

School of Computing, Engineering and Mathematics (CEM)

Faculty of Engineering, Environment and Computing (EEC)

**5001CEM SOFTWARE ENGINEERING** | 2122

**PROJECT REPORT**

**NAME:** Gurvinder Singh Nagra

**SID: 10248274**

1. **CODE PURPOSE**

Allows a user to login to a storefront with 10 or more books and add them to a shopping cart and checkout. For admins they can also work with stock.

1. **CODE LOCATION**

The location for the repository for the bookshop is at <https://github.coventry.ac.uk/5001CEM-2122/5001BookShopGurviner> and the location for the bank project is https://github.coventry.ac.uk/5001CEM-2122/5001Bankapp

1. **CODE INSTALLATION**
2. Go onto github.coventry.ac.uk and find the bookstore repository <https://github.coventry.ac.uk/5001CEM-2122/5001BookShopGurviner> and and bank repository <https://github.coventry.ac.uk/5001CEM-2122/5001Bankapp>
3. Unzip and set up a new python project in Codio, called shopping\_cart. Then follow the instructions below, adapting the filenames.

Open a terminal and do **sudo apt update**

cd into your directory which will beand do **sudo apt-get install python3-venv**

Then do **python3 -m venv venv**

Then activate the virtual environment: **. venv/bin/activate**

This is confirmed by the change to the prompt, which is now

**(venv) codio@emotion-theory:~/workspace/bookshop$** (of course, your box will have a different name).

Then do **pip install Flask**

We need a database, so also do

**pip install Flask-SQLAlchemy**

Do **export FLASK\_APP=app**

Do **flask run –-host=0.0.0.0** and select Box URL under project index if necessary

Make sure to run the bank project beforehand, the steps for that are below. As well as that the accounts the user can use to login are admin, customer1, customer 2 and the password for all of them is p455w0rd.

1. Open a terminal and do **sudo apt update to ensure that everything is updated before we begin installing and creating the virtual environment**

After that we will begin to create the virtual environment, do **sudo apt-get install python3-venv this will install everything needed to create the virtual environment.**

Then do **python3 -m venv venv to create the environment.**

Then activate the virtual environment: **. venv/bin/activate**

This is confirmed by the change to the prompt, which is now**(venv) codio@emotion-theory:~/workspace$** (Each user box will have a different name).

Then do **pip install Flask**

We need a database, so also do

**pip install Flask-SQLAlchemy**

Because we are working in the cloud (not on localhost) we’ll need to deal with cross-site requests. Because this is a potential security issue, it is not allowed by default. So we need to use CORS. Do

**pip install flask-cors**

**./venv/bin/pip3 install -e .** exactly as typed with the initial and final full stops.

Now do **pip3 install PyJWT**

Now do **pip install flask-wtf**

**Note for the bank project you will also need to make sure that you create a seller ID and get the variables PID, SID and secret token which will be placed into app.py in the bookshop project.**

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

**Here enter whatever values you wish for payment ID and amount just ensure you have the payment ID saved along with seller ID and the secret token.**

A screenshot of a computer

Description automatically generated with medium confidence

Run it the project by doing **env FLASK\_APP=payments FLASK\_ENV=development ./venv/bin/flask run --host=0.0.0.0**

Note that whenever the user tries to do any payments, they will need to recreate the seller otherwise it will say checksum str does not match, this is an error with codio as it seems to work once when the user pays however trying it multiple times seems to cause errors. This also means SID, PID and secret will need to be changed for this too work properly.

1. **CODE EXPLANATION**

#Modules that need to be imported for the code to work

**from** **flask** **import** Flask

**from** **markupsafe** **import** escape

**from** **flask** **import** url\_for

**from** **flask** **import** render\_template

**from** **flask** **import** request

**from** **flask** **import** redirect

**from** **flask** **import** abort

**from** **flask** **import** make\_response

**import** **sqlite3**

**from** **flask** **import** flash, session, render\_template, request, redirect, url\_for

**from** **werkzeug.security** **import** generate\_password\_hash, check\_password\_hash

**from** **hashlib** **import** md5

#Variables used to connect to the bank project and form a secure relationship for the payment system to work

sid = '3gI9AEx1a2U='

pid = ' meh'

secret = 'K1GkNIllKOX3AhZsOAE0SnZRX8YA'

app = Flask(\_\_name\_\_) #allows for the app to be exported before its run

app.secret\_key = "secret key" #secret key used within the code

**@app.route**('/index')

**def** **index**():

"""

Checks for username in session and will return either to home or login page depending on whether user is in session

"""

**print**(url\_for('index'))

**if** 'username' **in** session:

session['test'] = 'Test1' #creates a seperate session for logged in users, this is to ensure that the program will always know what user is logged in even when the pages change

