

School of Computing, Engineering and Mathematics (CEM)

Faculty of Engineering, Environment and Computing (EEC)

**5001CEM SOFTWARE ENGINEERING** | 2122

**PROJECT REPORT**

**NAME:** Ben Godfrey

**SID: 10146431**

# CODE PURPOSE

The program I have created is a flask server that hosts an auction website. This website will allow users to create an account and sell whatever they want using the auction system.

# CODE LOCATION

**https://github.coventry.ac.uk/5001CEM-2122/Ben\_Godfrey\_Auction**

I used GitHub to control my version. This was done by pushing all major changes into the GitHub repos so that it could track all my changes and so allow me to revert if I want.

# CODE INSTALLATION

First you would need to download the project using the provided github link. Open the terminal and navigate into the project file. Then you would need to install and open a virtual environment by executing the following code.

**sudo apt-get install python3-venv**

**. venv/bin/activate**

Then you would need to install the following python modules.

**pip3 install Flask**

**pip3 install Flask-SQLAlchemy**

**pip3 install flask-cors**

Now the project should be fully installed. To run the project you would type in.

**export FLASK\_APP=start**

**flask run –-host=0.0.0.0**

The flask server should now run.

# CODE EXPLANATION

## Overview of its structure

Python flask

(back-end)

SQLite and files

(storage)

HTML and JavaScript

(front-end)

The overall structure of the project is quite simple. It is split into three section main, the front-end, the back-end and the storage. The most important part of the project will be how these different levels of code communicate with each other in an effective manner.

### Front-end

The front-end is the part that the user sees. It displays the interface that the user is going to use as well as giving the user all the code that is required for the front-end uses. This means that the front-end cannot handle sensitive information and cannot do any vital processing. It is possible to transpile the front-end so that the code becomes very hard to read and therefore difficult to edit but that is unnecessary for a project this small in scope. This means that the front-end will do a limited amount of processing and any processing it does will only be used to aide the user in using the website and not for performing any vital data processing.

### Back-end

The back-end is the part that pushes the front-end to the user as well as running algorithms that not anyone can see. This part of the project can basically have access to all the information it wants since the user won’t have access to any of the code or any of the data that I don’t want them to have access to. The back-end could also be consider the backbone of the project since it has access to all the other levels of the code as well as where the bulk the programming was be done.

### Storage

The storage section is where I would store the data that is not temporary. Except for images where I am storing it on the file system, I am using sqlite. This is because it is easy to use since I can command just give it a command and it would do all the data sorting for me instead of forcing me to implement it in python unnecessarily.

## Storage

### Database structure

Items

Bids

Users

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Users | username, (primary key) string | name, string | Email, string | phoneNumber, integer |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Items | ID, (primary key) Integer | name, string | description, string | username, (foreign key from Users username) string | datetime, string | finished, integer |

|  |  |  |  |
| --- | --- | --- | --- |
| Bids | itemID, (foreign key from Items ID) integer, part of composite key | Username, (foreign key from Users username) string, part of composite key | maxPrice, integer |

I decided to use this database structure because it is very space efficient with no unnecessary repeats in data and because it is easy to search specific records since every record will have a unique primary key or composite key.

### Python implementation

#### Creation

**def** **databaseCreation**():

**try**: #if database is not found then it will create a database instead with the right tables

file = open("AuctionDB.db","r")

file.close()

**except** FileNotFoundError:

file = open("AuctionDB.db","w") #create the file

file.close()

connection = sqlite3.connect("AuctionDB.db") #connects to the new database

cursor = connection.cursor()

tableCreate = open("databaseTableCreation.txt","r") #this file stores the sql commands to create the right tables

tableCreate = tableCreate.readlines()

string = ""

**for** i **in** range(**0**,len(tableCreate)):

string+= tableCreate[i] #makes the array representing each line into only one single string

tableCreate = string

cursor.executescript(tableCreate) #executes the sql script that will create the tables

connection.close()

This is how a created the database in python. As you can see this code basically checks if the database file exists. If it does it does nothing. If it doesn’t however it will proceed to create a data based using the relevant commands stored in file databaseTableCreation.txt which is basically just creating the three tables that I need.

databaseTableCreation.txt

**CREATE** **TABLE** "Items" (

"ID" INTEGER **PRIMARY** **KEY** AUTOINCREMENT **UNIQUE**,

"name" TEXT **NOT** **NULL**,

"description" TEXT **NOT** **NULL**,

"username" TEXT **NOT** **NULL**,

"datetime" TEXT **NOT** **NULL**,

"finished" INTEGER,

**FOREIGN** **KEY**("username") **REFERENCES** "Users"("username")

);

