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](https://www.digitalocean.com), 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](https://www.gandi.net/en-CA) 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
   1. adduser yourusername
   2. usermod -aG sudo yourusername
   3. su yourusername
   4. sudo groupadd workgroup
   5. 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)

1. sudo systemctl restart mysql
2. sudo ufw allow 3306 (open port)
3. Login to mysql and create remote access accounts
   1. mysql -u root -p
   2. CREATE USER 'breweryapp'@'%' IDENTIFIED BY 'secret';
   3. CREATE DATABASE brewery;
   4. GRANT ALL PRIVILEGES ON brewery.\* TO 'breweryapp'@'%';
   5. FLUSH PRIVILEGES;
   6. EXIT;

At this point the database is remotely accessible

1. Install and configure Web Server
   1. sudo apt install nginx python3-pip python3-dev python3.10-venv build-essential libssl-dev libffi-dev python3-setuptools
   2. Create project directory and set group
      1. sudo mkdir /brewery
      2. sudo chgrp -R workgroup /brewery
      3. sudo chmod -R 2775 /brewery
   3. Create and activate python venv
      1. python3 -m venv .venv
      2. . .venv/bin/activate
      3. pip install flask wheel uwsgi python-dotenv mysql-connector-python flask\_login
2. 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.

## 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

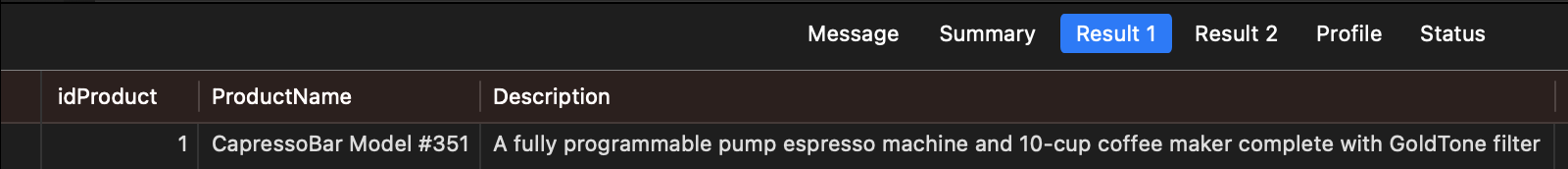
**WHERE** idProduct=inProductId;

**END**

# 

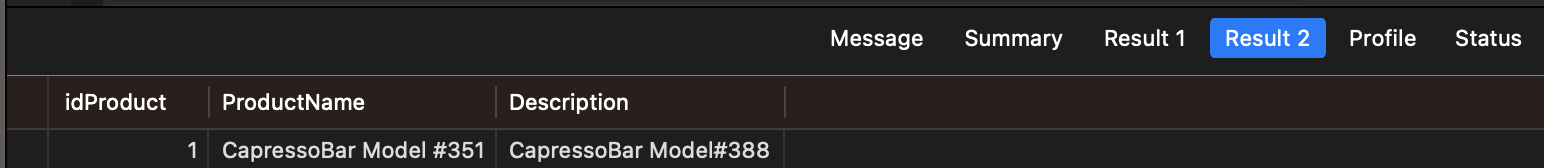
## Backend testing:

**SELECT** idProduct, ProductName, Description **from** bb\_product **WHERE** idProduct=**1**;



**CALL** update\_product(**1**, "CapressoBar Model #351", "CapressoBar Model#388", **1**, **NULL**, **NULL**, **NULL**);

**SELECT** idProduct, ProductName, Description **from** bb\_product **WHERE** idProduct=**1**;



# 

# 

# 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**

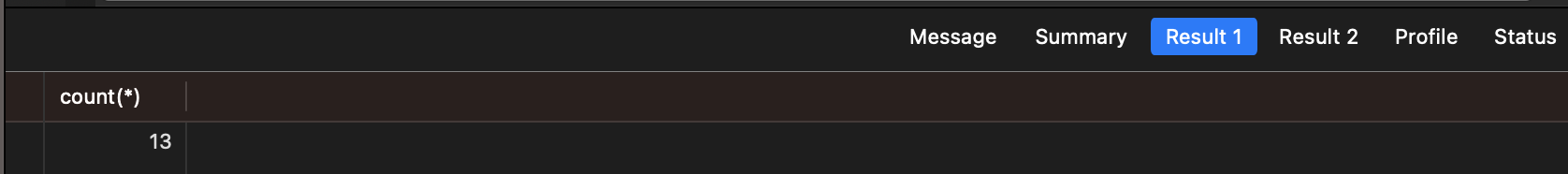
**INSERT** **INTO** bb\_product (ProductName, Description, ProductImage, Price, Active, idDepartment)

**VALUES** (inProductName, inDescription, inProductImage, inPrice, inActive, inIdDepartment);

**END**

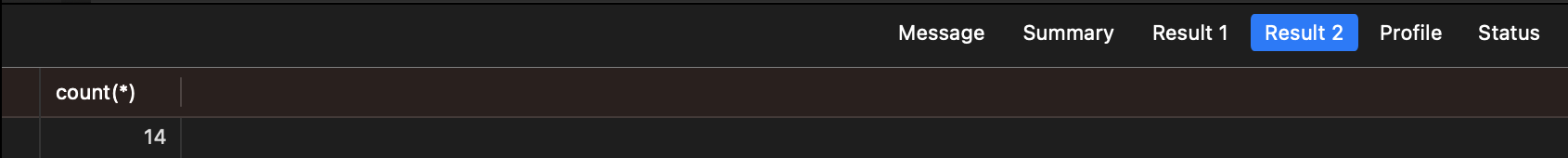
## Backend testing:

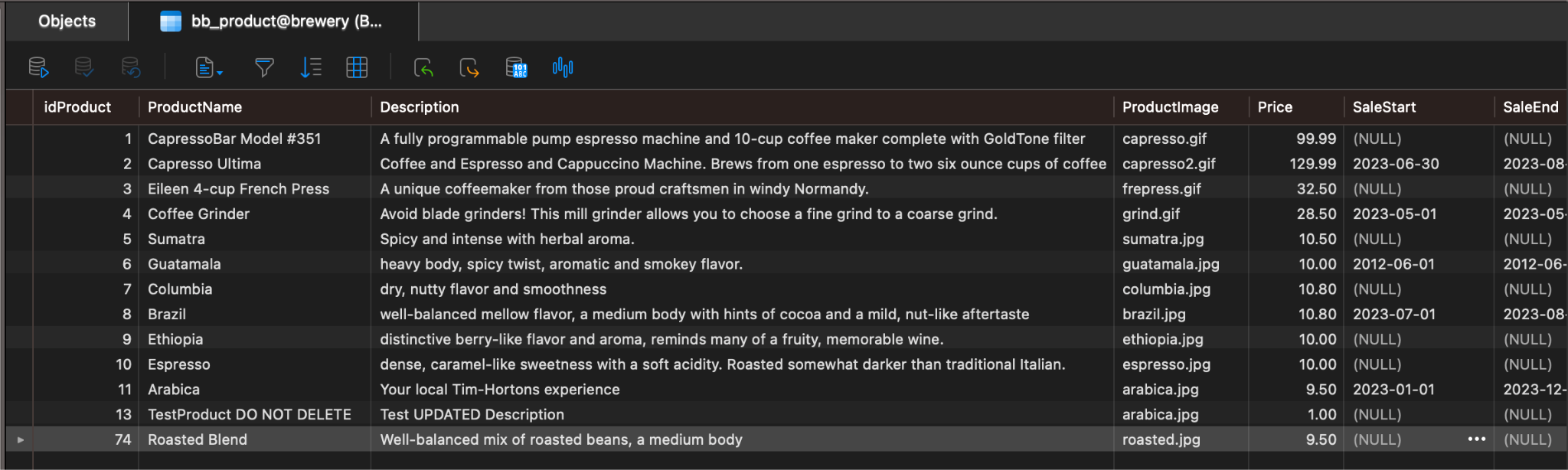
**SELECT** **COUNT**(\*) **FROM** bb\_product;



