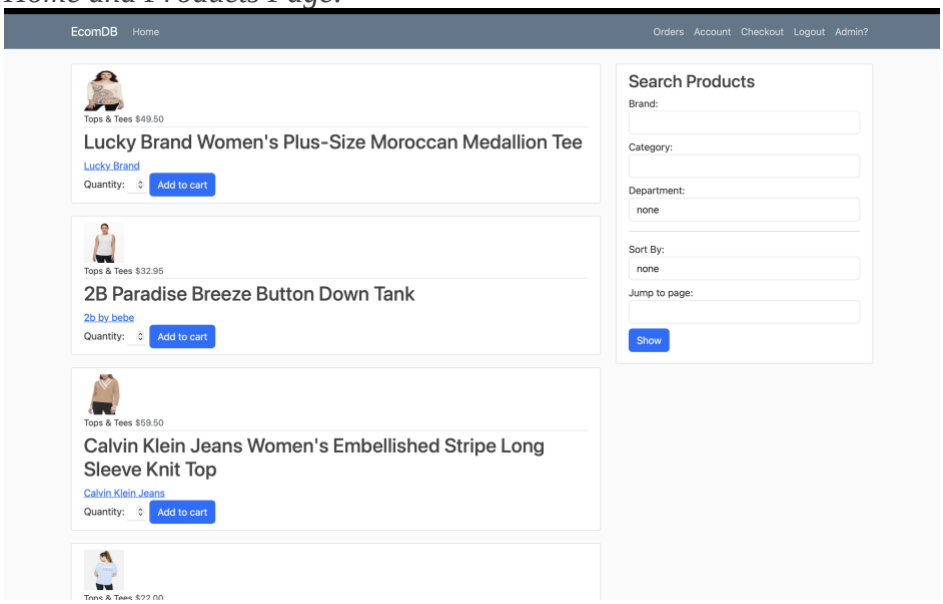
COL362 Milestone-3(Group 10)

Darshan Rakhewar - 2020CS10340
Vibhor Sengar - 2020EE30313
Nirbhay Kumar - 2020CS10365

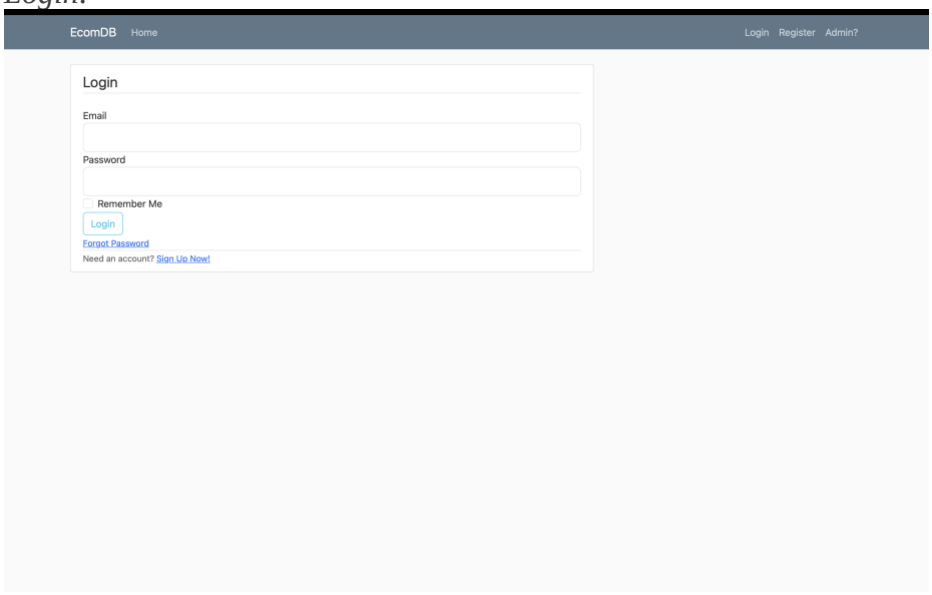
Application Front End Design

We show you the different front end pages in our application.