**CREATE** **TABLE** "Users" (

"username" TEXT **NOT** **NULL**,

"password" TEXT **NOT** **NULL**,

"email" TEXT **NOT** **NULL**,

"phoneNumber" TEXT **NOT** **NULL**,

**PRIMARY** **KEY**("username")

);

**CREATE** **TABLE** "Bids" (

"itemID" INTEGER **NOT** **NULL**,

"username" TEXT **NOT** **NULL**,

"maxPrice" INTEGER **NOT** **NULL**,

**PRIMARY** **KEY**("itemID","username"),

**FOREIGN** **KEY**("username") **REFERENCES** "Users"("username"),

**FOREIGN** **KEY**("itemID") **REFERENCES** "Items"("ID")

);

#### Reading Data

##### Does something exist

**def** **accountExists**(username): #checks if an account with a username exists (similar to checking if login is the same with identical logic)

connection = sqlite3.connect("AuctionDB.db")

cursor = connection.execute("SELECT username FROM Users WHERE username=:givenName", {"givenName": username})

result = **False**

**for** row **in** cursor:

result = **True** #account exists so result will be true

connection.close()

**return** result

This way of reading data from the database is the simplest where it just checks if an account exists or not. This is convenient to use because with just the username I can call this function and decide whether to do something (such as create a new account) based on a True or False output. The SELECT command is used to retrieve data and also I used WHERE to only pick the any occurrence of where a specific username exists. When inserting the python variables into the sql command I am using parameters to do so by substituting :var with the python variable. This is done because this will prevent an sql injection attack which may edit the database unknowingly or read the database unknowingly.

##### Retrieving attributes

**def** **account**(username): #gets information about account except password and username

connection = sqlite3.connect("AuctionDB.db")

cursor = connection.execute("SELECT email, phoneNumber FROM Users WHERE username=:givenName", {"givenName": username})

result = **False**

**for** row **in** cursor:

result = {"email": row[**0**], "phoneNumber": row[**1**]} #puts valid data in list to make it easy to read

connection.close()

**return** result

This way of reading data from the database is getting specific attributes of the record in a table. This is done by using the SELECT sql command to retrieve the data as well as using parameters in the sql command to prevent sql injection attacks. The result will be read and saved as a list so that the result is easily readable with the column names intact. This will only return one record since there can only be one account. But there are cases where multiple records are returned such as a user looking at the items they are selling. For this the result will be an array where it the result would be read row by row and appended to an array shown as below.

**def** **myItems**(username): #this will retreive the items that a specific user is selling

connection = sqlite3.connect("AuctionDB.db")

cursor = connection.execute("SELECT \* FROM Items WHERE username=:name", {"name": username}) #getting all items a specific username is selling

result = []

**for** row **in** cursor:

result.append({"ID": row[**0**], "name": row[**1**], "description": row[**2**], "username": row[**3**], "datetime": datetime.datetime.strptime(row[**4**], '%Y-%m-%d %H:%M:%S.%f')}) #this will store all the atributes of the objects returned

connection.close()

**return** result

#### Insert Data

**def** **hashPassword**(username, password): #this hashes the password, learnt in https://docs.python.org/3/library/hashlib.html

**return** str(hashlib.sha224(b"" + str(username).encode() + str(password).encode()).hexdigest())

**def** **newAccount**(username, password, email, phoneNumber):

password = hashPassword(username, password)

accountExist = accountExists(username)

**if** **not**(accountExist): #checks if the account exists or not before creating the account

connection = sqlite3.connect("AuctionDB.db")

connection.execute("INSERT INTO Users (username, password, email, phoneNumber) VALUES (:newUsername, :newPassword, :newEmail, :newPhoneNumber)", {"newUsername": username, "newPassword": password, "newEmail": email, "newPhoneNumber": phoneNumber}) #adds the new account to the database

connection.commit()

connection.close()

**return** accountExist #return if the account exists or not

This piece of code allows me to insert a new record that contains their username, password, email and phoneNumber which effectively creates a new account for a user. I have decided to hash the password so that if there is any data breach any potential hacker won’t be to get the password the user has. This function will also check if the account exists already before inserting any data. This is because I can’t have two accounts with the same username since there will be no real way to differentiate the accounts of the users using the auction website. I also use parameter in the sql command to prevent sql injection attacks. Once I have put the sql command in I need to write connection.commit() for sql to actually write the data then I closed the connection with the database. The function returns whether the account existed previously or not and therefore if an account was created.