**CALL** insert\_product('Roasted Blend', 'Well-balanced mix of roasted beans, a medium body', 'roasted.jpg', **9**.**50**, **1**, **1**);

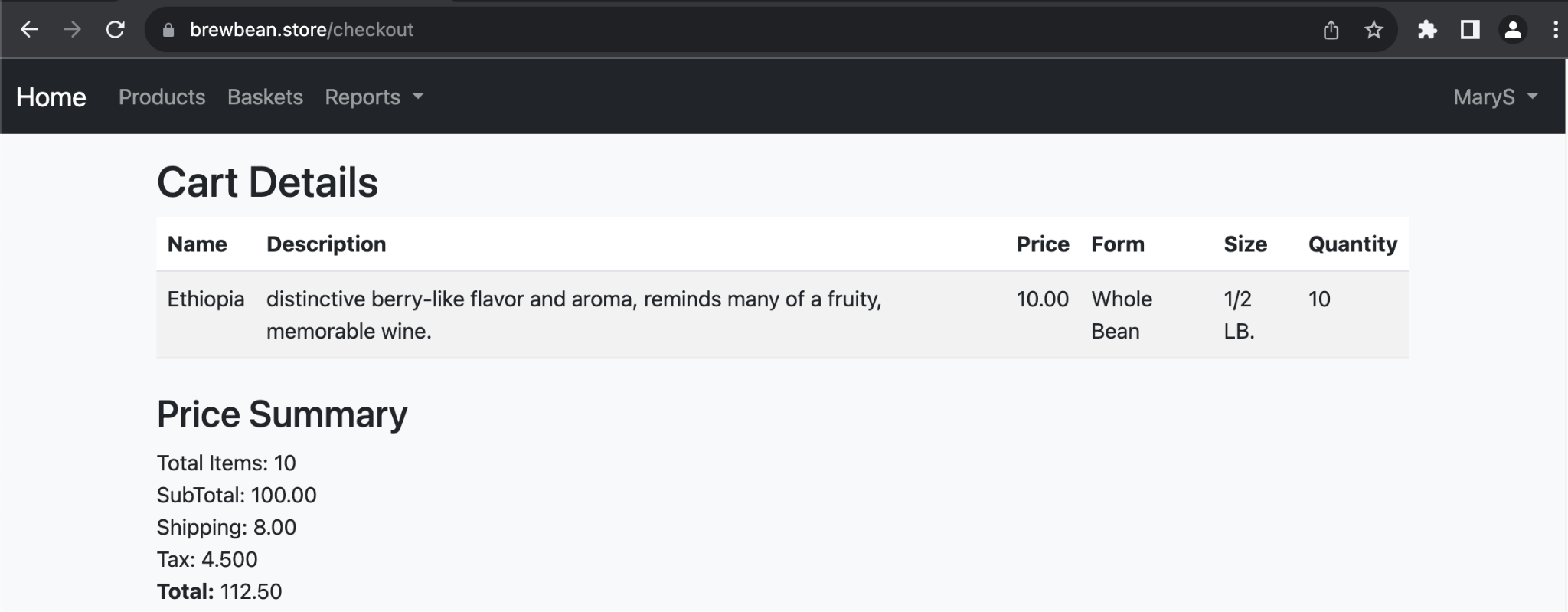
**SELECT** **COUNT**(\*) **FROM** bb\_product;





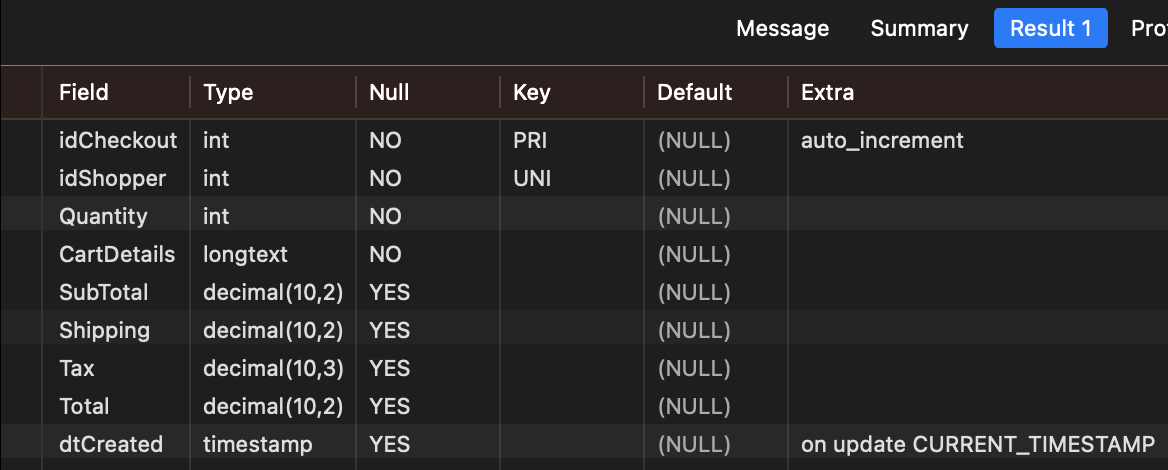
# Task 3 : Calculating the Tax on an Order.

