

School of Computing, Engineering and Mathematics (CEM)

Faculty of Engineering, Environment and Computing (EEC)

**5001CEM SOFTWARE ENGINEERING** | 2122

**PROJECT REPORT**

**NAME:** Jupiter Mellen

**SID:**

Table of Contents

[Purpose: 3](#_Toc89448221)

[Code Location and Installation: 4](#_Toc89448222)

[CODE EXPLANATION 4](#_Toc89448223)

[Project Structure 4](#_Toc89448224)

[Run.py 4](#_Toc89448225)

[\_\_init\_\_.py 5](#_Toc89448226)

[sql\_auction.py 5](#_Toc89448227)

[Base.html 6](#_Toc89448228)

[Bootstrap 6](#_Toc89448229)

[Navbar 7](#_Toc89448230)

[Message flashing 8](#_Toc89448231)

[Routes.py + Templates 9](#_Toc89448232)

[Home Page 9](#_Toc89448233)

[Back End 9](#_Toc89448234)

[Front End 9](#_Toc89448235)

[Item Page 11](#_Toc89448236)

[Back End 11](#_Toc89448237)

[Front End 11](#_Toc89448238)

[Login Page 13](#_Toc89448239)

[Back End 13](#_Toc89448240)

[Front End 14](#_Toc89448241)

[Register Page 15](#_Toc89448242)

[Back End 15](#_Toc89448243)

[Front End 16](#_Toc89448244)

[Sell Page 17](#_Toc89448245)

[Back End 17](#_Toc89448246)

[Front End 18](#_Toc89448247)

[User Page 19](#_Toc89448248)

[Back End 19](#_Toc89448249)

[Front End 20](#_Toc89448250)

[Testing 21](#_Toc89448251)

[Testing Regime 21](#_Toc89448252)

[Project Explanation 21](#_Toc89448253)

[Scope 21](#_Toc89448254)

[Test Structure 21](#_Toc89448255)

[External Test 23](#_Toc89448256)

[Quality Assurance 28](#_Toc89448257)

[Brief and requirements delivered 28](#_Toc89448258)

[Coding standards 28](#_Toc89448259)

[Documentation standards 28](#_Toc89448260)

[Team details and support 28](#_Toc89448261)

[Timely deliver 28](#_Toc89448262)

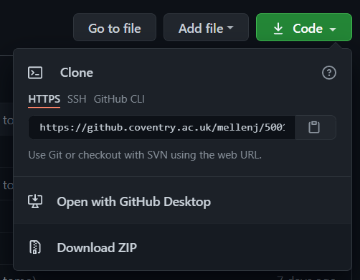
[Documentation 29](#_Toc89448263)

[Documentation List 29](#_Toc89448264)

# Purpose:

Pez-Bay is an Auction website that allows you to sell items through live bidding, it features User Accounts and their item listings.

# Code Location and Installation:



My code is located on the Coventry GitHub here:

[**https://github.coventry.ac.uk/mellenj/5001CEM\_Auction\_House**](https://github.coventry.ac.uk/mellenj/5001CEM_Auction_House)

From GitHub:

* You can either use the https to clone (depending on your ide)

Or Download the Zip file

* Modules required to run are
  + Flask - pip install flask
  + Flask sqlalchemy - pip install flask-sqlalchemy
* Then run:
  + run.py

# CODE EXPLANATION

## Project Structure

My project is split into different files and directories to form a package. Run.py Imports the Auction folder as a module and runs the app () function from \_\_init\_\_. This is good for several reasons: Firstly, it increases the readability of the code making it easier for other programmers to understand and work from. It increases the modularity of the project making it easy to change each section and stops you from rewriting lots of code. Finally testing the code becomes much easier, as each section is split into different files so you can individually test them. This idea was introduced to me in a YouTube video: Package Structure (Schafer, 2018) This thoroughly explained why package structure is needed and I believed it would benefit my Project.

## Run.py

from Auction import app

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

    app.run(debug=True)

Run.py imports app from the Auction Package. Line 3 Checks if it is currently being ran. Then line 4 starts up the flask application with debug as True allowing for updates to templates while the program is currently running

## \_\_init\_\_.py

from flask import Flask

from flask\_sqlalchemy import SQLAlchemy

from sqlalchemy import select

import os

# Setting flask configs like the path for images to be uploaded

app = Flask(\_\_name\_\_)

app.secret\_key = "test"

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///users.sqlite3'

app.config["SQLALCHEMY\_TRACK\_MODIFICATIONS"] = False

#Sets directory for images no matter where the folder origin is

path = os.path.dirname(os.path.abspath(\_\_file\_\_))

app.config["IMAGE\_UPLOADS"] = os.path.dirname(os.path.join(path+"\\static\\images\\"))

db = SQLAlchemy(app)

#imports after main imports so everything initialises correctly

from Auction import routes

This is used to correctly initialise the Auction module, it imports other files into one place and adds some app.configs for routes.py to use.

|  |  |  |
| --- | --- | --- |
| Imports: | from flask import Flask | Imports main flask module used mainly in routes.py |
|  | from flask\_sqlalchemy import SQLAlchemy | Imports SQL Alchemy used for SQL tables  Sql\_auction.py for creating tables  Routes.py for adding/altering rows in tables |
|  | import os | used to get correct directories |
|  | from Auction import routes | Imports routes after configs and other imports so it initializes correctly |

Line 17: Finds path for image folder, this is necessary as the user could have the folder in any directory, so it needs to be able to adapt depending on where it is stored.

## sql\_auction.py

this file creates both the Item and User SQLite tables using sqlalchemy module. This module uses blueprints to define the structure of SQL tables. I learnt how to Structure and get this working properly with a YouTube video by (Tim, 2018)

'''

User Table

takes:

Username , Password , Email , Phone Number

'''

class User(db.Model):

    user\_id = db.Column(db.Integer, primary\_key=True)

    username = db.Column(db.String(100))

    password = db.Column(db.String(50))

    email = db.Column(db.String(100))

    phone = db.Column(db.Integer)

    def \_\_init\_\_(self, name, email, password, phone):

        self.username = name

        self.password = password

        self.email = email

        self.phone = phone

The User Table contains attributes: id, username, password, email, and phone number, it is used to store users registration info, so it can be looked up when a user tries to log back in.