#### Images

**def** **newItem**(itemName, description, username, image): #this will just allow the user to add a new item

connection = sqlite3.connect("AuctionDB.db")

cursor = connection.execute("INSERT INTO Items (name, description, username, datetime) VALUES (:newName, :newDescription, :newUsername, :cDateTime)", {"newDescription": description, "newName": itemName, "newUsername": username, "cDateTime": datetime.datetime.now()})

SecondCursor = connection.execute("SELECT last\_insert\_rowid()") #learn sql command in https://alvinalexander.com/android/sqlite-autoincrement-insert-value-primary-key/

primaryID = **0**

**for** row **in** SecondCursor:

primaryID = row[**0**] #gets the primary key of the newly inserted item

connection.commit()

connection.close()

image.save("static/itemImages/" + str(primaryID)) #image is going to be saved using the primary key

This piece of code allows a user to add a new item so that other users can bid on it. This works in the same way as the other python code creating a new account until after the first sql command. Then I used the sql command SELECT last\_insert\_rowid() to get the primary key of the newly created record. I need to get the primary key because I want to store the image and so I am going to name the image file to be the same as the primary key of the item to link that record in the Items table and the file.

## Flask

### URLs and retrieving data from forms

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

**def** **loginPage**(validCredentials = ""):

**if** request.method == "POST": #if tis handling the forms

username = request.form["username"]

password = request.form["password"] #gets inputed username and password

**if** len(username) < **50** **and** len(password) < **50**:

**if** login.login(username, password): #if login credentials are valid then goes to right page, else goes back to log in so that they can put in valid credentials

session["username"] = username

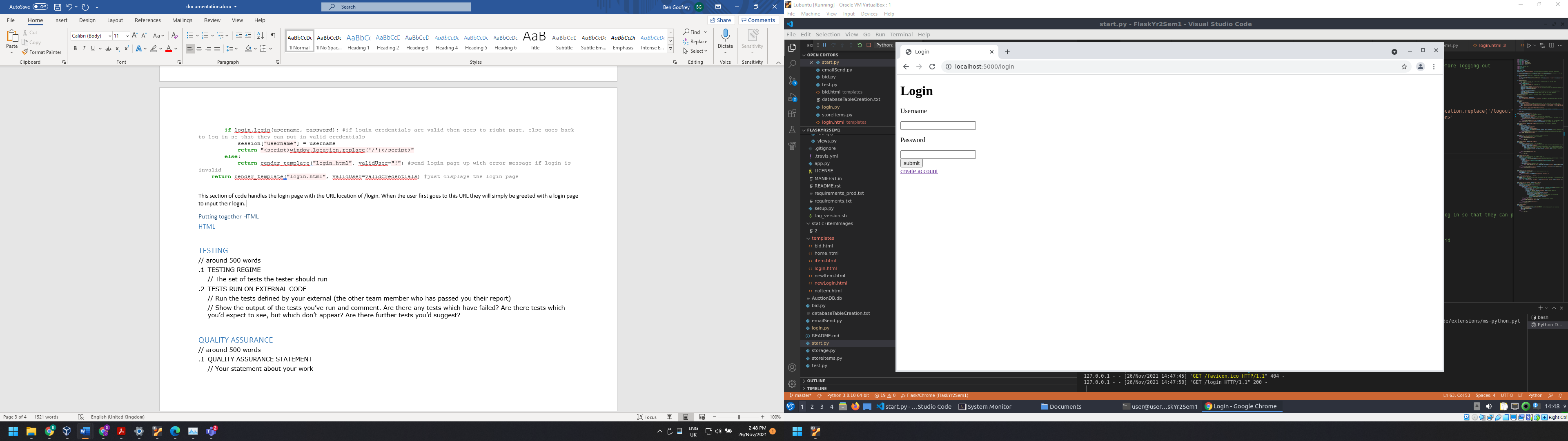
**return** "<script>window.location.replace('/')</script>"