**return** home() #after checking that there is a user already logged in it will return to the home page

**else**:

**return** "You are not logged in Please login to continue. <br><a href='/login'><button>Login</button></a>" #If the user is not logged in then it will prompt the user to log in and generate a html button for them to login

**@app.route**('/', methods=['GET', 'POST'])

**def** **login**():

"""

Takes no inputs, will generate a form for the user to use to enter username and password to login

"""

**if** request.method == 'POST':

**return** do\_the\_login(request.form['uname'], request.form['pwd'])

**else**:

**return** show\_the\_login\_form()

**def** **show\_the\_login\_form**():

**return** render\_template('login.html',page=url\_for('login')) #returns the template for the login page

**def** **do\_the\_login**(u,p):

"""

Takes no inputs

Connects to database which posses the table for the login information

If login information matches it will return the user to the index which will send them to the home page

"""

con = sqlite3.connect('Logins1.sqlite3')

cur = con.cursor();

cur.execute("SELECT count(\*) FROM Logins WHERE Name=? AND Password=?;", (u, p))

**if**(int(cur.fetchone()[**0**]))>**0**:# fetchone only takes 1 instance. if the fetchone is 0 then the user does not exist, otherwise it is 1.

session['username'] = u # will create a session for the user when they are confimred to be logged in so the website will not automatically log the user out

**return** redirect (url\_for('index'))

**else**:

**return** render\_template('unauthorised.html') # if the details do not match the user will not be logged in and sent to an unauthorised page

#def logged\_in(t,n):

# con = sqlite3.connect('logged\_in.db')

# try:

# con.execute('CREATE TABLE auth (title TEXT, name TEXT)')

# print ('Table created successfully');

# except:

# pass

# con.close()

# con = sqlite3.connect('logged\_in.db')

# con.execute("INSERT INTO auth values(?,?);", (t,n))

# con.commit()

# con.close()

**@app.route**('/add', methods=['POST'])

**def** **add\_product\_to\_cart**():

"""

Takes no paramethis inputs, will take inputs from form which adds products to the cart and will then use data from the products database to add costs and book codes

will then also link to the simple payments project to create a checkout when the user checkouts the products.

"""

cursor = None

**try**:

\_quantity = int(request.form['quantity'])

\_code = request.form['code']

# forms which take input for the quanityt of books being added and adding the books to the cart

**if** \_quantity **and** \_code **and** request.method == 'POST':

con = sqlite3.connect('products.db')

cur = con.cursor();

cur.execute("SELECT \* FROM products WHERE code=?;", [\_code])

row = cur.fetchone()

#connects to the product database and look for the product details and then add them to an array

itemArray = { row[**2**] : {'name' : row[**1**], 'code' : row[**2**], 'quantity' : \_quantity, 'price' : row[**4**], 'image' : row[**3**], 'total\_price': \_quantity \* row[**4**]}}

**print**('itemArray is', itemArray)

all\_total\_price = **0**

all\_total\_quantity = **0**

session.modified = True

**if** 'cart\_item' **in** session:

**print**('in session')

#This will check if the code for the book selected is within the cart and make sure to adjust the quantity depending on how many are added to the cart

**if** row[**2**] **in** session['cart\_item']:

**for** key, value **in** session['cart\_item'].items():

**if** row[**2**] == key:

old\_quantity = session['cart\_item'][key]['quantity']

total\_quantity = old\_quantity + \_quantity

session['cart\_item'][key]['quantity'] = total\_quantity

session['cart\_item'][key]['total\_price'] = total\_quantity \* row[**4**]

**else**:

session['cart\_item'] = array\_merge(session['cart\_item'], itemArray)

**for** key, value **in** session['cart\_item'].items():

individual\_quantity = int(session['cart\_item'][key]['quantity'])

individual\_price = float(session['cart\_item'][key]['total\_price'])

all\_total\_quantity = all\_total\_quantity + individual\_quantity

all\_total\_price = all\_total\_price + individual\_price

#this will check for the price and adjust the price within the cart

**else**:

session['cart\_item'] = itemArray

all\_total\_quantity = all\_total\_quantity + \_quantity

all\_total\_price = all\_total\_price + \_quantity \* row[**4**]

session['all\_total\_quantity'] = all\_total\_quantity

session['all\_total\_price'] = all\_total\_price

checksumstr = f"pid={pid:s}&sid={sid:s}&amount={all\_total\_price:.1f}&token={secret:s}" #this will use the variables sid, pid and the secret token to form the checksum which will form a secure connection to the payment system allowing payments to go through

#print('checksumstr is', checksumstr)

checksum = md5(checksumstr.encode('utf-8')).hexdigest()

session['checksum'] = checksum

#print('checksum is', checksum)

session['sid'] = sid

session['pid'] = pid

**return** redirect(url\_for('.home'))

**else**:

**return** 'Error while adding item to cart'

**except** **Exception** **as** e:

**print**(e)

**finally**:

cur.close()

con.close()

