShipEmail,inShipProvince,inShipCountry,

inBillFirstName,inBillLastName,inBillAddress,inBillCity,inBillState,inBillZipCode,inBillPhone,inBillFax,inBillEmail
,
         NOW(),inBillProvince,inBillCountry,inCardType,inCardNumber,inExpMonth,inExpYear,inCardName
      );

      SELECT LAST_INSERT_ID() INTO @last_inserted_id;
      SELECT SubTotal, Tax, Total, Quantity, CartDetails
      INTO checkout_subtotal, checkout_tax, checkout_total, checkout_quantity, cart_details
      FROM bb_checkout
```

```
        WHERE idShopper = inIdShopper
        ORDER BY dtCreated DESC
        LIMIT 1;

        UPDATE bb_basket
        SET Quantity = checkout_quantity,
            SubTotal = checkout_subtotal,
            Tax = checkout_tax,
            Total = checkout_total,
            OrderPlaced=1
        WHERE idBasket = @last_inserted_id AND IdShopper=inIdShopper;

        -- Get the number of items in the JSON array
        SET num_items = JSON_LENGTH(cart_details);

        -- Loop through each item in the JSON array
        WHILE idx < num_items DO
            SET cart_item = JSON_EXTRACT(cart_details, CONCAT('$[', idx, ']'));

            SET @productId = JSON_EXTRACT(cart_item, '$.productId');
            SET @productId = CAST(JSON_UNQUOTE(@productId) AS UNSIGNED);
            SET @price = CAST(JSON_EXTRACT(cart_item, '$.price') AS DECIMAL(6, 2));
            SET @quantity = CAST(JSON_EXTRACT(cart_item, '$.quantity') AS UNSIGNED);
            SET @option1 = JSON_EXTRACT(cart_item, '$.option1');
            IF @option1 IS NOT NULL THEN
                SET @option1 = JSON_EXTRACT(cart_item, '$.option1');
                                                        SET @option1 =
CAST(JSON_UNQUOTE(@option1) AS UNSIGNED);
            END IF;

            SET @option2 = JSON_EXTRACT(cart_item, '$.option2');
            IF @option2 IS NOT NULL THEN
                    SET @option2 = JSON_EXTRACT(cart_item, '$.option2');
                    SET @option2 = CAST(JSON_UNQUOTE(@option2) AS UNSIGNED);
            END IF;
            CALL insert_basketitem(@productId, @price, @quantity, @last_inserted_id, @option1, @option2);
            SET idx = idx + 1;
        END WHILE;

    END;

    IF done = 1 THEN
        ROLLBACK;
    ELSE
        COMMIT;
    END IF;