**else**:

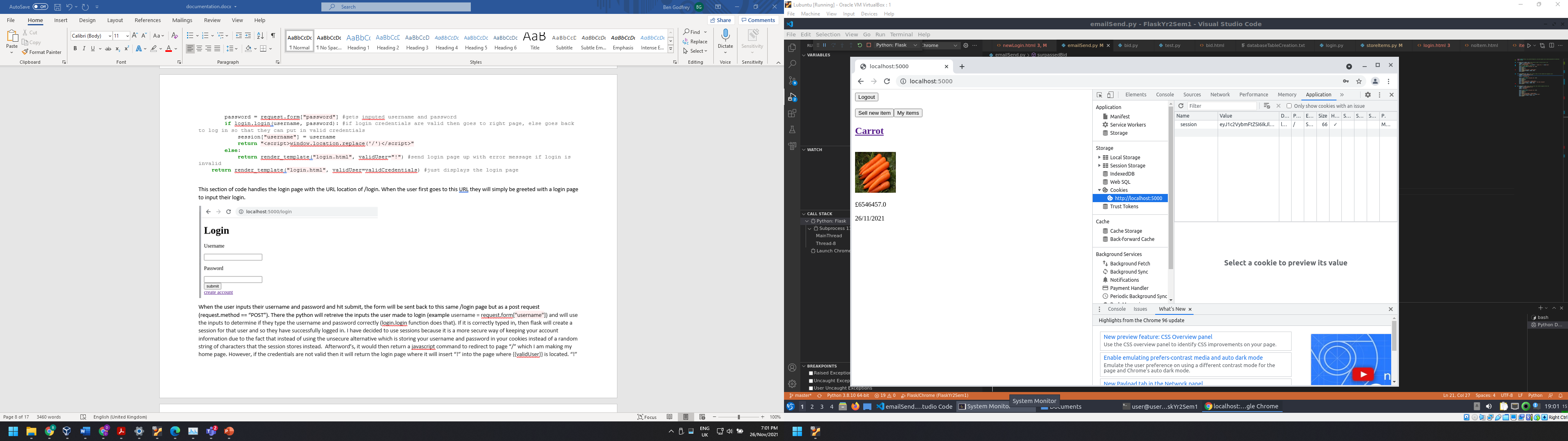
**return** render\_template("login.html", validUser="!") #send login page up with error message if login is invalid

**return** render\_template("login.html", validUser=validCredentials) #just displays the login page

This section of code handles the login page with the URL location of /login. When the user first goes to this URL they will simply be greeted with a login page to input their login.



When the user inputs their username and password and hit submit, the form will be sent back to this same /login page but as a post request (request.method == “POST”). There the python will retreive the inputs the user made to login (example username = request.form["username"]) and will use the inputs to determine if they type the username and password correctly (login.login function does that). Then python will checks if the password and username have less than 50 characters so that someone doesn’t take all the storage space with their large username and password. If the username and password is correctly typed in, then flask will create a session for that user and so they have successfully logged in. I have decided to use sessions because it is a more secure way of keeping your account information since, instead of using the unsecure alternative which is storing your username and password in your cookies it would store a random string to represent you.



Afterword’s, it would then return a javascript command to redirect to page “/” which I am making my home page. However, if the credentials are not valid then it will return the login page where it will insert “!” into the page where {{validUser}} is located. “!” in javascript is the same as not so I decided to use that to change a condition in a javascript condition. The change in condition will make the page notify the user that their credentials are not valid by inject a paragraph into a div.

Section of the html

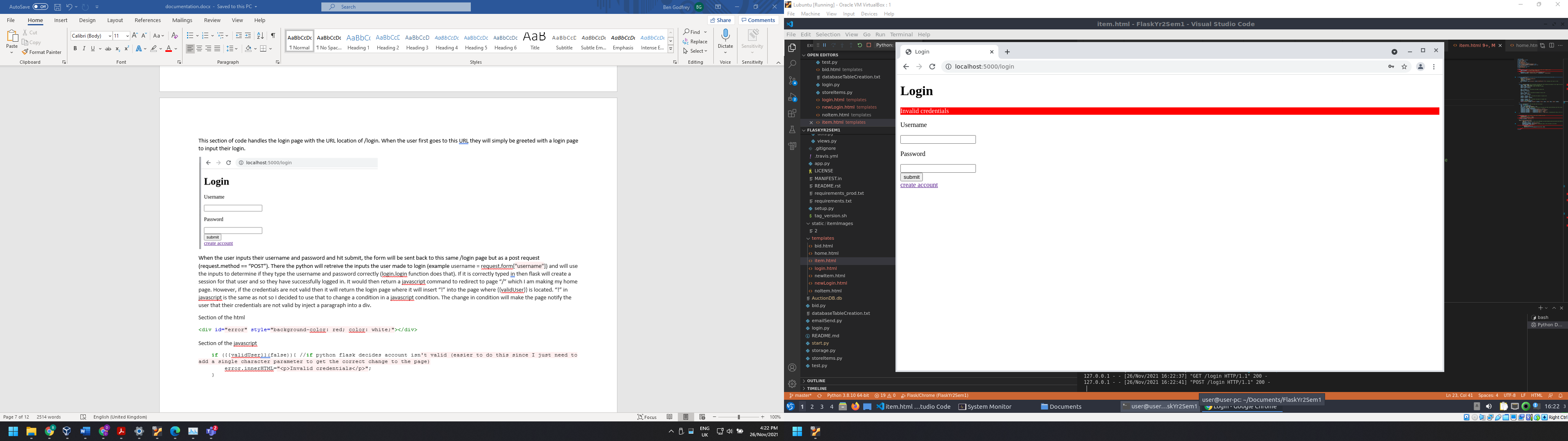
<div id="error" style="background-color: red; color: white;"></div>