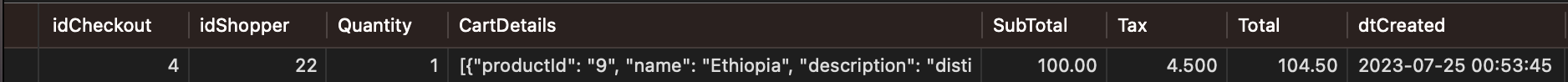
Note: In our application there is no separate tax calculator tool, instead we can simulate the exact same behavior by much more realistic means:



* 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;





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)

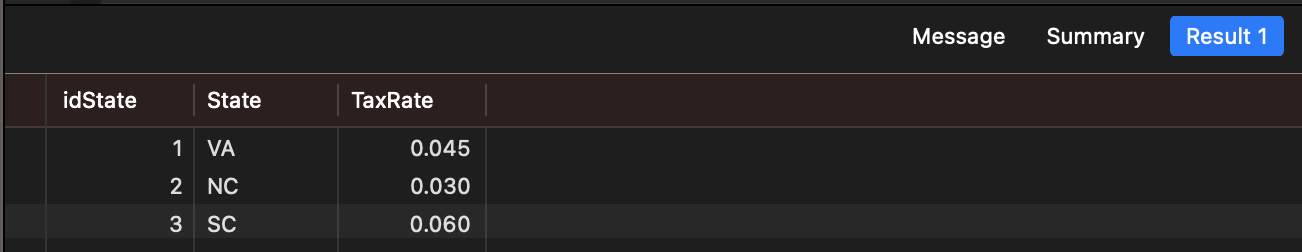
**BEGIN**

**SELECT** \* **FROM** bb\_checkout **WHERE** idShopper=inShopperId;

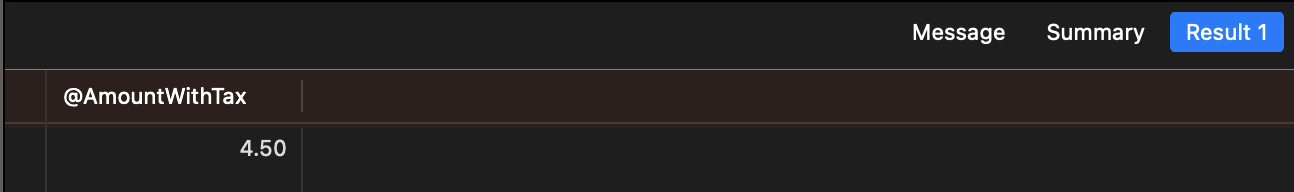
**END**

## Backend Testing (dummy tests from the problem spec):

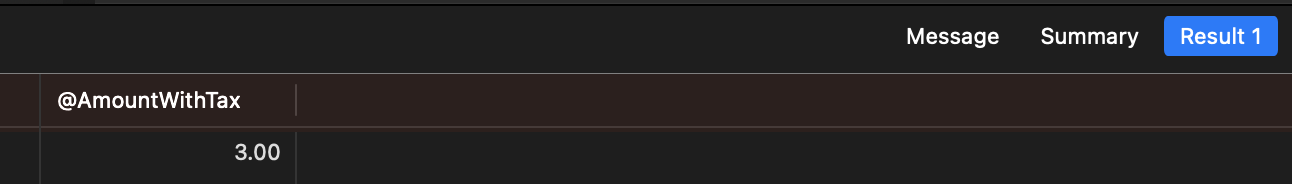
**SELECT** \* **FROM** bb\_tax

****

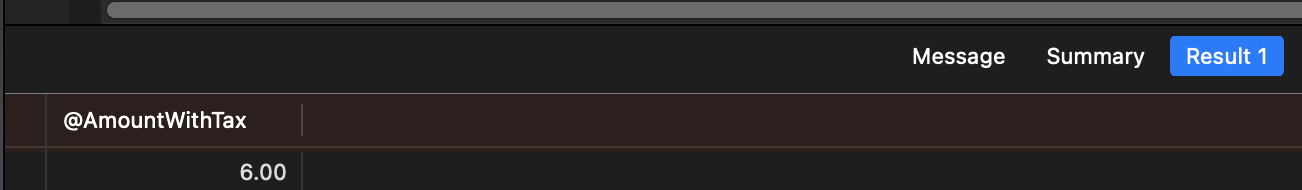
**CALL** get\_tax\_by\_state\_and\_amount('VA', **100**);



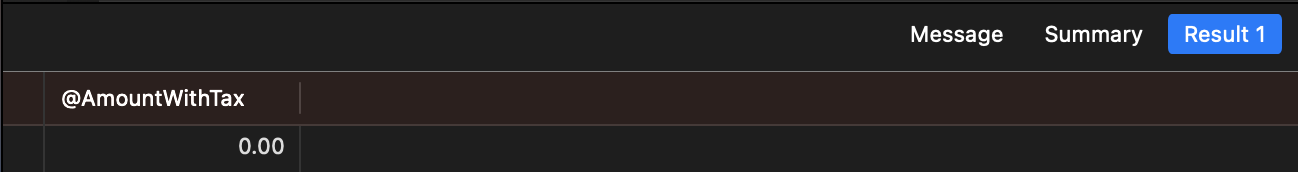
**CALL** get\_tax\_by\_state\_and\_amount('NC', **100**);



**CALL** get\_tax\_by\_state\_and\_amount('SC', **100**);



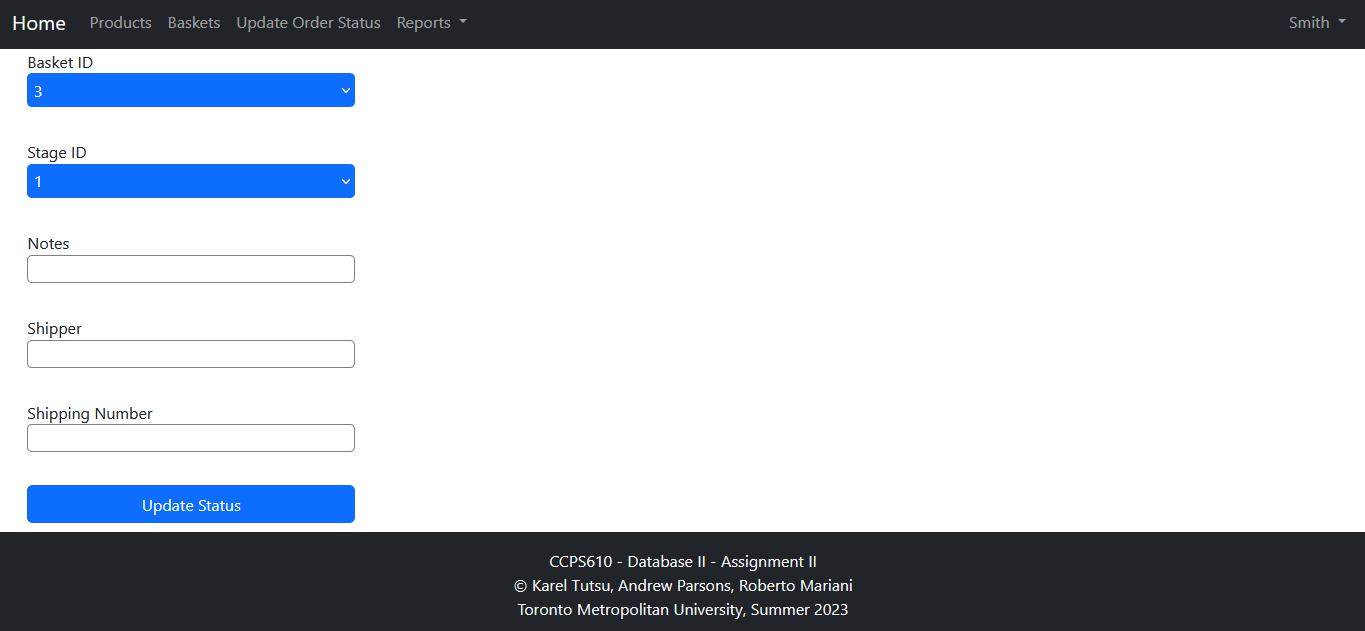
**CALL** get\_tax\_by\_state\_and\_amount('DC', **100**);



# 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:



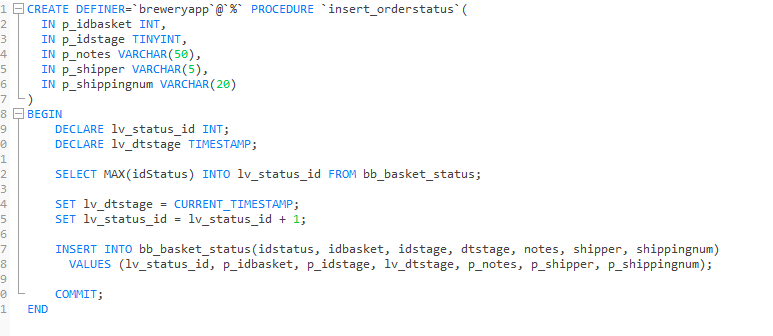
Screenshot of backend code, form data is passed to a function that calls the stored procedure in the database.