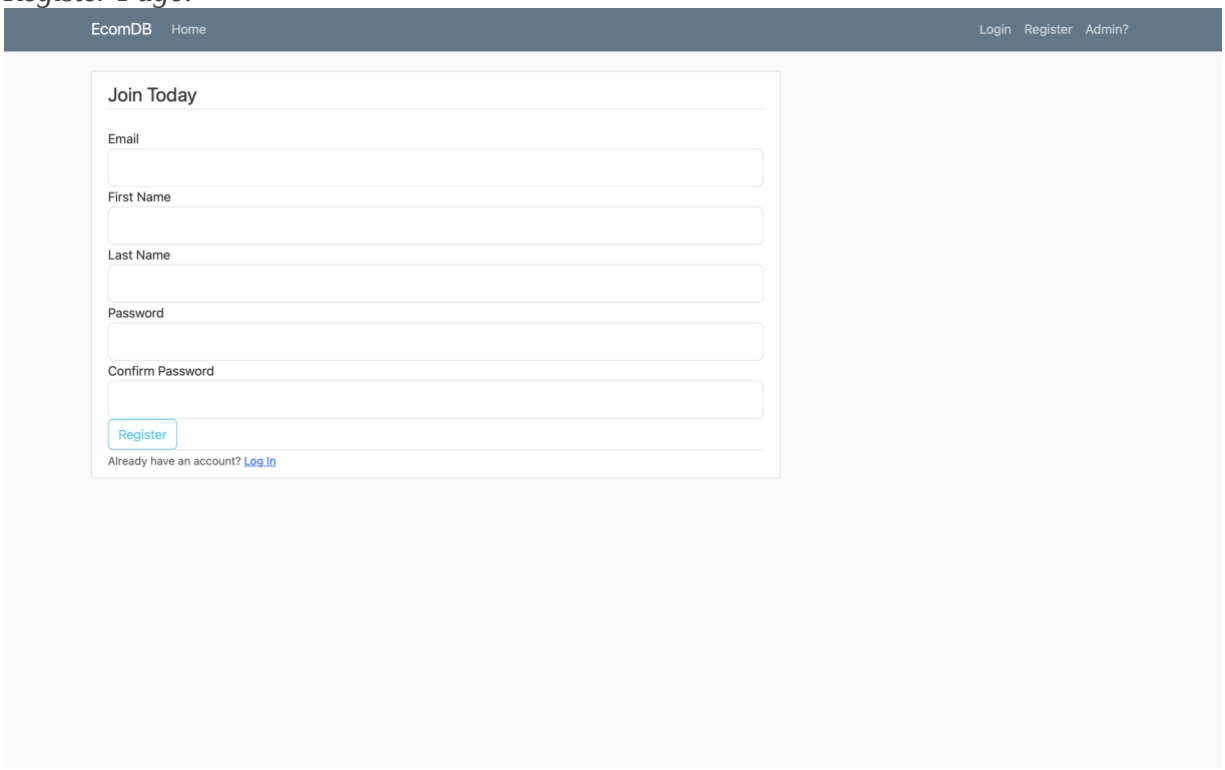
Home and Products Page:



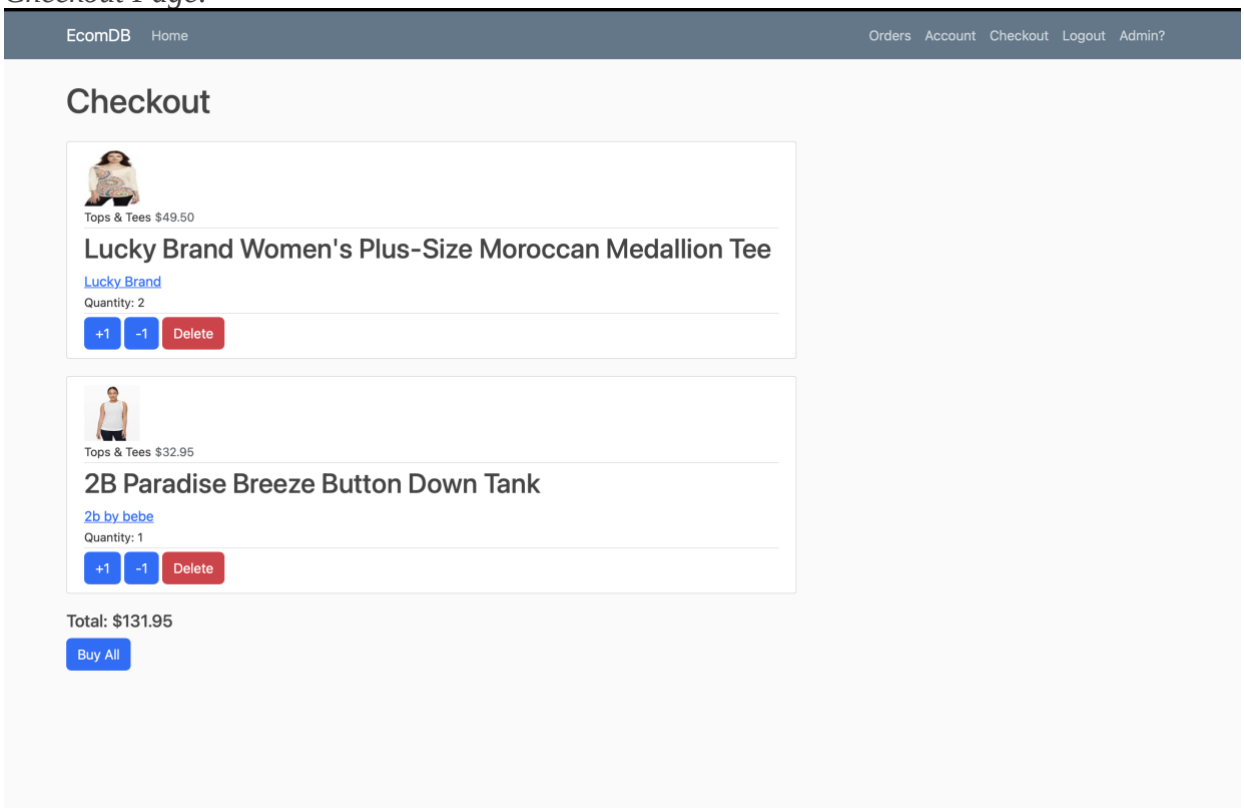
Login:



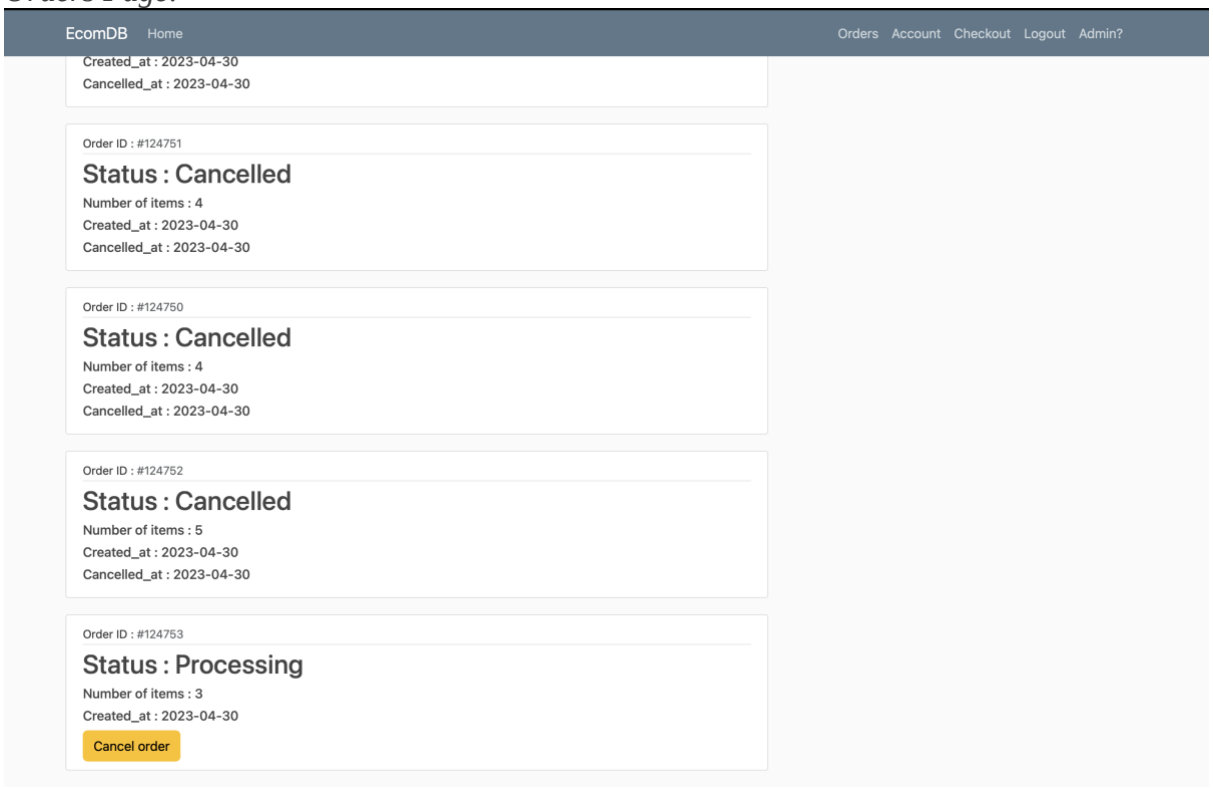
Register Page:



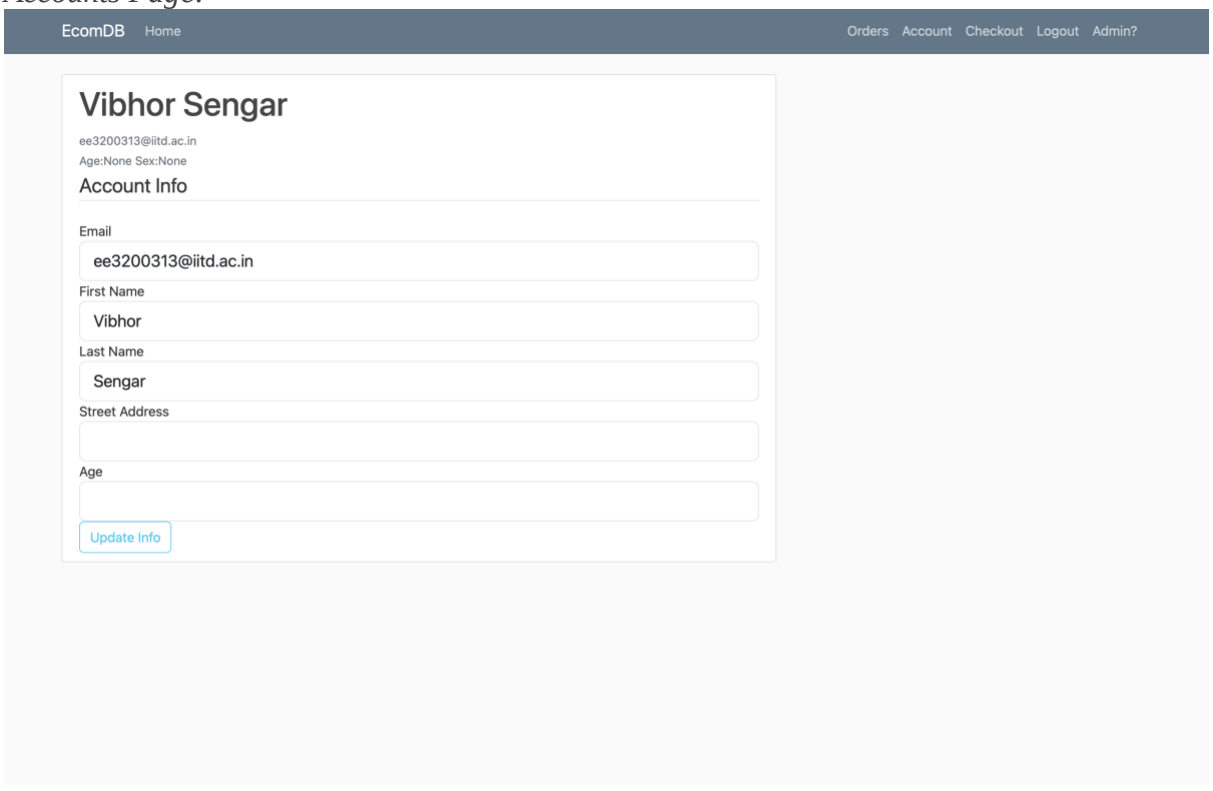
Checkout Page:



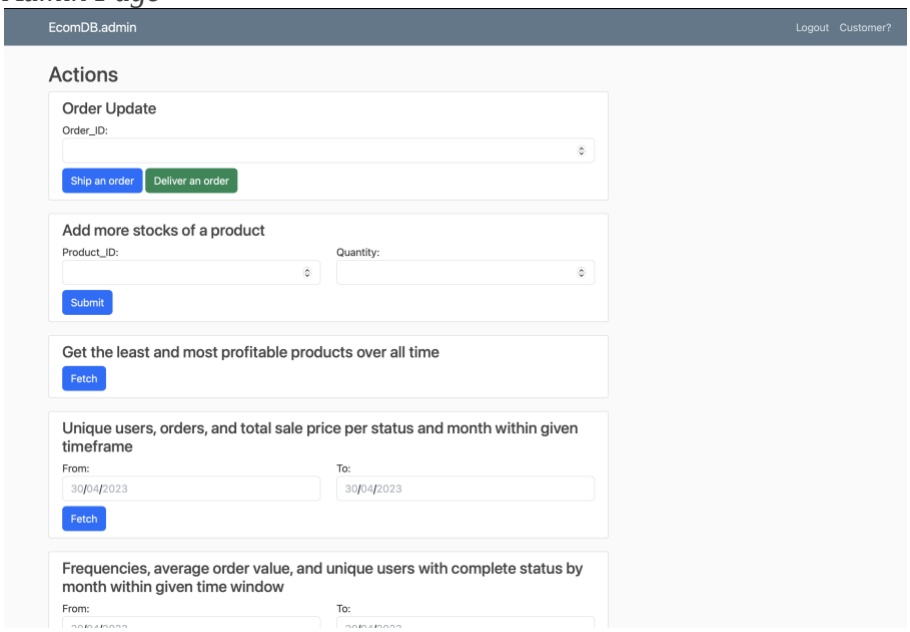
Orders Page:



Accounts Page:



Admin Page



Application Functionalities

There are 6 detailed independent functionalities in the application:

1. User Login, Register and Logout
2. User Account
3. Product Search and Sort
4. Product Checkout and Buy
5. Order Creation, Cancellation and Return
6. Admin Page

User Login:

1. The **/login** route handles user authentication. It checks if the user is already logged in by checking if the current user's session exists. If the user is already logged in, they are redirected to the home page. If the user is not logged in, the route connects to the database and checks the user's email and password against the stored credentials. If the credentials are valid, the user is logged in, and their session is created. The user is redirected to the home page. If the credentials are invalid, an error message is flashed, and the user is redirected to the home page.
2. The **/logout** route logs the user out of the application by deleting their session. The user is then redirected to the home page.
3. The **/register** route handles user registration. If the user is already logged in, they are redirected to the home page. If the user is not logged in, the route connects to the database and checks if the user's email already exists in the database. If the email already exists, an error message is flashed, and the user is redirected to the registration page. If the email does not exist, the user's details are inserted into the database, and a success message is flashed. The user is then redirected to the login page.
4. Overall, these routes provide a secure and straightforward way for users to register, log in, and log out of the web application.

User Account:

1. The **/account** route handles the user's account information and updates their account details. This route requires the user to be logged in. If the user is not logged in, they are redirected to the login page.
2. The route connects to the database and fetches the user's details to pre-populate the form for editing their account details. The user can then make changes to their details, including their first name, last name, email, age, and street address.
3. If the user submits the form with valid data, the route checks whether the email entered already exists in the database, and if it does, the user is redirected to the account page with an error message. Otherwise, the user's account details are updated in the database, and their session is also updated to reflect the changes. A success message is then flashed, and the user is redirected back to the account page.
4. If the form is not submitted, the route simply displays the user's account information for editing.
5. Overall, this route provides users with a way to manage their account details, update their information, and view their account information. It ensures that only logged-in users can access this information and makes it easy for users to update their details.

Product Search and Sort:

1. The **products** function is decorated with several routes that can handle different combinations of query parameters in the URL.
2. For example, the **/products/page:<page>/brand:<brand>** route can handle requests where the page and brand parameters are specified in the URL. The **products** function takes these parameters as arguments and uses them to query a database for product information. It also handles POST requests where users can search for products or add products to their cart.
3. The function first checks if the request is a POST request, and if so, it checks for the **search** and **add_to_cart** form data that is submitted. If the **search** form data is present, the function extracts the search parameters from the form data, sets the appropriate variables, and redirects to the **products** function with the new parameters. If the **add_to_cart** form data is present, the function adds the specified product and quantity to the user's cart if they are logged in, and displays an appropriate message if not.
4. The function then checks if an **order** parameter is present in the URL, and if so, queries the database for products ordered by the specified parameter. It also sets flash messages to notify the user of the current search and sort criteria.
5. The function finally queries the database for the products using the specified parameters and returns the results in an HTML template that displays the product information, as well as pagination links for navigating through the results.
6. Overall, this function is a crucial part of a Flask application that handles requests for product information and implements various search and sort criteria to help users find the products they are looking for.