'''

Item Table

takes:

Item Name , Image Name , Item Description , Start Date , End Date , Username(Forign key from User Table) , Current Bid , Sold

'''

class Item(db.Model):

    item\_id = db.Column(db.Integer,  primary\_key= True)

    name = db.Column(db.String(50))

    img\_name = db.Column(db.String(50))

    desc = db.Column(db.String(200))

    start\_date = db.Column(db.DateTime, default = datetime.utcnow())

    end\_date = db.Column(db.DateTime, default = datetime.utcnow()+timedelta(hours = 12))

    username = db.Column(db.String, db.ForeignKey("user.username"))

    current\_bid = db.Column(db.Integer, default = 0)

    sold = db.Column(db.Boolean, default = False)

    def \_\_init\_\_(self, name, img\_name, desc, username):

        self.name = name

        self.img\_name = img\_name

        self.desc = desc

        self.username = username

The Item table contains attributes: name, image name, description, start date, end date, username (this is a foreign key linked to the user table) , current bid and sold. A new row is made when an object is made and then passed to the

db.session.add()

function.

db.create\_all()

creates both tables in a new .db file if they don’t already exist

­­

## Base.html

Base.html is a template that is inherited by all other templates, this reduces rewritten code and saves time. Main components in this template include linking bootstrap, the Navbar and message flashing.

### Bootstrap

<head>

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

        <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css" rel="stylesheet"

        integrity="sha384-1BmE4kWBq78iYhFldvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3" crossorigin="anonymous">

        <title>{%block title%}{%endblock%}</title>

    </head>

The header of base.html includes the link to bootstrap which allows me to use their libraries that include layouts, Navbars, Buttons and more that help round my website and make it look professional without having to spend hours coding and designing them myself. I used the bootstrap docs (GetBootstrap.com, 2021) (<https://getbootstrap.com/docs/5.0/getting-started/introduction/>) To learn what to use and how to use it. The header also contains my CSS stylesheet. It is above the bootstrap link so it can overwrite any CSS settings that I wanted to change.

### Navbar

  <nav class="navbar navbar-dark bg-dark">

          <div class="container-fluid">

            <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarToggleExternalContent" aria-controls="navbarToggleExternalContent" aria-expanded="false" aria-label="Toggle navigation" >

              <span class="navbar-toggler-icon"></span>

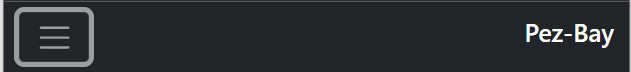
            </button>

            <h5 class="text-white h4, text-center">Pez-Bay</h5>

          </div>

        </nav>

When the Navbar is collapsed, you can see an open button and Pez-Bay title on the right side



 <div class="collapse" id="navbarToggleExternalContent">

          <div class="bg-dark p-4">

            <ul class="navbar-nav">

              <li class="nav-item">

                <a class="nav-link, text-muted" aria-current="page" href="/">Home</a>

              </li>

              <li class="nav-item">

                <a class="nav-link, text-muted" aria-current="page" href="/register">Register</a>

              </li>

              <li class="nav-item">

                <a class="nav-link, text-muted" aria-current="page" href="login">Login</a>

              </li>

              <!--If user is logged in it shows logout, Profile, Sell links-->

              {%if 'user' in session%}

              <li class="nav-item">

                <a class="nav-link, text-muted" aria-current="page" href="/logout">Logout</a>

              </li>

              <li class="nav-item">

                <a class="nav-link, text-muted" href="/user">Profile</a>

              </li>

              <li class="nav-item">

                <a class="nav-link, text-muted" href="/sell">Sell</a>

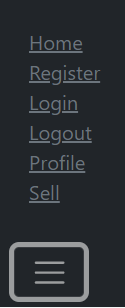
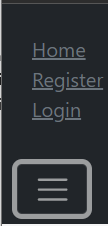
              </li>

              {%endif%}

            </ul>

          </div>

        </div>

 when opened links to Home, Register, And Login are shown. If the user is logged in then Logout, Profile and sell are show too (see figure 1&2) . This is done by using jinjas {%%} quotes, this allows you to add if statements inside of your html code. On line 34 it checks if the user key is in session and if so the code inside the jinja statement is outputted.

### Message flashing

  <div class="container-fluid">

          <!-- flashes messages

          Loops over each flashed message-->

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

            {%if messages%}

                {%for msg in messages%}

                    <p>{{msg}}</p>

                {%endfor %}

            {%endif %}

          {%endwith%}

          <!-- space for content of each daughter template-->

          {%block content%}

          {%endblock%}

        </div>

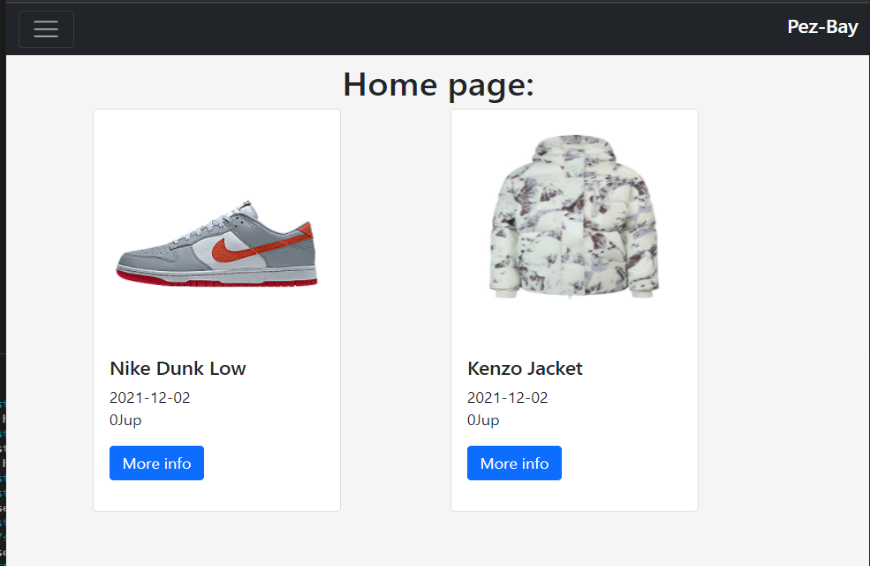
Message flashing is a flask function that allows you to flash a message on the page, this is useful so you can tell the user when something has happened e.g., ‘you’ve logged in’, or ‘Incorrect password’. This was done by using jinja statements to loop through each message, it is put in a <p> tag and then displayed on the website



**IMPORTANT!**

This next section I will join both html templates and there route.py function counterpart, so you can get a better understanding of how each page works

## Routes.py + Templates



## Home Page

The home page is the first thing the user sees, using Bootstrap Grid The page is made up of rows and columns, In each grid a bootstrap card is placed that includes the item image, name of the item, the date the item was put up for sale, the name of the user that put the item on sale and if the user is logged in it will also show and extra button displayed as More info. Using Bootstrap Grid, the page can change depending on the user’s screen size.