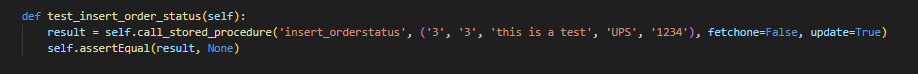
Python code to call the stored procedure:



Screenshot of stored procedure:



Test case for inserting order status:

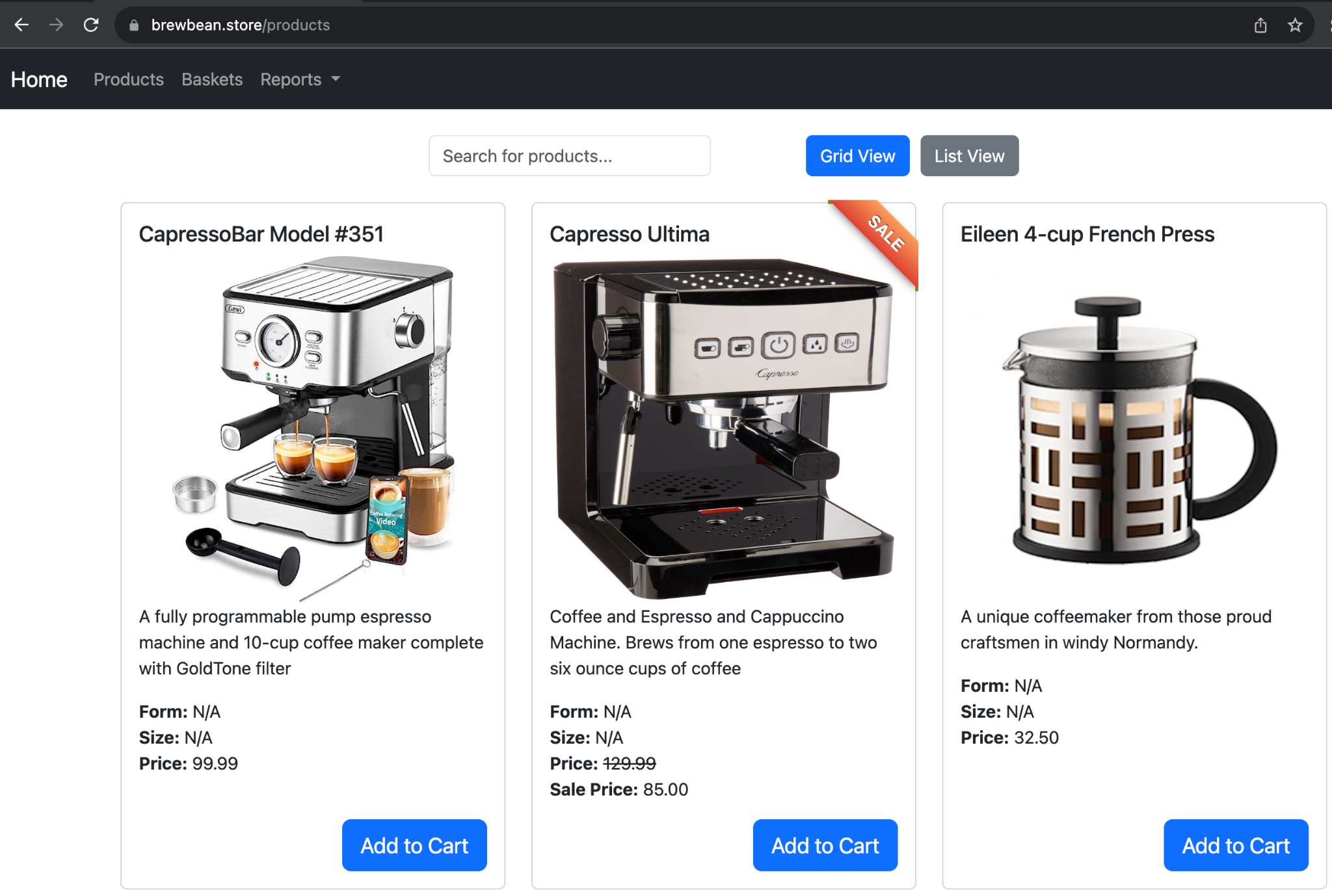


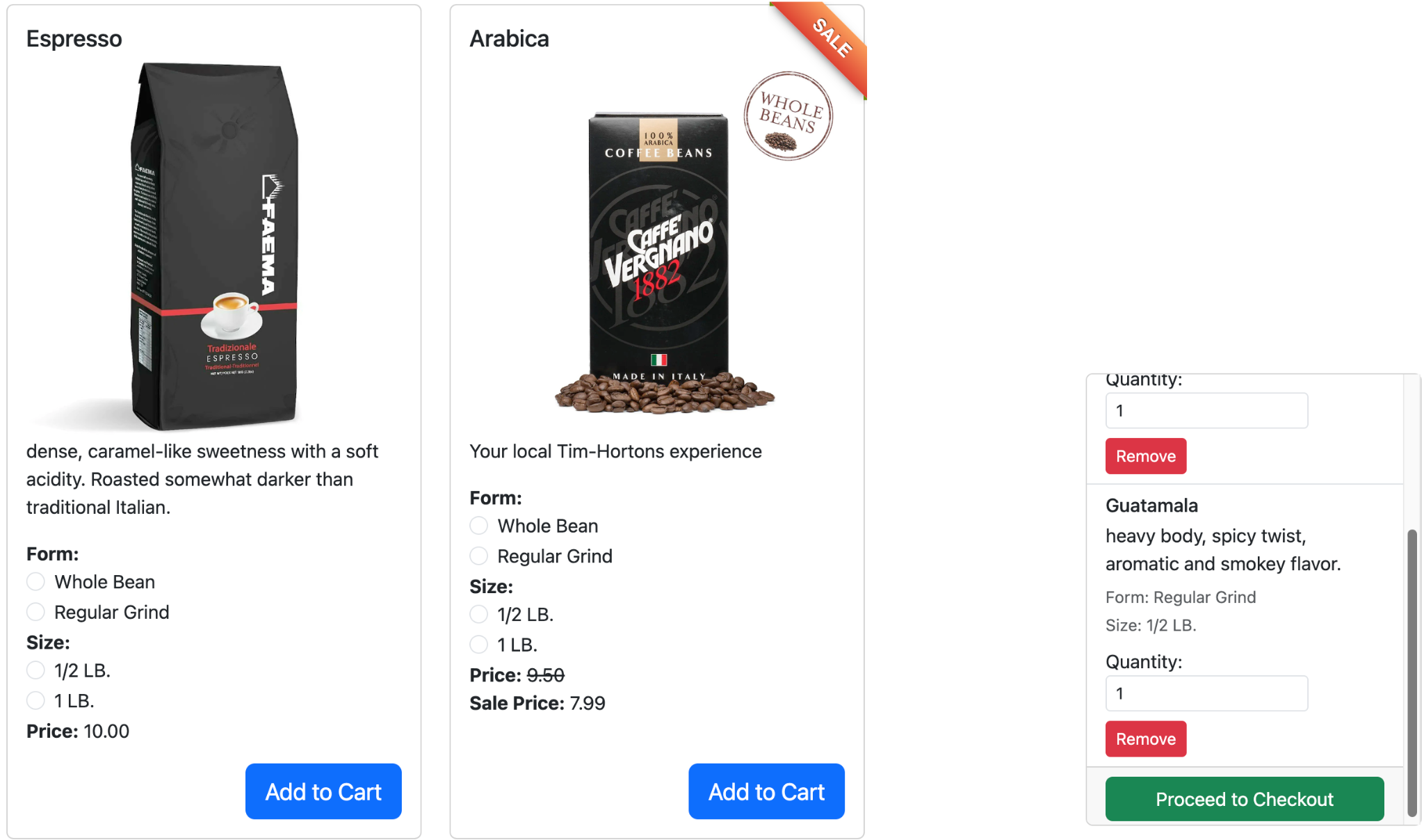
Screenshot showing test case passing:  