**@app.route**('/stock')

**def** **stocklevel**():

"""Takes no inputs, will connect to the products database and return the html template and make sure the stock is added to the template"""

con = sqlite3.connect('products.db')

con.row\_factory = sqlite3.Row

cur = con.cursor()

cur.execute("SELECT \* from products")

rows = cur.fetchall()

con.close()

**return** render\_template("stock.html",rows = rows)

**@app.route**('/stock/addnew', methods=['GET', 'POST'])

**def** **updatenewstock**():

"""Takes no paramethis inputs, will return the stock form in which the user can input stock information and add new stock, which then updates the stock databse"""

**if** request.method == 'POST':

**return** insertNewStock(request.form['aiidd'], request.form['bname'], request.form['ccode'], request.form['iidir'], request.form['aprice'], request.form['ddscr'], request.form['ddate'], request.form['tprice'], request.form['quant'])

**else**:

**return** showstockupd();

**def** **showstockupd**():

"""Takes no inputs, returns the addstock html template"""

**return** render\_template('addstock.html')

**def** **insertNewStock**(a, n, c, i, p, d, w, t, q):

"""Takes no inputs, will connect to the products database and insert values for the new stock added"""

con = sqlite3.connect('products.db')

con.execute("INSERT INTO products values(?,?,?,?,?,?,?,?,?);", (a, n, c, i, p, d, w, t, q))

con.commit()

con.close()

**return** showstockupd()

**@app.route**('/home')

**def** **home**():

"""Takes no inputs, will check whether the user is using an admin account and if they are it will return the admin template,

else if the user is not logging in with the admin account it will return them to the customer homepage"""

**print**(url\_for('home'))

**if** session['username'] == 'admin':

test = session['test']

con = sqlite3.connect('products.db')

cur = con.cursor();

cur.execute("SELECT \* FROM products")

rows = cur.fetchall()

**return** render\_template('adminProducts.html', products=rows, page=url\_for('home'))

**else**:

test = session['test']

**print**(test)

con = sqlite3.connect('products.db')

cur = con.cursor();

cur.execute("SELECT \* FROM products")

rows = cur.fetchall()

**return** render\_template('products.html', products=rows, page=url\_for('home'))

**@app.route**('/empty')

**def** **empty\_cart**():

#Takes no inputs, will pop all items out of the cart item array and empty the cart

**try**:

session.pop('cart\_item', None)

**return** redirect(url\_for('home'))

**except** **Exception** **as** e:

**print**(e)

**@app.route**('/delete/<string:code>')

**def** **delete\_product**(code):

#Takes input code which will remove the code for the book and then it will check if the session has an items and

#will adjust values for price and quantity and then return the user back to the homepage"""

**try**:

all\_total\_price = **0**

all\_total\_quantity = **0**

session.modified = True

**for** item **in** session['cart\_item'].items():

**if** item[**0**] == code:

session['cart\_item'].pop(item[**0**], None)

**if** 'cart\_item' **in** session:

**for** key, value **in** session['cart\_item'].items():

individual\_quantity = int(session['cart\_item'][key]['quantity'])

individual\_price = float(session['cart\_item'][key]['total\_price'])

all\_total\_quantity = all\_total\_quantity + individual\_quantity

all\_total\_price = all\_total\_price + individual\_price

**break**

**if** all\_total\_quantity == **0**:

session.pop('cart\_item', None)#if there is nothing in the cart the entire cart will be popped like the empty function

**else**:

session['all\_total\_quantity'] = all\_total\_quantity

session['all\_total\_price'] = all\_total\_price

**return** redirect(url\_for('home'))

**except** **Exception** **as** e:

**print**(e)

**def** **array\_merge**( first\_array , second\_array ):

**if** isinstance( first\_array , list ) **and** isinstance( second\_array , list ):

**return** first\_array + second\_array

**elif** isinstance( first\_array , dict ) **and** isinstance( second\_array , dict ):

**return** dict( list( first\_array.items() ) + list( second\_array.items() ) )

**elif** isinstance( first\_array , set ) **and** isinstance( second\_array , set ):

**return** first\_array.union( second\_array )

**return** False

**if** \_\_name\_\_ == "\_\_main\_\_":

app.run()

Software Design

The software presented here was developed to run a bookshop from which users can purchase books from. The code itself allows for the user to select books from a store page and add them to a shopping cart whilst checking through the prices and quantities they wish to see. It also allows an administrator to login, and they have the same functionality as the user, they also have the added functionality to check stock levels as well as what products the bookshop has in a table and even add more stock to the store.