### ­­­­Back End

'''

Home Function:

- checks if a user is logged in (In the session)

- Takes item\_id to be passed to item function

- Returns index.html

'''

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

def home():

    sold\_time\_out()

    if session:

        info = True

    else:

        info = False

    if request.method == "POST":

        current\_item = (request.form['id'])

        session['current\_item\_id'] = current\_item

        return redirect(url\_for('item'))

    return render\_template("home.html", values=Item.query.all(), info\_check = info)

### 

The Home function first checks if the user is in session and saves it to a variable that is passed to the html template with a SQL query that contains all the rows in the Item table. If the template sends a POST form it will take the item id that was sent, adds that item id to the session and redirects to the item page.

### Front End

<!--Home Page Template

Contains all items still on sale

allows user to see more info if logged in

Arguments:

Item:

    - sold

    - name

    - start date

    - username

    -

-->

{%block title%}Home Page{%endblock%}

{%block content%}

<h1 class="text-center">Home page:</h1>

<div class="container">

    <div class ="row">

        <!-- Loops over each item that hasent been sold-->

        {%for item in values%}

            {%if item.sold == 0%}

                <div class = "col">

                    <!--PLaces item in a bootstrap card with image, Title, and description

                    Also has more information button that links to item page-->

                    <div class="card" style="width: 250px !important;">

                        <img class="card-img-top" src = "{{url\_for('static', filename='images/'+item.img\_name)}}" alt="No Image Uploaded" style="height: 250px !important;">

                        <div class="card-body">

                            <h5 class="card-title">{{item.name}}</h5>

                            <p class="card-text">{{item.start\_date.date()}}<br>{{item.username}}</p>

                            <!--Checks if user is logged in to view more information -->

                            {%if info\_check%}

                                <form action="#" method="POST">

                                    <button type="submit" name="id" value="{{item.item\_id}}" class="btn btn-primary">More info</button>

                                </form>

                            {%endif%}

                        </div>

                    </div>

                </div>

            {%endif%}

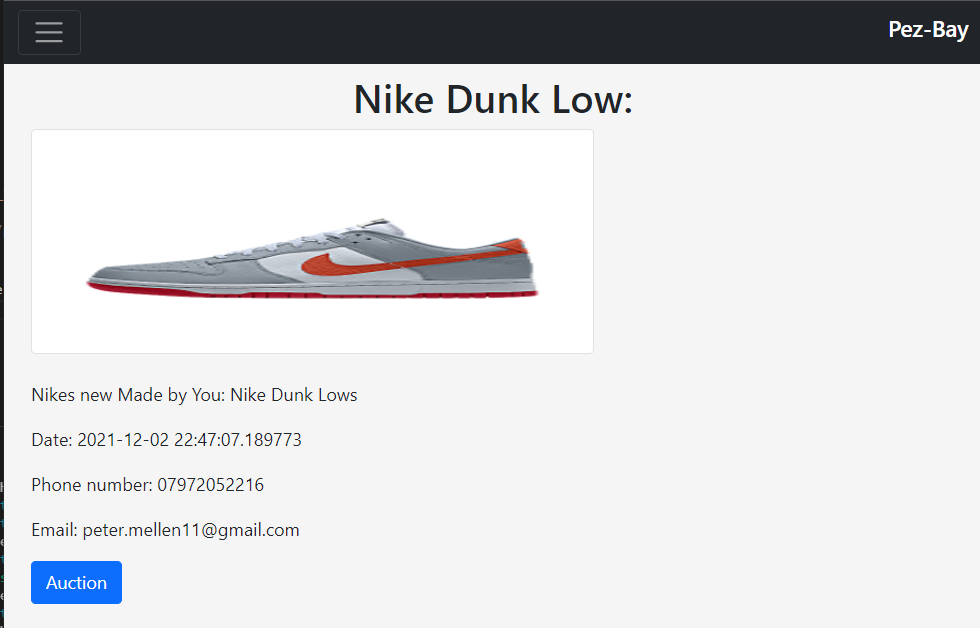
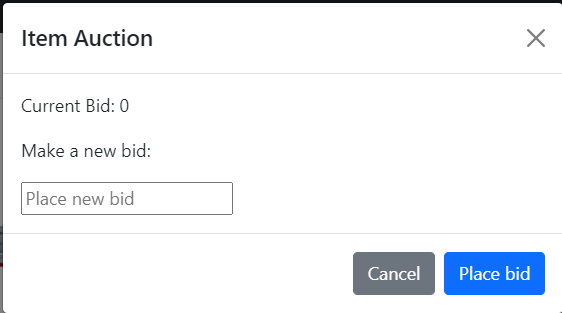
        {%endfor%}

    </div>

</div>

­­­­

The Home page displays each item that is currently being sold. It displays these items in a Bootstrap Card. This HTML file uses jinja for statements to loop over each Item. It checks to see if the item is sold and if False it then displays name, start date and username. Using jinja If statements it checks if the user is logged in to show the more info button. If the more info button is pressed it will send a form that includes the item id

­

## Item Page

The Item page is an extension of the Home page, it can only be accessed if the user is logged in. It allows the user to see more information and bid on the item that is currently being displayed. It uses the Bootstrap Modal to create a pop up that allows the user to place a new bid. The Modal Box includes the current bid And an Input box to enter your new bid.

### Back End

'''

Item Function

- querys a row in Item and User tables based on what was saved in the session

- Takes a new bid and inputs into the Item table

- Returns item.html

'''

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

def item():

    item = Item.query.filter\_by(item\_id = session['current\_item\_id']).first()

    user = User.query.filter\_by(username = session['user']).first()

    if request.method == "POST":

        bid = int(request.form['new\_bid'])

        if bid > item.current\_bid:

            item.current\_bid = bid

            db.session.commit()

        else:

            flash("Bid can't be lower than the current bid!")

    return render\_template("item.html", item\_data = item, user\_data = user)

### 

The item function takes two SQL query’s one from the item table, it filters by the current id stored in the session (see Home page), and one from the user table. This filters from the current user in the session, this would have been added when they were logged in. Then returns this to the item.html. If item.html has a POST form it takes a new bid and then updates the item table with the current bid.

### Front End

<h1 class="text-center">{{item\_data.name}}:</h1>

<!-- Item Template:

- shows detailed description of the current

- a bidding box

Arguments:

Item: Item Desctiption, Item Date

User: Phone number, Email

-->

<div class="container-fluid" ">

    {%with item = item\_data%}

        <img src="{{url\_for('static', filename='images/'+item.img\_name)}}" alt="..." class="img-thumbnail" style="height: 500px; width: 500px; ">

        <p><br>{{item.desc}}</p>

        <p>Date: {{item.start\_date}}</p>

    {%endwith%}

    {%with user = user\_data%}

        <p>Phone number: 0{{user.phone}}</p>

        <p>Email: {{user.email}}</p>

    {%endwith%}

        <!-- Button trigger Auction box  -->

    <button type="button" class="btn btn-primary" data-bs-toggle="modal" data-bs-target="#exampleModal">

        Auction

    </button>

The Item page displays image, description, start date, phone number and email through p tags and jinja statements using that data passes to it from the backend.