END
```

```
CREATE DEFINER=`breweryapp`@`%` PROCEDURE `insert_basketitem`(
  IN inIdProduct INT(2),
        IN inPrice DECIMAL(6, 2),
        IN inQuantity INT(2),
        IN inIdBasket INT(5),
        IN inOption1 INT(2),
        IN inOption2 INT(2))
BEGIN
  INSERT INTO bb_basket_item (idProduct, Price, Quantity, idBasket, option1, option2) VALUES (inIdProduct,
inPrice, inQuantity, inIdBasket, inOption1, inOption2);
END
```

Task 6 - Identifying Sale Products.

Within the **bb_products** table, there are three columns that provide information on specific sale periods and prices. This task was intended to identify if a product happened to be on sale on the day the information was requested and if so, present that item on the front-end as a sale item at sale price.

Snippet of bb_product table in the MySQL DB

idProduct	ProductName	Description	ProductImage	Price	SaleStart	SaleEnd	SalePrice
1	CapressoBar Model #351	A fully programmable pump espresso machine and 10-cup coffee maker complete with GoldTone filter	capresso.gif	99.99	(NULL)	(NULL)	(NULL)
2	Capresso Ultima	Coffee and Espresso and Cappuccino Machine. Brews from one espresso to two six ounce cups of coffee	capresso2.gif	129.99	2023-06-30	2023-08-05	85.00
3	Eileen 4-cup French Press	A unique coffeemaker from those proud craftsmen in windy Normandy.	frepress.gif	32.50	(NULL)	(NULL)	(NULL)
4	Coffee Grinder	Avoid blade grinders! This mill grinder allows you to choose a fine grind to a coarse grind.	grind.gif	28.50	2023-05-01	2023-05-31	19.99
5	Sumatra	Spicy and intense with herbal aroma.	sumatra.jpg	10.50	(NULL)	(NULL)	(NULL)
6	Guatamala	heavy body, spicy twist, aromatic and smokey flavor.	guatamala.jpg	10.00	2012-06-01	2012-06-15	8.00
7	Columbia	dry, nutty flavor and smoothness	columbia.jpg	10.80	(NULL)	(NULL)	(NULL)
8	Brazil	well-balanced mellow flavor, a medium body with hints of cocoa and a mild, nut-like aftertaste	brazil.jpg	10.80	2023-07-01	2023-08-01	8.00
9	Ethiopia	distinctive berry-like flavor and aroma, reminds many of a fruity, memorable wine.	ethiopia.jpg	10.00	(NULL)	(NULL)	(NULL)
10	Espresso	dense, caramel-like sweetness with a soft acidity. Roasted somewhat darker than traditional Italian.	espresso.jpg	10.00	(NULL)	(NULL)	(NULL)
11	Arabica	Your local Tim-Hortons experience	arabica.jpg	9.50	2023-01-01	2023-12-31	7.99
13	TestProduct DO NOT DELE	Test Original Description	arabica.jpg	1.00	(NULL)	(NULL)	(NULL)

Front-end Implementation:

- When the user navigates to products.html, the **get_products_by_usergroup** procedure is called and returns the products as well as a function call to **get_sale_info** to understand if that product should be on sale at the time the procedure was called (see back-end implementation below)

/products route within app.py, which calls **get_products_by_usergroup**

```
@app.route('/products')
def products():
   if current_user.is_authenticated:
       if current_user.usergroup in [const.ADMIN, const.EMPLOYEE]:
           products = sql.call_stored_procedure('get_products_by_usergroup',
(current_user.usergroup, ))
       else:
           products = sql.call_stored_procedure('get_products_by_usergroup', (const.GUEST, ))
   else:
       products = sql.call_stored_procedure('get_products_by_usergroup', (const.GUEST, ))

   # Render the products page with the retrieved products
   return render_template('products.html', data=products)
```

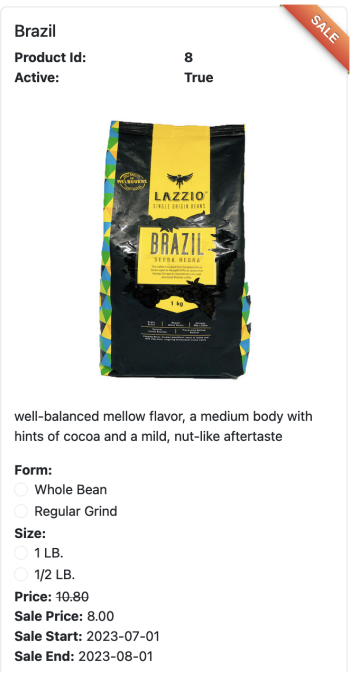
- The language to translate the 1 or 0 into the corresponding sale message was implemented on the front end in products.html, which shows a ribbon wrapper around the product if the product is on sale on the given day (as well as presenting the “sale price”)

Snippet of products.html

```
                        {% if row.SaleFlag == 1 %}
                            <div class="ribbon-wrapper">
                                <div class="ribbon">SALE</div>
                            </div>
                            <li class="card-price"><strong>Price:</strong> <s>{{ row.Price
}}</s></li>
```