Implementation

The program is executed by exporting the flask app using app = Flask(\_\_name\_\_) and exported using export FLASK\_APP=app and then run using flask run –host= 0.0.0.0. It will start by going to **@app.route**('/', methods=['GET', 'POST']), which will take the user to the login page and run the functions **def** **login**(): which will then return the function **def** **show\_the\_login\_form**(): which will return the ‘login.html’ template whilst also running **def** **do\_the\_login**(u,p): functions which check the login database for passwords and usernames after running the functions where they login and depending on whether they are using the administrator login or customer logins they will be taken to the homepage **@app.route**('/home'), however the administrator will be given the template 'adminProducts.html', and the user will be given 'products.html'., from the home page the user is able to add products to the cart using **@app.route**('/add', methods=['POST'])**def** **add\_product\_to\_cart**():which will then check what product is being added to the cart and gather all the data from the database to check what values will be needed when adding in cost and quantity and after it has done that it will update the page as well as the template to show products have been added to the cart and it will update the quantity within the cart.

The user can then empty the cart if they wish to using  [**@app.route**('/empty')](mailto:quantity.@app.route('/empty')) **def** **empty\_cart**(): this function will then pop all the items in the cart\_item session and return the user to the homepage where they can see the cart is empty.

The user is also able to delete individual items using **@app.route**('/delete/<string:code>') **def** **delete\_product**(code): this function will then pop the single item from the cart and adjust the price and quantity according to the single item taken out of the cart.

For the administrators they will have a stock button which will take them to **@app.route**('/stock')**def** **stocklevel**():this will connect to the products database and load the stock.html page and create a table showing the user the stock.

There is also a button on the stock page from which the administrator is able to add new stock to the database **@app.route**('/stock/addnew', methods=['GET', 'POST'])**def** **updatenewstock**():

From here the **def** **updatenewstock**(): will generate a form to add stock this then goes to **def** **insertNewStock**(a, n, c, i, p, d, w, t, q): from where the user is inserting values into the template and after they have inserted the values it will take them to the function **def** **showstockupd**(): from where it will return them back to the add stock form, from here they can go back to the homepage or the stock page.

Issues that are present with this code is that there is no security for the pages, though this is not required, meaning users can access any page including admin pages by changing the URL. Though this is not an issue with the specification it is a possible bug to work upon. There is also the fact that whenever the user tries to pay, the payment system will work however each time the user pays, the code must have its PID, SID and secret variables changed to a new seller, it is unknown why this is present though it seems to be an issue with Codio, the software provided, as it works perfectly fine once but after that it will not work again.

<!DOCTYPE html>

<html>

<head>

<title>BookShop but for admins </title>

<meta http-equiv="Content-Security-Policy" content="upgrade-insecure-requests">

<!-- The above line is to ensure that payments are able to go through, since security is not a concern of the project meaning without this line the payment system will expect more security features and will not work-->

<link href="{{ url\_for('static', filename='css/style.css') }}" rel="stylesheet" type="text/css" />

<!-- The above lines links the code to the static.css files which helps create a design for the store.-->

</head>

<body>

<div>

{% with messages = get\_flashed\_messages() %}

{% if messages %}

<ul class=flashes>

{% for message in messages %}

<li>{{ message }}</li>

{% endfor %}

</ul>

{% endif %}

{% endwith %}

</div>

<!-- This section of the code below is for the shopping cart itself, it will start by creating the main page and all the buttons for it. -->

<div id="shopping-cart">

<div class="txt-heading">Shopping Cart</div>

<p>Administator page</p>

<p><a href="{{url\_for('.stocklevel')}}"><button>Stock</button></a></p>

<!-- Below here it will check if the user has added a item to the cart, in which case it will modify the store to have the item in the cart and adjust price and quantity as well as generate an empty cart button and a remove item button in order to empty the cart if need be or to remove a particular item -->

{% if 'cart\_item' in session %}

<a id="btnEmpty" href="{{ url\_for('.empty\_cart') }}">Empty Cart</a>

<table class="tbl-cart" cellpadding="10" cellspacing="1">

<tbody>

<tr>

<th style="text-align:left;">Name</th>

<th style="text-align:left;">Code</th>

<th style="text-align:right;" width="5%">Quantity</th>

<th style="text-align:right;" width="10%">Unit Price</th>

<th style="text-align:right;" width="10%">Price</th>

<th style="text-align:center;" width="5%">Remove</th>

</tr>

{% for key, val in session['cart\_item'].items() %}

{% set quantity = session['cart\_item'][key]['quantity'] %}

{% set price = session['cart\_item'][key]['price'] %}

{% set item\_price = session['cart\_item'][key]['total\_price'] %}

<tr>

<td><img src="/static/images/{{ session['cart\_item'][key]['image'] }}" class="cart-item-image" />{{ session['cart\_item'][key]['name'] }}</td>