Product Checkout and Buy:

This is a Flask route with the endpoint **/checkout** that handles the checkout process of a user's shopping cart. Here are some features of this route:

1. This route accepts both GET and POST requests.
2. It checks if the user is logged in by verifying the presence of **current_user** in the session object. If the user is not logged in, the route redirects to the login page.
3. If the request method is POST, the route reads the data submitted from the checkout form, which includes the product id, quantity to add or remove, and delete command if applicable.
4. It updates the shopping cart data in the session object based on the user's input.
5. If the shopping cart is empty, the route redirects to the home page and shows a warning message.
6. If the shopping cart is not empty, the route retrieves product details from the database using the product id and calculates the total cost of all items in the cart.
7. It renders the **checkout.html** template and passes the product details and total cost as parameters, along with the current user's information for displaying on the template.
8. There is another Flask route with the endpoint **/buyall**, which is called when the user clicks on the "Buy All" button on the checkout page.
9. This route checks if the user is logged in and retrieves the product stock and quantity from the shopping cart for each item.
10. If any order item quantity is greater than the product stock, the route shows a warning message and redirects back to the checkout page.
11. Otherwise, the route starts a database transaction and creates a new order with the user ID, order status, creation date, and number of items.
12. It updates the product stock in the database by subtracting the purchased quantity.
13. The route also inserts new order item records in the database for each purchased item and updates the inventory item sold date to mark it as sold.
14. Once the database transaction is committed successfully, the route shows a success message, clears the shopping cart data in the session object, and redirects to the user's order history page.

Order Creation, Cancellation and Return

1. This Flask application route has two functions, "cancel" and "returns", which both handle order cancellations and returns, respectively. The "orders" function displays a list of orders associated with the currently logged-in user.
2. The "cancel" function allows the user to cancel an order by changing the order's status to "Cancelled" and setting the returned date to the current date. It also updates the status of the order's items to "Cancelled" and sets their returned date to the current date. The function then updates the sold_at value of the inventory items associated with the order's items to null and increases the stock of the products associated with those inventory items by 1. Finally, it displays a success message to the user and redirects them to their order list.
3. The "returns" function allows the user to return a completed order by changing the order's status to "Returned" and setting the returned date to the current date. It also updates the status of the order's items to "Returned" and sets their returned date to the current date. The function then updates the sold_at value of the inventory items associated with the order's items to null and increases the stock of the products associated with those inventory items by 1. Finally, it displays a success message to the user and redirects them to their order list.

Admin Login and Admin Page

1. The function **adminhome** performs several operations, depending on the value of the **request.method** variable. If it is a GET request, the function renders a template 'adminhome.html' using **render_template** function from Flask. If it is a POST request, the function checks the value of the **id** key in the submitted form data to determine the operation to be performed.
2. For **id=1**, the function checks if the submitted form data contains an **orderid** field. If yes, it queries the database to fetch the order details using the given order ID. If the order is not found, it flashes a message 'Order does not exist'. If the order is found, it checks whether the **ship** or **deliver** field is set in the submitted form data. If the **ship** field is set and the order status is 'Processing', it updates the order and order item status to 'Shipped' and flashes a success message. If the **deliver** field is set and the order status is 'Shipped', it updates the order and order item status to 'Complete' and flashes a success message.
3. For **id=2**, the function checks if the submitted form data contains **productid** and **qty** fields. If yes, it queries the database to fetch the product details using the given product ID. If the product is not found, it flashes a message 'Product does not exist'. If the product is found, it inserts inventory items in the database equal to the quantity given in the form data and updates the product stock accordingly. It then flashes a success message.
4. For **id=3**, the function executes a complex SQL query to fetch the 5 most and least profitable products and ranks them based on profit. It then renders a template 'table.html' using **render_template** function from Flask.
5. For **id=4**, the function checks if the submitted form data contains **date1** and **date2** fields. If yes, it queries the database to fetch various order-related statistics such as the number of orders, total revenue, and average order value for each month between the given dates. It then renders a template 'stats.html' using **render_template** function from Flask.
6. If none of the above conditions are satisfied, the function renders the 'adminhome.html' template with the current user's information.

Major Transactions/Queries:

Note flask connection object has in-built commit function so there was no need to begin and commit transactions in the queries themselves.

Login:

```
SELECT * FROM users WHERE email=%s
```

This is used to retrieve user of the email id entered

Register:

```
INSERT INTO users(id,first_name,last_name,email,password) VALUES((select max(id) from users) +
1,%s,%s,%s,%s)",[form.first_name.data,form.last_name.data,form.email.data,form.password.data]
```

This is use to insert a new user once the email and other fields have been validated and we are certain that the email doesn't already exist.