```
                                <li class="card-sale-price"><strong>Sale Price:</strong> <span
id="salePrice_{{ row.idProduct }}" class="sale-price">{{ row.SalePrice }}</span></li>
                            {% else %}
                                <li class="card-price"><strong>Price:</strong> {{ row.Price }}</li>
                            {% endif %}
```

Image of “On Sale” Product while logged in on an employee account (employees see sale start/end date)



Back-end Implementation:

Function get_sale_info(inIdProduct) from MySQL Database:

```
CREATE DEFINER=`breweryapp`@`%` FUNCTION `get_sale_info`(inIdProduct INT(2),inDate DATE) RETURNS int
BEGIN
  DECLARE sale_info INT;
  SELECT
    CASE WHEN inDate BETWEEN p.SaleStart AND p.SaleEnd THEN 1 ELSE 0 END INTO sale_info
  FROM bb_product p
  WHERE inIdProduct = p.idProduct;
  RETURN sale_info;
END
```

- First, a **get_sale_info** mysql function was created that accepts in a productId and a date and returns a value depending on if that product was considered “on sale” for the date that was passed in
    - If the item was considered “on sale”, the function returns a 1 - otherwise it returns a 0
- The language to translate the 1 or 0 into the corresponding sale message was implemented on the front end in products.html (see above in front-end implementation)
- The **get_products_by_usergroup** procedure calls this function when it needs to return products information to the front end:

Procedure get_products_by_usergroup(idUsergroup) from MySQL Database:

```
CREATE DEFINER=`breweryapp`@`%` PROCEDURE `get_products_by_usergroup`(IN idUsergroup INT)
BEGIN
    CASE idUsergroup
        WHEN 1 THEN
            SELECT *, get_sale_info(idProduct,CURRENT_DATE()) SaleFlag FROM bb_product_admin;
        WHEN 2 THEN
            SELECT *, get_sale_info(idProduct,CURRENT_DATE()) SaleFlag FROM bb_product_employee;
        WHEN 3 THEN
            SELECT *, get_sale_info(idProduct,CURRENT_DATE()) SaleFlag FROM bb_product_guest;
        ELSE
            SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = 'Invalid user group provided';
    END CASE;
END
```

Testing:

- Two test cases generated to ensure that the product with idProduct = 6 was listed as “on sale” during the sale period but NOT listed as “on sale” outside the test period

Relevant test case within test.py

```
def test_get_sale_info(self):
    test_jun10_list = self.call_mysql_function('get_sale_info', ('6', '2012-06-10'))
    test_jun10_dict = test_jun10_list[0]
    test_jun10_val = list(test_jun10_dict.values())[0]
    self.assertEqual(test_jun10_val,1)