Section of the javascript

**if** ({{validUser}}(false)){ //**if** python flask decides account isn't valid (easier to do this since I just need to add a single character parameter to get the correct change to the page)

error.innerHTML="<p>Invalid credentials</p>";

}



### Public JSON

**@app**.route("/item/<int:itemID>/jsonPrice")

**def** **JSONPrice**(itemID): #this is the json file with the current price of the item

firstBidding = bid.finalPrice(itemID)

**if** firstBidding != **None**: #if there is a bid for the item

**return** {"currentPrice": firstBidding["currentPrice"], "currency": "£"}

**else**:

**return** {"currentPrice": "0", "currency": "£"}

This gets a variable from its URL. This is done with the <int: itemID> part where it says that it is expecting an integer in that part of the screen. That integer will then be passed as a parameter (named itemID in this case) to the function handling what this URL is doing. This URL would use the itemID to get the current price of that item and return a json object (basically just a list). This is used because it will allow me (using javascript) update the price on the page the user is saying every few seconds so that people can see in almost real-time the price increasing as more bids come in. I send json object just as a one time as well since it can make for me the code more legible since for a page (specifically the item page) it can be repeating the same substitutions ({{substitute\_this}}) multiple times and having many substitutions making it difficult to read in code.

### Putting together html using python

**def** **htmlListItems**(items): #this lists a bunch of items viewable in html

html = ""

**for** i **in** range(**0**,len(items)): #loops through all the items

finalPrice = bid.finalPrice(items[i]["ID"])

price = finalPrice["currentPrice"]

**if** price % **1** == **0**:

price = str(int(finalPrice["currentPrice"]))

**else**:

price = str(finalPrice["currentPrice"]) + "0"

html+= "<div id='" + str(i) +"'><a href='/item/" + str(items[i]["ID"]) + "'><h2>" + str(items[i]["name"]) + "</h2></a><br><img src='/static/itemImages/" + str(items[i]["ID"]) + "' width='100' height='100'/><br>" + "<p>£" + price + "</p><p>" + stringDate(items[i]["datetime"]) + "</p><br></div>"

**return** html

**@app**.route("/")

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

html = "<!DOCTYPE html>"

**if** session.get("username"): #if logged in send this html, else send another bit of html (login link)

html+= "<button onclick=logout()>Logout</button><br><br> <script>function logout(){document.cookie=''; window.location.replace('/logout');}</script>"

html+= '<button type="button" onclick="window.location.href=' + "'" + "/newItem" + "'" + '">Sell new item</button>' #display these when the user is logged in

html+= '<button type="button" onclick="window.location.href=' + "'" + "/myItems" + "'" + '">My items</button>'

**else**:

html+= "<a href='/login'><small>login</small></a><br><br>" #display this when user isn't logged in

items = storeItems.allItems() #gets all the items that are being sold and makes it presentable

html+= htmlListItems(items) #list all items that user may want to buy

**return** html

To display a list of items which would naturally vary in length I have decided to make python to assemble the html for me as you can see with the code to display all the items. This is done in the htmlListItems function by looping through all the elements in the array storing the relevant items attributes and simply adding the relevant attributes to the html in a way that makes a coherent page the user can use.