Account:

```
UPDATE users SET first_name=%s,last_name=%s,email=%s WHERE id=%s
",[form.first_name.data,form.last_name.data,form.email.data,session.get('current_user')[0]]
```

Used to update account info

Products:

The add to cart button adds the specific product to cart using the session object. We simply add the product number (returned using a hidden form field from the template) in the cart.

```
if order:
    if department:
        if brand and category:
            flash('Showing products of brand '+brand+' and category '+category+' in department '+department+' ordered by '+order+ 'price','success')
            curr.execute("SELECT * FROM products WHERE brand=%s AND category=%s AND department=%s ORDER BY retail_price "+order+ " LIMIT 10 OFFSET %s",[brand, category, department, (int(page)-1)
        elif brand:
            flash('Showing products of brand '+brand+' in department '+department+' ordered by '+order+ 'price','success')
            curr.execute("SELECT * FROM products WHERE brand=%s AND department=%s ORDER BY retail_price "+order+ " LIMIT 10 OFFSET %s",[brand, department, (int(page)-1)*10])
        elif category:
            flash('Showing products of category '+category+' in department '+department+' ordered by '+order+ ' price','success')
            curr.execute("SELECT * FROM products WHERE category=%s AND department=%s ORDER BY retail_price "+order+ " LIMIT 10 OFFSET %s",[category, department, (int(page)-1)*10])
        else:
            flash('Showing products in department '+department+' ordered by '+order+ ' price','success')
            curr.execute("fSELECT * FROM products WHERE department=%s ORDER BY retail_price "+order+ " LIMIT 10 OFFSET %s",[department, (int(page)-1)*10])
```

This is a simple query that returns the tuples which satisfy the specific brand, category, order and page number. The conditional statement is to handle the cases where either one of the above is not specified.

Checkout:

In Checkout we iterate over the cart and create a product and quantity tuple list to be passed to the template.

```
"select * from products where id="+str(item)
```

We use this simple query to retrieve the product info to be passed.

Orders:

1.
```
select stock from products where id=<key>
```

   This query retrieves the stock count of a product with the given key from the 'products' table.

2.
```
select max(id) from order_items
```

   This query retrieves the maximum id from the 'order_items' table.

3.
```
select max(order_id) from orders
```

   This query retrieves the maximum order_id from the 'orders' table.

4. `INSERT into orders (order_id, user_id, status, created_at, num_of_item) values (<max_order_id+1>, <user_id>, 'Processing', current_date, <num_of_items>):`

This query inserts a new row into the 'orders' table with the given data. The order_id is set to the maximum order_id plus one, the user_id is set to the current user's id, the status is set to 'Processing', the created_at date is set to the current date and the num_of_items is set to the total number of items in the user's cart.

5. `update products set stock=stock-<session['cart'][key]> where id=<key>:`

This query updates the stock count of a product with the given key in the 'products' table by subtracting the number of items in the user's cart for that product.

6. `insert into order_items (id, order_id, inventory_item_id, status, created_at, sale_price) select <max_order_items_id+row_number() over (order by id)>, <max_order_id+1>, id, 'Processing', current_date, cost from inventory_items where product_id=<key> and sold_at is null limit <session['cart'][key]>:`

This query inserts multiple rows into the 'order_items' table for a given product based on the number of items in the user's cart for that product. The id is set to the maximum order_items_id plus the row number, the order_id is set to the maximum order_id plus one, the inventory_item_id is set to the id of the inventory item for the given product, the status is set to 'Processing', the created_at date is set to the current date and the sale_price is set to the cost of the inventory item. The 'where' clause ensures that only inventory items that have not been sold yet are selected, and the 'limit' clause limits the number of rows inserted to the number of items in the user's cart for that product.

7. `update inventory_items set sold_at=current_date from (select id from inventory_items where product_id=<key> and sold_at is null limit <session['cart'][key]>) as t where t.id=inventory_items.id:`

This query updates the 'sold_at' column in the 'inventory_items' table for a given product based on the number of items in the user's cart for that product. Only inventory items that have not been sold yet and are in the list of items being purchased are updated. The 'sold_at' column is set to the current date.

Cancel and Return

Since there are a lot of queries in these routes we shall write the practical explanation of each. The exact queries can be seen in the code itself.