 <!-- Auction box

    Includes:

    Current bid

    Allows you to send a new bid

    Bid must be > current bid

    -->

    <form action="#" method="POST" id="auction\_form"> </form>

    <div class="modal fade" id="exampleModal" tabindex="-1" aria-labelledby="exampleModalLabel" aria-hidden="true">

        <div class="modal-dialog">

            <div class="modal-content">

                <div class="modal-header">

                    <h5 class="modal-title" id="exampleModalLabel">Item Auction</h5>

                    <button type="button" class="btn-close" data-bs-dismiss="modal" aria-label="Close"></button>

                </div>

                <div class="modal-body">

                <p>Current Bid: {{item\_data.current\_bid}}</p>

                <p>Make a new bid:</p>

                <input type="number" name="new\_bid" placeholder="Place new bid" form="auction\_form" required>

                </div>

                <div class="modal-footer">

                    <button type="button" class="btn btn-secondary" data-bs-dismiss="modal">Cancel</button>

                    <button type="submit" class="btn btn-primary" form="auction\_form">Place bid</button>

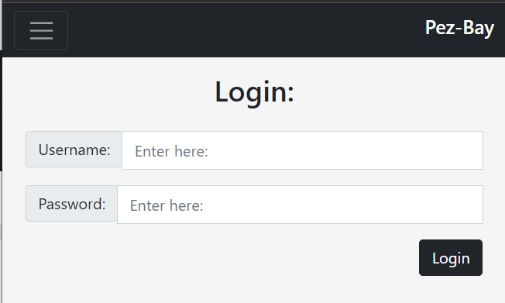
                </div>

            </div>

        </div>

    </div>

below this Is a button that leads to the Bootstrap component Modal (GetBootstrap.com, 2021). The Modal contains the items current bid and an input box and two buttons one to place bid and one to exit the modal



## Login Page

The Login page takes users input compares it to SQL tables and the redirects to the home page if a user is found

### Back End

'''

Login function

- checks if input is in User table, and adds user to the session

- Takes username and password

- Returns login.html

'''

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

def login():

    if request.method == "POST":

        user = request.form["Username"]

        passw = request.form["Password"]

        found\_users = User.query.filter\_by(username=user).first()

        if found\_users:

            if found\_users.password == passw:

                flash(f"{user}, You have logged in successfully.")

                session["user"] = user

            else:

                flash("Incorrect password")

                return redirect(url\_for("login"))

        else:

            flash("User not found")

        return redirect(url\_for("user"))

    else:

        if "user" in session:

            flash("Already logged in")

            return redirect(url\_for("home"))

    return render\_template("login.html")

The login function checks for a POST and then retrieves the username and password. Then uses the username to make a SQL query and filter by it. After if a match has been found the password is checked. A flash message is sent: “You have logged in successfully”, and the user is added to the session to stay logged in while using the website. If the username is not found a flash message is returned saying “user is not found “and if the password does not match to the one in the table, Incorrect password is flashed. I final check to see if the user is already in the session and a flash message is returned “Already logged in”

­­

### Front End ­­­­

<!--

Login Page

Allows user to input Username and Password

to log into the website and allow you to fully access the website

-->

{%block title%}Login Page{% endblock %}

{%block content %}

<!--Form to send both inputs-->

<form action="#" method="POST" id="login\_post" ></form>

<h1 class="text-center">Login:</h1>

<!--Container is in rows for each input-->

<div class="container pt-3">

    <div class="row">

        <div class="input-group mb-3">

            <div class="input-group-prepend">

                <span class="input-group-text" id="basic-addon1">Username:</span>

            </div>

            <input type="text" class="form-control" placeholder="Enter here:" name="Username" form="login\_post" aria-label="Username" aria-describedby="basic-addon1" >

            <p><br></p>

        </div>

    </div>

    <div class="row">

        <div class="input-group mb-3">

            <div class="input-group-prepend">

                <span class="input-group-text" id="basic-addon1">Password:    </span>

            </div>

            <input type="password" class="form-control" placeholder="Enter here:" name="Password" form="login\_post"  aria-label="Password" aria-describedby="basic-addon1" >

            <p><br></p>

        </div>

    </div>

    <!--Submit button to trigger form-->

    <input type="submit" value="Login" class="btn btn-dark" style="float: right;" form="login\_post">

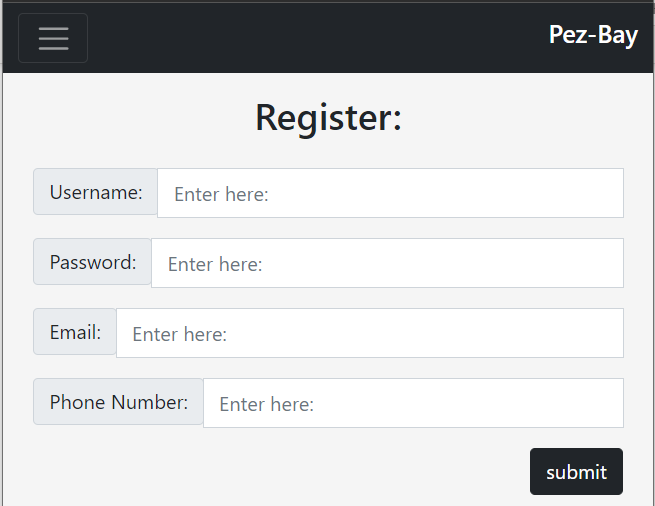
</div>

Login.html takes two inputs username and Password, it uses bootstrap Span and Inputs, Span being on the left of the input so it is clear what needs to be typed in the input box, The password box is also Type password to obscure the plaintext from anyone watching.

These inputs are then sent through one form named login\_post through POST.

<form action="#" method="POST" id="login\_post" ></form>

    <input type="submit" value="Login" class="btn btn-dark" style="float: right;" form="login\_post">



## ­Register Page

This page is where the user can create a new account, It takes four inputs: Username, Password, Email and Phone Number. Once all fields are filled out in the correct format there is a submit button that will send the data to the back end and store it in a new row in the user table.