<td>{{ session['cart\_item'][key]['code'] }}</td>

<td style="text-align:right;">{{ quantity }}</td>

<td style="text-align:right;">**&#8356;** {{ price }}</td>

<td style="text-align:right;">**&#8356;** {{ item\_price }}</td>

<td style="text-align:center;">

<a href="{{ url\_for('.delete\_product', code=session['cart\_item'][key]['code']) }}" class="btnRemoveAction">

<img src="/static/images/icon-delete.png" alt="Remove Item" />

</a>

</td>

</tr>

{% endfor %}

<tr>

<td colspan="2" align="right">Total:</td>

<td align="right">{{ session['all\_total\_quantity'] }}</td>

<td align="right" colspan="2"><strong>**&#8356;** {{ session['all\_total\_price'] }}</strong></td>

<td></td>

</tr>

</tbody>

</table>

<!-- This part of the code will generate the payment form by linking to the bank project which sends the user the form in which they can enter payment details and pay for the product -->

<form action="https://topic-telex-5000.codio-box.uk/pay" method="POST">

<input type="hidden" name="pid" value={{ session['pid'] }}>

<input type="hidden" name="sid" value={{ session['sid'] }}>

<input type="hidden" name="amount" value={{ session['all\_total\_price'] }}>

<input type="hidden" name="success\_url" value="http://localhost:8080/payment/success">

<input type="hidden" name="cancel\_url" value="http://localhost:8080/payment/cancel">

<input type="hidden" name="error\_url" value="http://localhost:8080/payment/error">

<input type="hidden" name="checksum"value={{ session['checksum'] }}>

<input type="submit" value="Pay">

Amount to be paid {{ session['all\_total\_price'] }}

</form>

{% else: %}

<div class="no-records">Your Cart is Empty</div>

{% endif %}

</div>

<div id="product-grid">

<div class="txt-heading">Products</div>

<!-- The code below is for when a product is added to cart, it will allow the user to click add to cart which will then trigger the add to cart function -->

{% for product in products %}

<div class="product-item">

<form method="post" action="/add">

<div><img src="/static/images/{{ product[3] }}" class="product-image">></div>

<div class="product-tile-footer">

<div class="product-title">{{ product[1] }}</div>

<div class="product-price">**&#163;** {{ product[4] }}</div>

<div class="cart-action">

<input type="hidden" name="code" value="{{ product[2] }}"/>

<input type="text" class="product-quantity" name="quantity" value="1" size="2" />

<input type="submit" value="Add to Cart" class="btnAddAction" />

</div>

</div>

</form>

</div>

{% endfor %}

</div>

</body>

</html>

This is for the admin page, this template will display the storefront of the page with the added features needed for the admin account, it will style.css to generate a style for the page on which all the books will be shown in sections and ensuring they are sized correctly and in the right area. It will also generate the buttons needed to navigate the store and perform different functions. The only difference between the buttons here and the products.html for the user is that it has a button to take the admin to the stock page, another difference that isn’t buttons is that in <title>BookShop but for admins </title> which will make it to the page title specifies that page is for admins. More features of this code are that it uses <meta http-equiv="Content-Security-Policy" content="upgrade-insecure-requests"> and <form action="https://topic-telex-5000.codio-box.uk/pay" method="POST">, these pieces of code help establish a secure connection with the bank project as well as create a form on the page for the user to input payment details and proceed to payment.

<!DOCTYPE html>

<html lang="en">

<head>

<title>Add stock</title>

</head>

<body>

<H1>Add stock</H1>

<P>Please enter your desired book to add to stock, please note you can only add genuine books without a duplicate code:</P>

<form action="{{page}}" method="post">

<!--Form below with all the values that need to be added in order to add a new book to the store and database for products -->

<label for="aiidd">ID:</label><br>

<input type="text" id="aiidd" name="aiidd"><br>

<label for="bname">Book name:</label><br>

<input type="text" id="bname" name="bname"><br>

<label for="ccode">Code:</label><br>

<input type="text" id="ccode" name="ccode"><br><br>

<label for="iidir">Image directory:</label><br>

<input type="text" id="iidir" name="iidir"><br>

<label for="aprice">Price:</label><br>

<input type="range" step= "0.01" min="1.00" max= "50.00"id="aprice" name="aprice" oninput="this.nextElementSibling.value = this.value"><output>25.00</output><br>

<label for="ddscr">Description:</label><br>

<textarea id="ddscr" name="ddscr" rows="4" cols="30"></textarea><br>

<label for="ddate">Date:</label><br>

<input type="date" id="ddate" name="ddate"><br>

<label for="tprice">Trading price:</label><br>

<!--Below is a slider which has a limit on a maximum of 100 for trading price and using step it will only increment the same way normal money would.-->

<input type="range" step= "0.01" min="1.00" max="100.00" id="tprice" name="tprice" oninput="this.nextElementSibling.value = this.value"><output>50.00</output><br>

<label for="quant">Quantity:</label><br>

<input type="range" min="1" max="20" id="quant" name="quant" oninput="this.nextElementSibling.value = this.value"><output>10</output><br>

<input type="submit" value="Submit">

</form>

<!-- Below are buttons which will redirect the user to the main stock page and the home page if they click them, by doing this it will return them to the url for a different page -->

<p><a href="{{url\_for('.stocklevel')}}"><button>Stock</button></a></p>

<p><a href="{{url\_for('.home')}}"><button>Home</button></a></p>

</body>

</html>

This is the add stock html template, this will generate a form which will allow the user to input book details, each part of the form will have <label for="aiidd">ID:</label><br> <input type="text" id="aiidd" name="aiidd"><br>, of course this will be different variables for each label depending on the value that needs to be added to the database.