    test_jun19_list = self.call_mysql_function('get_sale_info', ('6', '2012-06-19'))
    test_jun19_dict = test_jun19_list[0]
    test_jun19_val = list(test_jun19_dict.values())[0]
    self.assertEqual(test_jun19_val, 0)
```

Relevant result

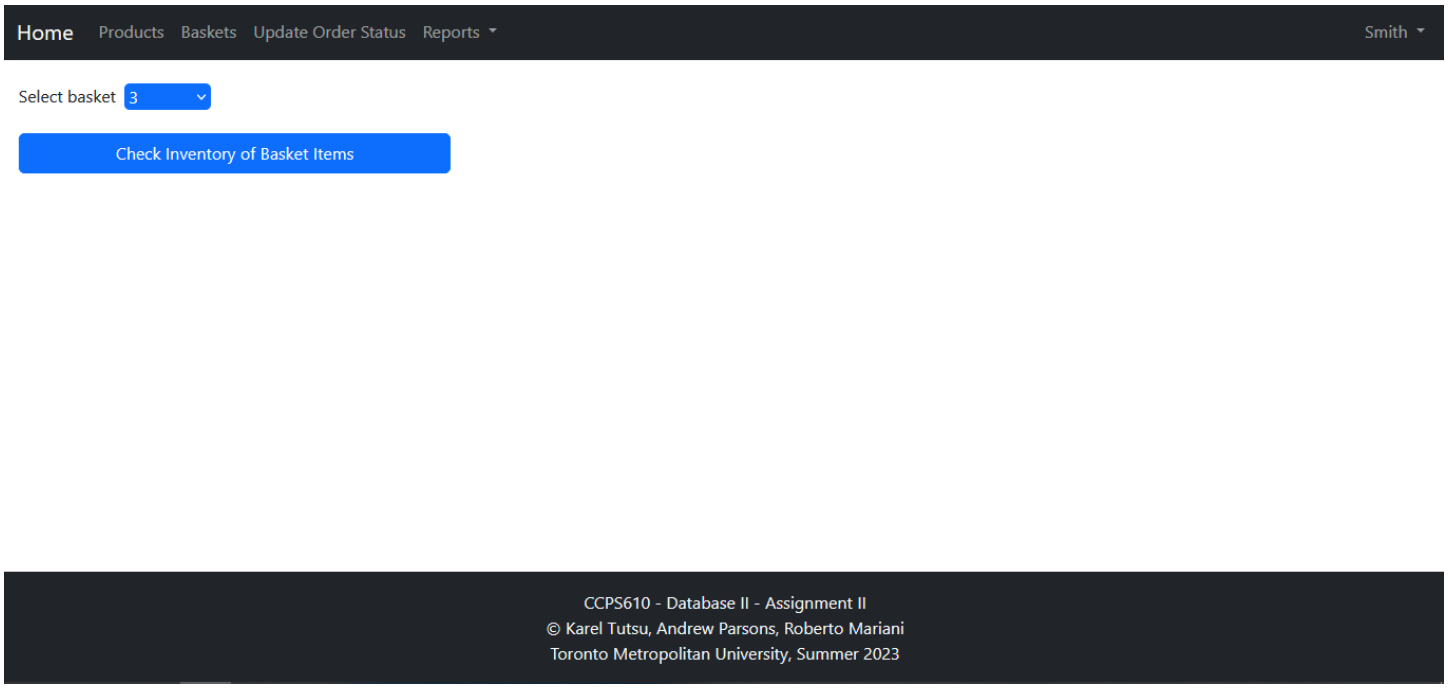
```
test_get_sale_info (__main__.TestDatabase) ... ok
```

Report 1

Report to show whether all items in her/his basket are in Stock or not? Using an Explicit Cursor.

When the user navigates to the check “all items in stock” report page the user will see a form where they can select the basket id. When the user presses the “Check Inventory of Basket Items” button a post request will be sent to the server. The server will call the stored procedure that checks if all the items in the basket have a stock greater than 0. The stored procedure takes as input the basket id, then all the items from the basket are selected and stored in a cursor. The cursor is then iterated over to and the stock of each item is checked. If the stock is 0 then the procedure exits the loop and a message saying “All items NOT in stock” is returned. If all items are in stock then a message saying “All items in stock” is returned. Once the procedure is done, the server updates the html template and sends the updated template with the message to the user.

Check inventory page:



Back end code:

```python
@app.route('/report/all_items_in_stock', methods=['GET', 'POST'])
@login_required
def all_items_in_stock():
    baskets = []
    message = None
    result = None

    if current_user.usergroup in [1,2]:
        result = sql.get_baskets(current_user)
    else:
        result = sql.get_baskets_by_shopper_id(current_user.id_shopper)

    for item in result:
        baskets.append(item)

    if request.method == 'POST':
        shopperId = request.form['basketId']
        result = sql.all_items_in_stock(shopperId)

        if result:
            message = result['message']


    return render_template('allitemsinstock.html', baskets=baskets, message=message)
```

Function that calls stored procedure:

```python
def all_items_in_stock(shopper_id):
    return call_stored_procedure('all_items_in_stock', (shopper_id,), fetchone=True, update=True)
```

Stored procedure in the database:

```
CREATE DEFINER=`breweryapp`@`%` PROCEDURE `all_items_in_stock`(
  IN basketId INT
)
BEGIN
  DECLARE not_in_stock TINYINT DEFAULT 1;
  DECLARE finished INTEGER DEFAULT 0;
  DECLARE p_stock INT;
  DECLARE p_id INT;