### Back End

'''

Register function:

- Validates inputs, and enters new fields into User table

- Takes username, password, email and phone

- Returns register.html

'''

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

def register():

    if request.method == "POST":

        log = request.form

        rname = log["Username"]

        password = log["Password"]

        email = log["Email"]

        phone = log["Phone"]

        found\_user = User.query.filter\_by(username=rname).first()

        if  not rname or not password or not email or not phone:

            flash("Please fill out all fields")

        elif found\_user:

            flash("User already exists")

        else:

            new\_user = User(rname, email, password, phone )

            db.session.add(new\_user)

            db.session.commit()

            flash("New user registered successfully")

            flash("Go to /login to login to your new account")

    return render\_template("register.html")

The register function checks if a Post request has come through, it then creates variables for each input that is needed for the SQL User table. It checks to make sure the user doesn’t already exist, it then checks to make sure all fields are not empty, and flashes “("Please fill out all fields” if they are. And flashes “User already exists” if user is found in the table. If everything is valid a new\_user object is made with all the inputs and added. I found a very good stack overflow page that explained this to me perfectly (Siroky, 2013), Then “db.session.add(new\_user)” and “db.session.commit()” adds the row to the table. A flash message is returned to show the user the account creation was successful.

### Front End

<!--Register Template:

Inputs:

    - Username , Password , Email and Phone

Sends inputs through one form to be proccessed in routes.py

Validation is done through back end-->

<form action="#" method="POST" id="register\_post" ></form>

<h1 class="text-center">Register:</h1>

<div class="container pt-3">

    <div class="row">

        <div class="input-group mb-3">

            <div class="input-group-prepend">

                <span class="input-group-text" id="basic-addon1">Username:    </span>

            </div>

            <input type="text" class="form-control" placeholder="Enter here:" aria-label="Username" aria-describedby="basic-addon1" name="Username" form="register\_post"> required

            <p><br></p>

        </div>

    </div>

    <div class="row">

        <div class="input-group mb-3">

            <div class="input-group-prepend">

                <span class="input-group-text" id="basic-addon1">Password:    </span>

            </div>

            <input type="password" class="form-control" placeholder="Enter here:" aria-label="Password" aria-describedby="basic-addon1" name="Password" form="register\_post" required>

            <p><br></p>

        </div>

    </div>

    <div class="row">

        <div class="input-group mb-3">

            <div class="input-group-prepend">

                <span class="input-group-text" id="basic-addon1">Email:       </span>

            </div>

            <input type="email" class="form-control" placeholder="Enter here:" aria-label="Email" aria-describedby="basic-addon1" name="Email" form="register\_post" required>

            <p><br></p>

        </div>

    </div>

    <div class="row">

        <div class="input-group mb-3">

            <div class="input-group-prepend">

                <span class="input-group-text" id="basic-addon1">Phone Number:</span>

            </div>

            <input type="number" class="form-control" placeholder="Enter here:" aria-label="Phone" aria-describedby="basic-addon1" name="Phone" form="register\_post" required>

            <p><br></p>

        </div>

    </div>

    <input type="submit" value="submit" class="btn btn-dark" style="float: right;" form="register\_post">

Register.html is split up into div rows to split each input neatly and centred to fill that space of the page. Each input uses Bootstrap Span and Input, this allows text to be next to the input box so it is easy to understand what input must be given. Password input uses a password type to obscure the plain text, email uses an email type to make sur an @ is used and phone uses number type so only numbers are entered. All inputs have the required field as they must be filled out to submit a POST request. I used w3 schools page on this to find the correct types I needed to use (w3 Schools, n.d.).

<form action="#" method="POST" id="register\_post" ></form>

All inputs are submitted through one form and sent to the back end

## Sell Page

The sell page allows the user to upload items to the Item table and allows another user to bid on these items. It takes three inputs: item name, Item image and item description. There is a submit button that sends all this data to the back end

### Back End

'''

Sell Function:

- checks if user is in session,

validates inputs are not NULL,

makes sure image name is unique and saves image it image folder

adds item name desc and image name to Item table

- Takes item name, description and image

- Returns sell.html

'''

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

def sell():

    if not session:

        flash("Please log in")

        return redirect(url\_for("login"))

    elif request.method == "POST":

        if request.form['item\_name'] != '' and request.form['item\_desc'] != '':

            name = request.form['item\_name']

            desc = request.form['item\_desc']

            if request.files:

                img = request.files['item\_img']

                item\_user = session["user"]

                img\_name = img.filename

                img\_name = unique\_img\_name(img\_name)

                img.save(os.path.join(app.config["IMAGE\_UPLOADS"], img\_name))

                item\_data = Item(name, img\_name, desc, item\_user)

                db.session.add(item\_data)

                db.session.commit()

                flash("Item saved successfully")

                return redirect(url\_for("sell"))

        else:

            flash("please enter all fields")

    return render\_template("sell.html")

The sell function first checks If the user is in session, and flashes “Please log in” then redirects you to the login page. The function then checks for a POST request. And validates that the items are not empty and flashes “please enter all fields” if they are. It then saves the name and description inputs to a variable. It then checks that a file has been received it takes the image filename( if the filename Is not unique it sends it to the unique function ) and saves the image to the image folder. Item name, description and image filename is then saved as a new row to the Item table and committed (Siroky, 2013). A flash is returned “Item saved successfully”.

### Front End

<form action="#" method="POST" id="sell\_post" enctype="multipart/form-data"></form>

<h1 class="text-center">Sell Item:</h1>

<div class="container pt-3">

    <div class="row">

        <div class="input-group mb-3">

            <div class="input-group-prepend">

                <span class="input-group-text" id="basic-addon1">Item name</span>

            </div>

            <input type="text" class="form-control" placeholder="Enter here:" aria-label="Username" aria-describedby="basic-addon1" name="item\_name" form="sell\_post"required >

            <p><br></p>

        </div>

    </div>

    <div class="row">

        <input type="file" class="custom-file-input" name="item\_img"  accept="image/\*" form="sell\_post"required

>

        <p><br></p>

    </div>

    <div class="row">

        <div class="input-group">

            <div class="input-group-prepend">

                <span class="input-group-text">Item<br>Description</span>

            </div>

            <textarea class="form-control" aria-label="With textarea" name="item\_desc" form="sell\_post" required></textarea>

            </div>

    </div>

    <p><br></p>

    <input type="submit" value="submit" class="btn btn-dark" style="float: right;" form="sell\_post">