## Email

**def** **bidSet**(item, maxBid, user): #this sends an email of when the bid is set

fromaddr = "devmail8022@gmail.com"

toaddr = user["email"]

text = "you have bidded " + str(maxBid) + " pounds for a " + item["name"]

#subject = "you have set a bid"

server = smtplib.SMTP('smtp.gmail.com', **587**)

server.ehlo()

server.starttls()

server.ehlo()

server.login("devmail8022", "DevMail1024")

server.sendmail(fromaddr, toaddr, text)

server.quit()

This function will allow python to send an email to a user (in this case when the user sets a bid). This first is done by the email address "devmail8022@gmail.com" that I created a couple of days ago as a throwaway address in python. Then after logging in it would send a mail to the users email based on what parameters was enters. The parameters user and item will be in a list form that contains the important attributes name and email. This is because it is easier to do it like that since the functions that read the database for the items and users attributes return in that form so I can just put the list in that form into this function and it works. This basic framework was used to send basically all the emails the server needed to send for whatever reason.

### Background tasks

For performing background tasks (such as finding and acting bid that close at specific times) I would need to make python perform two tasks at the same time, the background tasks and serving the users with pages and relevant information. To do this I used the threading module which allows python to perform in a pseudo-asynchronous way (only uses one core).

**def** **searchAndCloseItems**(): #this is invloved in handling when the bid closes https://smirnov-am.github.io/background-jobs-with-flask/

**while** **True**:

time.sleep(**10**)

closeItems = storeItems.dueClose()

**for** i **in** range(**0**,len(closeItems)): #cycles through all the items that has been close

log = bid.finalPrice(closeItems[i]["ID"])

account = login.account(log["username"])

emailSend.wonBid(closeItems[i], log["currentPrice"] ,account) #sends email to winner

thread = Thread(target=searchAndCloseItems)

thread.daemon = **True**

thread.start()

The function is running in an infinite loop checking if any item should close bidding. And with any items that where close an email would be sent to the winner. There is a time delay of 10 seconds so that there won’t be unnecessary strain on the machine (if it wasn’t there it will max out a core).

## Javascript

### Countdown

fetch("/item/{{idImage}}/json")

.then(res => res.json())// https://www.codegrepper.com/code-examples/javascript/javascript+read+json+file+from+url

.then((out) => {

description.innerHTML = out["description"];

nameHeading.innerHTML = out["name"]; //displaying some of the attributes in the html

title.innerHTML = out["name"];

**if**({{sold}}(**true**)){

time(out["datetime"]);

setInterval(**function**(){time(out["datetime"])}, **100**);

}

});

This piece of javascript code handles getting some attributes from a json file and creating a timer counting down to the end of the auction period for that item. The json object is retrieved from the URL "/item/{{idImage}}/json" and is first used to just display some of the attributes to the HTML (displaying the name and description of the item). Then if it wasn’t sold already it would then get the datetime attribute of the json object and put it into a time() function that will be called every second using setInterval. The time function will display how long it is until the end of the auction.

**function** timeLeft(timeM){//this calculates the weeks days hours minutes and seconds it will take for the bid to close

timeM = timeM / **1000**; //makes it into seconds only

weeks = timeM / (**60**\***60**\***24**\***7**); //gets the number of weeks

leftover = weeks % **1**;

weeks = weeks - leftover;

days = leftover \* **7**; //calculates the number of days left

leftover = days % **1**;

days = days - leftover;

hours = leftover \* **24**; //calculates the amount of hours left

leftover = hours % **1**;

hours = hours - leftover;

minutes = leftover \* **60**; //calculates the amount of minutes left

leftover = minutes % **1**;

minutes = minutes - leftover;

seconds = leftover \* **60**; //calulates the amount of seconds left

leftover = seconds % **1**;

seconds = Math.round(seconds);

**return** {"seconds": seconds, "minutes": minutes, "hours": hours, "days": days, "weeks": weeks};

}

**function** time(dateTime){ //gets the time left for the bid

now = **new** Date();

then = **new** Date(dateTime);

then = then.setDate(then.getDate() + **14**);

difference = then - now //adds week to the date from python and then finds difference with the current datetime

timeLeftList = timeLeft(difference);

timeDisplay.innerHTML = timeLeftList["seconds"] + " seconds";

timeDisplay.innerHTML+= " " + timeLeftList["minutes"] + " minutes";