  DECLARE basket_items CURSOR FOR SELECT idProduct FROM bb_basket_item WHERE idBasket = basketId;

  DECLARE CONTINUE HANDLER
        FOR NOT FOUND SET finished = 1;

  OPEN basket_items;

  myLoop: LOOP
    FETCH basket_items INTO p_id;

    IF finished = 1 THEN
      LEAVE myLoop;
    END IF;

    SELECT stock INTO p_stock FROM bb_product WHERE idProduct = p_id;

    IF p_stock = 0 THEN
      SET not_in_stock = 0;
      LEAVE myLoop;
    END IF;
  END LOOP;

  IF not_in_stock = 0 THEN
    SELECT 'All items NOT in stock!' AS message;
  ELSE
    SELECT 'All items in stock!' AS message;
  END IF;

END
```

Test cases to test stored procedure:

```
def test_all_items_in_stock(self):
    result = self.call_stored_procedure('all_items_in_stock', ('3', ), fetchone=True, update=True)
    message = result['message']
    self.assertEqual(message, 'All items in stock!')

def test_all_items_not_in_stock(self):
    result = self.call_stored_procedure('all_items_in_stock', ('16', ), fetchone=True, update=True)
    message = result['message']
    self.assertEqual(message, 'All items NOT in stock!')
```

Screenshot of test cases passing:

```
test_all_items_in_stock (__main__.TestDatabase.test_all_items_in_stock) ... ok
test_all_items_not_in_stock (__main__.TestDatabase.test_all_items_not_in_stock) ... ok
```

Report 2 - Calculating a Shopper's Total Spending.

Added to the NAV bar on the front end, the purchase report provides an employee or admin user with the access to see all shoppers and the total spending they have done across all their baskets in a simple table.

Front-end Implementation:

- When the user navigates to report/purchasereport.html, the **get_total_purchase_amt** procedure is called and returns the shopper id, first and last name, as well as a function call to **get_purchase_amt_by_shopperid** to sum the total value of their baskets (see back-end implementation below)

/report/purchasereport.html route within app.py, which calls **get_total_purchase_amt**

```
@app.route('/report/purchasereport')
def purchasereport():
   shopper_totals = {}
   if current_user.is_authenticated:
       if current_user.usergroup in [const.ADMIN, const.EMPLOYEE]:
           shopper_totals = sql.call_stored_procedure('get_total_purchase_amt', ())

   # Render the reports page with the total amount report
   return render_template('purchasereport.html', shopper_totals=shopper_totals)
```

Snippet of purchasereport.html

```
{% if shopper_totals %}
      <table class="table table-striped table-hover table-bordered" id="purchaseReportTable">
          <thead class="table-dark">
              <tr>
                  {% for column in shopper_totals[0].keys() %}
                      <th>{{ column }}</th>
                  {% endfor %}
              </tr>
          </thead>
          <tbody>
          {% for shopper_total in shopper_totals %}
                  <tr>
                      {% for k,v in shopper_total.items() %}
                          <td>{{ v }}</td>
                      {% endfor %}
                  </tr>
              {% endfor %}
          </tbody>
      </table>
      {% else %}
          <p class="text-center text-muted">No shoppers found.</p>
      {% endif %}
```

Image of Reports tab in NAV bar → Purchase Report

Report - Total Spending by Shopper

idShopper	FirstName	LastName	total_purchase_amt
21	John	Carter	370.59
22	Margaret	Somner	619.3
23	Kenny	Ratman	81.75
24	Camryn	Sonnie	61.1
25	Scott	Savid	329.22
26	Monica	Cast	761.74
27	Pete	Parker	21.69
28	Moe	Snow	None
29	foo	bar	None

Back-end Implementation:

Function get_purchase_amt_by_shopperid(inShopperId) from MySQL Database:

```
CREATE DEFINER=`breweryapp`@`%` FUNCTION `get_purchase_amt_by_shopperid`(inShopperId INT) RETURNS float
BEGIN
  DECLARE lv_result   FLOAT;
  SELECT SUM(Total) INTO lv_result FROM bb_basket b WHERE b.idShopper=inShopperId;
RETURN lv_result;
END
```

- First, a **get_purchase_amt_by_shopperid** mysql function was created that accepts in a shopperId and returns the sum total in dollar value of all baskets owned by that shopperId
- The **get_total_purchase_amt** procedure calls this function when it needs to run the report on the front end (i.e. in the purchasereport route within app.py)

Procedure get_total_purchase_amt() from MySQL Database:

```
CREATE DEFINER=`breweryapp`@`%` PROCEDURE `get_total_purchase_amt`()
BEGIN
  SELECT idShopper, FirstName, LastName, get_purchase_amt_by_shopperid(idShopper) as total_purchase_amt FROM bb_shopper;
END
```

Testing:

- One test case generated to ensure that the product with idShopper = 25 was equal to the sum of its basket in bb_basket
- One test to ensure that the **get_total_purchase_amt** successfully generated correct totals for all shopper ids

Relevant test cases within test.py

```python
def test_get_purchase_amt_by_shopperid(self):
    test_single_shopper_id_list = self.call_mysql_function('get_purchase_amt_by_shopperid', (25,))
    test_single_shopper_id_dict = test_single_shopper_id_list[0]
    test_single_shopper_id_val = list(test_single_shopper_id_dict.values())[0]
    self.assertEqual(test_single_shopper_id_val,329.22)
```

```python
def test_get_total_purchase_amt(self):
    # first test get_purchase_amt_by_shopperid for all listed shoppers in bb_shopper
    try:
        # Create SQL statement to call the function
        sql = f"SELECT idShopper FROM bb_shopper;"
        args = []
        self.cursor.execute(sql, args)
        return_vals = self.cursor.fetchall()
        shopper_ids = [d['idShopper'] for d in return_vals]
    except Exception as ex:
        print("An error occurred: ", str(ex))
        self.connection.rollback()  # Rollback the transaction
        shopper_ids = []

    # get the individual purchase amounts for each shopper
    exp_results_dict = {}
    for shopper_id in shopper_ids:
        single_shopper_id_list = self.call_mysql_function('get_purchase_amt_by_shopperid', (shopper_id,))
        single_shopper_id_dict = single_shopper_id_list[0]
        exp_results_dict[shopper_id] = list(single_shopper_id_dict.values())[0]

    # fetch results using get_total_purchase_amt
    response_list = self.call_stored_procedure('get_total_purchase_amt', ())
    result = {}
    for item in response_list:
        shopper_id = item['idShopper']
        shopper_spend = item['total_purchase_amt']
        result[shopper_id] = shopper_spend

    self.assertEqual(result,exp_results_dict)
```

Relevant results

```
test_get_purchase_amt_by_shopperid (__main__.TestDatabase) ... ok
```

```
test_get_total_purchase_amt (__main__.TestDatabase) ... ok
```

Account Registration

To register an account the user can fill out all the required fields in the form and then click the Register Account button. Once the user clicks the button a post request will be sent to the user and all the form fields will be passed to the stored procedure. The stored procedure will first check if the user supplied email and username are already registered by another user. If they are then an error message will be outputted and the server will update the html template with the error message shown to the user. Otherwise the procedure will insert a record into the bb_account table and the bb_shopper table and a success message will be outputted. Then the server will update the html template with the success message to let the user know that their account has been created successfully.

Screenshot of registration page:

Backend code for registration page:

```python
@app.route('/registerShopper', methods=['GET', 'POST'])
def registerShopper():
    message = None