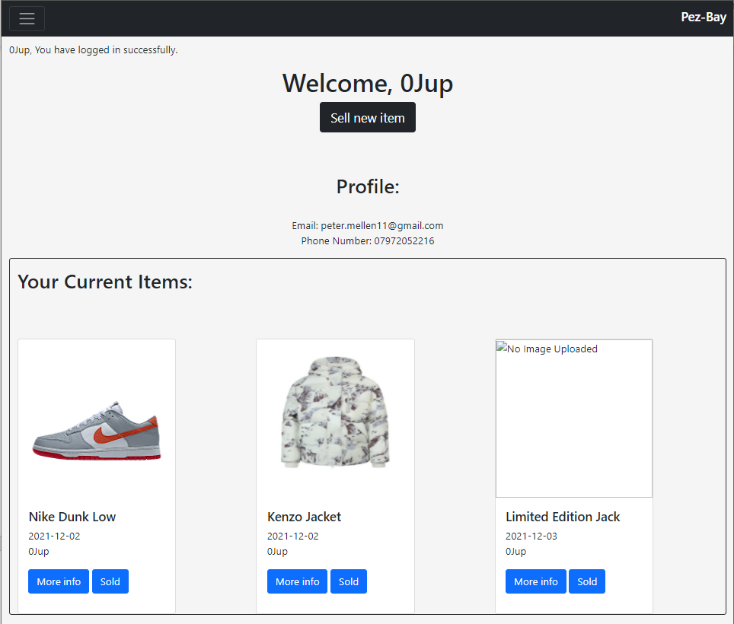
</div>

Register.html is split into rows for each input. It takes the item name using a standard Bootstrap Span and Input (GetBootstrap.com, 2021), Then in the second row I used an image input to take the user image, finally for the third row I used a Bootstrap Span and Textarea for the user to enter the description, this is useful as the text area allows for more space for multiline descriptions. There is a submit button at the bottom that activates the “sell\_post ” form

<form action="#" method="POST" id="sell\_post" enctype="multipart/form-data"></form>

This form sends all the inputs to the back end, I used “enctype multipart/form-data” so I could send the image as well as the text in one form.

## User Page

The user page displays information about the user and the items they currently have on for sale. There is a button that sends you to the sell page. I have added an item with incorrect filetype to display what happens if an image is not uploaded or of incorrect file type. The images are the same as the home page and have more info buttons, they also have an extra sold button to end there auction.

### Back End

'''

User function:

- checks for a button click on user page

    - sold button alters field in Item table to True

    - id (more details) adds id to session and redirects to item page for more details about that item

- Returns user.html

'''

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

def user():

    if "user" in session:

        user = session["user"]

        user\_data = User.query.filter\_by(username=user).first()

        user\_items = Item.query.filter\_by(username=user).all()

        if request.method == "POST":

            if 'id' in request.form:

                current\_item = (request.form['id'])

                session['current\_item\_id'] = current\_item

                return redirect(url\_for('item'))

            elif 'sold' in request.form:

                current\_item = Item.query.filter\_by(item\_id=request.form['sold']).first()

                current\_item.sold = True

                db.session.commit()

                flash("Item sold!")

                return redirect(url\_for('user'))

        return render\_template("user.html",  user = user\_data, values = user\_items)

    else:

        flash("You are not logged in!")

        return redirect(url\_for("login"))

The user function first checks if the user is in session and flashes “You are not logged in!” if they’re not. It makes two SQL queries getting the users data row by filtering by the username in the session, then gets the user item rows by filtering with the username again, it can do this because there is a foreign key in the Item table. Then it checks for a POST request, if it is id (For the more information button) it adds the id to the session and redirects to the item page. And if sold (sold button is pressed) it query’s the item id in that table to get the row and changes the sold attribute to True, it returns a flash saying” Item sold!”. Then redirect to refresh the page

Finally, it returns the template with user data values and the users item values

### Front End

<!--User Template

This is the users profile page

Includes:

    - User details:

        - Name , Email , Phone number

        - Sell new item Button

    - Users Items:

        - Name, Start date

        - More info Button

        - Sold Button

Arguments:

user:

    - username, email, phone

item:

    - name, date , username

These are used to dipslay the current user in session and that users specific items

-->

{%block title%}User{%endblock%}

{%block content%}

<!-- User info Block: -->

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

    <h1>Welcome, {{user.username}}</h1>

    <a href="/sell" class="btn btn-dark btn-lg " role="button" aria-disabled="true">Sell new item</a>

    <p>

    <br>

    <br>

    <h2>Profile:</h2>

    <br>

    Email: {{user.email}}

    <br>

    Phone Number: 0{{user.phone}}

    </p>

</div>

This first section of user.html displays some user information including: Username, Email, and the users phone number it also has a Bootstrap button (GetBootstrap.com, 2021) to send you to the sell page. It uses jinja statements and the values passed in the back end to display this information.

<!-- Users item Block-->

<div class="container-fluid" style="border-radius:.25rem!important; border:1px solid #313131; box-shadow: darkgray !important;">

    <p>

        <h2>

            Your Current Items:

        </h2>

        <br><br>

    </p>

    <div class ="row">

        <!-- Loops over each item that was passed and puts it in a bootstrap card-->

        {%for item in values%}

            <div class = "col">

                <div class="card" style="width: 250px !important;">

                    <img class="card-img-top" src = "{{url\_for('static', filename='images/'+item.img\_name)}}" alt="No Image Uploaded" style="height: 250px !important;">

                    <div class="card-body">

                        <h5 class="card-title">{{item.name}}</h5>

                        <p class="card-text">{{item.start\_date.date()}}<br>{{item.username}}</p>

                            <form action="#" method="POST">

                                <button type="submit" name="id" value="{{item.item\_id}}" class="btn btn-primary">More info</button>

                                <button type="submit" name="sold" value="{{item.item\_id}}" class="btn btn-primary">Sold</button>

                            </form>

                    </div>

                </div>

            </div>

        {%endfor%}

    </div>

This next section is in a container with a grey outline, it has a “Your current items” header. It then Loops over each Item that was passed in the back end using a Bootstrap Card (GetBootstrap.com, 2021) displaying the item name, start date and username. There are two buttons. One for more information and One for Sold

# Testing

## Testing Regime

### Project Explanation

This Project is named Pez-Bay. This is a Flask Web App; it is an Auction Website that includes User Accounts with easy registration and login features. It allows these users to sell Items on a Sell page and to view their profile that includes their user details and the items they have sold or are currently selling, With the added ability to mark their items as Sold and end any Auction. Pez-Bay allows you to view all current selling items on the Home page, you can see more information on each item and can Bid on items too.