For the first route function, "/cancel/orderid:<orderid>", the SQL queries executed are:

1. Update the status of the order to "Cancelled" and set the returned_at field to the current date in the orders table where the order_id is equal to the given orderid and the status is either "Processing" or "Shipped".
2. Update the status of the corresponding order items to "Cancelled" and set the returned_at field to the current date in the order_items table where the order_id is equal to the given orderid and the status is either "Processing" or "Shipped".
3. Select the inventory_item_id from the order_items table where the order_id is equal to the given orderid.
4. For each inventory_item_id selected, update the sold_at field to null in the inventory_items table where the id is equal to the current inventory_item_id.
5. Select the product_id from the inventory_items table where the id is equal to the current inventory_item_id.
6. Increase the stock of the product in the products table by 1 where the id is equal to the current product_id.

The overall practical meaning of these queries is to cancel an order and update the status of the corresponding order items, inventory items, and products accordingly.

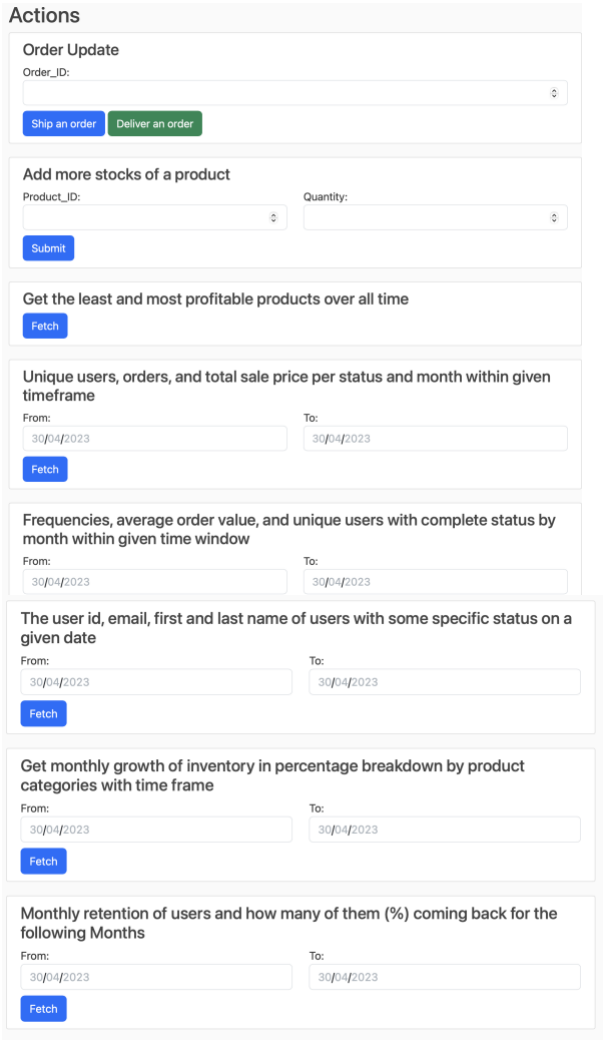
For the second route function, "/returns/orderid:<orderid>", the SQL queries executed are:

1. Update the status of the order to "Returned" and set the returned_at field to the current date in the orders table where the order_id is equal to the given orderid and the status is "Complete".
2. Update the status of the corresponding order items to "Returned" and set the returned_at field to the current date in the order_items table where the order_id is equal to the given orderid and the status is "Complete".
3. Select the inventory_item_id from the order_items table where the order_id is equal to the given orderid.
4. For each inventory_item_id selected, update the sold_at field to null in the inventory_items table where the id is equal to the current inventory_item_id.
5. Select the product_id from the inventory_items table where the id is equal to the current inventory_item_id.
6. Increase the stock of the product in the products table by 1 where the id is equal to the current product_id.

The overall practical meaning of these queries is to mark an order as returned and update the status of the corresponding order items, inventory items, and products accordingly.

Admin Analysis

This page consists of 8 analysis/admin queries whose meaning is explicitly mentioned in the frontend. The sql queries are mainly read queries with specific conditions as can be seen in the code.



Demonstration Scenario

1. Simulate two different users on two IP
2. User logs into the website with valid credentials.
3. User is redirected to the homepage where they can see a list of products available for purchase.
4. User can change his account
5. User can add products to their cart and proceed to checkout when they are ready.
6. Once the user places an order, they are redirected to a confirmation page where they can see the details of their order.
7. The admin logs into the website with valid credentials.
8. The admin is redirected to the admin dashboard where they can see a list of orders that have been placed.
9. The admin can view the details of each order, update the status of an order, and add new products to the inventory.
10. The admin can also view reports on the most and least profitable products, as well as sales data for a given time period.
11. The admin logs out of the website.