# 

# 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,ShipProvince,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,inShipFax,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**

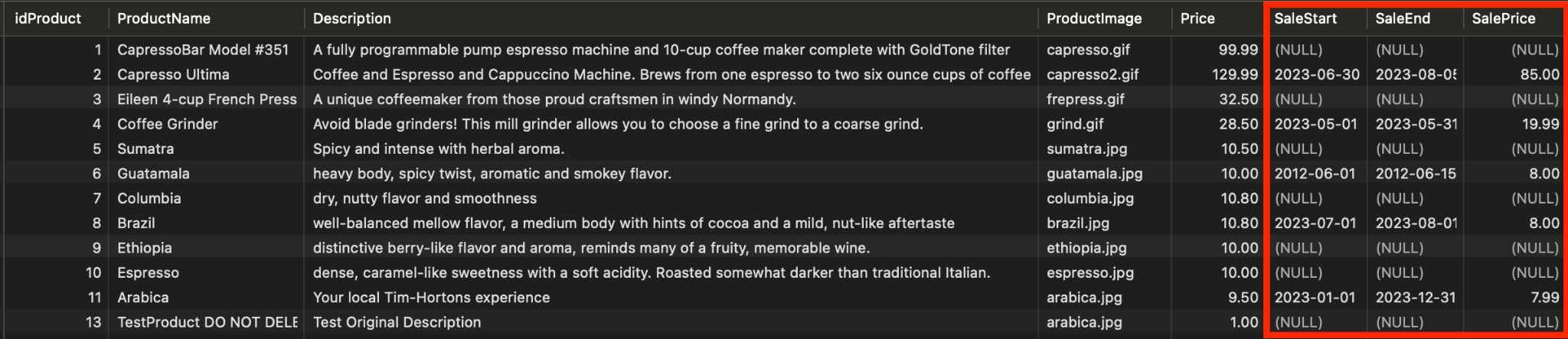
**INSERT** **INTO** bb\_basket\_item (idProduct, Price, Quantity, idBasket, option1, option2) **VALUES** (inIdProduct, inPrice, inQuantity, inIdBasket, inOption1, inOption2);

**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



### 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>

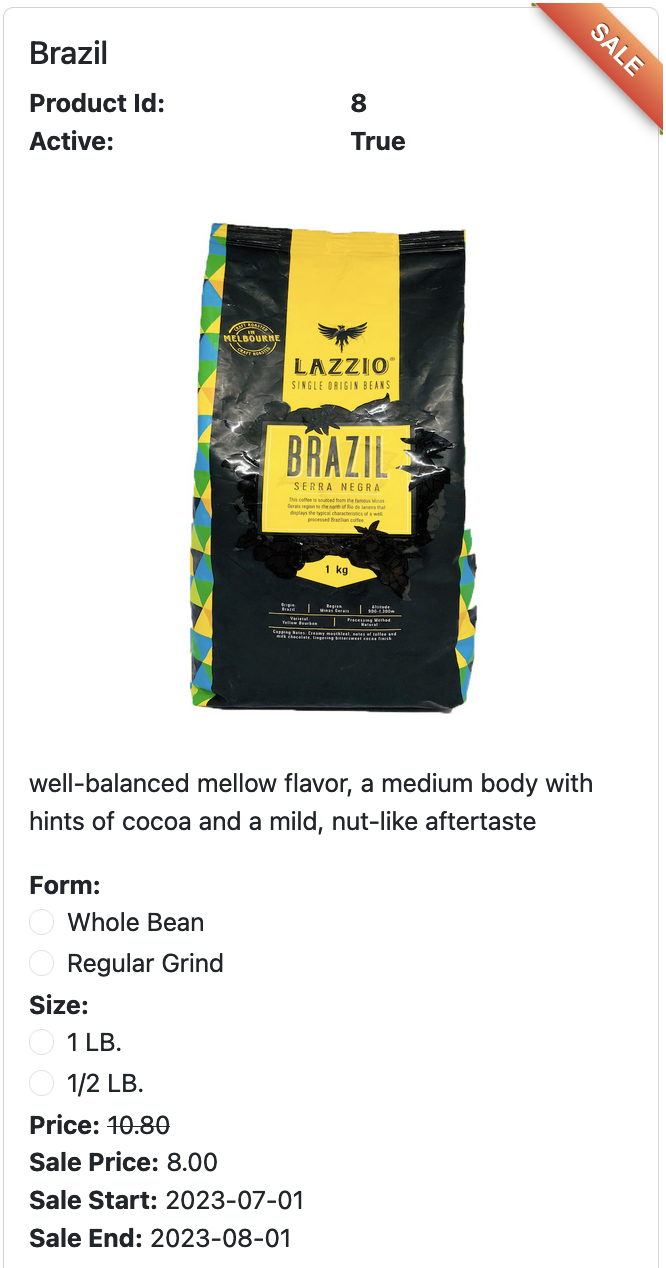
<li class="card-sale-price"><strong>Sale Price:</strong> <span id="salePrice\_{{ row.idProduct }}" class="sale-price">{{ row.SalePrice }}</span></li>