### Scope

The tests will be split into Sections, testing different pages. These Tests should commence over the period of 1 Hour These Include:

* Home page
* Register Page
* Login Page
* Items Page
  + Bidding on items
* Sell Page
* User Page
  + Sold Items

### Test Structure

* The Test being done
* The Input Data
  + Normal, Range, Erroneous
* Expected Result
* If the test Passed or Failed
* And Any Interesting comments

Home Page:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test** | **Input** | **Result** | **Pass or Fail** | **Comments** |
| a). Item Display | **/** | All Items currently on sale are displayed on the page |  |  |
| b). Item Data | **/** | The items displayed have the correct data including: Name, Date, And Username |  |  |
| c). Item Image | **/** | All images are displayed correctly and in boxed dimensions, If item has no image Alt text I shown |  |  |
| d). User Logged In/Out | **/** | If User is logged in Card Element shows More information Button, This Button is removed if User is not logged in |  |  |

Register Page:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test** | **Input** | **Result** | **Pass or Fail** | **Comments** |
| a). Input elements | / | All Input Boxes Correctly Displayed |  |  |
| b). Email text  validation | -[test@test.com](mailto:test@test.com)  [-test@test.com@](mailto:-test@test.com@)  -test.com | Accepts only emails, Requires one @ |  |  |
| c). Phone integer validation | 07972052216  067272727.88  test | Only accepts integers |  |  |
| d). Password Obscuring | / | Passwords characters are Obscured |  |  |
| e). User Already Exists | / | If User already in the User Table Regster page will flash “User already Exists” |  |  |

Login Page:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test** | **Input** | **Result** | **Pass or Fail** | **Comments** |
| a). Input elements | / | All Input Boxes Correctly Displayed |  |  |
| b). Username Validation | / | If The incorrect Username is Input the page will flash “User not found” |  |  |
| b). Password  Validation | / | If The incorrect Password is Input the page will flash “Incorrect Password” |  |  |

Items Page:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test** | **Input** | **Result** | **Pass or Fail** | **Comments** |
| 1a). Item Data | / | Correct information for The current Item is displayed this includes: Item name, date, user phone number, User email |  |  |
| 1b). Item Image | / | Correct Image is displayed if there is one to display |  |  |
| 2a). Auction Modal | / | Auction Modal button should appear bellow the item information |  |  |
| 2b). Current Bid | / | The correct Current Bid should show up in the Auction Modal |  |  |
| 2c). Bid Type Validation | / | Input should only take integers |  |  |
| 2d). Bid amount Validation | / | Bid must be above the current bid, if not then item page should flash “Bid can’t be lower than current bid!” |  |  |

Sell Page:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test** | **Input** | **Result** | **Pass or Fail** | **Comments** |
| a). Input elements | / | All Input Boxes Correctly Displayed |  |  |
| b). Name input Validation | / | Name must be entered |  |  |
| b). File input Validation | .png/.jpg/.gif  .exe  No file selected | File must be selected and must be image |  |  |
| b). Description input Validation | / | Description must be entered |  |  |

User Page:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test** | **Input** | **Result** | **Pass or Fail** | **Comments** |
| 1a). User Data | **/** | Returns all the Users correct data this includes: Username, Email, Phone number. |  |  |
| 2a). User Item | **/** | Users Items must all show up, With correct data, |  |  |
| 2b). Item Sold | **/** | Must show Sold Button and change item to Sold on button press, should also flash “Item sold!” |  |  |

# External Test

These Test were done on my teammate Joseph Less project:

Tests involving input fields will include 3 tests (NRE):

* Normal – normal ‘expected’ data
* Range – data on the edges of expected and erroneous data
* Erroneous – wrong datatype, anything that shouldn’t be accepted

1. **Register**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test** | **Expected result** | **Test data:** N / R / E | **Pass/ fail** | **Comments** |
| 1.1 – Register forms | Forms requesting user information | N/A | Pass | Takes all inputs correctly and adds new row to table correctly |
| 1.2 – Email input validation | Accepts valid email addresses, tells user if email is already in use | [example@email.com](mailto:example@email.com), [ex@ample@email.uk](mailto:ex@ample@email.uk), example3mail.com, ‘email address already registered’ | Pass | Only valid addresses were allowed, And email already in use was not allowed |
| 1.3 – Phone input validation | Only accepts integers | 02476123123, 4946825.341, ‘phonenumber’ | Pass | Invalid input was not allowed |
| 1.4 – Password validation | Passwords under 5 characters are not accepted | ‘password’, ‘pass’ | Pass | -~ - |
| 1.5 – Password confirmation | Message saying both passwords must match | Matching fields, non-matching fields | Pass | -~ - |
| 1.6 – Submit | Tells user the account has been made and redirects to the home page | N/A | Pass | -~ - |

1. **Login**

|  |  |  |  |
| --- | --- | --- | --- |
| **Test** | **Expected result** | **Pass/ fail** | **Comments** |
| 2.1 – Login forms | Forms requesting email and password | Pass | Email and password forms requested correctly |
| 2.2 – Login | Given an existing account, logs user in when submit is clicked and redirects to home page, when no account exists, tells user | Pass | Querys correct data from table and redirects to home page, when an invalid user is entered correct message is flashed |

1. **Sell item**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test** | **Expected result** | **Test data** Normal/range/ erroneous | **Pass/ fail** | **Comments** |
| 3.1 – Sell item forms | Forms requesting item information | N/A | Pass | -~ - |
| 3.2 – Price input validation | Accepts positive numbers to 2 decimal places | 20, 20.51, ‘Five’ | Pass | -~ - |
| 3.3 – File input validation | Only accepts jpg files | Jpg file, non-jpg file | Pass | -~ - |
| 3.4 – Submit button | Adds item to the database, redirects to the home page | N/A | Pass | -~ - |

1. **My items**

|  |  |  |  |
| --- | --- | --- | --- |
| **Test** | **Expected result** | **Pass/ fail** | **Comments** |
| 4.1 – My items page | Once items are added, my items page displays all user items with a photo, name, price, date added. | Pass | -~ - |
| 4.2 – Data displayed is accurate | data matches the item | Pass | -~ - |
| 4.3 – Actions button | When clicked, pops up a series of actions | Pass | -~ - |

1. **Home page**

|  |  |  |  |
| --- | --- | --- | --- |
| **Test** | **Expected result** | **Pass/ fail** | **Comments** |
| 5.1 – Home page | Displays all items with a photo, name, price, date added and username of seller | Pass | -~ - |
| 5.2 – Data displayed is accurate | Data matches the item | Pass | -~ - |
| 5.3 – Multiple users items | Items shown are from all users on the site | Pass | -~ - |
| 5.4 - Logout | Users can access home page without logging in | Pass | -~ - |

1. **More information**

|  |  |  |  |
| --- | --- | --- | --- |
| **Test** | **Expected result** | **Pass/ fail** | **Comments** |
| 6.1 – Item page | The expected item information | Pass |  |
| 6.2 – Data shown | Item photo, name, full description, date added, price/current bid(if one exists), auction end date/time | Pass |  |
| 6.3.1 – viewing as Item owner | Under the information is a link to the myitems page. When clicked, redirects user | Pass |  |
| 6.3.2 – Viewing item as buyer | When an item is viewed by users that don’t own it a bid button shows under the information | Pass |  |

1. **Marking as sold**

|  |  |  |  |
| --- | --- | --- | --- |
| **Test** | **Expected result** | **Pass/ fail** | **Comments** |
| 7.1 – Sold button | On the my items page, the actions button on each item displays a “mark as sold” option | Pass |  |
| 7.2 – Sold function | When clicked, the page refreshes and the item diplays a marker to say it has been sold | Pass |  |
| 7.3 – Home page | Once an item is marked as sold, it does not show on the home page | pass |  |
| 7.4 - Email | When an item that has been bid on is marked as sold, the user who bid on the item is emailed notifying them they have won | Fail | Could have something to do with the other account having a fake email as I didn’t have another one to test with |
| 7.5 – Sold actions | Once an item is marked as sold, the mark it as sold button is removed | Pass |  |

1. **Bid on an item**

|  |  |  |  |
| --- | --- | --- | --- |
| **Test** | **Expected result** | **Pass/ fail** | **Comments** |
| 8.1 – Bid pop up | When the bid button is clicked, pop up area to place a bid on an item | Pass |  |
| 8.2 – Bid input validation | Allows users to make a bid of the starting price, or 10p above the current bid if one exists  **Test data:** Relevant price, relevant price with 2 decimal places, “wlg%nlwng$” | Pass |  |
| 8.3 – Bid made | Current bid on the item page, home page and my items page is updated to the new value | Pass |  |
| 8.4 – Email | When a bid is made on an item that had a bid previously, the user who bid previously is emailed | Fail | \*\*Email will be sent to the same email address as in table 7 once email is amended\*\* |

1. **Delete an item**

|  |  |  |  |
| --- | --- | --- | --- |
| **Test** | **Expected result** | **Pass/ fail** | **Comments** |
| 9.1 – Delete | From myitems when delete item is pressed from the actions menu, the item is deleted from the database, not showing on the myitems or home page | Pass |  |
| 9.2 – Owner only | Trying to delete an item you don’t own returns an error  **Test data:** type: /delete/(an item id you don’t own) e.g 1 | Pass |  |

Almost all tests were passed and there was very good validation on all the inputs that were passed, The user experience was stress free and I had a very easy time testing this code. Emailing unfortunately failed but I have a susupicion its due to the other test user haaving a fake email (ii didn’t have another one to test with) so this is most likely due to my error. I would only add one more type of validation in the registration section: To add a User Validator as I was able to re register a user that was already in the table, although this was outside of the testing spec so this code still passed overall with flying colours.

# Quality Assurance

## Brief and requirements delivered

* The brief details a flask web application, Base on an Auction. Some features include the ability to sell items and bid on items from other users. Pez-Bay has been able to fulfill a majority of features excluding some small features that I didn’t have time to complete. The features that were in my scope were completed to a high standard and work simply and efficiently.

## Coding standards

* My Code was commented concisely and only where needed to stop repetition and increase readability, With multiline comments at the top of each program bullet pointing key features, inputs, outputs and single line comments only on key parts of my code that needed extra explanation. My Code is packaged meaning that it is split into modules and imported for my run.py function to use. This makes my program modular decreasing rewritten code an increasing readability and will allow less maintenance overall.

## Documentation standards

* I have tried to keep my documentation concise by splitting it into sections. The code explanation was also split into each function and its html counterpart to make the program easier to understand. I have tried to only write necessary details and kept only the important sections of my code in the explanation. I hope this will make it easier to read and stop it from being tedious to review. The testing regime details a clear process to follow and what tests should be done in which section. I Have reference where I have used code and where I have learnt from.

## Team details and support

* Our team had meeting for project planning, and another meeting after the code freeze to make sure what each member was supposed to do. We assigned roles to each member:

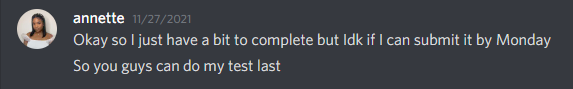
Table

Description automatically generated

Unfortunately, one of our members had to defer so our role diagram was changed. I ended up testing and Documentation expert for Joes Project and doing the QA evaluation for Nicks Project.

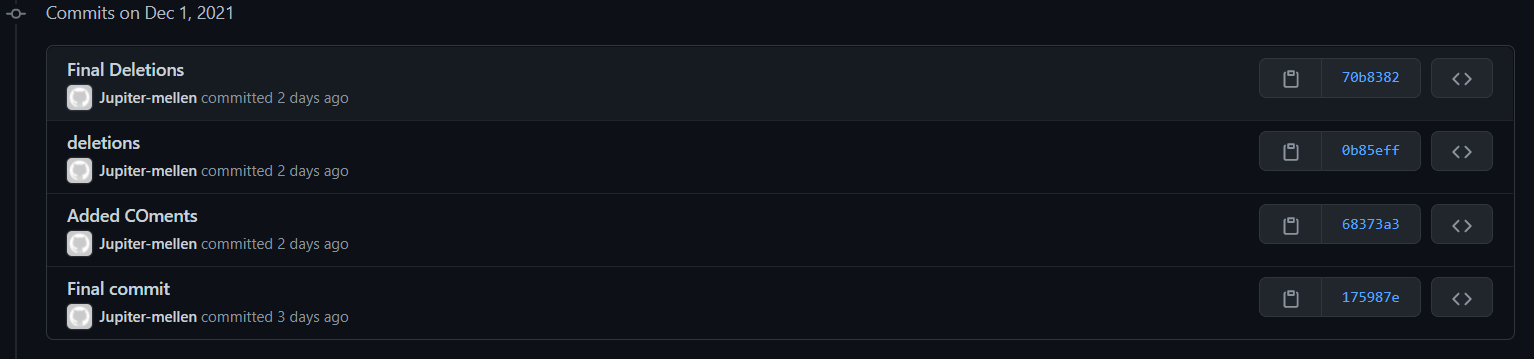
Timely deliver

My Delivery was not on time as at first, we were under the assumption that Annette was still going to submit her project we moved our deadlines back and waited until she was able to submit



We found out much later that she deferred and had to rearrange all our roles that took even more time to sort out.

My final commits were on the 1st of December for the rest of my team to use:



1. **QUALITY ASSURANCE**

// around 500 words

.1 QUALITY ASSURANCE STATEMENT

// Your statement about your work

.2 EXTERNAL QA EVALUATION

// Your evaluation of the external’s QA statement

# Documentation

## Documentation List

* Code with comments: [**https://github.coventry.ac.uk/mellenj/5001CEM\_Auction\_House**](https://github.coventry.ac.uk/mellenj/5001CEM_Auction_House)
* Code purpose (page 3)
* Code location (page 4)
* Installation instructions (page 4)
* Code explanation (page 4 – 20)
* Testing regime (page 20-27)
* QA statement (page 28)
* References (section 6)

Documentation Inspection

1. **DOCUMENTATION**

// around 500 words

.1 DOCUMENTATION LIST

// List all documentation you have included

.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**

// 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.