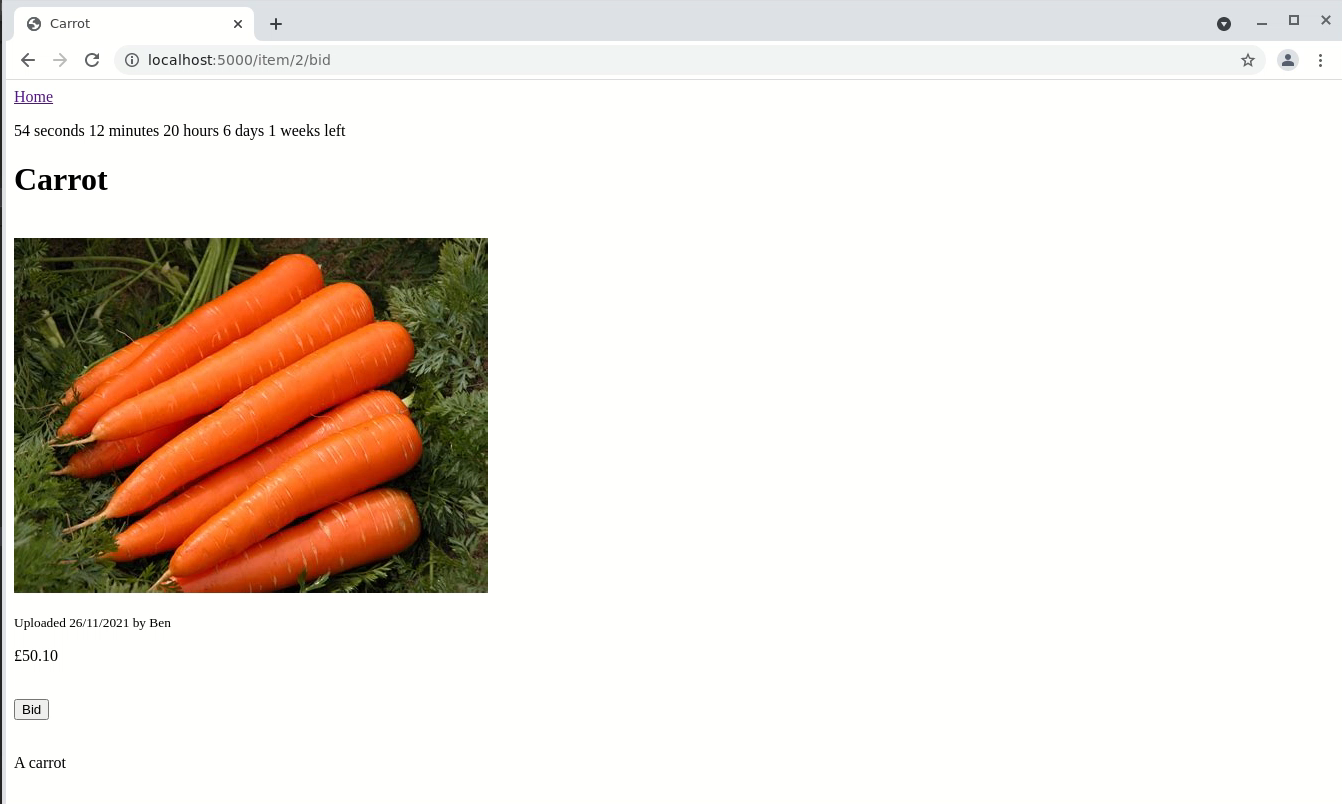
timeDisplay.innerHTML+= " " + timeLeftList["hours"] + " hours";

timeDisplay.innerHTML+= " " + timeLeftList["days"] + " days";

timeDisplay.innerHTML+= " " + timeLeftList["weeks"] + " weeks left"; //displaying how long is left of the acution

}

The time function would first get the current datetime and then find the difference between the current datetime and the parameter dateTime + 14 days (since I want the duration of all auctions to be two weeks) in milliseconds. Then the difference is passed into the timeLeft function where it will calculate the difference in seconds, minutes, hours, days, and weeks then display the results to the html. This plus the setInteval allows for the countdown to exist.



### Updating price

**function** currentPrice(){ //retrieves the json that gives the current price of the item

attributes = {};

fetch("/item/{{idImage}}/jsonPrice")

.then(res => res.json())

.then((out) => {

**if**(out["currentPrice"] % **1** == **0**){

price.innerHTML = out["currency"] + out["currentPrice"]; //displays the result in html

}**else**{

price.innerHTML = out["currency"] + out["currentPrice"] + "0"; //displays the result in html

}

});

}

**if**({{sold}}(**true**)){ //this checks if the item is deemed sold yet

currentPrice();

setInterval(currentPrice, **10000**);

}**else**{

price.innerHTML = "Bid over!!!";

}

The price updating works in a very similar way to the countdown except it continuously asks for the json object from the server. This code basically calls the currentPrice function every 10 seconds. That function then retrieves the json object that contains the current price and then displays the current price.

### Input checker

**function** insideString(character, string){ //this checks the email address has an @ inside it

array = string.split("")

**for**(i = **0**; i < array.length; i++){

**if**(array[i] == "@"){

**return** **true**;

}

}

**return** **false**;

}

**function** submit(){

document.cookie=''; //removes any cookies that may be there

**if**(document.getElementById("password").value == document.getElementById("password2").value){ //checks if the two passwords

**if**(document.getElementById("phoneNumber").value.length < **11** && document.getElementById("phoneNumber").value.length > **0**){

inside = insideString("@", document.getElementById("email").value) //sees if "@" is inside the email

**if**(document.getElementById("email").value.length < **50** && inside){

document.form.submit(); //submits the form, learnt in https://www.javascript-coder.com/javascript-form/javascript-form-submit/

}**else**{

error.innerHTML = "<p>Invalid email</p>"; //inject a message in the html when phone number isn;t valid

}

}**else**{

error.innerHTML = "<p>Invalid phone number</p>"; //inject a message in the html when phone number isn;t valid

}

}

**else**{

error.innerHTML = "<p>Two passwords are not the same</p>"; //inject a message in the html when the two passwords do not match

}

}

This is a basic validator of the inputs that the user types in to create the account. This is used so that it can say to the user what they have gotten wrong when they make a mistake in the details, they have type in. This won’t bother the server and is easy to implement. If the user wants to go around this and disable the basic checker, then it wouldn’t really matter since it would just mean that their account won’t work properly and won’t affect the functioning of the server (will have limiters on the server side of how long each input is so that the server won’t be filled up).

# TESTING

Test the registration system. Can you create an account? Can you create two accounts with the same username? Can you create an account with missing inputs?

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

Test the login system. Can you login to an account? What happens if you delete all cookies (deletes session stored in browser) while on a page that requires a login, does it stay the same on reload?

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

Check if in the home page that all the items displayed are still biddable (was added less than two weeks ago).

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

Check if the prices that are displayed for each item are correct (10p higher than second highest bidder or the bid if there is only one bidder).

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

Does the price auto update on another browser when a separate user adds a new maximum bid (should take 10 seconds for that to occur)?

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

Check if the price is formatted correctly (£14.20 instead of £14.2 and £14 instead of £14.0 as examples).

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

Check if the images are being displayed properly. Does it display for the right items? Does the images display at all?

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

Check the actual bidding system. Can the user make the bid easily? What happens if you try to put invalid input? What occurs when a bid is placed? All functions related to the bidding is in bid.py.

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

Check the email system. Email will be sent when you set a bid, when someone surpasses your bid and when the item is closed for bidding. All functions related to emails are place in the emailSend.py file.

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

Check if you can add a new item. Is it possible? Does it associate with the right username? Are images correctly uploaded?

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

# QUALITY ASSURANCE

// around 500 words

.1 QUALITY ASSURANCE STATEMENT

The project is for the most part finished. It can allow the user to create their account, bid for an item while getting email notifications of the progress of their bid. The user can also bid an item of their own too. However, I couldn’t implement the payment system for when someone wins a bid. Although there is email functionality and a basic checker of the email for the email, I haven’t implemented an email validator where the website will send a random number to confirm it is your email.

Most of the code is commented and split into smaller functions and files so that potential errors are more easily pinpointed and so the code is more maintainable. The only part that may be difficult to maintain is the homepage where I made python assemble the page instead of using a template since it was easier to do it that way with the simple page.

.2 EXTERNAL QA EVALUATION

// Your evaluation of the external’s QA statement

# DOCUMENTATION

Purpose of the code (section 1)

Statement of the location of the code (section 2)

How to install the project (section 3)

Overall structure of the code (section 4)

Explanation of the commented code (section 4)

Testing regime (section 5)

QA statement (section 6)

**REFERENCES**

[**https://www.pythonforbeginners.com/code-snippets-source-code/using-python-to-send-email**](https://www.pythonforbeginners.com/code-snippets-source-code/using-python-to-send-email)

<https://www.codegrepper.com/code-examples/javascript/javascript+read+json+file+from+url>

<https://alvinalexander.com/android/sqlite-autoincrement-insert-value-primary-key/>

<https://docs.python.org/3/library/hashlib.html>

<https://smirnov-am.github.io/background-jobs-with-flask/>

// do not count towards word limit

// You must reference any sites consulted and code used in your work. Code re-use is fine and a common practice (one reason why documentation is so important). Lifting entire code blocks including complete applications without attribution is an academic conduct offence and this has consequences.

// I’ll be saying more about how to reference code.