For the sliders there is more for the input values as we use <input type="range" step= "0.01" min="1.00" max= "50.00"id="aprice" name="aprice" oninput="this.nextElementSibling.value = this.value"><output>25.00</output><br>, the way that this works is that it will create a range from which the slider can go to, depending on the value in this caseprice the minimum is £1 and maximum is £50, furthermore with the step=0.01 we are able to make it so the cost goes up in proper increments for money when moving the slider.

As well as that at the bottom of the page there are buttons placed so that the user is able to travel back to the stock page or back to the homepage with ease.

One issue with the stock form, is that it cannot upload an image, instead the user will need to add the image to the product directory and then write the path for the image down in the section for adding an image.

<!DOCTYPE html>

<html lang="en">

<head>

<title>Login</title>

</head>

<body>

<!-- This will generate a small form in which the user is able to enter details and submit them for the main app to then check -->

<H1>Login</H1>

<P>Please enter your username and password:</P>

<form action="{{page}}" method="post">

<label for="uname">Username:</label><br>

<input type="text" id="uname" name="uname" value="Logger"><br>

<label for="pwd">Password:</label><br>

<input type="text" id="pwd" name="pwd" value="Password"><br><br>

<input type="submit" value="Submit">

</form>

</body>

</html>

This is the login code; it allows the user to input values for username and password which are then sent to app.py which will compare the values to the database and if they match the available logins, it will allow the user to move to the homepage.

<!doctype html>

<html>

<body>

<!-- Below are the buttons which allow the user to return to the homepage or go to the add stock page. -->

<p><a href="{{url\_for('.home')}}"><button>Home</button></a></p>

<p><a href="{{url\_for('.updatenewstock')}}"><button>Add Stock</button></a></p>

<table border = 1>

<!-- The code below will generate a table and will populate rows with data that has been taken from the products database as well as resize images so they fit within the table properly. -->

<thead>

<td>ID</td>

<td>Name</td>

<td>Code</td>

<td>Price</td>

<td>Date</td>

<td>Description</td>

<td>TradePrice</td>

<td>Quantity</td>

</thead>

{% for row in rows %}

<tr>

<td>{{row["id"]}}</td>

<td>{{row["name"]}}</td>

<td>{{row['code']}}</td>

<td><img src="/static/images/{{row['image']}}" width = "100px" height = "150px"></td>

<td>{{row["price"]}}</td>

<td>{{row['date']}}</td>

<td>{{row['description']}}</td>

<td>{{row["tradeprice"]}}</td>

<td>{{row["quantity"]}}</td>

</tr>

{% endfor %}

</table>

</body>

</html>

This is the code for the stock page, this template will generate a table through the use of rows which are supplied in app.py con = sqlite3.connect('products.db')

con.row\_factory = sqlite3.Row

cur = con.cursor()

cur.execute("SELECT \* from products")

rows = cur.fetchall()

con.close()

**return** render\_template("stock.html",rows = rows)

this allows for the HTML template to take the stock supplied from the database and then arrange it into a table and using <td><img src="/static/images/{{row['image']}}" width = "100px" height = "150px"></td> the template will resize the image to ensure that it will always fit within the table and each image is the same size meaning the structure of the page will look reasonable.

This template will also show more information than the homepage as it will allow the user to check trade prices, how much quantity is in stock as well as dates for any book displayed. As well as that this page also possess buttons which allow for the user to travel back to the homepage but also move forward to the add stock page.

<!DOCTYPE html>

<!-- The code below will simply generate a page and insert an image from online which will show the user has not been able to log in -->

<html>

<head>

<meta http-equiv="content-type" content="text/html; charset=UTF-8">

<title></title>

</head>

<body>

<h1 align="center"><br>

</h1>

<div align="center"><img moz-do-not-send="true"

alt="Sorry please renter the url and try logging in again" width="187" height="200"><br>

<br>

<br>

<h1>Bokshop</h1>

<p><br>

</p>

<h2>Unauthorised access</h2>

<img moz-do-not-send="true"

src="https://upload.wikimedia.org/wikipedia/commons/0/03/Forbidden\_Symbol\_Transparent.svg"

alt="Forbidden" width="400" height="400">

<h2>This access attempt has been reported to the relevant

authorities</h2>

</div>

</body>

</html>

This is the template for the unauthorised access page, if a user logins with the wrong details they will be redirected here, this is a very basic page as it has no way to return and only has the instruction to re enter the URL to return to the login page and try again. It does use an image and resizes it as well.

/\*The code below will generate different fonts as well as different spacings and colours for the bookshop making sure it has a certain look too it and that everything is sized and spaced out properly, each part of the code will specify what part of the bookshop it modifies to look a certain way. \*/

body {

font-family: Arial;

color: #211a1a;

font-size: 0.9em;

}

#shopping-cart {

margin: 40px;

}

#product-grid {

margin: 40px;

}

#shopping-cart table {

width: 100%;

background-color: #F0F0F0;

}

#shopping-cart table td {

background-color: #FFFFFF;

}

.txt-heading {

color: #211a1a;

border-bottom: 1px solid #E0E0E0;

overflow: auto;

}

#btnEmpty {

background-color: #ffffff;

border: #d00000 1px solid;

padding: 5px 10px;

color: #d00000;

float: right;

text-decoration: none;

border-radius: 3px;

margin: 10px 0px;

}

.btnAddAction {

padding: 5px 10px;

margin-left: 5px;

background-color: #efefef;

border: #E0E0E0 1px solid;

color: #211a1a;

float: right;

text-decoration: none;

border-radius: 3px;

cursor: pointer;

}

#product-grid .txt-heading {

margin-bottom: 18px;

}

.product-item {

float: left;

background: #ffffff;

margin: 30px 30px 0px 0px;

border: #E0E0E0 1px solid;

}

.product-image {

height: 250px;

width: 155px;

background-color: #FFF;

}

.clear-float {

clear: both;

}

.demo-input-box {

border-radius: 2px;

border: #CCC 1px solid;

padding: 2px 1px;

}

.tbl-cart {

font-size: 0.9em;

}

.tbl-cart th {

font-weight: normal;

}

.product-title {

margin-bottom: 20px;

}

.product-price {

float:left;

}

.cart-action {

float: right;

}

.product-quantity {

padding: 5px 10px;

border-radius: 3px;

border: #E0E0E0 1px solid;

}

.product-tile-footer {

padding: 15px 15px 0px 15px;

overflow: auto;

}

.cart-item-image {

width: 30px;

height: 30px;

border-radius: 50%;

border: #E0E0E0 1px solid;

padding: 5px;

vertical-align: middle;

margin-right: 15px;

}

.no-records {

text-align: center;

clear: both;

margin: 38px 0px;

}}

This is the CSS file, which is utilised by all the HTML templates, this helps create designs for pages throughout the entire project as well as ensure that tables and images are sized properly and have correct alignment so they will look reasonable.

**TESTING**

**Project**

The project is a bookshop its scope is that it can allow the user to login into the bookshop and that they are able to add products into a cart and remove them and then they are able to go to checkout page and pay for the products. As well as that have a system for administators to be able to see and manage stock within the bookshop. It will have simple workflow in Flask/SQL/HTML/CSS and security and UX will not be a major focus.

**Test scope**

For the project the basic functional tests will be:

* If the user can login into admin and customer accounts
* The shopfront can be accessed
* They can add and remove from the cart
* If using the admin page they can manage stock levels
* Checking out

For the project tests that are out of scope would be:

* Security/UX
* Uploading images directly instead of creating filepaths for them to be found.
* Registration with other accounts
* Removing stock using the stock levels page

**Test time**

In total the test should take 55 minutes

**Test regime**

Login (10 mins)

Test 1: Ensure the user can login with the customer and admin accounts provided

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Pass | Yes / No | Fail | Yes / No | Comments |  |

Test 2: Ensure that the user cannot login with the wrong password or username

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Pass | Yes / No | Fail | Yes / No | Comments |  |

Stock system (15 minutes)

Test 1: Is the stock visible in the table with all the required attributes

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Pass | Yes / No | Fail | Yes / No | Comments |  |

Test 2: Can the user go to add stock from here

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Pass | Yes / No | Fail | Yes / No | Comments |  |

Test 1: Does the form function correctly with set amounts on sliders and correct inputs for other values

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Pass | Yes / No | Fail | Yes / No | Comments |  |

Test 2: Does the form add more stock to the database when filled out

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Pass | Yes / No | Fail | Yes / No | Comments |  |

Test 3: Does the form update stock values that already exist

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Pass | Yes / No | Fail | Yes / No | Comments |  |

Shopfront (20 mins)

Test 1: inspect number of database records and cross reference to display. These should match

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Pass | Yes / No | Fail | Yes / No | Comments |  |

Test 2: use alternative database with expanded records and repeat Test 1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Pass | Yes / No | Fail | Yes / No | Comments |  |