{% else %}

<li class="card-price"><strong>Price:</strong> {{ row.Price }}</li>

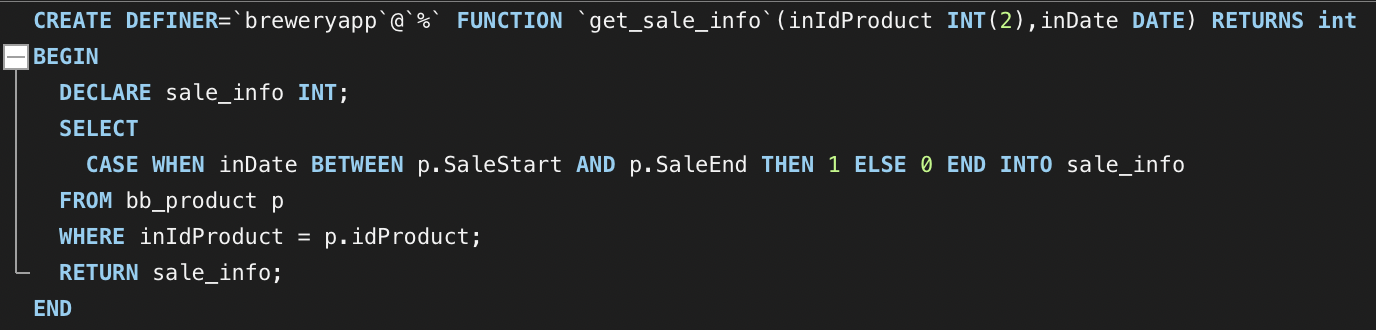
{% endif %}

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



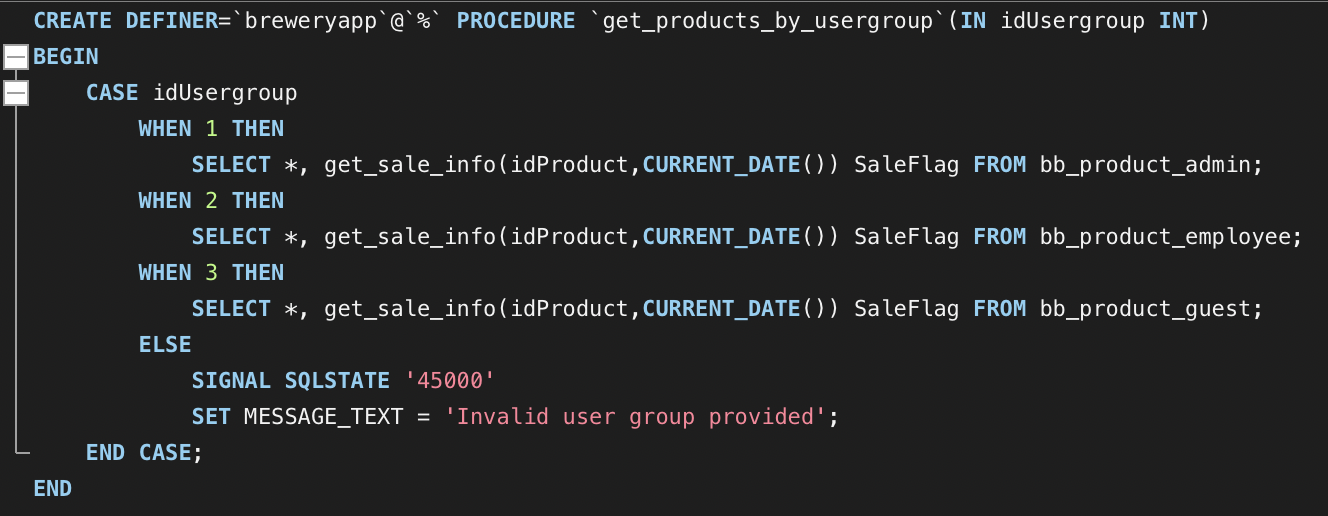
### Back-end Implementation:

Function get\_sale\_info(inIdProduct) from MySQL Database:



* 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:

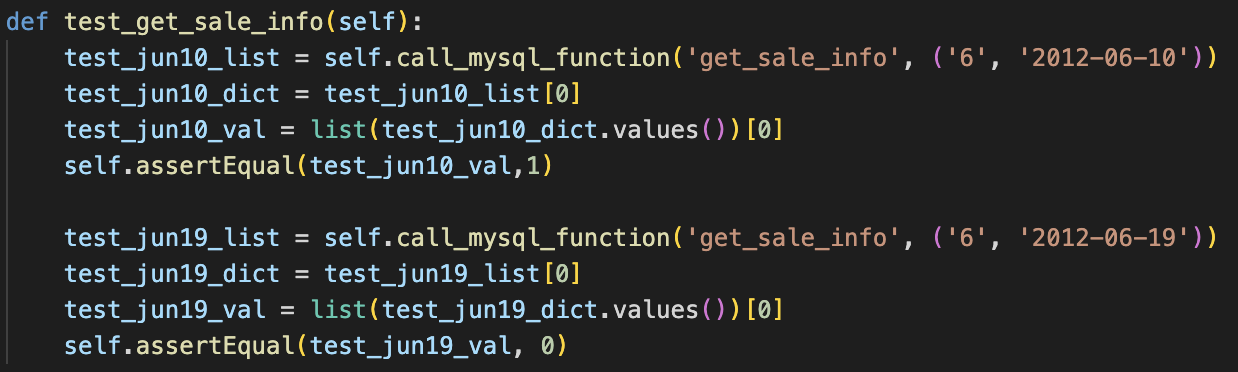
Procedure get\_products\_by\_usergroup(idUsergroup) from MySQL Database:



### 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



Relevant result

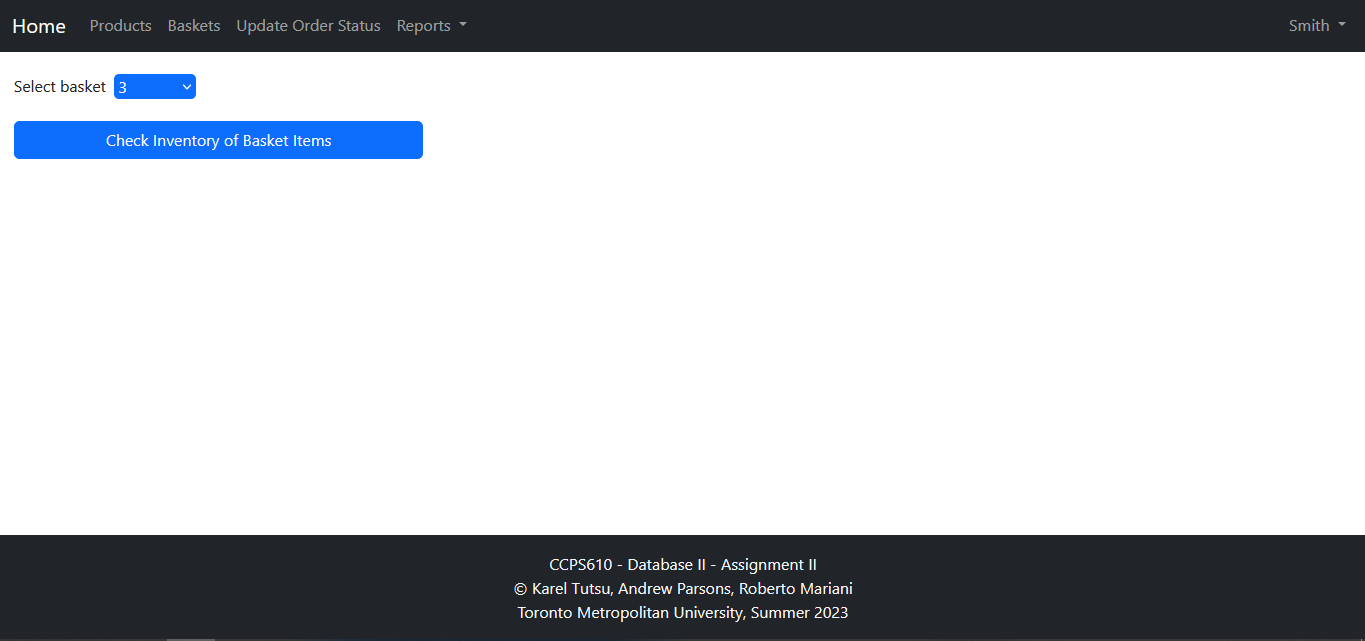


# Report 1

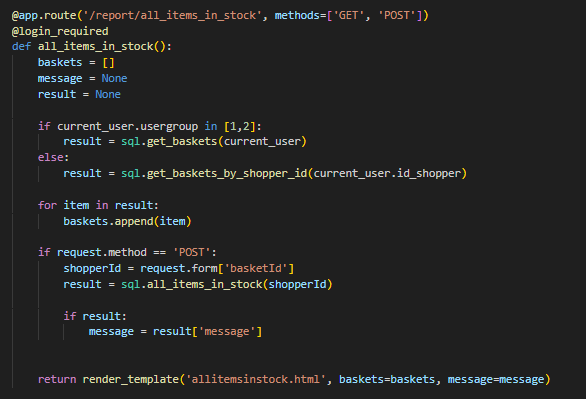
Report to show whether all items in her/his basket are in Stock or not? Using an Explicit Cursor.

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:



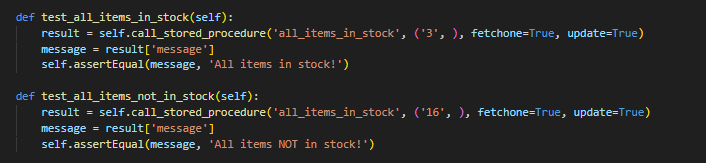
Function that calls stored procedure:



Stored procedure in the database:



Test cases to test stored procedure:



Screenshot of test cases passing:



# 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>

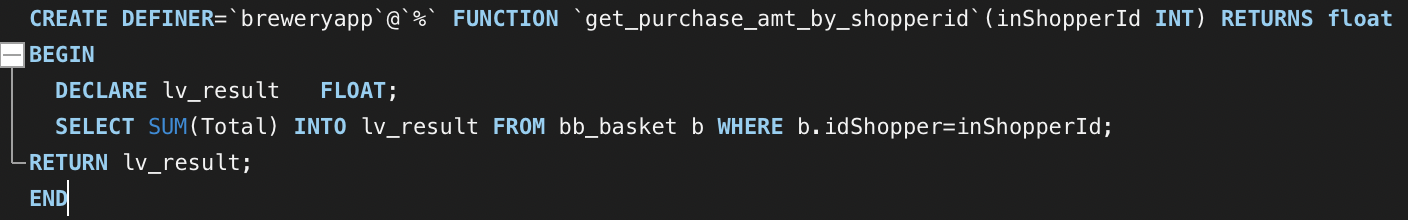
{% endif %}

Image of Reports tab in NAV bar → Purchase Report



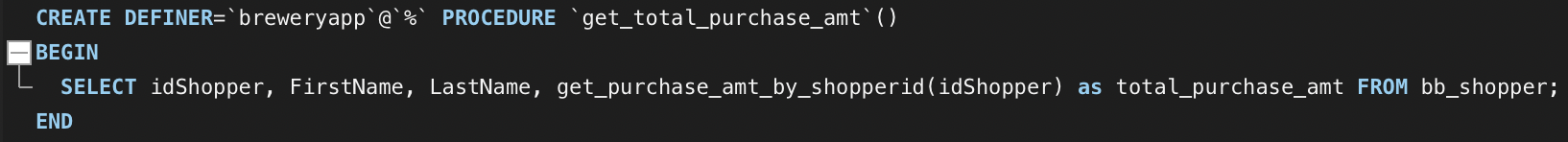
### Back-end Implementation:

Function get\_purchase\_amt\_by\_shopperid(inShopperId) from MySQL Database:



* 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)

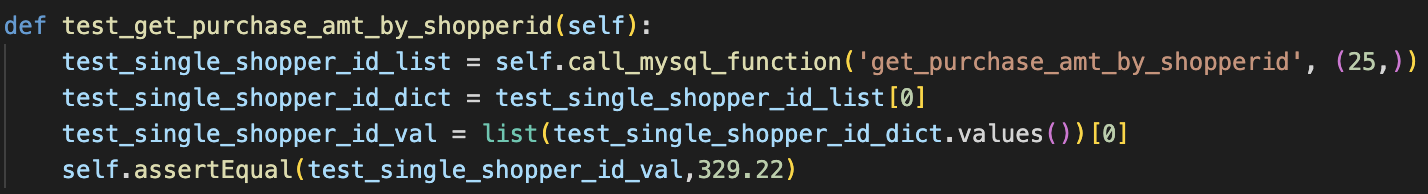
Procedure get\_total\_purchase\_amt() from MySQL Database:

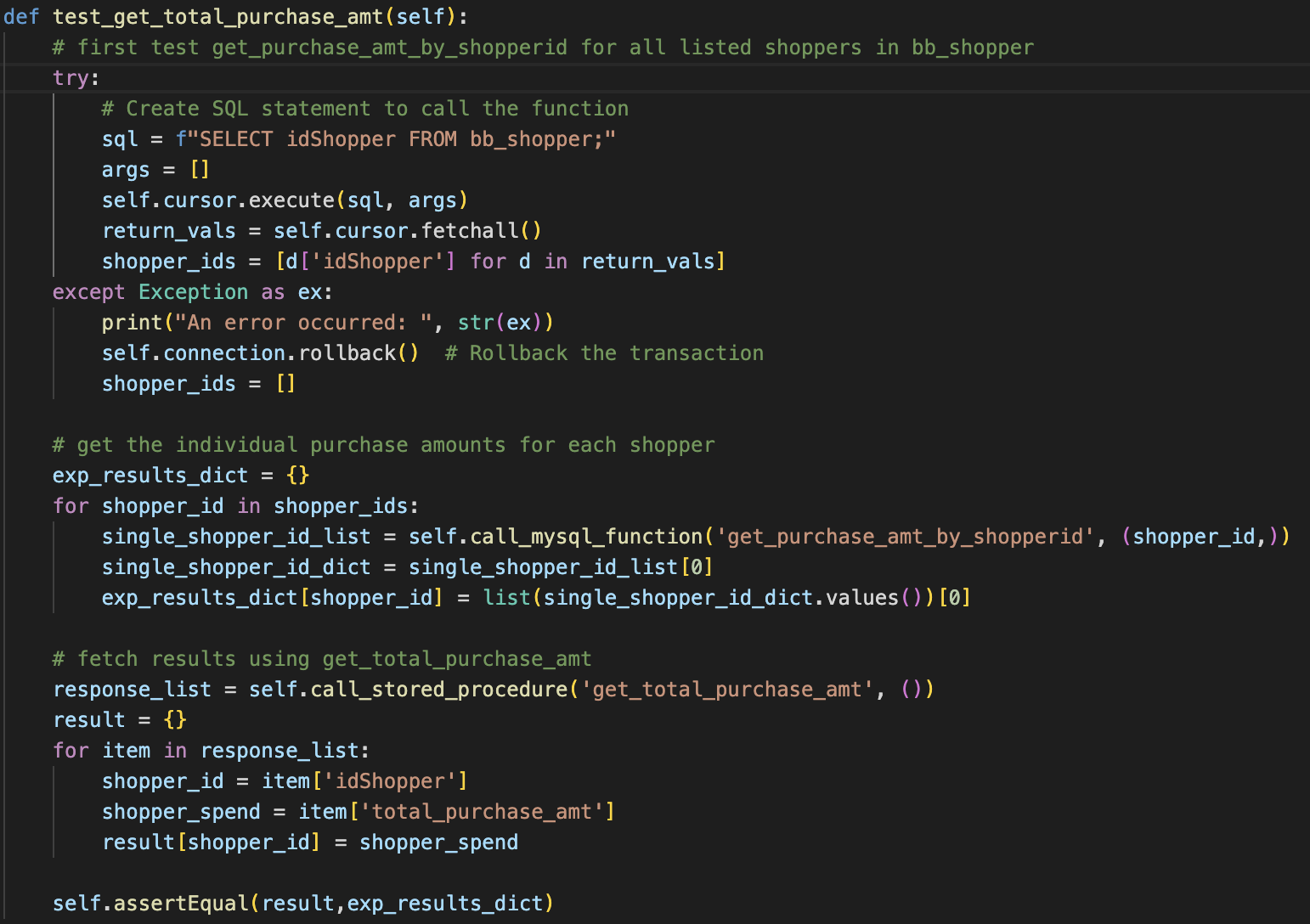


### Testing:

* 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





Relevant results

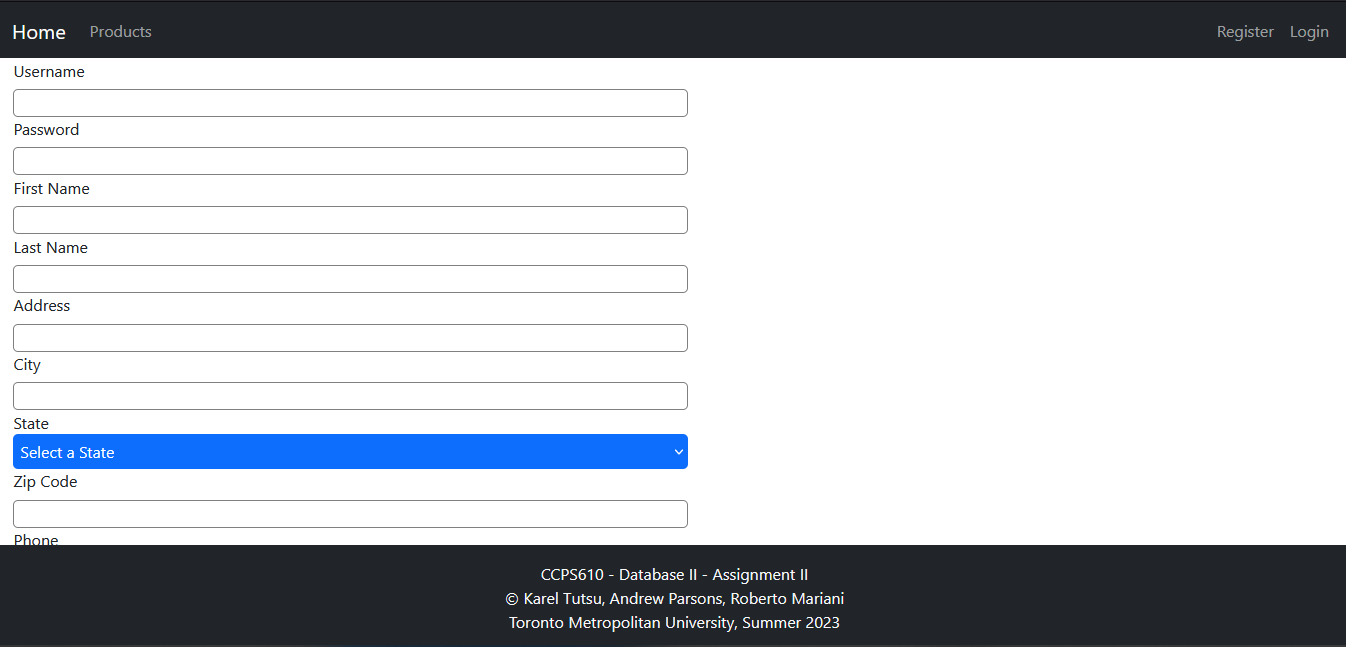




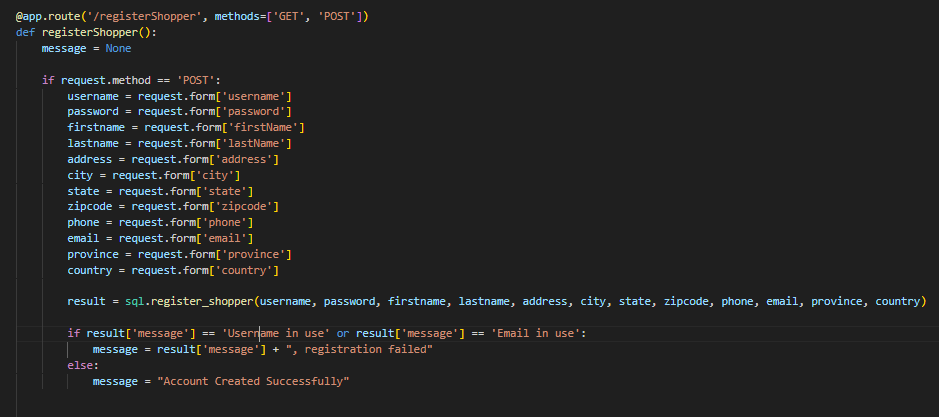
Account Registration

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.

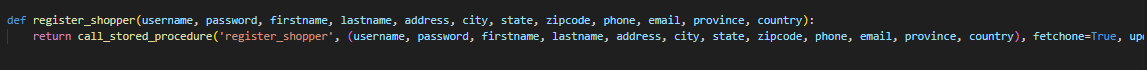
Screenshot of registration page:



Backend code for registration page:



Function to call stored procedure:



Screenshot of stored procedure:



Test cases used to test stored procedure:



Screenshot of test cases passing:

