OCR COMPUTER SCIENCE H446

COMPONENT 03

PROGRAMMING PROJECT

H446-03

By Dhyan Patel

Candidate Number: 7184 | Centre Name: Preston Manor School | Centre Number: 12336

Table of Contents

Analysis

Problem & Client Identification:

Living in a densely populated area, there are many stores which need to provide daily goods to local residents. While contacting the nearby businesses about their inventory solutions, my client Ranchoddas Shamaldas Chanchad has requested that I create an easy and efficient way to manage details of customers, stocks, and inventory to ensure products are kept in check and don't run out.

There are ways to solve this problem without using computational methods, such as a paper-made database involving several spreadsheets that are filled out and edited as stock changes. This method is extremely cheap however there is no way to keep track of what stocks need to be reordered other than memory.

For these reasons, a computational approach is preferred. Algorithms can be used to sort and search data. Changes in stock levels can be visualised to provide data on what products are popular and will be prone to running out. All in all, this will make data tracking less complicated for the business. The app will be web-based so that the client can access it from any device. I will have to create a database to keep track of products.

Using computational methods will allow me to produce code more efficiently. Not all computational methods are suitable, however, so I will be analysing each one to see their suitability to this project.

Computational Methods	Advantages	Disadvantages
Thinking Abstractly	Abstraction is used to hide components of the program which the user does not need to see, e.g the user does not need to see an algorithm running in the background. This allows for a program to be more approachable for a user as the complexity is hidden. The GUI will have 'buttons' which will run code that the user will not see, despite it being there. Abstraction can also be used in the creation of a program to lower the complexity of code, reducing the number of lines which saves system resources	As the solution begins to get more complex, it will become more and more difficult to hide processes that users do not need to see, this can result in overcomplicating the code if the programmer does this for all tasks, rather than making it more efficient.

	such as processor time and RAM.	
Thinking Ahead	Thinking ahead is necessary for creating a program as you must plan the program before you code it. It allows you to recognize preconditions for tasks, e.g knowing that a database needs to interface with a GUI needs to be able to interact with a database, and designing it around that. Thinking ahead also allows me to see if there will be any code that may be reused in the program, which is beneficial as I will not need to write and test all of my code as it has already been tested.	Some problems are difficult to model before you start coding them, so all inputs and outputs may not be clear at the start. Thinking ahead will not help in these situations and the programmer will have to tackle the problem head on.
Thinking Procedurally	Using stepwise refinement to break down tasks into smaller pieces will allow me to approach larger problems. Also ensures all inputs and outputs are considered, allowing data to go to the correct places in programs, e.g from one sub-procedure to another.	Procedural thinking is not always the most efficient way to carry out a problem, and it may make code less readable/maintainable if it has to follow a strict path.
Thinking Logically	Involves planning out decisions made throughout the program, which allows coding the program to be much easier, e.g mapping out all the inputs a user can make at a certain point in the program, then mapping out what will occur after one of those inputs are sent. This allows for all potential scenarios to be planned for.	Mapping out complex processes can take a lot of time as just a single decision can change the entire outcome of the program, so many time consuming tests would need to be made to ensure the program functions properly.

Thinking Concurrently	I can develop multiple modules at the same time, e.g. working on interface and login systems at the same time, as inputs that go into the login system will have to be submitted through the GUI. Another way I can make use of thinking concurrently is by testing solutions as I am developing them.	Working on multiple tasks at once could lead to overcomplicating the program, which leads to inefficient code.
Caching	Could be used to save data the user uses frequently, which will make it easier for the user, as that data can be loaded at the start of the program	System resources are occupied to load data that may not get used.
Backtracking	A way to work towards a solution partially and going back to the last most successful point if it doesn't work. If a particular solution does not need thorough planning then this method allows for faster coding.	For most solutions, backtracking takes a lot of time compared to other methods because of its recursive nature.
Data Mining	By analysing data, we can utilise techniques such as forecasting to predict future patterns in stock. This way we can pre-order items before they run out, check what products are most popular and do other data comparisons.	Can be complex to implement and this project may not have a very complex database, so implementing it may just be more effort than is necessary.
Divide and Conquer	Being able to split bigger problems into smaller subtasks makes coding easier as the criteria of subtasks is much easier to manage than the criteria of an entire problem.	Merging subtasks into a final solution is a difficult final step.
Heuristics	Works as a substitute for classical methods for when they are too slow. Allows for a quick solution that is functional. Sometimes building a thorough solution is not necessary when comparing the time that would need to	Solutions are not fully optimal and can even be completely incorrect, which can lead to inaccurate judgements.

	be put into it to the problem it is trying to solve. Here, a solution that is 'good enough' can be implemented. When a problem is intractable, heuristics is applied, while the solution is not fully optimal, it is given in a reasonable timeframe.	
Brute Force	When applicable, it outperforms other methods.	Evaluating many different solutions one by one to see which one is most suitable can be very time consuming, brute force is just a less effective version of backtracking.

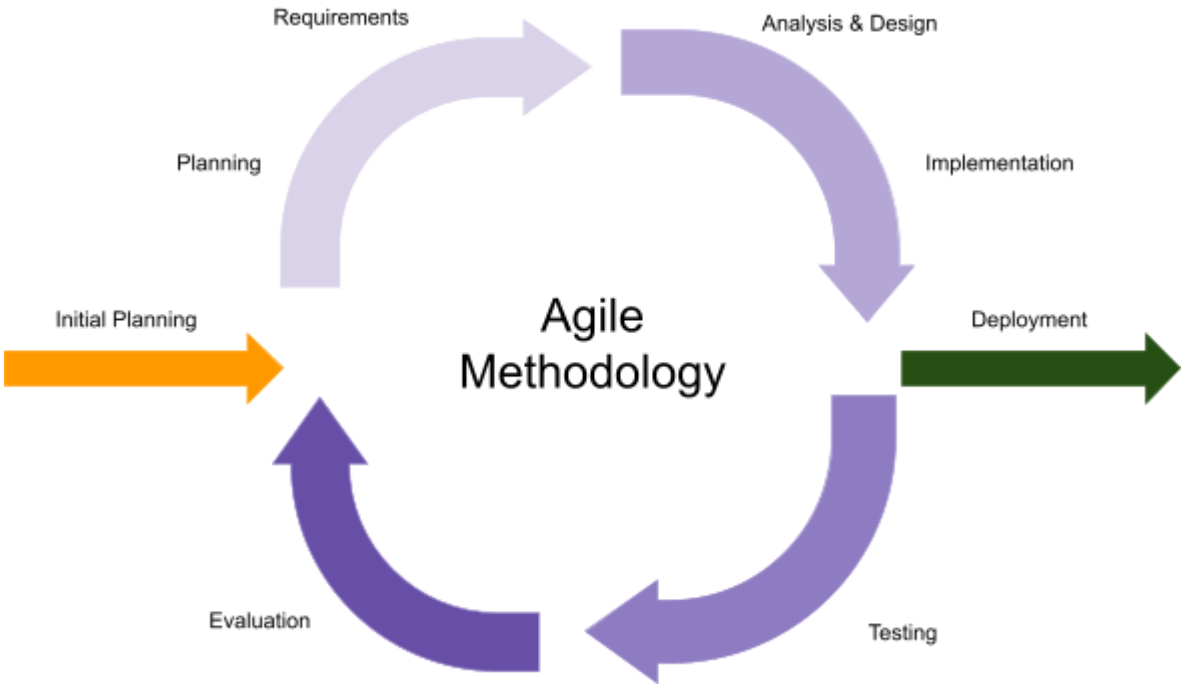
I will be making use of many methodologies. One of which is abstraction, which can hide detail, save memory, and make the program easier to use.

There will always be a need to hide unnecessary data from the user. When creating the GUI, code will be hidden behind easily understandable buttons, and abstraction can be applied while creating the design of the GUI too. By reducing the GUI to only what is necessary, system resources can be saved that would otherwise be wasted.
To make the product more client-friendly, abstraction will be used to display error messages in a format the user can understand.

Thinking ahead will be used throughout the development process, as it allows me to see the preconditions for each task, this will be seen in my use of flowcharts and other visual methods of displaying the project structure. You cannot create a database without having planned its structure and layout, so this method is essential to the project. By planning out the program structure, I can identify where I can generate reusable components, saving time when coding the project.

Thinking concurrently is an unnecessary method for this project. Trying to implement this method will lead to an unnecessarily complex program which will take too much time to develop, most processes will need data from previous processes in order to run so concurrency will not be applicable to most of the project anyways. Due to this, thinking procedurally is very important to this project's development. It will allow me to plan the program structure more easily using flowcharts, and will be useful in the generation of functions and subprocedures.

Data mining allows seeing patterns and trends in data, e.g what stocks are more popular, which stocks can be discontinued etc. Data mining algorithms are complex mathematical algorithms but this is a small project so there is no need to waste system resources and development time creating them. While data mining algorithms will increase the business's efficacy, the investment needed is too much.



(Source: original)

For my methodology of choice, I will be continuing forward with the agile methodology. It features an iterative approach to project management. It allows modular development of the program, where I can frequently bug test each part of the program individually before it is implemented. Due to its iterative nature, I can talk to my client frequently making sure the product meets their expectations, however their collaboration will be important for this to take place.

Successive methodologies such as waterfall are easier to use however they assume that customer requirements are gathered at the beginning of the project. In reality, this is a very difficult thing to achieve as requirements can change during project development, an iterative methodology allows me to change the final product during development.

This narrows down the methodology of choice to just two, spiral and agile. I have chosen agile over spiral as code is not checked as regularly in spiral. Code is developed, a prototype is produced then testing and evaluation takes place. This means things like functions and subroutines are not individually tested before being put into the program, making it more difficult to maintain the code.

Agile methodology also makes use of thinking logically. Thinking logically is meant to show me how a problem can be broken down into smaller problems, each new subproblem can be solved in each of the development iterations.
As each iteration can be its own self contained module, it is extremely easy to save code to be reused as it can be used later in development, this way time is saved and I can be confident that the code I am using has already been fully tested.

If the project is not to the stage where the client is satisfied, it is possible to simply start another development iteration. This ensures that the client will receive a usable product that is bug free and meets all of their requirements.

Stakeholders

My primary stakeholder is Rancho, who owns a convenience store nearby. At his store he sells snacks, alcoholic beverages, fizzy drinks, and essential groceries. Any other retail/grocery business will be able to utilise this program as long as they have inventory to track.

The database will have fields fillable with products which any business is able to utilise. Abstracting SQL queries using a simple UI will allow the program to be very user-friendly so less training is required for staff, another stakeholder.

Staff in the store will benefit from this program as inventory is tracked on the database, meaning they no longer need to search the store to see what needs restocking, as purchases are tracked. The program can also highlight when something needs to be restocked as its inventory count gets low. Rancho and his employees can also view graphs which will visualise stock data and allow them to make better informed decisions on what to order.

Rancho will also be able to access the program from home, meaning he will not have to enter the store to check stock. On his days off he can check in on stock and order more as needed from wherever he is. This will also stop him from losing out on sales as long as he is diligent in ordering or the system self orders for him, as stock will never be missing and he will have a continuous stream of sales.

Rancho's store's customers are the main stakeholder for this program. By making the store's stock management more efficient, less items will be missing less often. Otherwise they would have to travel to other stores, increasing time spent shopping. A potential feature for this program would be to allow customers to view if items are in stock or not before they reach the store but this may be limited due to security concerns by Rancho.

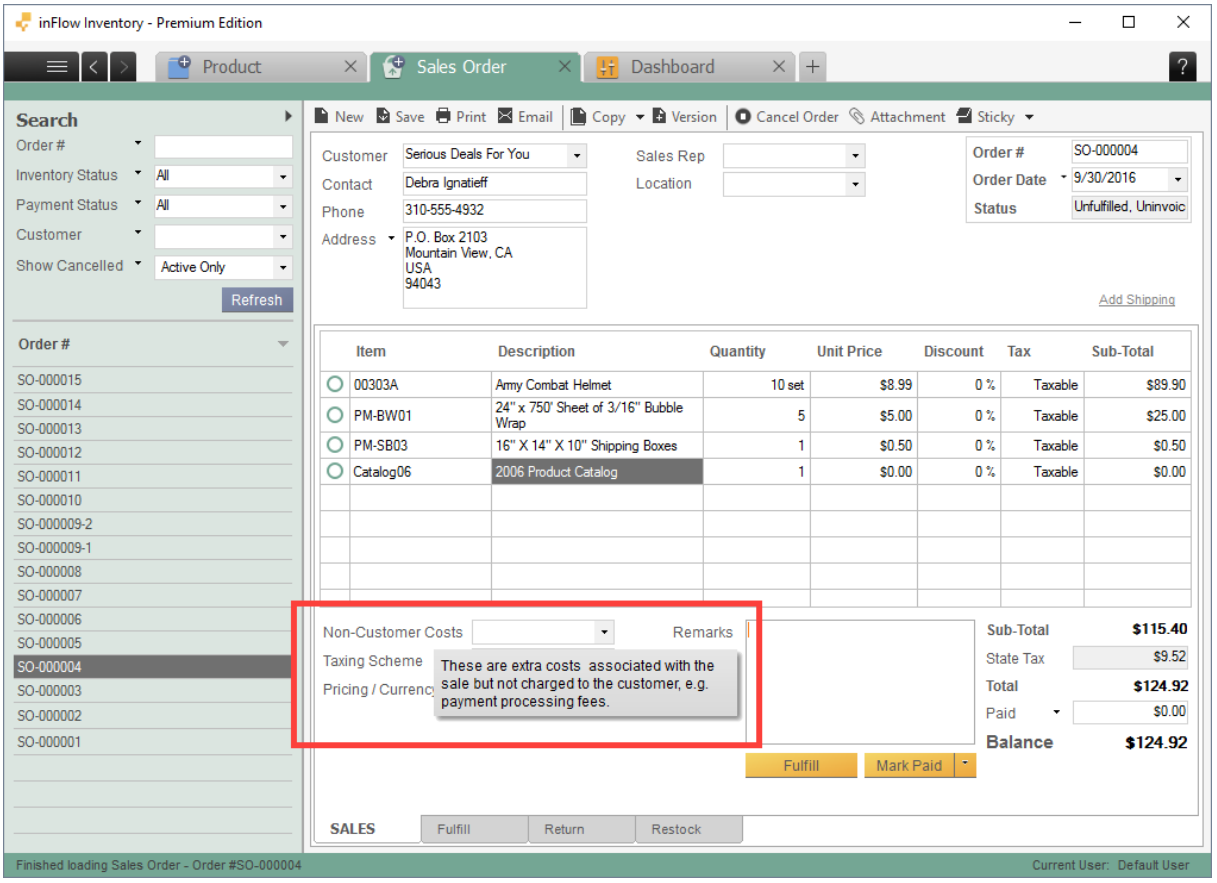
Research

So that the solution provided is of use to Rancho, I have researched pre-existing solutions to the problem so that I could analyse them and maybe add elements I have not yet thought about.

Existing solutions

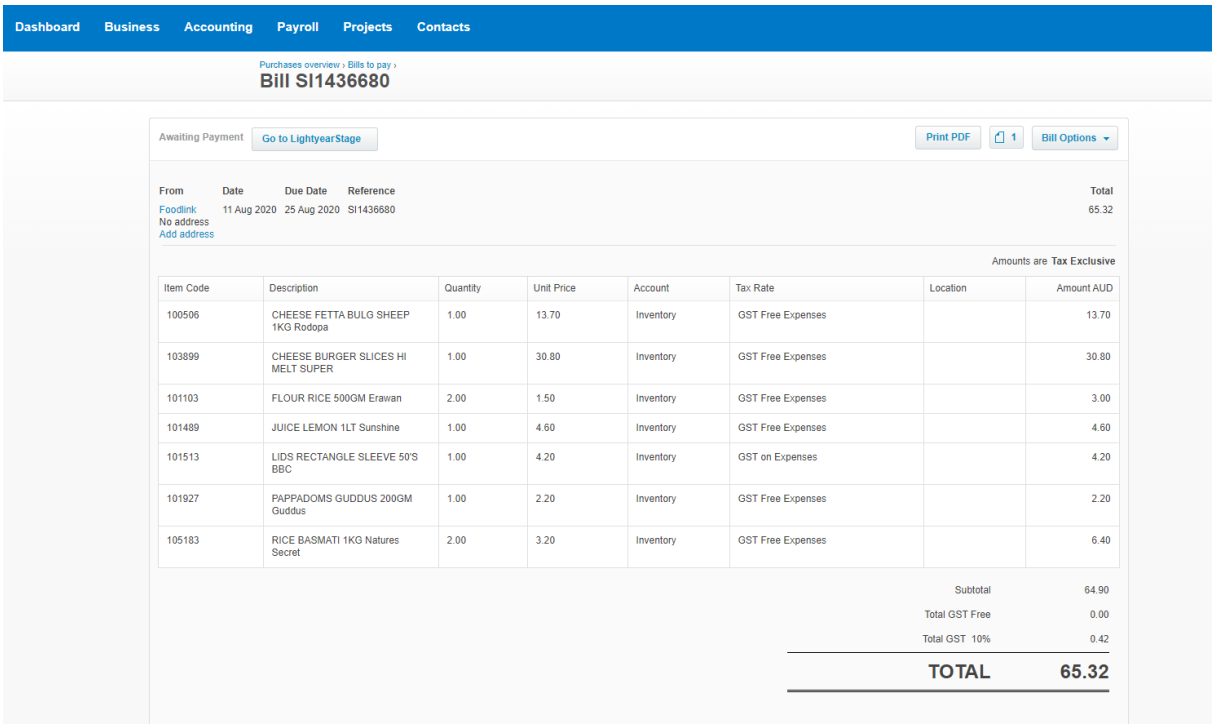
inFlow Inventory:

There are many tools available in this software which help with stock management and their feature list is massive. While this software is versatile it contains a lot of redundant features which would be useful for a larger business but for a local convenience store features such as asset management, making its pricing a waste. The number of users that can be logged onto the software simultaneously is also limited and placed behind subscription paywalls. Many of the features also require other softwares being purchased to function so to make full use of inFlow would require a lot of set up, cost and maintenance.



Xero Inventory Management:

Xero is a New Zealand-based technology company that provides cloud-based accounting software for small and medium-sized businesses, their software meets many of the requirements and is easy to use as an end user, however the downside is that this makes the software very simplistic. It also requires a monthly subscription which will eat into profits constantly. As the software is hosted by xero any downtime on their servers will heavily affect Rancho's business. It is better for a longstanding business to have bespoke software which caters to their exact needs. The xero GUI is very easy to navigate so I will be taking some inspiration from them.



Dashboard Business Accounting Payroll Projects Contacts

Purchases overview › Bills to pay ›

Bill SI1436680

Awaiting Payment | Go to LightyearStage | Print PDF | 1 | Bill Options

From	Date	Due Date	Reference	Total
Foodlink No address Add address	11 Aug 2020	25 Aug 2020	SI1436680	65.32

Amounts are Tax Exclusive

Item Code	Description	Quantity	Unit Price	Account	Tax Rate	Location	Amount AUD
100506	CHEESE FETTA BULG SHEEP 1KG Rodopa	1.00	13.70	Inventory	GST Free Expenses		13.70
103899	CHEESE BURGER SLICES HI MELT SUPER	1.00	30.80	Inventory	GST Free Expenses		30.80
101103	FLOUR RICE 500GM Erawan	2.00	1.50	Inventory	GST Free Expenses		3.00
101489	JUICE LEMON 1LT Sunshine	1.00	4.60	Inventory	GST Free Expenses		4.60
101513	LIDS RECTANGLE SLEEVE 50'S BBC	1.00	4.20	Inventory	GST on Expenses		4.20
101927	PAPPADOMS GUDDUS 200GM Guddus	1.00	2.20	Inventory	GST Free Expenses		2.20
105183	RICE BASMATI 1KG Natures Secret	2.00	3.20	Inventory	GST Free Expenses		6.40

Subtotal	64.90
Total GST Free	0.00
Total GST 10%	0.42
TOTAL	**65.32**

Client interview

I prepared a questionnaire to gain more insight into the problem, to see what features Rancho wants prioritised for this software and other miscellaneous information. I have listed the questions and the justification of why I want to ask about them below.

Question 1: What is your current solution and how is it meeting your requirements?

This allows me to understand what methods are already working for the client and allows me to make a solution that will be comfortable for the client to use.

Question 2: What do you find difficult doing with your current solution?

By finding out what the client finds difficult I can make a more suitable solution to solve them.

Question 3: Do you have both a computer and internet connection available at the store?

By finding out the hardware limitations I can immediately cross out certain solutions.

Question 4: Are there any additional features that you would like to see in the software?

I can implement features that the user wants and create a better product.

Below are the questions and answers:

Question	Answer
What is your current solution and how is it meeting your requirements?	Stocks are recorded after they are purchased. Low volume items are tracked by hand on pen and paper as they are bought and high volume items are checked on the shop floor and in the storage area to see if they are running low. An order is put into the supplier if stock is running low.
What do you find difficult doing with your current solution?	Staff can overlook missing stock since it isn't automatically tracked. If items go missing or stolen then we will not know as we are not tracking how many items have been sold.
Do you have a computer and an internet connection available at the store?	Yes
Are there any additional features that you would like to see in the software?	Allow multiple people to access the software

Solution

The proposed solution will be a web app that will work off user inputted data and data from the tilling system. The user will be able to manage the inventory by adding stocks, removing stocks, and editing information on stocks.

A login system will be present as this store hires multiple employees, creating the need for different access levels.

By making it a web app, the client will be able to access it from any location given they have internet access. This allows them to view stock levels and be aware of what is going on in the store, and if they need to order restocks while not being in the store. Due to it being web based, any device with a standard internet browser will be able to access the program. This eliminates the need to match hardware and OS requirements for the client's devices. There will be different levels of access for each type of user to ensure the permissions the store owner has are different from what a staff member who doesn't need unlimited access to the program.

Limitations of the solution

As the solution is to create a web application, the user will require a device with an internet browser to access it. The store already has an internet connection and a local network, meaning if network connectivity ever drops, the program and server will both still be accessible by setting up a LAN from the server. However this would mean the server cannot be accessed from outside of the store, so changes to stock level would have to be made in store.
Another limitation is the server itself. If the local network goes down or the server stops working, the program is no longer accessible. Running the server also incurs some cost in the form of electricity, as it would be running 24/7. Getting the hardware for the initial setup of the server will also incur costs for the client, however it is better than hiring a hosting provider as outside services will be more unreliable than having a local server as an internet connection blackout would only lead to partial loss of features rather than losing complete access to the program.

Requirements

Requirements Specifications

#	Requirement	Justification
Design		
1.1	GUI: Menus Buttons to select options Input boxes (text and numbers) Labels	A GUI interface will make the program easier to use and it will make abstraction possible so any complex processes are hidden from the user. There will be a menu based interface as those are commonplace, meaning the user will not be unfamiliar with the program. Input boxes are used to input information.
1.2	Use of styles: Reasonable font selection and size Consistent theme Buttons and text boxes coloured appropriately	Makes using the program easier as consistent font and theme usage means user will not spend a long time trying to distinguish between different fonts. Buttons having contrasting colours will make them easier to spot.

		Neutral colours will aid in productivity.
1.3	Use of tables	Easiest way to represent a database is to use tables. This will make the inventory easy to view.
Input		
2.1	Login fields need inputs: User's name User's password Register new user account by entering user details	Used to add security to the program so nobody unauthorised can edit the database. Also allows the creation of new user accounts for other staff in the store.
2.2	Searching through stock: Product ID Search fields	Remembering unique IDs for every product is impossible so SQL will be utilised to find products depending on the parameters the user selects using the search fields.
2.3	Alterations of the database: Change stock level	New stock can be added after it is acquired and can also be removed.
Processing		
3.1	Login page: The username will be referenced in the credentials table and then the password will be matched against the input, leading to the program continuing as normal or an error message	Depending on the access level the credentials have, the program will be different.
3.2	Changing of database when user edits records	When data is edited or added the database should be accurately updated with the new changes on the corresponding record(s).
3.3	Data validation checks: Make sure correct data types are entered into database Prevent security threats on login page	Make sure only valid data types can be entered so the program functions as normal. Invalid data can break the database.
Output		
4.1	Message if login has failed	User should be alerted if their login fails so they may try again

4.2	Data validation errors	In the case that the user enters non-valid data into the database they will be alerted as the data is rejected.
4.3	Display stock search results	Queries can be easily made and displayed when the user decides to make a search.

Software Requirements

Requirement	Justification
Server OS: Windows 8 or higher	While the server software would work on Windows 7 and higher, Windows 7 has many security vulnerabilities as it is unsupported by microsoft. Program will probably function on mac and linux devices through the use of interpreters or virtual machines, but it is unrecommended and untested.
User devices: Web browser capable	To access the program on the server, a device must be able to use the internet, via which they will access the web app on the server. This eliminates all other OS and hardware requirements for user devices.

Hardware Requirements

Requirement	Justification
Computer with 4 GB RAM, 2 GHZ processor and 2 GB of storage space free. Computer must possess networking capabilities to serve as a server.	Server requires enough RAM to load the OS, program as well as data on stocks. This is the minimum spec requirements for the server which the program will be running on, more ram and a better CPU will always help to carry out changes made to the database.
Monitor/touch screen output	Used to view the program and information on stock
Mouse/touch screen	Used to navigate the GUI which is based on menus and buttons

Keyboard/touch screen	Used to enter information such as login details and enter data

Success Criteria

This is a measurable set of goals for the project which I plan to reach.

#	Criteria	Why/How
1	Login page and levels of access for different credentials	Allows only some staff to have access to the entire program, so something like temporary employees can only update when stocks go down while the owner and a manager will have full access to the program. The GUI used should be simple and password will be hidden behind (*) to ensure it is unseen.
2	Use of a CSV file so the software can interact with the tilling system	Easier to keep track of stock if software interacts with tills.
3	Stock management: 1. Update stock levels 2. View product information 3. Update product information	Allows user to set stock levels, and to amend mistakes on products that have been entered.
4	Error messages: 1. When input is sanitised 2. Wrong login details entered	Alert the user on invalid inputs so they may try again. Alert the user when login details are incorrect so they can amend them.
5	Software must be accessible from a variety of different devices from different locations	So the software is accessible from the client's home on a home device if needed, as this is a web app this is very achievable.

Design

Decomposition allows for the breaking down of big tasks into smaller subtasks. By breaking a problem down into smaller modules, each smaller problem can be analysed much better, and it also structures the problem.
To decompose problems. I will use flowcharts which will visualise all the different modules and what part of the program they are relevant to. This will help me code much more efficiently. Top-down design is effective in identifying all the main components that make up a problem so I can approach them as free-standing modules to implement them with different methods such as subroutines.

Database Design

Everything in the project relies on accessing a database so it is essential that the database be the first thing that is created.
I have opted out of using a flat file database, instead opting for a relational database. Flat file databases can result in redundant data, make stocks more difficult to update due to its structure and make complex queries troublesome to carry out. A relational database allows me to make multiple tables connected by keys.

To begin I created a simple layout showing the tables and their attributes.

There will be a table for information on users containing their credentials as well as identifying information about them. This table will be accessed if a user attempts to log in.

User Tbl	**Stockcounter Tbl**	**Products Tbl**
UserID	ProductID	ProductID
Name	Stock Left	Product Name
Password	Critical level threshold	Pricing
Admin		Product Barcode

The user table will only be used to hold user credentials and login information.

The products table will be used to differentiate what each stock is while the stockcounter table holds information on how many of each product is left in stock.

The primary key of the products table will be used as a secondary key in the stock counter table however it still will be a completely unique field in the stock counter table. This creates a one to one relationship between the stockcounter table and the products table.

This layout is how I will maintain simplicity while still keeping the database efficient. It is the most optimal solution, there are no overlaps in data.

User Table			
Field	**Data Type**	**Validation**	**Justification**
UserID	Integer	Primary Key Not NULL Unique	This field will be the primary key used to identify the user, so it must be unique and a user record cannot exist if it is not present.
Name	String	Unique Not NULL String limit=100	The name is used to both log into the program and identify the user, so it must be unique and present. The string limit is set to 100 for the sake of completeness, while a name that long will probably not exist, this prevents an input of a large size from accidentally being submitted.
Password		Not NULL String limit= 150	Must have a password for login credentials. String limit of 150 but unlikely any password will reach that length.
Admin	Boolean	Not NULL	This field has only a true or false value to designate a user as an admin. It cannot be null so the program knows what access level a user has.

Stockcounter Table			
Field	**Datatype**	**Validation**	**Justification**
ProductID	Integer	Secondary Key Foreign Key Not NULL	This table lists the stock count of each product. To identify which product is

			being referred to, this field cannot be empty thus it is Not NULL. The validation test for uniqueness was already carried out in the products table therefore it is not needed here.
Stock Left	Integer	Not NULL	Cannot be left empty as it displays the current stock level. If stock runs out, instead of being empty it will simply be listed as the value ‘0’.
Critical level threshold	Integer		Optional field that can be left empty. Used to alert the user that stock is running low once the value in the stock left field reaches or falls below the critical level threshold.

Products Table			
Field	**Datatype**	**Validation**	**Justification**
ProductID	Integer	Primary Key Not NULL Unique	Main identifying key for each product so it cannot be left empty and must be unique for each record.
Product Name	String	String limit=150 Not NULL Secondary Key	Product name is used as the secondary key and must not be missing as a human will not be able to tell what product the record is storing by looking at the primary key, which is just an integer.
Pricing	Numeric Value	Not NULL	Numeric value is used instead of integer as price can be a floating number

			with decimals. It must be filled in otherwise the product would be sold for free.
Product Barcode	String	Not NULL String limit=13 Secondary Key Unique	Not NULL as the barcode is the main identification standard used in Europe (the EAN numbering system). It also functions as a secondary key as it is unique to each product.

My product table has 5 columns, one of which is the barcode column. After doing some research on barcodes and the datatype I could store one under, I found that they are known as EANs (European Article Numbers), which is a standard describing a barcode symbology and numbering system used in global trade to identify a specific retail product type, in a specific packaging configuration, from a specific manufacturer.

The standard used for retail products is the EAN-13, which is exactly 13 characters long. The field for barcodes in the products table is exactly 13 characters long, so I have made the code more efficient by carrying out a validation test in the table itself.

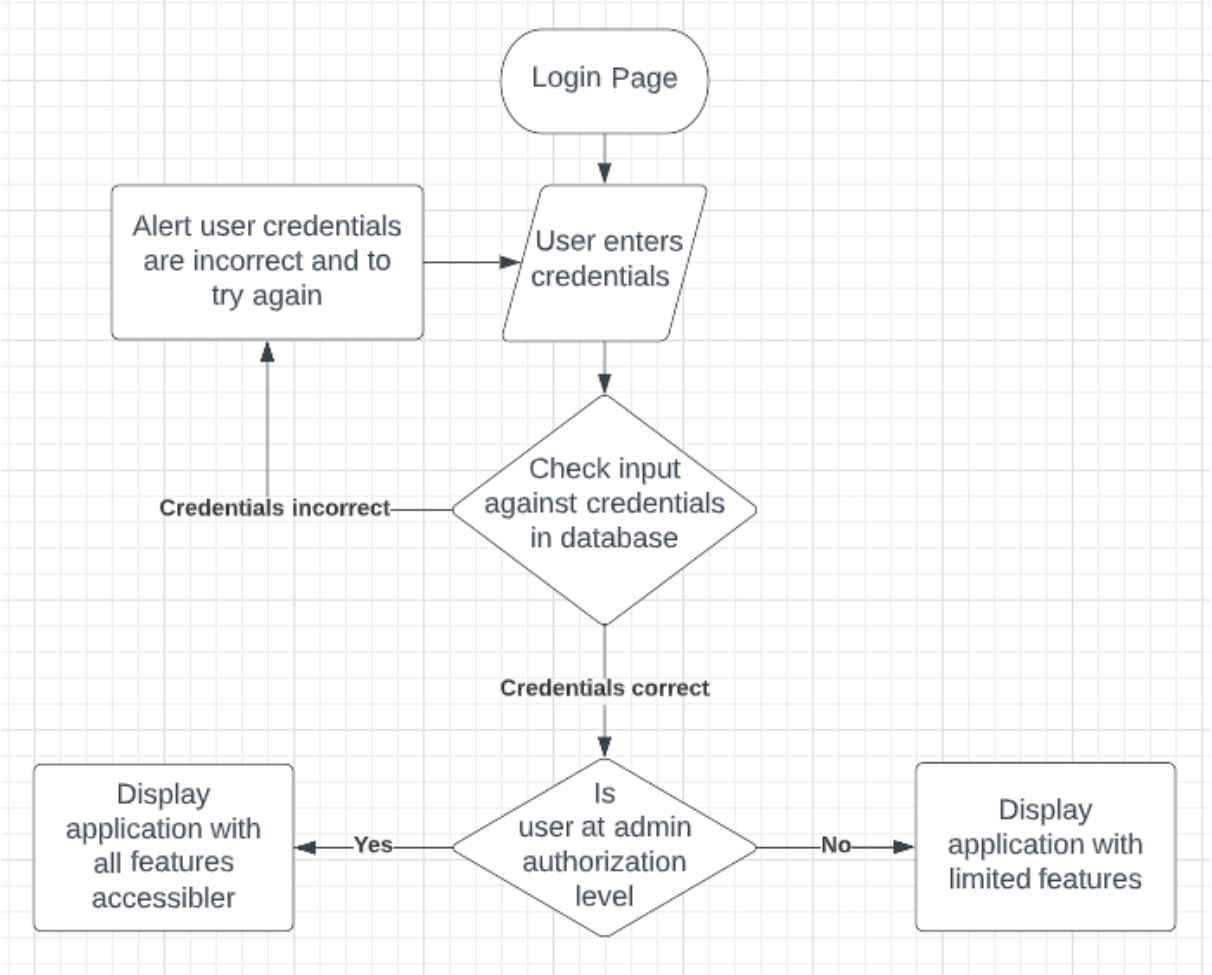
```
class product(db.Model):
    ProductID = db.Column(db.Interger, primary_key=True)
    ProductName = db.Column(db.String(150))
    Pricing = db.Column(db.Numeric())
    barcode = db.Column(db.String(13))
    image = db.Column(db.String(300))
```

Login Page

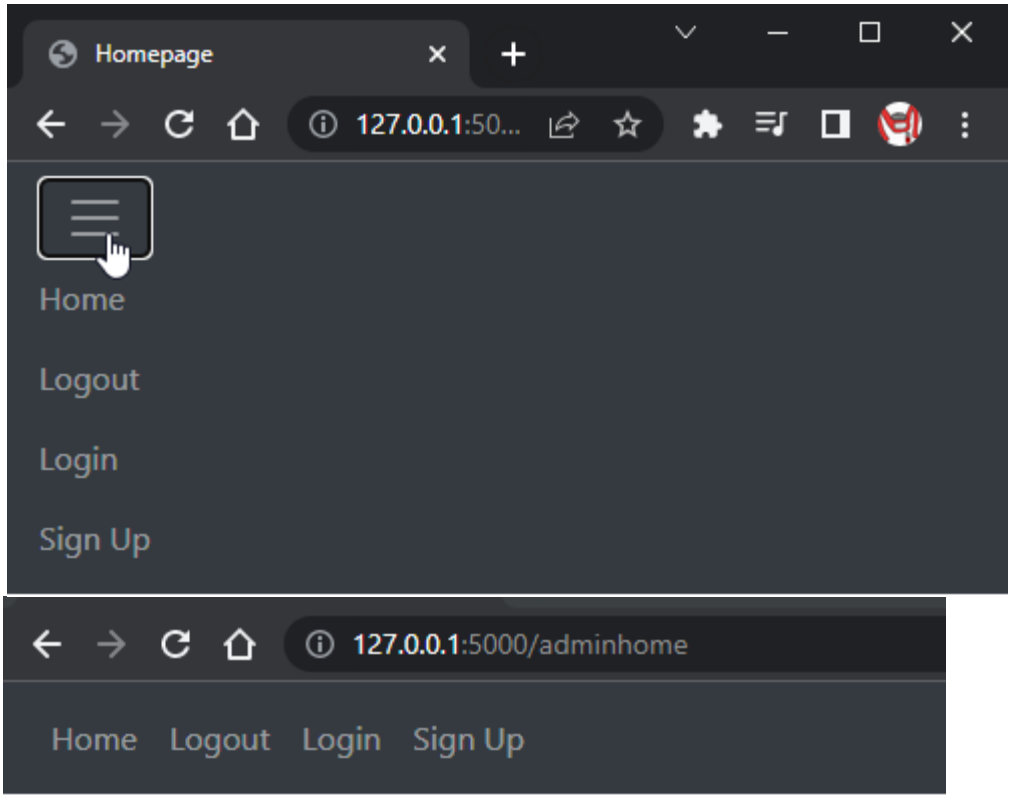
Once the program is opened, the user is presented with a login page. There are three input options for the user, the username, the password, and the enter button.

Upon entering details the provided username and password are checked against the user table to see if they are correct.
The entry for the username is pulled up, and the password for that entry is checked against the entered password.
If the provided details are incorrect, the program displays an error message citing that the login details are incorrect. The user will be able to try and enter their details again.

An SQL will be carried out at this part of the program and there will only be two possible options, a TRUE value, or a FALSE value which dictate whether the user can log in or not

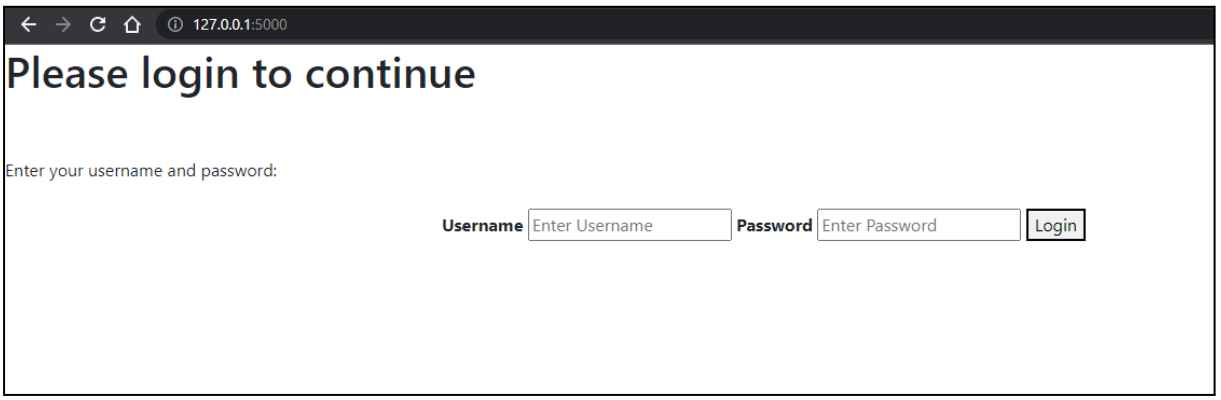


Once the username and password correspond to the correct details listed in the user table, the program reads a value in the Admin field in the user table to check if the user has admin permissions. If admin=TRUE, the user has full access to the program. This is to ensure only experienced staff and the store owner can do actions such as add stocks and create accounts.



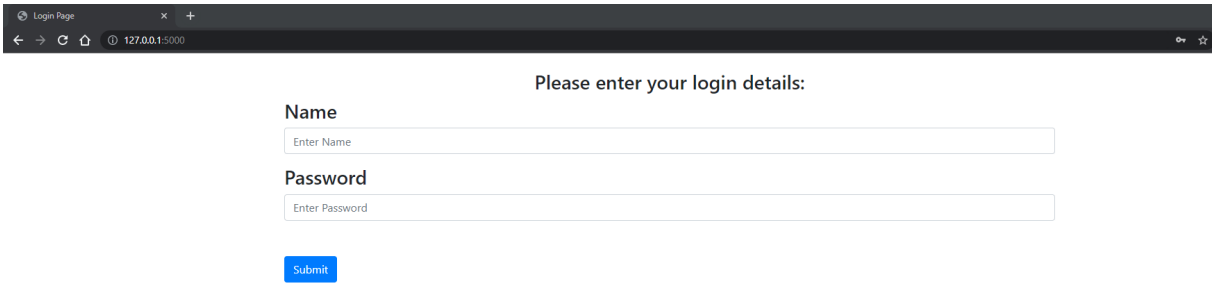
Usability features

Shown above is the basic navbar. The buttons on it are currently place holders until the application is built upon further. To ensure the application is easy to use on all types of devices regardless of screen size, I have added a navbar toggler button which will collapse and expand the navbar on click/tap. This allows the application to be used on smaller devices such as phones and tablets. The button only appears when the screen/window size is too small to fit all the items on the navbar to ensure no space is uselessly taken up.



Pictured above is a mock up for the login page of the application. The theme for the application was meant to be kept plain and simple to encourage productivity but this GUI is harsh on the eyes, I will be making amendments to it as per Rancho's requests.

After meeting with Rancho, I have created a new GUI design which is easier on the eyes and more user friendly.




```python
if request.method== 'POST':
    name=request.form.get('name')
    password = request.form.get('password')

    user= Users.query.filter_by(name=name).first()
    if user:
        if check_password_hash(user.password, password):
            flash('Logged in successfully!', category='success')
            login_user(user, remember=True)
            return redirect(url_for('views.home'))

        else:
            flash('Incorrect details, try again', category='error')
```

This is the backend for the login page, there is a post sent to the server containing the login credentials entered.

If the credentials are correctly matched, the user is logged in using the function login_user. This designation dictates what views the user can access, and I have used it to ensure the application is secure.

```python
#LOGOUT
@auth.route('/logout')
@login_required
def logout():
    logout_user()
    return redirect(url_for('auth.login'))
```

Pictured above is the logout page, you cannot logout without being logged in therefore I have added @login_required to the view.

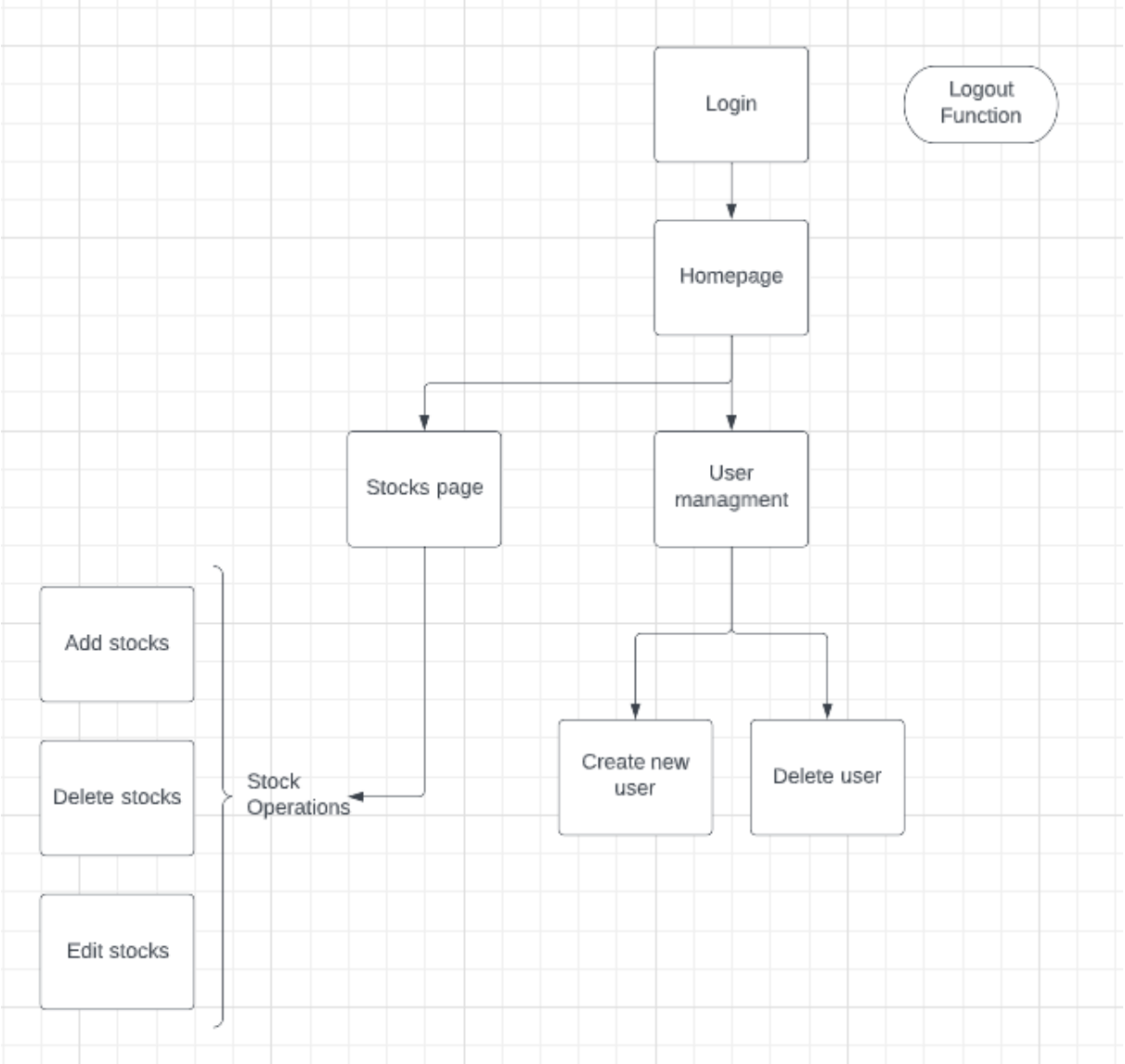
```
login_manager= LoginManager()
login_manager.login_view = 'auth.login'
login_manager.init_app(app)

@login_manager.user_loader
def load_user(id):
    return Users.query.get(int(id))
```

If a user is not logged in at any moment, they will be redirected to the login page, and that will be the only accessible page for them. Login manager is initialised directly after models are imported and the credentials database is initialised.

Homepage and Program Flow

From the homepage the user can go to the stocks page or user management.
The stocks page will have a table and link to 3 forms which can be used to edit the database.

The user management page will display all users and actions that can be taken to edit users.



Stocks and Stock updating

Stocks page displays a table with sorting and search functionalities, there are 3 buttons which allow the user to manipulate the database in an approachable manner. Through the use of abstraction the user will only see a simple form for all 3 operations.

Editing stocks will require the user to enter the stock's ID, then enter what information they want changed. Only text boxes which are filled will edit fields in the selected record. Validation checks are carried out on all inputs and an error message will be displayed if inputs are invalid. If data is valid, the database is updated and a success message is outputted to show the action was successful.

Adding stocks will require the user to submit

Testing plan

To ensure the final product is free of bugs testing needs to take place. It will be used to filter out any logic and syntax errors made during coding.

Alpha testing:

A method of testing done by the development team to make sure the program functions as intended.

White box testing:
Done internally by the development team. Typically a test plan is drafted to ensure every path a program can take is covered, and the tester can view the code while testing. This covers most bugs that may arise.

Black box testing:
Testing carried out without knowledge of the code, this type of testing focuses only on the inputs and outputs of a program.

Beta testing:
Beta testing is an opportunity for real users to use a product to uncover any bugs or issues before a general release. Real life usage can differ heavily from what a developer assumes an application would be used. This way, more covert bugs can be identified so that they can be fixed.

The program does not require any installation or set up on the side of the client during the development process due to the nature of a web app, letting me run it on my systems. This allows me to see logs more easily and diagnose bugs as they come up without needing to wait to gain access to the store's server.

—----------

Testing and debugging is an arduous process and it would be incredibly inefficient to test/debug after the program is fully developed, but I have a modular program, allowing me to test each process separately.
After internal testing, I can release the program to my stakeholders, allowing them to use it and give feedback on the program. This allows me to find any bugs that I may have missed in my own testing.

Beta testing questionnaire	
Question	**Justification**
Are you able to log into the program without issues?	Ensures the database and login process are functioning as intended
Were there any errors that arose while trying to manipulate the stocks table?	As stock management is the main focus of this project, the client will be asked if there were any issues with manipulating the stocks table.
Are there any concerns with the GUI?	The client must be satisfied with the GUI as they will be using the program on a daily basis.
Any miscellaneous errors/issues/bugs?	To find out if anything needs to be fixed

To test the program I will need test data so I have filled in some fields on my tables:

User Table			
UserID	**Name**	**Password**	**Admin**
Automatically assigned	Alucard	qwerkz	True
Automatically assigned	Ingrid	WeWras	False
Automatically assigned	Caess1	feIJLf	False

Product Table		
ProductID	**ProductName**	**barcode**
Automatically assigned	Pepsi	0101010101010
Automatically assigned	Milk	2323232323232
Automatically assigned	Bread	1257890864378

Login page tests:

Test type	**Data type**	**Test data**	**Expected result**	**Justification**
Logging in normally using admin login	Normal	Username- Alucard Password- qwerkz	Logs into homepage	Login process needs to be checked to see if it will allow those with valid details to log in
Logging in normally using regular login	Normal	Username- Ingrid Password- WeWras	Logs into homepage	Login process needs to be checked to see if different access levels may cause them to not be able to log in
Leaving login fields blank	Erroneous	-	Error shown and cannot log in	Make sure a username and password are

				required to log into the program
Enter login details with a character missing	Erroneous	Username-Caess1 Password- felJ	Error shown (password incorrect) and cannot log in	Make sure program is secure and that an incorrect password flash is shown as error, in this case the username is correct but password isn't, this allows the program to access the record for the relevant user and read the password field on the database, so I am checking if the password auth is working as intended and will block incorrect passwords.
Completely incorrect details	Erroneous	Username- Afneuye Password- acvetjiujuh	Error shown and cannot log in	Covers all bases, system should show an error message and login will be blocked.

Navigation testing:

Input	**Data type**	**Expected result**	**Justification**
Click on log out	Normal	Logs out and program automatically redirects unauthenticated user to login page	Make sure the user is able to log out of the application after they are finished using it.
Click on user management	Normal	Leads to user management page	Make sure button is functioning
Click on stocks	Normal	Leads to stocks page	Make sure button is functioning

Click on stock add	Normal	Leads to stock add page	Make sure button is functioning
Click on stock edit	Normal	Leads to stock edit page	Make sure button is functioning
Click on stock delete	Normal	Leads to stock delete page	Make sure button is functioning

Stock edit page:

Input	Data type	Expected result	Justification
ID- 6 Barcode- 1234567890000 stockcount- 12.4	Erroneous	Errors flashed, incorrect ID will be the first erroneous data checked by validation code, stopping any changes being made to the database	Make sure records of a stock that does not exist are not changed
ID- 2 Barcode- 4564564564567 ProductName- Candy Stockcount- 800	Normal	Stock stored in 2nd row is updated according to only the data that is submitted and nothing else, so pricing will be left the same	Make sure null inputs are not processed on the stock edit page by the program, and confirm the database records are edited properly
ID- 1 Barcode- 123456789123456789	Erroneous	Barcode invalid, error message sent explaining barcode needs to be valid	Make sure data validation works
No data	Erroneous	ID is null, no action occurs, error message shown	Make sure system does not do anything wrong if no data is posted

Stock delete page:

Input	Data type	Expected result	Justification
ID- 3 Click submit button	Normal	Success message flashed to browser stating which stock was deleted	Make sure stocks can be deleted by the user.
ID- 789 Click submit button	Erroneous	Error message shown to tell user that the process was not executed	Trying to delete a record that does not exist may cause issues, so this test will be done to make sure the action is stopped

Stock add page:

Input	Data type	Expected result	Justification
ProductName- Cookies stocklevel- 45 barcode-096678456 4818 Pricing=2.4	Normal	New record is added called ‘Cookies’, user is flashed with a message confirming it was successful.	To see if user can add products and receive confirmation that it has happened
ProductName- 42mint stocklevel- 8.7 barcode-096678456 48182222222222 Pricing=AAAAAA	Erroneous	Error is flashed with message stating data is invalid.	Makes sure data validation is functional

User management page:

Input	Data type	Expected result	Justification
Click on add user	Normal	Leads to a form to add a user	Ensures button is functional
Click on delete user	Normal	Leads to a form to delete a user	Ensures button is functional

Add user page:

Input	Data type	Expected result	Justification
Name- yuo1 Password- chipmunks Password(confirm)- chipmunks Accesslevel- user	Normal	Creates a new user with the credentials entered, flashes a message confirming it has happened	Checks the user has a way to create new credentials and that there is a confirmation message once they do so
Name-yuo2 Password-chimunk Password(confirm)-chimuk1	Erroneous	Error message flashed stating passwords did not match, new record is not created in the database	Make sure validation on passwords is functional and that error message is shown properly

Delete user page:

Input	Data type	Expected result	Justification
UserID- 1	Normal	First record on database is deleted	Make sure credentials can be removed from the database by the user

UserID- 34	Erroneous	As there is no record with ID 34 an error will be shown	Make sure error is shown correctly if user tries to delete non-existent records

Implementation

I will be using python to code my solution, as python has a large variety of libraries that I can utilise. My IDE is vscode dev, which I have chosen due to the variety of extensions and developing tools available to it. It has a file explorer which can neatly lay out files for applications that are made using flask, making it much easier to see the layout of the program. Syntax errors are also highlighted and problems in the code are checked as you are coding, making sure easily avoidable bugs are caught as you are coding. It also auto formats code for HTML, removing menial effort while I create the GUI.
HTML is a standard markup language for web page creation. As this is a web app, it is essential I use HTML. The use of Django allows me to write python statements directly into HTML code with minimal effort.

Flask

I have chosen to use Flask, a web application framework written in Python. It has no pre existing database abstraction layer, form validation, or any other components but through the use of third party libraries and extensions I can gain the same functionality as other web frameworks.

The reason I choose Flask is its simplicity to use and its benefits over other frameworks such as Sanic, which runs faster than Flask, but upon adding any sort of functionality to the application Flask ends up being the more effective option.

Django

Django is another high-level python web framework that I will be utilising. As python was my language of choice, django is one of the most versatile options available to me, which is what puts it above other options. It is also more effective in fast application development, which will let me provide my client a product faster.

Flask-SQLAlchemy

Flask-SQLAlchemy is an extension for flask that adds support for SQLAlchemy to the application.
SQLAlchemy will handle database logic in this program. Usage of this toolkit will prevent me writing overly verbose statements to run queries or add/remove any records.

MySQL

MySQL is an open-source relational database management system. I need to use MySQL to create my database.

Werkzeug

A WSGI (handles how a web server communicates with web applications) application library that contains various essential utilities.
It will primarily be used to handle password hashing for the program to ensure the application is kept secure. Typically hashing would be difficult to code but this pallets library handles it for me, saving time.

Javascript

Used to store table data so it can be displayed and work with HTML.

Base page template

The app is structured so that it can utilise django. There is a base template which contains any css, scripts etc which will be utilised by every single web page.
The base template will feature a navbar used to navigate the program.

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

'Blocks' are used to define sections of code that a child template can replace. Main page content will be held inside a content block. Page titles will be held in a title block.

```
{% with messages = get_flashed_messages(with_categories=true) %}
{% if messages %}
  {% for category, message in messages %}
  {% if category == 'error'%}
  <div class="alert alert-danger alter-dismissable fade show" role="alert">
    {{ message }}
    <button type="button" class="close" data-dismiss="alert">
      <span aria-hidden="true">&times;</span>
    </button>
  </div>
  {% else %}
  <div class="alert alert-success alter-dismissable fade show" role="alert">
    {{ message }}
    <button type="button" class="close" data-dismiss="alert">
      <span aria-hidden="true">&times;</span>
    </button>
  </div>
  {% endif %}
  {% endfor %}
{% endif %}
{% endwith %}
```

Django allows the usage of python statements inside of html files. HTML allows me to flash a message given a condition occurs, the code above in the base template defines conditions for when a message should be flashed and defines the flashed message. Depending on why

a message needs to be flashed, the program will substitute a variable holding the message into the block {{ message}}.
I have also included script blocks for pages that may need to utilise scripts, e.g pages with tables.

Login page and module

```
<div class="container">
  <form method="POST">
    <h3 align="center"> Please enter your login details: <h3>
    <div class="form-group">
        <label for="Username">Name</label>
        <input
            type="text"
            class="form-control"
            id="name"
            name="name"
            placeholder="Enter Name"
        />
    </div>
    <div class="form-group">
        <label for="password1">Password</label>
        <input
            type="password"
            class="form-control"
            id="password"
            name="password"
            placeholder="Enter Password"
        />
    </div>
    <br>
    <button type="Login" class="btn btn-primary"> Submit </button>
</div>
```

A login page has a few components the user needs, a form and a submit button. It should be the first thing a user views after they try to access the application.

The above code shows a form which contains 2 input boxes and a button which posts the entered data to the server.

The username is held under variable ‘name’ and variable ‘password’.

In order to actually display the login page, I need to create a view that will render the HTML once a user enters the website.

As it is the first page that a user should see, I have stored it under the view ‘/’.

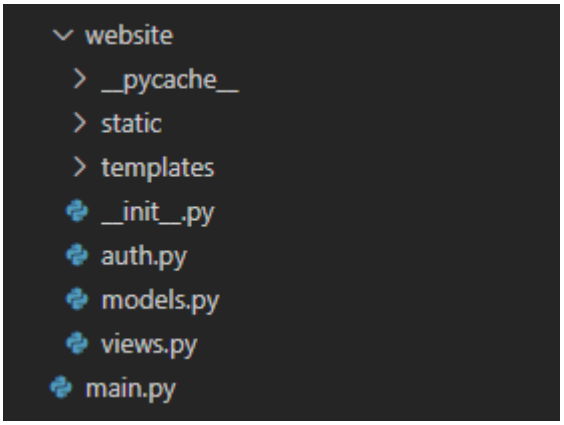
```python
#LOGIN PAGE
@auth.route('/', methods=['GET', 'POST'])
def login():
    if request.method== 'POST':                                     #IF USER POSTS, TAKE INPUTS
        name=request.form.get('name')
        password = request.form.get('password')

        user= Users.query.filter_by(name=name).first()              #QUERY DATABASE WITH INPUTED USERNAME
        if user:
            if check_password_hash(user.password, password):        #VALIDATE PASSWORD AND LOGIN
                flash('Logged in successfully!', category='success')
                login_user(user, remember=True)
                return redirect(url_for('views.home'))

            else:                                                  #ERROR IF DETAILS INCORRECT
                flash('Incorrect details, try again', category='error')



    data=request.form

    return render_template('login.html')                        #LOADS THE HTML PAGE INTO THE VIEW

```

Once the user presses the submit button, the server is specifically looking for variables listed as ‘name’ and ‘password’, and to keep the code simple, I have kept the variables consistent in the backend.
The server will query a database containing user details, taking the correct record. If the correct record is not found, the user is flashed with an error message telling them to try to login again. If the username is valid, the password is checked against a hash value. This keeps the program more secure as password records are not plain text but instead are encrypted. If the password is incorrect, an error message is flashed.

General layout of program

main.py will be used to launch the web app, and it is stored outside of the website module. The website folder itself acts as a module which can have imports made from it.
__init__.py will have everything needed to initialise the application.
Templates folder will store HTML pages, which will form the GUI. Static folder will store any data that is not code, so that it can be used in the program, e.g images, fonts etc.

auth.py and views.py designate views for urls which the user can visit. They designate which HTML templates are rendered when a user visits a page. They will also tell the app what to do with any inputs sent from the user side. auth.py will handle logins and logouts, views.py will handle everything actually related to the app.



Database and initialization of application

The database will be made and managed using SQLAlchemy, which is a toolkit that will facilitate communication between python and the database.

```
db = SQLAlchemy()            #CREATES DATABASE
DB_NAME = "database.db"

def create_database(app):                          #Function checks if database DOES NOT EXIST, then creates database if so
    if not path.exists('website/' + DB_NAME):
        db.create_all(app=app)
        print('Created Database!')
```

This code creates the database in the app directory, function create_database(app) will create a database if a database with the appropriate name under the directory is not found. This prevents overwriting pre-existing data.

The webapp itself is initialised using the function create_app() which will be run as soon as the program is executed.

```
def create_app():
    app = Flask(__name__)
    app.config['SECRET_KEY'] = 'hjshjhdjah kjshkjdhjs'
    app.config['SQLALCHEMY_DATABASE_URI'] = f'sqlite:///{DB_NAME}'
    db.init_app(app)
```

What I'm doing here is I'm saying that I want to create a database. I need to pass in the base directory that I want it to go to, and then I want it to be named database.db.

The secret key is used for all security related needs by extensions or the application, e.g securely signing the session cookie. The secret key will be changed once the app is deployed.

```
1  from website import create_app
2
3  app = create_app()
4
5  if __name__ == '__main__':
6      app.run(debug=True)
7
8
```

Function create_app() is imported into this executable then executed in debug mode, to make development easier.

Now that a database is created and the app can be initialised, we can create models for our database.

```
class product(db.Model):
    ProductID = db.Column(db.Integer, primary_key=True)
    ProductName = db.Column(db.String(150))
    stocklevel = db.Column(db.Integer)
    barcode = db.Column(db.String(13))
    Pricing = db.Column(db.Numeric())

    def to_dict(self):
        return {
            'ProductID':self.ProductID,
            'ProductName':self.ProductName,
            'stocklevel':self.stocklevel,
            'barcode':self.barcode,
            'Pricing':self.Pricing
        }
```

This creates the product table, all validation required for the database is done as the user posts variables to the server, meaning I do not have to set any attributes for any of the record fields.

The table is converted to a dictionary for later usage, once I implement a table which will be displayed on the stocks page.

```
class Users(db.Model, UserMixin):
    def get_id(self):
            return (self.UserID)
    UserID = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(100), unique=True)
    password = db.Column(db.String(150))
    accesslevel = db.Column(db.String())
```

The users table has the function get_id(self) in it, which returns the ID once a user attempts to login.
As these are login credentials, unique has been assigned to the name to differentiate logins.

Now that the database has a layout and I have everything to initialise the app, I start to edit the create_app() function more.

Error and debugging

```
Traceback (most recent call last):
  File "c:\Users\dhypa\source\repos\lets try again\main.py", line 1, in <module>
    from website import create_app
  File "c:\Users\dhypa\source\repos\lets try again\website\__init__.py", line 5, in <module>
    from .models import Users
  File "c:\Users\dhypa\source\repos\lets try again\website\models.py", line 1, in <module>
    from website import db
    ^^^^^^^^^^^^^^^^^^^^^^
ImportError: cannot import name 'db' from partially initialized module 'website' (most likely due to a circular import) (c:\Users\dhypa\source\repos\lets try again\websit
PS C:\Users\dhypa\source\repos\lets try again>
```

For the program to initialise the database
Here, I could not import ‘db’ due to a circular dependency, I was trying to import the Users class/table from my models file before the models file was initialised.

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from os import path
from flask_login import LoginManager
from .models import Users

db = SQLAlchemy()
DB_NAME = "database.db"

def create_database(app):
    if not path.exists('website/' + DB_NAME):
        db.create_all(app=app)
        print('Created Database!')
```

The error was fixed by removing the import on line 5, ending the circular dependency. The import was placed on the line after the database was fully initialised, allowing the import to take place.

```
from .views import views                        #imports views from both auth and views files
from .auth import auth

app.register_blueprint(views, url_prefix='/')   #sets the default url prefix for views in both files as '/'
app.register_blueprint(auth, url_prefix='/')

from .models import Users, product              #imports table models from database

with app.app_context():                         #initializes database with tables
    db.create_all()
```

This is the code after I have fixed the error made. I have correctly imported the database table models.

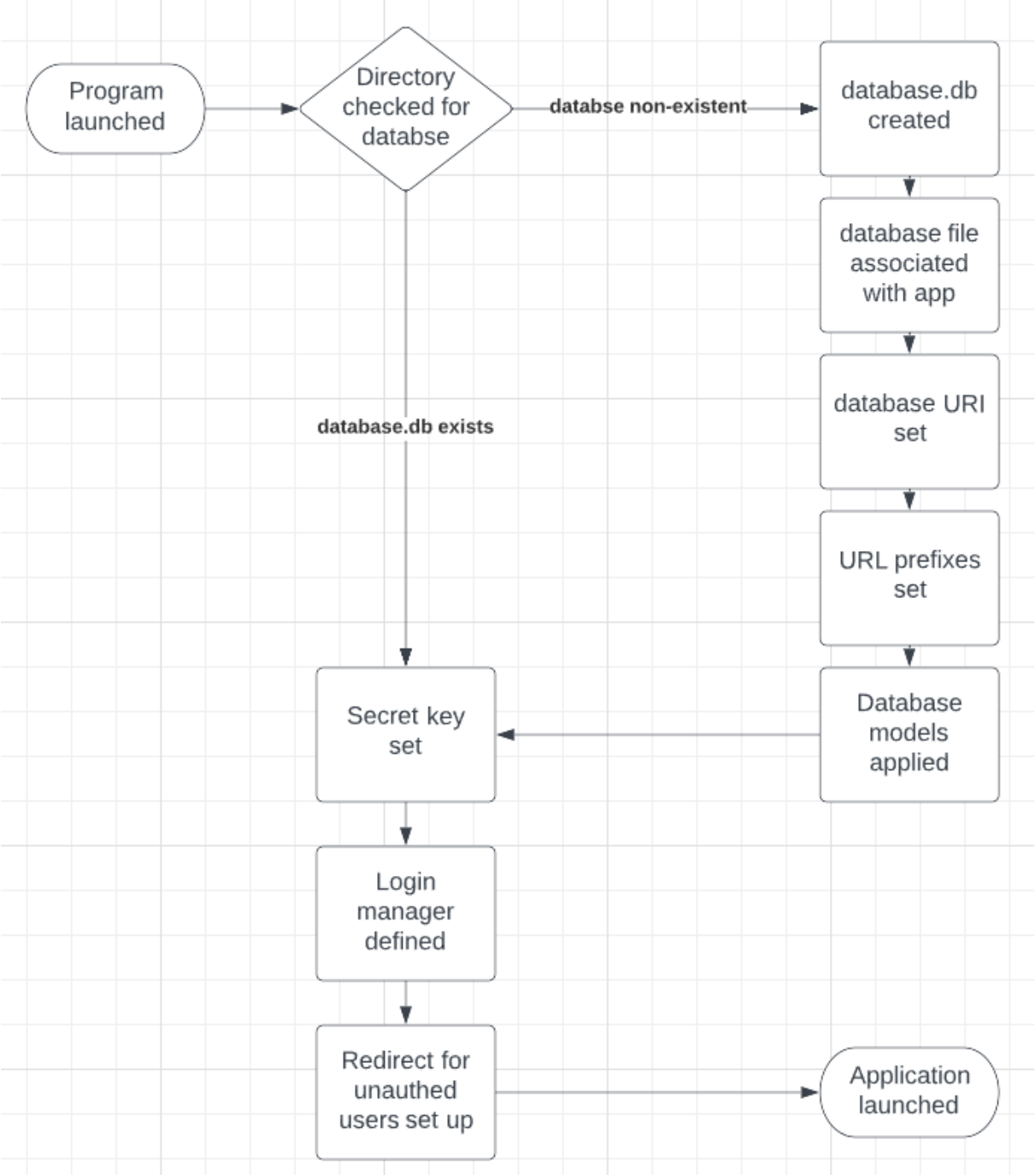
The above code also shows importing of auth and views to set a default url prefix.

```
login_manager= LoginManager(app)
login_manager.login_view = 'auth.login'    #if a user is not logged in, they are automatically directed to the view 'auth.login'
login_manager.init_app(app)

@login_manager.user_loader              #This callback is used to reload the user object from the user ID stored
def load_user(id):
    return Users.query.get(int(id))

return app
```

login_view sets the default view which users are redirected to if they are not authenticated. I have set this to ‘auth.login’ as that is where my login page is. At the end of create_app() the app is returned. After this function is executed, an instance of the application as well as a database is created in the file directory. To prevent the database being overwritten or deleted, I have set the function create_database() to only create a new database if one does not exist in the first place.

```
def create_database(app):
    if not path.exists('website/' + DB_NAME):
        db.create_all(app=app)
        print('Created Database!')
```

Presented is a flowchart which shows the process of how the application initialises:

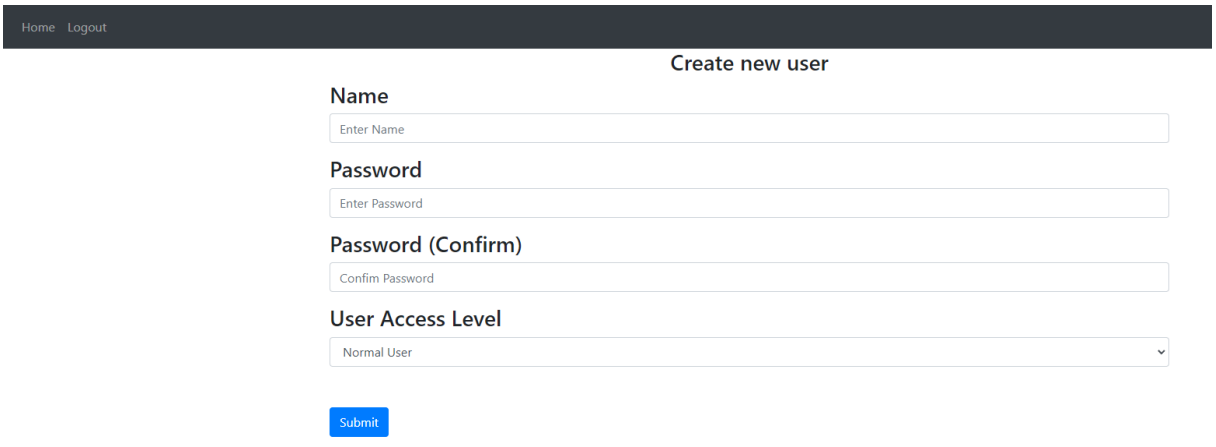


Add User

To actually log into the application, a user requires login credentials.

```
<form method="POST">
    <h3 align="center"> Create new user <h3>
    <div class="form-group">
      <label for="Name">Name</label>
      <input
         type="text"
         class="form-control"
         id="name"
         name="name"
         placeholder="Enter Name"
    />
    </div>

    <div class="form-group">
      <label for="password1">Password</label>
      <input
         type="password"
         class="form-control"
         id="password1"
         name="password1"
         placeholder="Enter Password"
    />
    </div>

    <div class="form-group">
      <label for="password2">Password (Confirm)</label>
      <input
         type="password"
         class="form-control"
         id="password2"
         name="password2"
         placeholder="Confim Password"
     />
    </div>
    <div class="form-group">
      <label for="accesslevel"> User Access Level </label>
      <select class="form-control" id="accesslevel" name="accesslevel" placeholder="Select User Access Level">
      <option>Normal User</option>
      <option>Admin</option>
      </select>

    </div>

    <br>
    <button type="submit" class="btn btn-primary"> Submit </button>
```

The HTML above codes for a form which will be used by the user to add new users.
The page renders 4 input boxes, one for a username, 2 boxes for password and password confirmation and a box which displays 2 options between admin and normal user. By setting the ‘type’ as password on the input boxes, the user’s input will be censored as they type in their password of choice.



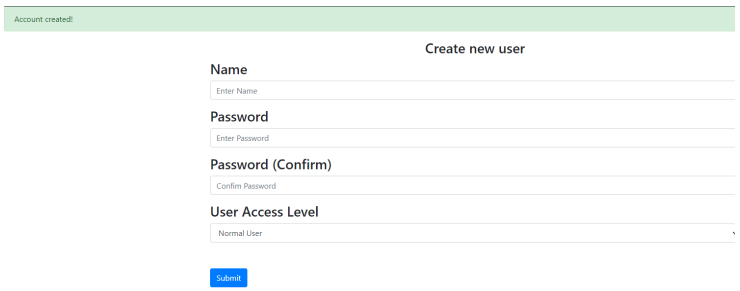
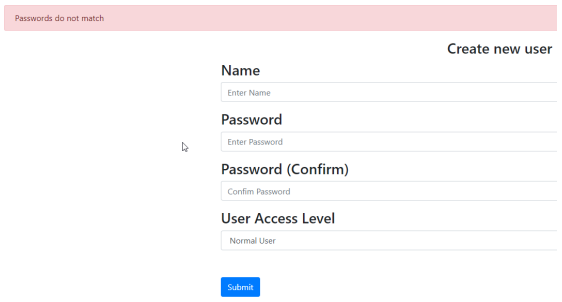
```python
@views.route('/userManagement/createuser', methods=['GET', 'POST'])
@login_required
def UMcreateuser():
    if request.method == 'POST':
        name = request.form.get('name')
        password1 = request.form.get('password1')
        password2 = request.form.get('password2')
        password1 = request.form.get('password1')
        accesslevel = request.form.get('accesslevel')
        #input validation tests
        if len(name) < 2:
            flash('Name must be valid', category='error')
        elif password1 != password2:
            flash('Passwords do not match', category='error')
        elif len(password1) < 6:
            flash('Passwords must be atleast 6 characters', category='error')
        else:
            new_user = Users(name=name, password=generate_password_hash(password1, method='sha256'))
            db.session.add(new_user)
            db.session.commit()
            flash('Account created!', category='success')
    return render_template("user create.html")
```

The above code designates a view to the url '/userManagement/createuser', which renders the user creation form html. After the server receives a post request, the server checks the data received for the variables listed:

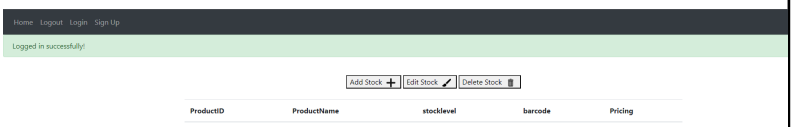
- pame
- password1
- password2
- accesslevel

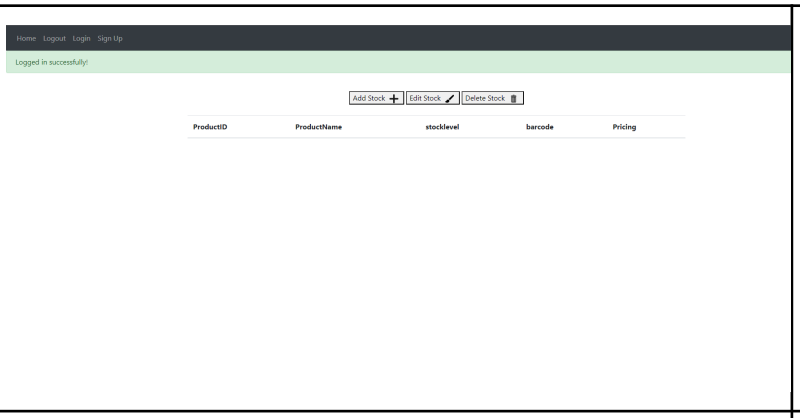
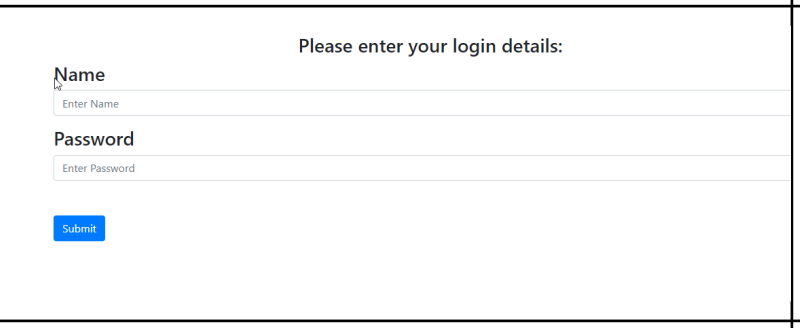
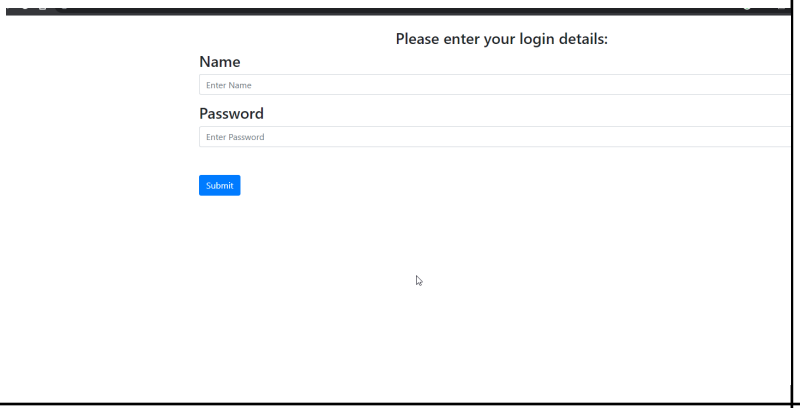
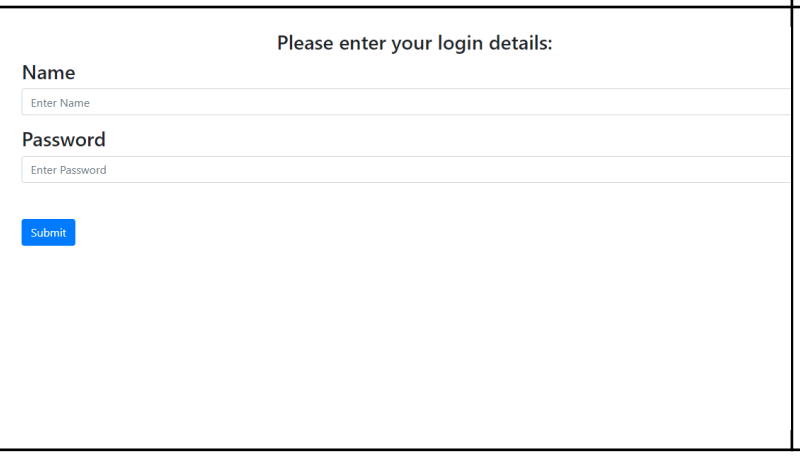
After this, validation checks are performed on the variables ensuring the credentials are valid. If all checks are successful, the users table is updated with a new record and the changes to the database are saved.

If any validation checks are failed, an error message is immediately displayed. If the new record was successfully added, a success message is flashed. Now that we have a method to create users, both user creation testing and login page testing can be performed.

Add User Testing:			
Input	**Data type**	**Output**	**Pass/Fail**
Name- yuo1 Password-chipmunks Password(confirm)- chipmunks Accesslevel- user	Normal		pass
Name-yuo2 Password-chimunk Password(confirm)-chimuk1	Erroneous		pass

While preparing the test data for login page testing, I had inputted wrong credentials multiple times, and some of my test data credentials actually had passwords which were too weak, causing me to have to edit them to make them more secure. This shows effective validation by the application.

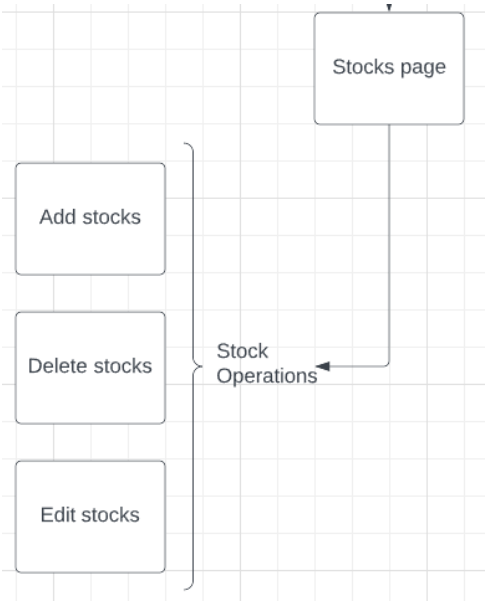
Login Page Testing:				
Input	**Data type**	**Required Output**	**Output**	**Pass/Fail**
Username-Alucard Password-qwerkz	Normal	Logs in		Pass

Username-Ingrid Password-WeWras	Normal	Logs in		Pass
Leaving login fields blank	Erroneous	Login fails		Pass
Username-Caess1 Password- felJ	Erroneous	Login fails		Pass
Username-Afneuye Password-acvetjiujuh	Erroneous	Login fails		Pass

Stocks page and stock actions

I have already created a stocks table in the database, so I now need a way to view it, populate it and manage its contents.

The stock page will display a table containing all information on the stocks, and will have 3 functional pages which will allow the user to add, delete and edit stocks.



I have already prepared the products database model in preparation for rendering a table on the stocks page (usage of the todict() function).

First thing I am required to do is to make a way to populate the products table in the database, then after it is populated I can create the rendered table, after which edit and delete functions will be added.

The method for creating new stock records will be very similar to how new users are created, the user will enter details on the stock which they wish to add, the server will receive the data and create a new record with it.

```
<form method="POST">
  <h3 align="center"> Create new stock entry <h3>
  <div class="form-group">
    <label for="">Product Name</label>
    <input
        type="text"
        class="form-control"
        id="ProductName"
        name="ProductName"
        placeholder="Enter ProductName"
  />
  </div>

  <div class="form-group">
    <label for="stocklevel">stocklevel</label>
    <input
        type="number"
        class="form-control"
        id="stocklevel"
        name="stocklevel"
        placeholder="Enter stocklevel"
  />
  </div>

  <div class="form-group">
    <label for="barcode">Barcode</label>
    <input
        type="text"
        class="form-control"
        id="barcode"
        name="barcode"
        placeholder="Enter barcode"
   />
  </div>

  <div class="form-group">
    <label for="Pricing">Pricing</label>
    <input
        type="text"
        class="form-control"
        id="Pricing"
        name="Pricing"
        placeholder="Enter Pricing"
   />
  </div>
  <br>
  <button type="submit" class="btn btn-primary"> Submit </button>
```

Form has method ‘POST’ so the data is posted to the server.
4 input boxes are created to ask for product name, stock level, barcode and price of stock.

A submit button will be used to send the data.

```python
@views.route('/stocks/add', methods=['GET', 'POST'])
def addstock():
    if request.method== 'POST':
        ProductName= request.form.get('ProductName')   #recieves data from form
        stocklevel= request.form.get('stocklevel')
        barcode= request.form.get('barcode')
        Pricing= request.form.get('Pricing')

        if len(barcode)>='14':                                          #validation checks and errors
            flash("Invalid barcode, try again!", category='error')
        elif isinstance(stocklevel, (str,bool, float, complex)) == True:
            flash("Invalid entry for stock level, try again!",category='error')

        elif isinstance(Pricing, (float, int))==False:
            flash("Invalid entry for pricing, try again!",category='error')
        else:                                                           #commits change to database
            new_product = product(ProductName=ProductName, stocklevel=stocklevel, barcode=barcode, Pricing=Pricing)
            db.session.add(new_product)
            db.session.commit()
            flash('Stock successfully added!', category='success')
```

The view for adding stocks will be under the main stocks page. Variables are set from the data received from the form on the page, all variables are checked to make sure they are valid. Error messages are sent if they are not valid.
Barcode length is confirmed, stocklevel is checked to make sure it is the correct data type, so is Pricing.
If validation checks are successful, the record is added to the database.

Now that the database can be populated, I need to create a way to present the data.

```python
#STOCKS AND STOCK CONTROL
@views.route('/stocks')
def stocks():
    return render_template('stocks.html', title='Stocks')

#STOCK TABLE DATA
@views.route('/data')
def data():
    return {'data': [product.to_dict() for product in product.query]}
```

I start off by creating a view under url ‘/stocks’. The table is the main thing the user will see after going to the stocks module of the program.
stocks.html is the template that will contain the buttons and table for the stock page. This page does not have any special actions so the view code for it is very short.

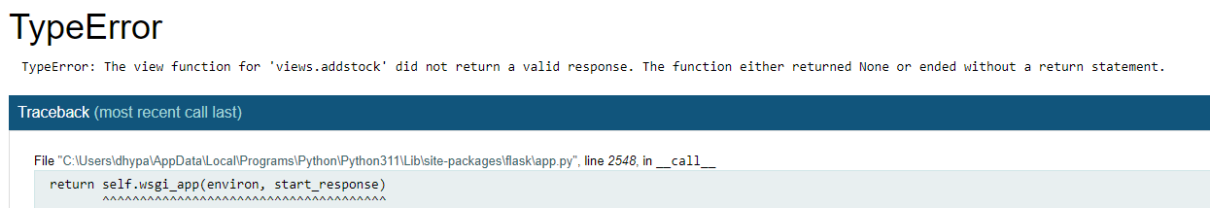
I am using a javascript table to represent data, which requires me to use an ajax data source which will be held at the end point ’/data’. The dictionary prepared earlier in the products table comes in useful here.

```
<table id="data" class="table table-striped">
  <thead>
    <tr>
      <th>ProductID</th>
      <th>ProductName</th>
      <th>stocklevel</th>
      <th>barcode</th>
      <th>Pricing</th>
    </tr>
  </thead>
  <tbody>
  </tbody>
</table>
```

I create a simple table with the table headings being the fields in the product table

Error and debugging

While attempting to go to the stock addition page to make sure the form is loaded correctly, I realised I failed to actually render the templates for the stock module, resulting in the program not functioning.



To fix this I added code to return the correct HTML pages for each view I have in the stocks module.

```
return render_template("stock create.html")
```

The stock addition page now loads as intended with the correct fields.

Now with a way to populate the the products table, I can confirm the stocks page table renders correctly. I will be populating the table with test data.

Error and debugging

```
File "c:\Users\dhypa\source\repos\lets try again\website\views.py", line 51, in addstock
    if len(barcode)>='14':                                          #validation checks and errors
       ^^^^^^^^^^^^^^^^^^
TypeError: '>=' not supported between instances of 'int' and 'str'
```

I had made a syntax error in my validation checks, and after looking closely, my validation checks were written incorrectly.

```
if len(barcode)>'13':                                          #validation checks and errors
    flash("Invalid barcode, try again!", category='error')
```

‘13’ was being compared to the length of the string barcode as an integer, which is not possible.

```
stocklevel= int(request.form.get('stocklevel'))
barcode= request.form.get('barcode')
Pricing= float(request.form.get('Pricing'))
```

I set the inputs coming in from the form to the correct data types that the validation checks were expecting to receive, as numbers and floats were not being communicated properly between the HTML form and python, despite both the pricing and stocklevel input boxes having validation checks for numbers and the input boxes for stocklevel and Pricing in the form itself had designation of type= ‘number’.

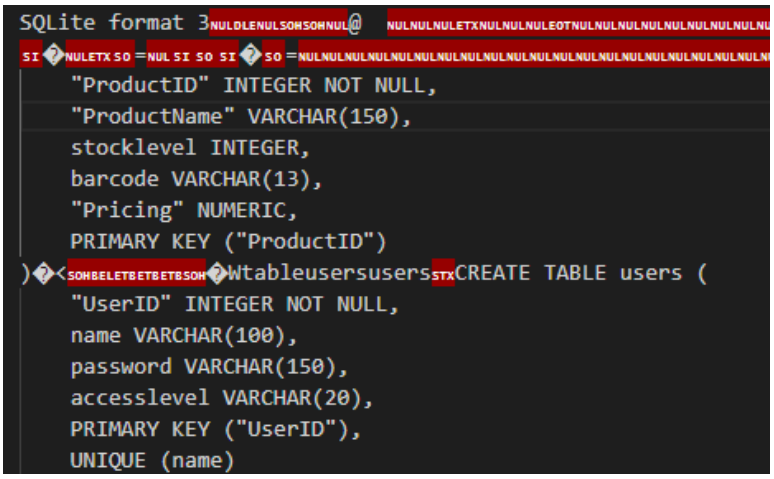
```
<label for="Pricing">Pricing</label>
<input
    type="number"
```

The database also did not initialise properly, as the stocklevel column was missing from the product table. This was not an error on side of the code, to confirm this I deleted the pre-existing database and re-initialized the application,
With these bugs fixed, the stock addition page is functional.

DB file before:

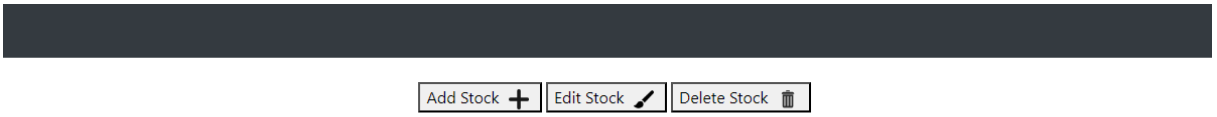
```
SQLite format 3
    "ProductID" INTEGER NOT NULL,
    "ProductName" VARCHAR(150),
    barcode VARCHAR(13),
    "Pricing" NUMERIC,
    PRIMARY KEY ("ProductID")
)CREATE TABLE users (
    "UserID" INTEGER NOT NULL,
    name VARCHAR(100),
    password VARCHAR(150),
    accesslevel VARCHAR(20),
    PRIMARY KEY ("UserID"),
    UNIQUE (name)
)indexsqlite_autoindex_users_1users
```

DB file after:



ProductID	ProductName	stocklevel	barcode	Pricing
1	Pepsi	52	0101010101010	£1
2	Milk	12	2323232323232	£2
3	Bread	23	1234123412341	£1.50

This is the outputted table with the data I imputed into the stock creation form.
Now that the stocks page is operational, I have added buttons leading to the operations that can edit the database.



ProductID	ProductName	stocklevel	barcode	Pricing
1	Pepsi	52	0101010101010	£1
2	Milk	12	2323232323232	£2
3	Bread	23	1234123412341	£1.50

Editing stocks is different from adding or deleting a record. The user may only want to change certain fields in the record but not all of them.

```
productbeingeditted = product.query.filter_by(ProductID=ProductID).first_or_404(description='ProductID is not valid'.format(ProductID))

try:                                                    #Ensures only correct fields are editted
    if nProductName != "":
        productbeingeditted.ProductName=nProductName
    if nstocklevel !="":
        productbeingeditted.stocklevel=nstocklevel
    if nbarcode != "":
        productbeingeditted.barcode=nbarcode
    if nPricing != "":
        productbeingeditted.Pricing=nPricing


except:    #if the above if statements fail, error is flashed
    flash("There was an error",category='error')
finally:   #database is editted
    edittedrecord = product(ProductID=ProductID, ProductName=nProductName, stocklevel=nstocklevel, barcode=nbarcode, Pricing=nPricing)
    db.session.update(edittedrecord)
    db.session.commit()
    flash("Operation has been completed", category='success')
```

Just like all the previous forms, the code in the views document will follow the same format with the methods defined and the form variables being stored, but things are different after that.

A first_or_404() function means if the ID the user wants to edit does not come up in the database query for some reason, an error is automatically sent with the given message in the function.

A try statement means the code in it will be run, and if for some reason it cannot be run, the program will skip to the except statement which will flash an error.

The final statement updates the database and flashes a success message.
The code on the HTML is just another form which posts the relevant information.

```
if request.method== 'POST':
    ProductID= request.form.get('ProductID')
    nProductName= request.form.get('ProductName')
    nstocklevel= request.form.get('stocklevel')
    nbarcode= request.form.get('barcode')
    nPricing= request.form.get('Pricing')
```

I had already set up validation on the stock addition page, and as the variables are essentially identical, I can reuse the same code here, saving time on testing and development.

Stock deletion page is quite simple, a HTML form in which a product ID is entered into, then that ID is queried then deleted.
The ID inputted is used to query the database, if it is not found an error message is displayed.
The record is then deleted.

Validation on the ID entered is not required, because if it is an invalid datatype, it does not exist in the database in the first place, so an error will be displayed if incorrect datatype is entered.

```python
@views.route('/stocks/delete', methods=['GET', 'POST'])
def deletestock():
    if request.method== 'POST':
        ProductID= request.form.get('ProductID')
        stock_to_delete = product.query.filter_by(ProductID=ProductID).get_or_404(description='ProductID is not valid'.format(ProductID))

        try:
            db.session.delete(stock_to_delete)
            db.session.commit()
            flash("Stock ", ProductID," deleted successfully!", category='success')
            return redirect(url_for('views.stocks'))#
        except:
            flash("Error deleting stock, try again!", category='error')
```

I have added return statements to all stock related pages which will redirect the user back to stocks after they have completed an operation successfully.

```python
return redirect(url_for('views.stocks'))#
```

The stocks module is now finished and is ready for testing.

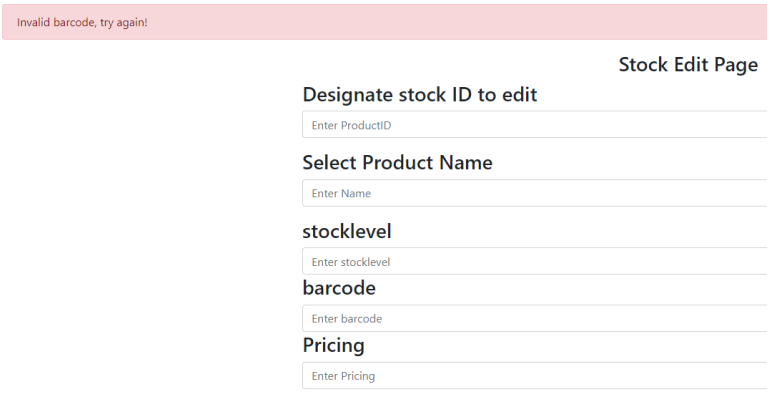
Test data:

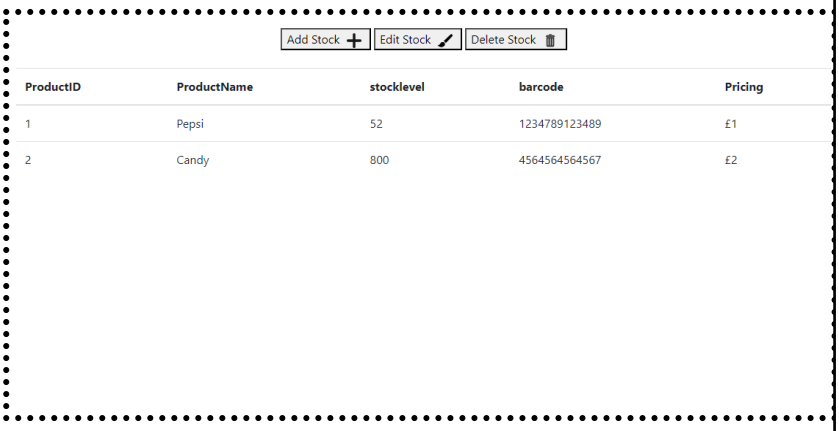
ProductID	ProductName	stocklevel	barcode	Pricing
1	Pepsi	52	0101010101010	£1
2	Milk	12	2323232323232	£2
3	Bread	23	1234123412341	£1.50

Above is the current state of the stocks table before testing.

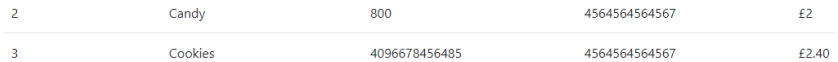
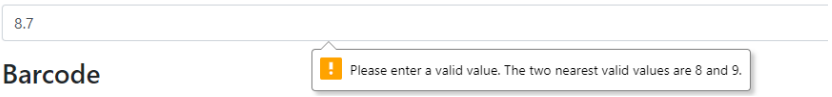
Stock edit page testing:

Input	Data type	Required Output	Output	Pass/Fail
ID- 1 Barcode- 1234789123489	Normal	Only barcode edited	ProductID: 1 | ProductName: Pepsi | stocklevel: 52 | barcode: 1234789123489 | Pricing: £1	Pass
ID- 2 Barcode- 4564564564567 ProductName- Candy Stockcount- 800	Normal	Only barcode, productname and stockcount editted	2 Candy 800 4564564564567 £2	Pass

ID- 1 Barcode- 1234567891234567 89	Erroneous	Error message flashed		Pass
-	Erroneous	Error displayed		Pass

Stock delete page testing:				
Input	**Data type**	**Required Output**	**Output**	**Pass/Fail**
ID- 3 Click submit button	Normal	Record at ID 3 deleted		Pass
ID- 789 Click submit button	Erroneous	Error displayed		Pass

Stock addition page testing:

Input	Data type	Required Output	Output	Pass/Fail
ProductName-Cookies stocklevel-45 barcode-0966784564818 Pricing=2.4	Normal	New record added		Pass
ProductName- 42mint stocklevel-8.7 barcode-09667845648182222222222 Pricing=AAAAAA	Erroneous	Error flashed		Pass

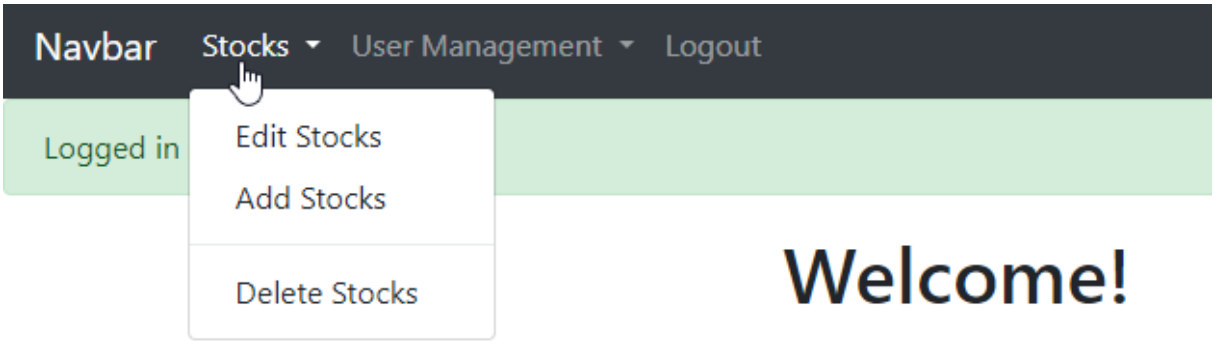
2	Candy	800	4564564564567	£2
3	Cookies	4096678456485	4564564564567	£2.40

Basic Navigation

A navbar will be utilised as it is easy to use and can be very versatile.

```
<div class="collapse navbar-collapse" id="navbar">
  <ul class="navbar-nav mr-auto">
    <li class="nav-item dropdown">
      <a class="nav-link dropdown-toggle" href="/stocks" id="navbarDropdown" role="button" data-toggle="dropdown" aria-haspopup="true" aria-expanded="false">
        Stocks
      </a>
      <div class="dropdown-menu" aria-labelledby="navbarDropdown">
        <a class="dropdown-item" href="/stocks/edit">Edit Stocks</a>
        <a class="dropdown-item" href="/stocks/add">Add Stocks</a>
        <div class="dropdown-divider"></div>
        <a class="dropdown-item" href="/stocks/delete">Delete Stocks</a>
      </div>
    </li>
```

Using bootstrap's collapsable navbar, I can create a navbar which is friendly to manny screen sizes, meaning the website has functionality even if it is accessed from devices such as phones.



This code creates a navbar which has buttons that can act as dropdown lists. This makes it so the user is not overwhelmed by options and makes the program look more sleek.

Users and user management

The user addition page has already been created to acquire login credentials, and testing on it has already been done. Now I will be completing the users module by implementing a way to delete users and add a main users page which will display all existing users and related information stored about them in the database, excluding passwords.

For a method to delete users, I have a form which posts data to the server. Inputted user is queried against the database, if valid then the record found will be deleted. If the user is not found, an error message is displayed.

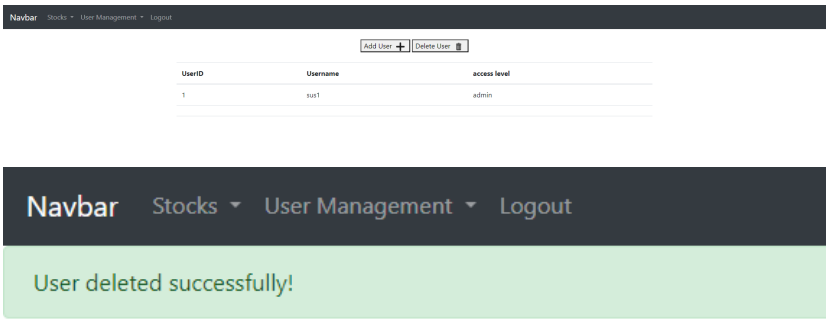
```
def UMdeleteuser():
    if request.method== 'POST':
        UserID= request.form.get('UserID')
        user_to_delete = Users.query.filter_by(UserID=UserID).first_or_404(description='UserID is not valid'.format(UserID))

        try:
            db.session.delete(user_to_delete)
            db.session.commit()
            flash("User deleted successfully!", category='success')
        except:
            flash("Error deleting user, try again!", category='error')
    render_template('user delete.html')
```

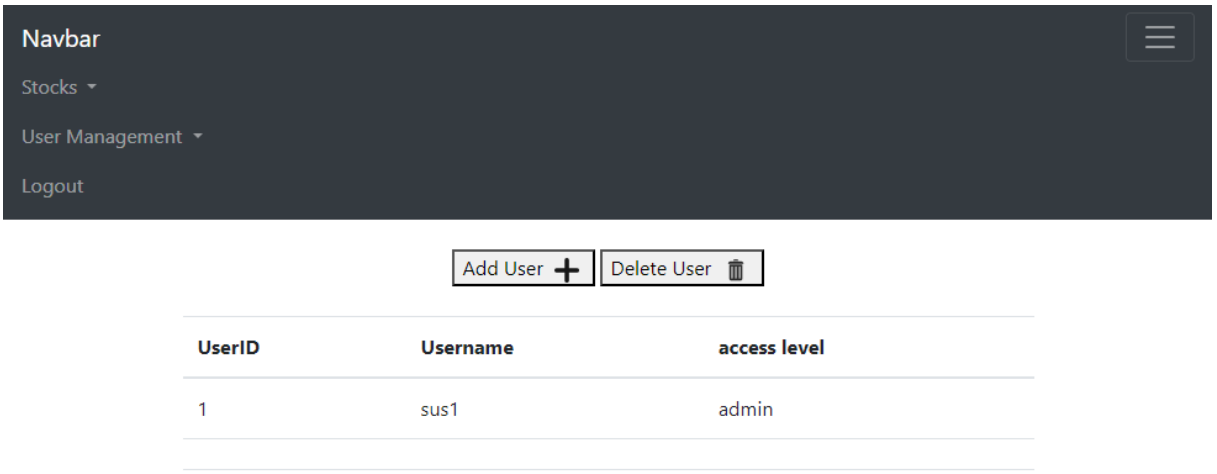
```
<form method="POST">
    <h3 align="center"> Enter UserID for deletion <h3>
    <div class="form-group">
      <label for="Name">UserID</label>
      <input
          type="int"
          class="form-control"
          id="UserID"
          name="UserID"
          placeholder="Enter UserID"
    />
    </div>
    <button type="submit" class="btn btn-primary"> Submit </button>
```

Above is the HTML code which takes the UserID which will be deleted.

The users table is created in the same way the stocks table was, except with different columns.

As user addition was already tested earlier, I will only be testing the user deletion operations.

User delete page testing:				
Input	**Data type**	**Required Output**	**Changes**	**Pass/Fail**
ID- 2	Normal	Record at ID 2 deleted		Pass



Pictured above is the users page, displaying all users stored in the database, with buttons leading to forms which allows the user to manipulate the users table.

Access levels and securing the program

Now that the main content of the web app is complete, I need to add code which will differentiate between admins and normal users, and will hide functions which normal users cannot access. Due to django, I can write python statements directly into html.

By utilising has_role(role) function I can create conditional statements which will only load elements in if a user is authorised. This allows me to block access to pages normal users should not be visiting, e.g user management.

```
from flask_security import RoleMixin
```

I start by importing rolemixin then creating a model for the roles table.

```
#holds roles
class Role(db.Model, RoleMixin):
    __tablename__ = 'role'
    RoleID = db.Column(db.Integer(), primary_key=True)
    rolename = db.Column(db.String(80), unique=True)

# table used to assign roles
roles_users = db.Table('roles_users',
        db.Column('UserID', db.Integer(), db.ForeignKey('user.id')),
        db.Column('RoleID', db.Integer(), db.ForeignKey('role.id')))
```

Each role has a name and a unique ID, with the rolename being a string. The roles_users table is used to assign roles to users, the userID and roleID are taken as foreign keys and put into the table.

The information in the database needs to be recognised by flask_security so it can make a connection between the two, so I import SQLAlchemySessionUserDatastore then pass the table containing users and roles.

```
from flask_security import RoleMixin, Security, SQLAlchemySessionUserDatastore
```

Security is also imported which binds instance of app with the data.

```
user_datastore = SQLAlchemySessionUserDatastore(db.session, Users, Role)

security = Security(app, user_datastore)
```

Now that this is set up, we go back to the usercreation function.

```
else:
    new_user = Users(name=name, password=generate_password_hash(password1, method='sha256'))
    db.session.add(new_user)

    role = Role.query.filter_by(id=request.form['options']).first()

    Users.role.append(role)
    db.session.commit()
    flash('Account created!', category='success')
```

Now when a user is created, their role is added to the Users table.

I can gate views by using the statement @roles_accepted('admin'), which will prevent all non-admins from seeing a view, instead the app will render the login page, where the user

can sign in. This is so if an admin is accidentally on a different account, try to access a view that they frequently, then realise that their account does not have the required access level.

```
@roles_accepted('admin')
def UMcreateuser():
    from models import Role
    if request.method == 'POST':
        name = request.form.get('name')
        password1 = request.form.get('password1')
        password2 = request.form.get('password2')
        password1 = request.form.get('password1')
        accesslevel = request.form.get('accesslevel')
        #input validation tests
```

All views can now be gated through both requiring the user to be logged in and by role.

```
@views.route('/api/data')
@login_required
@roles_accepted('admin')
def data():
    query = product.query
```

Above is a view which holds data for the stocks table so that it can be rendered, @login_required gates logins and @roles_accepted('admin') limits access to users assigned the admin role.

```
{% if current_user.has_roles('admin') %}
  <li class="nav-item dropdown">
    <a class="nav-link dropdown-toggle" href="/userManagement" id="navbarDropdown" role="button" data-toggle="dropdown" aria-haspopup="true" aria-expande
     User Management
    </a>
    <div class="dropdown-menu" aria-labelledby="navbarDropdown">
     <a class="dropdown-item" href="/adminhome/userManagement/createuser">Create user</a>
     <a class="dropdown-item" href="/adminhome/userManagement/deleteuser">Remove user</a>
    </div>
  </li>
{% endif %}
```

The above code displays usable of current_user.has_roles() to find out if the user loaded into the view has the admin role, and the following HTML content will only be rendered if the user is authenticated. The reason I have created it this way is so that separate templates do not need to exist between normal users and admin users, instead it can all be coded together and content can simply be gated using simple conditional statements, saving time.

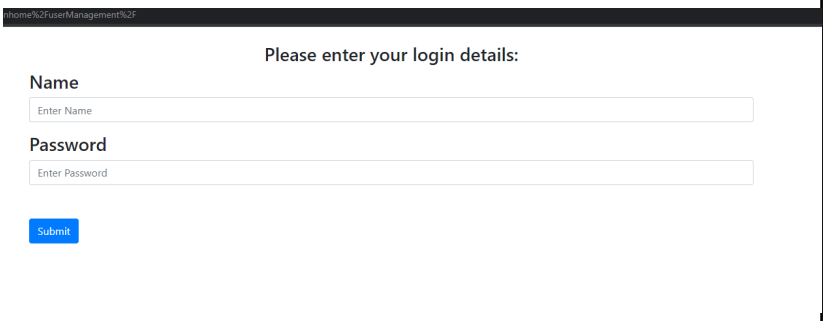
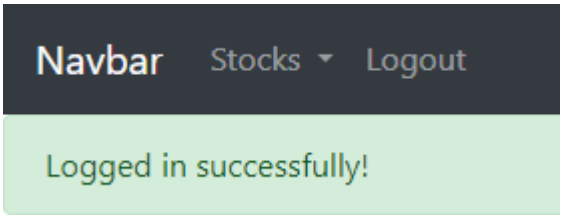
Small error and debugging:

```
  File "C:\Users\dhypa\AppData\Local\Programs\Python\Python311\Lib\site-packages\sqlalchemy\util\langhelpers.py", line 1138, in __get__
    obj.__dict__[self.__name__] = result = self.fget(obj)
                                           ^^^^^^^^^^^^^^
  File "C:\Users\dhypa\AppData\Local\Programs\Python\Python311\Lib\site-packages\sqlalchemy\sql\schema.py", line 2918, in column
    return self._resolve_column()
           ^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\dhypa\AppData\Local\Programs\Python\Python311\Lib\site-packages\sqlalchemy\sql\schema.py", line 2958, in _resolve_column
    return self._link_to_col_by_colstring(
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\dhypa\AppData\Local\Programs\Python\Python311\Lib\site-packages\sqlalchemy\sql\schema.py", line 2879, in _link_to_col_by_colstring
    raise exc.NoReferencedColumnError(
sqlalchemy.exc.NoReferencedColumnError: Could not initialize target column for ForeignKey 'role.id' on table 'roles_users': table 'role' has no column named '
PS C:\Users\dhypa\source\repos\lets try again>
```

The foreign key access in the role_users table was named incorrectly.

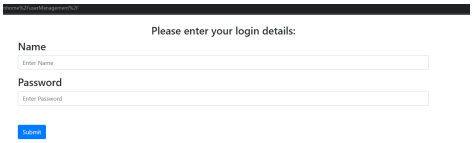
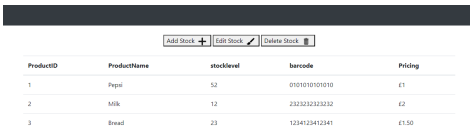
```
('users.UserID'))
('role.RoleID')))
```

Mistake has been fixed and the code functions as intended.
I will run 2 tests, one which will confirm HTML content can be gated individually by roles, the other test will see if entire viewpoints can be gated by roles.

Hidden content testing:				
Input	**Data type**	**Required Output**	**Output**	**Pass/Fail**
Try to access view as unauthorised user	Normal	Trying to access user management page results in redirect to sign in	 Redirected to login view	Pass
View navbar with hidden content as a normal user	Normal	User management button is hidden	 Button is missing as intended	Pass

Evaluation

Evidence for success criteria completion

#	Criteria	Why/How	Evidence	Was it met and comments
1	Login page and levels of access for different credentials	Allows only some staff to have access to the entire program, so something like employees can only update when stocks while the owner and a manager will have full access to the program. The GUI used should be simple and password will be hidden behind (*) to ensure it is unseen.		Criteria met. Tested in implementation. Any attempt to access unauthorised content redirects users to the login page where they can log into an account which has the required access levels.
2	Use of a CSV file so the software can interact with the tilling system	Easier to keep track of stock if software interacts with tills.	-	Criteria not met.
3	Stock management: 1. Update stock levels 2. View product information 3. Update product information	Allows user to set stock levels, and to amend mistakes on products that have been entered.		Criteria met. Stocks table accurately displays information, and the table can be updated.

4	Error messages: 1. When input is sanitised 2. Wrong login details entered	Alert the user on invalid inputs so they may try again. Alert the user when login details are incorrect so they can amend them.	 {% with messages = get_flashed_messages(with_categories=true) %} {% if messages %} {% for category, message in messages %} {% if category == 'error'%} {{ message }} <button type="button" class="close" data-dismiss="alert"> × </button> {% else %} {{ message }} <button type="button" class="close" data-dismiss="alert"> × </button> {% endif %} {% endfor %} {% endif %} {% endwith %}	Criteria met. The original criteria had just 2 sub-requirements, input sanitisation and error messages for the login page. During development, I have utilised error messages in many places such as database querying.
5 r	Software must be accessible from a variety of different devices from different locations	So the software is accessible from the client's home on a home device if needed, as this is a web app this is very achievable.	Nature of application N/A	Criteria met. This criteria is naturally met due to the solution being a webapp, which only requires basic HTML and basic networking capabilities to run.

Conclusion

From the original success criteria,I have delivered every requirement except interaction with the csv file. The program can be accessed from anywhere given it is running and an internet connection is available. I have created a GUI that is easy on the eyes, being incredibly user friendly due to clear font, consistent theme and very visible buttons. The usage of a webapp means most of the work required to set up a networked application is skipped, and it also allows me to use all of HTML and CSS's advantages as just a front end GUI, while the solution's backend is coded completely using python, sql and some javascript.

I have not met the criteria which would allow the program to interact with a tilling system via a shared CSV file, this is due to the fact that I do not have the relevant expertise or the hardware to test/develop this feature. In the future I would like to research on how to do this but unfortunately I was unable to deliver. In the future, this feature may be added by allowing the program to read and write to files stored by the tilling system, for this to happen the tilling system must be network capable so it can communicate with the server over LAN.

During development, I was forced to edit the database design as SQL did not support the design I had made. I reduced the number of tables by one and kept the stock count in the same table as the stock itself, which did not cause any data integrity issues.

Limitations and improvements

1. No way to update user passwords, a new account has to be created and the old one deleted

Creating a new view and form which can receive information from the user and modify the database accordingly. Admins could be given the ability to change any normal user's passwords, and normal users can be given the ability to only change their own password (database query would have the UserID stored in the class passed directly into it, meaning user would only be able to change their own password).

2. Program is susceptible to SQL injection, which can lead to data loss and downtime of the program, which cuts into profits.

Research into common SQL attacks would be needed, and after studying them, validation checks can be implemented into the program. This would block most attacks.

3. Adding products is tedious and time consuming

The user needs to enter a 13 character barcode every time they want to add a stock to the database, by creating a way for the program to access the device camera, the barcode can be scanned and metadata can be automatically gathered on the internet or any other publicly available product databases then filled into relevant fields. This would save time therefore increase profit, not to mention make the program more user friendly. As the program is already accessible via phone devices, using this feature would be very easy.

4. There is a limited selection of roles.

Currently, there is only a normal user role and an admin user role. If the client wants to limit employee access to only some features e.g allow new employees to only view the stocks table but not modify it, this is very impractical. There is already a roles table set up, and a user creation form. The user creation form can be edited to allow selection of many more roles. Each operation on stocks can have its own designated role. Changes made to the program would be as followed:

- GUI elements would be blocked behind roles through the use of djanjo to insert python code directly into the HTML
- New role based access statements on every single view in the program
- A new master admin role which has access to everything
- New roles module which would only be visible to the master admin role, which allows setting of roles
- Newly created users would have limited access to the program until an admin assigns roles to their user class

Improvements:

1. The program can be improved by having it track stock fluctuations. Changes in stock can be recorded every 30 minutes and tracked for a long period of time. Datamining algorithms can then take this information and find trends which can advise the client to more carefully control their stock. This would increase profits and keep the store from running out of stock by predicting when it may be purchased. New database models would need to be created and I would have to look into data mining, which is foreign to me.

2. The program could record changes in stock and calculate profits based on time. This would give the client a good business overview and allow them to see what stocks are selling well, which stocks are unpopular etc, increasing profits.

Maintenance

This program can operate without technical support, as all possible inputs have been accounted for, and admins can manage users, so support is not needed in setting up the program either. Any removal of staff can also be carried out using the user management forms which means extra help is not required, however, there is the small but existent possibility that the user may delete every single credential stored in the database. If this occurs, the client may need me or someone else to manually create a credential in the database or create methods to edit the database which are not locked behind the admin role or flask login. This can be done either by:

- editing the program's code to remove security, creating user credentials on the pre-existing user creation form
- Creating a new user creation form and using it to edit the database
- Manually editing the database, which is not recommended as it may corrupt the database and mess with the password hashing

Limitation and improvement: Seeing as deleting every user credential is possible, this is a limitation of my program which I failed to see. This can be fixed by doing a check on user input when they attempt to delete a user. If the inputted ID matches the ID stored in the current user class, an error is flashed and the operation is stopped. This just means if the user enters their own ID then the program does not let them.

Aside from software maintenance, as the database grows storage space upgrades may be needed by the client so that the database can store more information. This is a relatively simple process and the client has options, which include plugging in more secondary storage drives, replacing the pre-existing drive, utilising cloud storage etc. Replacing the pre-existing drive would result in downtime of the application which is not ideal, however this should not have to occur as the information being stored is just text, it does not have a large magnitude.

If the software is upgraded over time, more memory may be required to hold the program, and any redundant code should be purged so it does not take up space in memory.

Changing requirements

If the client's needs of the program ever change, it can always have modules added to it due to its structure, new views can be added with different functionalities, but care should be taken not to overwrite pre-existing functionality, especially as the program gets more complex.

Bibliography

Flask documentation
https://flask.palletsprojects.com/en/2.3.x/

Basic flask tutorial
https://youtu.be/dam0GPOAvVI

Bootstrap content distribution network - Used to style the application
https://getbootstrap.com/

Tables blog post by Miguel Grinberg - Used to format and create viewable tables in HTML
https://blog.miguelgrinberg.com/post/beautiful-interactive-tables-for-your-flask-templates

Role-based access control tutorial
https://www.geeksforgeeks.org/flask-role-based-access-control/

Code Listing

All pages after this simply contain the coded solution, no documentation.

This is a full print out of all code in colour.

RUN

```
from website import create_app

app = create_app()

if __name__ == '__main__':
    app.run(debug=True)
```

APP INITIALIZER

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from os import path
from flask_login import LoginManager, login_manager, login_user
from flask_security import Security, SQLAlchemySessionUserDatastore

db = SQLAlchemy()
DB_NAME = "database.db"

def create_database(app):
    if not path.exists('website/' + DB_NAME):
        db.create_all(app=app)
        print('Created Database!')


def create_app():
    app = Flask(__name__)
    app.config['SECRET_KEY'] = 'hjshjhdjah kjshkjdhjs'
    app.config['SQLALCHEMY_DATABASE_URI'] = f'sqlite:///{DB_NAME}'
    db.init_app(app)


    from .views import views
    from .auth import auth

    app.register_blueprint(views, url_prefix='/')
    app.register_blueprint(auth, url_prefix='/')
```

```
    from .models import Users, Role

    with app.app_context():
        db.create_all()




    login_manager= LoginManager(app)
    login_manager.login_view = 'auth.login'
    login_manager.init_app(app)

    @login_manager.user_loader
    def load_user(id):
        return Users.query.get(int(id))

    return app


def create_database(app):
    if not path.exists('website/' + DB_NAME):
        db.create_all(app=app)
        print('Created Database!')
```

VIEWS

```
from flask import Blueprint, render_template, request, flash, redirect,
url_for
from flask_login import login_user, login_required, logout_user,
current_user, login_url
from .models import Users, product
```

```
from . import models, db
from werkzeug.security import generate_password_hash,
check_password_hash
from django.db import models
from flask_security import RoleMixin, Security,
SQLAlchemySessionUserDatastore


views = Blueprint('views', __name__)

@views.route('/adminhome')
@login_required
def adminhome():
    return render_template("admin home.html")

@views.route('/adminhome/userManagement/')
@login_required
def UM():
    return render_template('user management.html')

@views.route('/adminhome/userManagement/createuser', methods=['GET',
'POST'])
def UMcreateuser():

    if request.method == 'POST':
        name = request.form.get('name')
        password1 = request.form.get('password1')
        password2 = request.form.get('password2')
        password1 = request.form.get('password1')
        accesslevel = request.form.get('accesslevel')
        #input validation tests
        if len(name) < 2:
            flash('Name must be valid', category='error')
        elif password1 != password2:
            flash('Passwords do not match', category='error')
        elif len(password1) < 6:
            flash('Passwords must be atleast 6 characters',
category='error')
        else:
            new_user = Users(name=name,
password=generate_password_hash(password1, method='sha256'))
            db.session.add(new_user)
```

```
            db.session.commit()
            flash('Account created!', category='success')
    return render_template("admin user create.html")

@views.route('/adminhome/userManagement/deleteuser', methods=['GET',
'POST'])
def UMdeleteuser():
    if request.method== 'POST':
        UserID= request.form.get('UserID')
        user_to_delete =
Users.query.filter_by(UserID=UserID).first_or_404(description='UserID
is not valid'.format(UserID))

        try:
            db.session.delete(user_to_delete)
            db.session.commit()
            flash("User deleted successfully!", category='success')
        except:
            flash("Error deleting user, try again!", category='error')
    return render_template('user delete.html')
#STOCKS AND STOCK CONTROL
@views.route('/stocks')
@login_required
def stocks():
    return render_template('stocks.html')

#STOCK TABLE DATA
@views.route('/stocks/data')
def tabledata():
    return {'data': [product.to_dict() for product in product.query]}


@views.route('/stocks/add', methods=['GET', 'POST'])
def addstock():
    if request.method== 'POST':
        ProductName= request.form.get('ProductName')   #recieves data
from form
        stocklevel= int(request.form.get('stocklevel'))
        barcode= request.form.get('barcode')
        Pricing= float(request.form.get('Pricing'))

        if len(barcode)> 13 :
#validation checks and errors
            flash("Invalid barcode, try again!", category='error')
```

```
        elif isinstance(stocklevel, (str,bool, float, complex)) ==
True:
            flash("Invalid entry for stock level, try
again!",category='error')
        elif isinstance(Pricing, (float, int))==False:
            flash("Invalid entry for pricing, try
again!",category='error')
        else:
#commits change to database



            new_product = product(ProductName=ProductName,
stocklevel=stocklevel, barcode=barcode, Pricing=Pricing)
            db.session.add(new_product)
            db.session.commit()
            flash('Stock successfully added!', category='success')
    return render_template("stock create.html")

@views.route('/stocks/edit', methods=['GET', 'POST'])
def editstock():
    if request.method== 'POST':
        ProductID= request.form.get('ProductID')
        nProductName= request.form.get('ProductName')
        nstocklevel= request.form.get('stocklevel')
        nbarcode= request.form.get('barcode')
        nPricing= request.form.get('Pricing')

        productbeingeditted =
product.query.filter_by(ProductID=ProductID).first_or_404(description='
ProductID is not valid'.format(ProductID))
        if len(nbarcode)> 13 :
#validation checks and errors
            flash("Invalid barcode, try again!", category='error')
        elif isinstance(nstocklevel, (str,bool, float, complex)) ==
True:
            flash("Invalid entry for stock level, try
again!",category='error')
        elif isinstance(nPricing, (float, int))==False:
            flash("Invalid entry for pricing, try
again!",category='error')
        else:
            try:
#Ensures only correct fields are editted
                if nProductName != "":
```

```
                    productbeingeditted.ProductName=nProductName
                if nstocklevel !="":
                    productbeingeditted.stocklevel=nstocklevel
                if nbarcode != "":
                    productbeingeditted.barcode=nbarcode
                if nPricing != "":
                    productbeingeditted.Pricing=nPricing


            except:    #if the above if statements fail, error is
flashed
                flash("There was an error",category='error')
            finally:   #database is editted
                edittedrecord = product(ProductID=ProductID,
ProductName=nProductName, stocklevel=nstocklevel, barcode=nbarcode,
Pricing=nPricing)
                db.session.update(edittedrecord)
                db.session.commit()
                flash("Operation has been completed",
category='success')
    return render_template("stock edit.html")

@views.route('/stocks/delete', methods=['GET', 'POST'])
def deletestock():
    if request.method== 'POST':
        ProductID= request.form.get('ProductID')
        stock_to_delete =
product.query.filter_by(ProductID=ProductID).first_or_404(description='
ProductID is not valid'.format(ProductID))

        try:
            db.session.delete(stock_to_delete)
            db.session.commit()
            flash("Stock ", ProductID," deleted successfully!",
category='success')
            return redirect(url_for('views.stocks'))#
        except:
            flash("Error deleting stock, try again!", category='error')
    return render_template('stock delete.html')




@views.route('/api/data')
```

```python
@login_required
#@roles_accepted('admin')
def data():
    query = product.query

    # search filter
    search = request.args.get('search[value]')
    if search:
        query = query.filter(db.or_(
            product.ProductName.like(f'%{search}%'),
            product.barcode.like(f'%{search}%')
        ))
    total_filtered = query.count()

    # sorting
    order = []
    i = 0
    while True:
        col_index = request.args.get(f'order[{i}][column]')
        if col_index is None:
            break
        col_name = request.args.get(f'columns[{col_index}][data]')
        if col_name not in ['ProductName', 'barcode', 'ProductID']:
            col_name = 'ProductName'
        descending = request.args.get(f'order[{i}][dir]') == 'desc'
        col = getattr(product, col_name)
        if descending:
            col = col.desc()
        order.append(col)
        i += 1
    if order:
        query = query.order_by(*order)

    # pagination
    start = request.args.get('start', type=int)
    length = request.args.get('length', type=int)
    query = query.offset(start).limit(length)

    # response
    return {
        'data': [user.to_dict() for user in query],
        'recordsFiltered': total_filtered,
        'recordsTotal': product.query.count(),
        'draw': request.args.get('draw', type=int),
    }
```

DATABASE MODELS

```
from website import db
from flask_login import UserMixin, FlaskLoginClient
from sqlalchemy.sql import func
from flask_security import RoleMixin, Security,
SQLAlchemySessionUserDatastore









class Users(db.Model, UserMixin):
    __tablename__='users'
    def get_id(self):
           return (self.UserID)
    UserID = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(100), unique=True)
    password = db.Column(db.String(150))
    accesslevel = db.Column(db.String(20))



class product(db.Model):
    ProductID = db.Column(db.Integer, primary_key=True)
    ProductName = db.Column(db.String(150))
    stocklevel = db.Column(db.Integer)
    barcode = db.Column(db.String(13))
    Pricing = db.Column(db.Numeric())

    def to_dict(self):
        return {
            'ProductID':self.ProductID,
            'ProductName':self.ProductName,
            'stocklevel':self.stocklevel,
            'barcode':self.barcode,
            'Pricing':self.Pricing
        }

#class stockcounter(db.Model):
#    ProductID = db.Column(db.Integer,
db.ForeignKey('product.ProductID'))
```

```
#    stockremaining = db.Column(db.Integer)
#    criticalthreshold = db.Column(db.Integer)

#holds roles
class Role(db.Model, RoleMixin):
    __tablename__ = 'role'
    RoleID = db.Column(db.Integer(), primary_key=True)
    rolename = db.Column(db.String(80), unique=True)

# table used to assign roles
roles_users = db.Table('roles_users',
        db.Column('UserID', db.Integer(),
db.ForeignKey('users.UserID')),
        db.Column('RoleID', db.Integer(),
db.ForeignKey('role.RoleID')))
```

MORE VIEWS

```
from flask import Blueprint, render_template, request, flash, redirect,
url_for
from werkzeug.security import generate_password_hash,
check_password_hash
from flask_login import login_user, login_required, logout_user,
current_user, login_manager
from . import models
from .models import Users

auth = Blueprint('auth', __name__)

#LOGIN PAGE
@auth.route('/', methods=['GET', 'POST'])
def login():
    if request.method== 'POST':
        name=request.form.get('name')
        password = request.form.get('password')

        user= Users.query.filter_by(name=name).first()
        if user:
            if check_password_hash(user.password, password):
                flash('Logged in successfully!', category='success')
                login_user(user, remember=True)
                return redirect(url_for('views.adminhome'))
```

```
            else:
                flash('Incorrect details, try again', category='error')



    data=request.form

    return render_template('login.html')

#LOGOUT
@auth.route('/logout')
@login_required
def logout():
    logout_user()
    return redirect(url_for('auth.login'))
```

TEMPLATES

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">

    <link rel="stylesheet" type="text/css"
href="https://cdn.datatables.net/1.10.25/css/dataTables.bootstrap5.css"
>

    <link
      rel="stylesheet"

href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.
min.css"

integrity="sha384-Vkoo8x4CGsO3+Hhxv8T/Q5PaXtkKtu6ug5TOeNV6gBiFeWPGFN9Mu
hOf23Q9Ifjh"
      crossorigin="anonymous"
    />
    <link
      rel="stylesheet"
```

```
href="https://stackpath.bootstrapcdn.com/font-awesome/4.7.0/css/font-awesome.min.css"
      crossorigin="anonymous"
    />
    <link
      rel="stylesheet"
      href="https://use.fontawesome.com/releases/v5.7.0/css/all.css"

integrity="sha384-lZN37f5QGtY3VHgisS14W3ExzMWZxybE1SJSEsQp9S+oqd12jhcu+A56Ebc1zFSJ"
      crossorigin="anonymous"
    />
    <title>{% block title %} Home {% endblock %} </title>
</head>
<body>


  <nav class="navbar navbar-expand-lg navbar-dark bg-dark">
  <a class="navbar-brand" href"#"> Navbar </a>
    <button
      class="navbar-toggler"
      type="button"
      data-toggle="collapse"
      data-target="#navbar"
    >
      <span class="navbar-toggler-icon"></span>
    </button>
    <div class="collapse navbar-collapse" id="navbar">
      <ul class="navbar-nav mr-auto">
        <li class="nav-item dropdown">
          <a class="nav-link dropdown-toggle" href="/stocks" id="navbarDropdown" role="button" data-toggle="dropdown" aria-haspopup="true" aria-expanded="false">
            Stocks
          </a>
          <div class="dropdown-menu" aria-labelledby="navbarDropdown">
            <a class="dropdown-item" href="/stocks/edit">Edit Stocks</a>
            <a class="dropdown-item" href="/stocks/add">Add Stocks</a>
            <div class="dropdown-divider"></div>
            <a class="dropdown-item" href="/stocks/delete">Delete Stocks</a>
          </div>
        </li>
```

```
        <li class="nav-item dropdown">
          <a class="nav-link dropdown-toggle" href="/userManagement"
id="navbarDropdown" role="button" data-toggle="dropdown"
aria-haspopup="true" aria-expanded="false">
            User Management
          </a>
          <div class="dropdown-menu" aria-labelledby="navbarDropdown">
            <a class="dropdown-item"
href="/adminhome/userManagement/createuser">Create user</a>
            <a class="dropdown-item"
href="/adminhome/userManagement/deleteuser">Remove user</a>
          </div>
        </li>









      <div class="navbar-nav">


        <a class="nav-item nav-link" id="logout"
href="/logout">Logout</a>

      </div>
    </div>
  </nav>

  {% with messages = get_flashed_messages(with_categories=true) %}
  {% if messages %}
    {% for category, message in messages %}
    {% if category == 'error'%}
    <div class="alert alert-danger alter-dismissable fade show"
role="alert">
      {{ message }}
      <button type="button" class="close" data-dismiss="alert">
        <span aria-hidden="true">&times;</span>
      </button>
    </div>
    {% else %}
    <div class="alert alert-success alter-dismissable fade show"
role="alert">
      {{ message }}
```

```
      <button type="button" class="close" data-dismiss="alert">
        <span aria-hidden="true">&times;</span>
      </button>
    </div>
    {% endif %}
    {% endfor %}
  {% endif %}
  {% endwith %}


  <!--BEYOND THIS POINT IS ACTUAL CONTENT ON THE PAGE! -->

  <!-- CONTENT BLOCK HOLDS ACTUAL PAGE, NAVBAR ALWAYS PRESENT -->
  <div class="container">
  {%block content%}
  {%endblock%}
  </div>









  <script
    src="https://code.jquery.com/jquery-3.2.1.slim.min.js"

integrity="sha384-KJ3o2DKtIkvYIK3UENzmM7KCkRr/rE9/Qpg6aAZGJwFDMVNA/GpGF
F93hXpG5KkN"
    crossorigin="anonymous"
  ></script>
  <script

src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.12.9/umd/popper
.min.js"

integrity="sha384-ApNbgh9B+Y1QKtv3Rn7W3mgPxhU9K/ScQsAP7hUibX39j7fakFPsk
vXusvfa0b4Q"
    crossorigin="anonymous"
  ></script>
  <script
```

```
src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/js/bootstrap.min.j
s"

integrity="sha384-JZR6Spejh4U02d8jOt6vLEHfe/JQGiRRSQQxSfFWpi1MquVdAyjUa
r5+76PVCmYl"
    crossorigin="anonymous"
  ></script>
</body>
</html>
```

```
{% extends "admin base.html" %}
{% block title%} Homepage {% endblock%}

{%block content%}
<h1> Welcome! <h1>




{%endblock%}
```

```
{% extends "admin base.html" %} {%block title%} User Management
{%endblock%}

{% block content %}
<div class="container">

    <form method="POST">
        <h3 align="center"> Create new user <h3>
        <div class="form-group">
          <label for="Name">Name</label>
          <input
             type="text"
             class="form-control"
             id="name"
             name="name"
             placeholder="Enter Name"
        />
        </div>

        <div class="form-group">
          <label for="password1">Password</label>
          <input
             type="password"
```

```
            class="form-control"
            id="password1"
            name="password1"
            placeholder="Enter Password"
        />
        </div>

        <div class="form-group">
          <label for="password2">Password (Confirm)</label>
          <input
             type="password"
             class="form-control"
             id="password2"
             name="password2"
             placeholder="Confim Password"
         />
        </div>
        <div class="form-group">
          <label for="accesslevel"> User Access Level </label>
          <select class="form-control" id="accesslevel"
name="accesslevel" placeholder="Select User Access Level">
          <option>Normal User</option>
          <option>Admin</option>
          </select>

        </div>

        <br>
        <button type="submit" class="btn btn-primary"> Submit </button>

</div>




{%endblock%}
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
```

```
    <link
      rel="stylesheet"

href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.
min.css"

integrity="sha384-Vkoo8x4CGsO3+Hhxv8T/Q5PaXtkKtu6ug5TOeNV6gBiFeWPGFN9Mu
hOf23Q9Ifjh"
      crossorigin="anonymous"
    />
    <link
      rel="stylesheet"

href="https://stackpath.bootstrapcdn.com/font-awesome/4.7.0/css/font-aw
esome.min.css"
      crossorigin="anonymous"
    />
    <link
      rel="stylesheet"
      href="https://use.fontawesome.com/releases/v5.7.0/css/all.css"

integrity="sha384-lZN37f5QGtY3VHgisS14W3ExzMWZxybE1SJSEsQp9S+oqd12jhcu+
A56Ebc1zFSJ"
      crossorigin="anonymous"
    />
    <title> Login Page </title>

</head>

<body>
<br>
<div class="container">
  <form method="POST">
    <h3 align="center"> Please enter your login details: <h3>
    <div class="form-group">
        <label for="Username">Name</label>
        <input
            type="text"
            class="form-control"
            id="name"
            name="name"
            placeholder="Enter Name"
        />
    </div>
    <div class="form-group">
```

```
        <label for="password1">Password</label>
        <input
            type="password"
            class="form-control"
            id="password"
            name="password"
            placeholder="Enter Password"
        />
    </div>
    <br>
    <button type="Login" class="btn btn-primary"> Submit </button>
  </div>

<br>









  <script
    src="https://code.jquery.com/jquery-3.2.1.slim.min.js"

integrity="sha384-KJ3o2DKtIkvYIK3UENzmM7KCkRr/rE9/Qpg6aAZGJwFDMVNA/GpGF
F93hXpG5KkN"
    crossorigin="anonymous"
  ></script>
  <script

src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.12.9/umd/popper
.min.js"

integrity="sha384-ApNbgh9B+Y1QKtv3Rn7W3mgPxhU9K/ScQsAP7hUibX39j7fakFPsk
vXusvfa0b4Q"
    crossorigin="anonymous"
  ></script>
  <script

src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/js/bootstrap.min.j
s"
```

```
integrity="sha384-JZR6Spejh4U02d8jOt6vLEHfe/JQGiRRSQQxSfFWpi1MquVdAyjUa
r5+76PVCmYl"
    crossorigin="anonymous"
  ></script>
</body>
</html>
```

```
{% extends "admin base.html" %}
{% block content %}

<form method="POST">
  <h3 align="center"> Create new stock entry <h3>
  <div class="form-group">
    <label for="">Product Name</label>
    <input
       type="text"
       class="form-control"
       id="ProductName"
       name="ProductName"
       placeholder="Enter ProductName"
  />
  </div>

  <div class="form-group">
    <label for="stocklevel">stocklevel</label>
    <input
       type="number"
       class="form-control"
       id="stocklevel"
       name="stocklevel"
       placeholder="Enter stocklevel"
  />
  </div>

  <div class="form-group">
    <label for="barcode">Barcode</label>
    <input
       type="text"
       class="form-control"
       id="barcode"
       name="barcode"
       placeholder="Enter barcode"
   />
  </div>
```

```
  <div class="form-group">
    <label for="Pricing">Pricing</label>
    <input
       type="number"
       class="form-control"
       id="Pricing"
       name="Pricing"
       placeholder="Enter Pricing"
    />
  </div>
  <br>
  <button type="submit" class="btn btn-primary"> Submit </button>









{%endblock%}
```

```
{% extends "admin base.html" %}
{% block content %}


<div class="container">

    <form method="POST">
        <h3 align="center"> Enter ProductID for deletion <h3>
        <div class="form-group">
          <label for="Name">ProductID</label>
          <input
             type="int"
             class="form-control"
             id="ProductID"
             name="ProductID"
             placeholder="Enter ProductID"
        />
        </div>
        <button type="submit" class="btn btn-primary"> Submit </button>
</div>
<br>
```

```
{%endblock%}
```

```
{% extends "admin base.html" %}
{% block content %}

<div class="container">

    <form method="POST">
        <h3 align="center"> Stock Edit Page <h3>
        <div class="form-group">
          <label for="ProductID">Designate stock ID to edit</label>
          <input
             type="number"
             class="form-control"
             id="ProductID"
             name="ProductID"
             placeholder="Enter ProductID"
        />
        </div>

        <div class="form-group">
          <label for="ProductName">Select Product Name</label>
          <input
             type="text"
             class="form-control"
             id="ProductName"
             name="ProductName"
             placeholder="Enter Name"
        />
        </div>

        <div class="form-group">
          <label for="stocklevel">stocklevel</label>
          <input
             type="number"
             class="form-control"
             id="password2"
             name="password2"
             placeholder="Enter stocklevel"
         />
         <div class="form-group">
            <label for="barcode">barcode</label>
            <input
```

```
            type="number"
            class="form-control"
            id="barcode"
            name="barcode"
            placeholder="Enter barcode"
        />
        <div class="form-group">
           <label for="Pricing">Pricing</label>
           <input
            type="number"
            class="form-control"
            id="Pricing"
            name="Pricing"
            placeholder="Enter Pricing"
        />
       </div>
       <br>
       <button type="submit" class="btn btn-primary"> Submit </button>




{%endblock%}
```

```
{% extends "admin base.html" %}

{% block content %}



  <br>
  <div class:"container" align="center">
    <a href="/stocks/add">
        <button class="block">
            Add Stock <img src="/static/plus_icon.png"
            style="width: 30px; height: 20px; object-fit: contain;"
            ></i>
        </button>
      </a>


    <a href="/stocks/edit">
        <button class="block">
            Edit Stock <img src="/static/edit_icon.png"
            style="width: 30px; height: 20px; object-fit: contain;"
```

```
            ></i>
        </button>

    </a>
  <a href="/stocks/delete">
        <button class="block">
            Delete Stock <img src="/static/recycle-bin-icon-25.png"
            style="width: 30px; height: 20px; object-fit: contain;"
            ></i>
        </button>

    </a>







</div>

<br>


<table id="data" class="table table-striped">
  <thead>
    <tr>
      <th>ProductID</th>
      <th>ProductName</th>
      <th>stocklevel</th>
      <th>barcode</th>
      <th>Pricing</th>
    </tr>
    <tr>
      <td>1</td>
      <td>Pepsi</td>
      <td>52</td>
      <td>1234789123489</td>
      <td>£1</td>
    </tr>
    <tr>
      <td>2</td>
      <td>Candy</td>
      <td>800</td>
      <td>4564564564567</td>
      <td>£2</td>
```

```
      </tr>
      <tr>
        <td>3</td>
        <td>Cookies</td>
        <td>45</td>
        <td>0966784564818</td>
        <td>£2.40</td>
      </tr>



    </thead>

    <tbody>
    </tbody>
  </table>
{% endblock %}

{% block scripts %}

  <script type="text/javascript" charset="utf8"
src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
  <script type="text/javascript" charset="utf8"
src="https://cdn.datatables.net/1.10.25/js/jquery.dataTables.js"></scri
pt>
  <script type="text/javascript" charset="utf8"
src="https://cdn.datatables.net/1.10.25/js/dataTables.bootstrap5.js"></
script>
  <script>
    $(document).ready(function () {
      $('#data').DataTable({
        ajax: '/stocks/data',
        columns: [
          {data: 'ProductID'},
          {data: 'ProductName'},
          {data: 'stocklevel'},
          {data: 'barcode'},
          {data: 'Pricing'}
        ],
      });
    });
  </script>
{% endblock %}
```

```
{% extends "admin base.html" %}
```

```
{% block content %}


<div class="container">

    <form method="POST">
        <h3 align="center"> Enter UserID for deletion <h3>
        <div class="form-group">
          <label for="Name">UserID</label>
          <input
             type="int"
             class="form-control"
             id="UserID"
             name="UserID"
             placeholder="Enter UserID"
        />
        </div>
        <button type="submit" class="btn btn-primary"> Submit </button>
</div>
<br>









{%endblock%}
```

```
{% extends "admin base.html" %}

{% block content %}



  <br>
  <div class:"container" align="center">
    <a href="/adminhome/userManagement/createuser">
        <button class="block">
            Add User <img src="/static/plus_icon.png"
            style="width: 30px; height: 20px; object-fit: contain;"
            ></i>
        </button>
```

```
    </a>


  <a href="/adminhome/userManagement/deleteuser">
      <button class="block">
          Delete User <img src="/static/recycle-bin-icon-25.png"
          style="width: 30px; height: 20px; object-fit: contain;"
          ></i>
      </button>

    </a>









</div>

<br>


<table id="data" class="table table-striped">
  <thead>
    <tr>
      <th>UserID</th>
      <th>Username</th>
      <th>access level</th>
    </tr>
    <tr>
      <td>1</td>
      <td>sus1</td>
      <td>admin</td>

    </tr>
    <tr>
      <td></td>
      <td></td>
      <td></td>

    </tr>
    <tr>
      <td></td>
      <td></td>
```

```
        <td></td>

      </tr>



    </thead>

    <tbody>
    </tbody>
  </table>
{% endblock %}

{% block scripts %}

  <script type="text/javascript" charset="utf8"
src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
  <script type="text/javascript" charset="utf8"
src="https://cdn.datatables.net/1.10.25/js/jquery.dataTables.js"></scri
pt>
  <script type="text/javascript" charset="utf8"
src="https://cdn.datatables.net/1.10.25/js/dataTables.bootstrap5.js"></
script>
  <script>
    $(document).ready(function () {
      $('#data').DataTable({
        ajax: '/stocks/data',
        columns: [
          {data: 'ProductID'},
          {data: 'ProductName'},
          {data: 'stocklevel'},
          {data: 'barcode'},
          {data: 'Pricing'}
        ],
      });
    });
  </script>
{% endblock %}
```