Images are correctly displayed

Test 1: ensure that images are correctly resized and fit bounding box

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Pass | Yes / No | Fail | Yes / No | Comments |  |

Currencies are correctly displayed

Test 1: interface should display correct symbol for currency ($, £ etc.)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Pass | Yes / No | Fail | Yes / No | Comments |  |

Prices are correctly displayed

Test 1: Amounts should agree with database and be formatted in the same way (2 decimal points)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Pass | Yes / No | Fail | Yes / No | Comments |  |

Shopping basket (5 minutes)

Test 1: Add item

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Pass | Yes / No | Fail | Yes / No | Comments |  |

Test 2: Remove item

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Pass | Yes / No | Fail | Yes / No | Comments |  |

Test 3: Empty Cart

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Pass | Yes / No | Fail | Yes / No | Comments |  |

Checkout (5 minutes)

Test 1: The user is redirected to a payment page

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Pass | Yes / No | Fail | Yes / No | Comments |  |

Test 2: Pay for items in basket; payment accept or cancel screen displays

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Pass | Yes / No | Fail | Yes / No | Comments |  |

1. **QUALITY ASSURANCE**

The requirements for this code are that it the user can login with 3 accounts, 2 for customers and 1 for admin which is fully working, though the user cannot register or use other accounts however this was not listed as required. They also should be both able to use the storefront which should allow the user to add and remove items, in the code that has been made this is possible and the user is also able to add different quantities of items to the cart as well as see how much the total costs will be. However, the storefront and cart are in one page and not separate out, though they are fully functional and work as intended. Another feature is if the user is logged in with an admin account, they should also have stock page, which has been implemented, as well as a form to add new stock which is also working. Though if the user tries to add the same book again it will not increase the quantity but instead add it again. The final part of the required specification is the checkout, which is working when the user clicks pay, however a success template for the success page has not been set up to show the user that the payment is success, however the user is directed to the success page showing that it does work although they will not be able to view anything that shows them it has worked.

.2 EXTERNAL QA EVALUATION

// Your evaluation of the external’s QA statement

1. **DOCUMENTATION**

The documentation that has been included is the code purpose which is to inform the purpose of the code and what it will do when it is run. As well as that there is code location which allows the user to locate the code in its repository so that they are able to install and run it themselves. There is the code explanation which explains important sections of the code and how specific functions may work and what certain areas will do, as well as that it helps people understand how different areas of code work if they would want to reuse the code or modify it for something else. There is the testing section which defines the project scope as well as the testing scope which a regime of tests that can be run by the tester and checked to see if the project functions and passes certain tests.

.2 EXTERNAL DOCUMENTATION INSPECTION

// Does all documentation appear? Is it usable – can you install and run? Do the tests make sense and can you run them?

// Do you understand the code explanation? Is there anything missing? Are there improvements you’d suggest?

**REFERENCES**

[1] week 4 lab materials

These were used and adapted/modified to help set up a project and create a login system for the user to be able to login into the homepage, as well as that code was taken from here and readapted for other areas such as connecting to databases.

<https://apiv2.coventry.aula.education/file/download/?objectName=e925966235e4e4b64192ba70d9f4d6325001cem_2122_basic_flask_project_setup.docx&name=5001CEM-2122-Basic_Flask_project_setup.docx>

<https://apiv2.coventry.aula.education/file/download/?objectName=1897241ace77cac1dd778c026750c03a5001cem_2122_basic_register_and_login.docx&name=5001CEM-2122-Basic_Register_and_Login.docx>

[2] week 5 lab materials

This was used to help create the front of the shopping cart and well as the homepage, further the use of sessions was taken from here and modified so that the user is able to have sessions for each time they login and sessions will not clear when the cart is emptied but instead pop items from the cart and still have the same functionality.

<https://apiv2.coventry.aula.education/file/download/?objectName=74ed05dea1ca87104b59cac0db51d7165001cem_2122_5_shopping_cart.docx&name=5001CEM-2122-5-shopping-cart.docx>

[3] week 6 lab materials

This was used to help create the checkout system, there is also code from the aula feed for 5001 CEM that was not included in this document as it was an older version, and the fixes were put on the aula feed.

<https://apiv2.coventry.aula.education/file/download/?objectName=4d5c7763258c8b95607c6d394df9373b5001cem_2122_6_flask_aalto_checkout.docx&name=5001CEM-2122-6-Flask-Aalto-checkout.docx>

Luke SID: 10107753

Gave help on creating a button from which I was able to adapt multiple buttons as well as that he helped modify add stock and stock so that it would work properly. As well as advice on the index and some issues with sessions as well.

Ernest SID: 10135033

Gave help on modifying values for different forms as well as working with the CSS and HTML templates to ensure that they would resize images properly and guidance on inserting new stock with the right variables and some small help on sliders.