    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
        firstname = request.form['firstName']
        lastname = request.form['lastName']
        address = request.form['address']
        city = request.form['city']
        state = request.form['state']
        zipcode = request.form['zipcode']
        phone = request.form['phone']
        email = request.form['email']
        province = request.form['province']
        country = request.form['country']

        result = sql.register_shopper(username, password, firstname, lastname, address, city, state, zipcode, phone, email, province, country)

        if result['message'] == 'Username in use' or result['message'] == 'Email in use':
            message = result['message'] + ", registration failed"
        else:
            message = "Account Created Successfully"
```

Function to call stored procedure:

```
def register_shopper(username, password, firstname, lastname, address, city, state, zipcode, phone, email, province, country):
    return call_stored_procedure('register_shopper', (username, password, firstname, lastname, address, city, state, zipcode, phone, email, province, country), fetchone=True, up
```

Screenshot of stored procedure:

```
CREATE DEFINER=`breweryapp`@`%` PROCEDURE `register_shopper`(
    IN lv_username VARCHAR(255),
    IN lv_password VARCHAR(255),
    IN lv_firstname VARCHAR(15),
    IN lv_lastname VARCHAR(20),
    IN lv_address VARCHAR(40),
    IN lv_city VARCHAR(20),
    IN lv_state CHAR(2),
    IN lv_zipcode VARCHAR(15),
    IN lv_phone VARCHAR(10),
    IN lv_email VARCHAR(25),
    IN lv_province VARCHAR(15),
    IN lv_country VARCHAR(15)
)
registerShopper:BEGIN
    DECLARE lv_idShopper INT;
    DECLARE lv_username_count INT;
    DECLARE lv_email_count INT;

    SELECT COUNT(*) INTO lv_username_count FROM bb_account WHERE Username = lv_username;
    SELECT COUNT(*) INTO lv_email_count FROM bb_shopper WHERE Email = lv_email;

    IF lv_username_count > 0 THEN
      SELECT 'Username in use' as message;
      LEAVE registerShopper;
    END IF;

    IF lv_email_count > 0 THEN
      SELECT 'Email in use' as message;
      LEAVE registerShopper;
    END IF;

    INSERT INTO bb_shopper(FirstName, LastName, Address, City, State, ZipCode, Phone, Email, Province, Country)
    VALUES(lv_firstname, lv_lastname, lv_address, lv_city, lv_state, lv_zipcode, lv_phone, lv_email, lv_province,lv_country);

    SET lv_idShopper = LAST_INSERT_ID();

    INSERT INTO bb_account(idUsergroup, idShopper, Username, `Password`, Active)
    VALUES(4, lv_idShopper, lv_username, lv_password, 1);

    COMMIT;

    SELECT 'Success' as message;

  END
```

Test cases used to test stored procedure:

```
def test_register_user(self):
    result = self.call_stored_procedure('register_shopper', ('test', '1234', 'mike', 'smith', '1234 Random Street', 'Toronto', '', '1234', '4161234567', 'test@gmail.com',
    message = result['message']
    self.assertEqual(message, 'Success')

    self.call_stored_procedure('delete_user_by_email_and_username', ('test', 'test@gmail.com'), fetchone=False, update=True)

def test_register_user_same_username(self):
    result = self.call_stored_procedure('register_shopper', ('Smith', '1234', 'mike', 'smith', '1234 Random Street', 'Toronto', '', '1234', '4161234567', 'test2@gmail.com'
    message = result['message']
    self.assertEqual(message, 'Username in use')

def test_register_user_same_email(self):
    result = self.call_stored_procedure('register_shopper', ('test3', '1234', 'mike', 'smith', '1234 Random Street', 'Toronto', '', '1234', '4161234567', 'Crackjack@aol.co
    message = result['message']
    self.assertEqual(message, 'Email in use')
```

Screenshot of test cases passing:

```
test_register_user (__main__.TestDatabase.test_register_user) ... ok
test_register_user_same_email (__main__.TestDatabase.test_register_user_same_email) ... ok
test_register_user_same_username (__main__.TestDatabase.test_register_user_same_username) ... ok
test_tax_rates (__main__.TestDatabase.test_tax_rates) ... ok
```