Table of Contents

1 Introduction

This is a README file describing the implementation of the system designed for the first assignment, which I have chosen to call Marketplace..

2 Instructions

2.1 Code Execution

The main actor in the system is the user. She has been modelled as a concrete superclass, The database, in my workspace, is called marketplace. To initialise the database, please execute:

```
mysql -h localhost -u root marketplace < database/schema.sql
```

To run the unit tests:

```
python3 unit_tests.py
```

To start the application:

```
python3 application.py
```

To use the application, please go to the url:

https://rival-system-5000.codio-box.uk/login

2.2 Sample User Login

From the login page, you can choose to log in as a customer, or a seller. Both logins will lead to different flows, and gain access to different pages of the application.

Sample Customer Details

username: cust1 password: 123456 name: Ollie	username: cust2 password: 118118 name: James	username: cust3 password: abc123 name: Michael
cvv: 123	cvv: 999	cvv: 555

Sample Seller Details:

username: sell1 password: mypass name: Billy	username: sell2 password: secret name: Sarah	

3 Functionality

3.1 Customer Functionality

The customer can add products to their basket. Upon checkout, the basket's contents are shown, and the total cost displayed. To pay, the payment details page displays three types of payment: Card, OnlinePayment, Voucher. The user's card and online payment details stored in the database are used to prepopulate the form.

Click the radio button to indicate which type of payment to make. If using the card payment, enter the cvv manually - it is not populated automatically for security reasons. :) To use a voucher, please use the sample details provided in the box below.

Sample Voucher Details:

number:	`1111`	number:	`2222`	number:	`3333`
value:	`50.00`	value:	`100.00`	value:	`5.00`

NOTE: Incorrect details, e.g. manually changing the number of the online payment system so that it doesn't match that which is connected to their account, an error message will be displayed. Similarly, a voucher of insufficient value to cover a purchase results in an error.

Upon successful payment, the basket is emptied and the customer is returned to the initial page of products for sale.

3.1 Seller Functionality

A seller can add new products to their catalogue. They can edit the price of any of the products in their catalogue. They can also update their shopfront by entering a different png/jpeg file.

In the sample data, user `sell1` has seven products in their catalogue, while `sell2` has two. Both sellers stock `biscuits` but for different prices, so `biscuits` should be listed twice in the products view that the customer gets, each with its respective price.

Sample Voucher Details:

number:	1111	number:	2222	number:	3333
value:	50.00	value:	100.00	value:	5.00

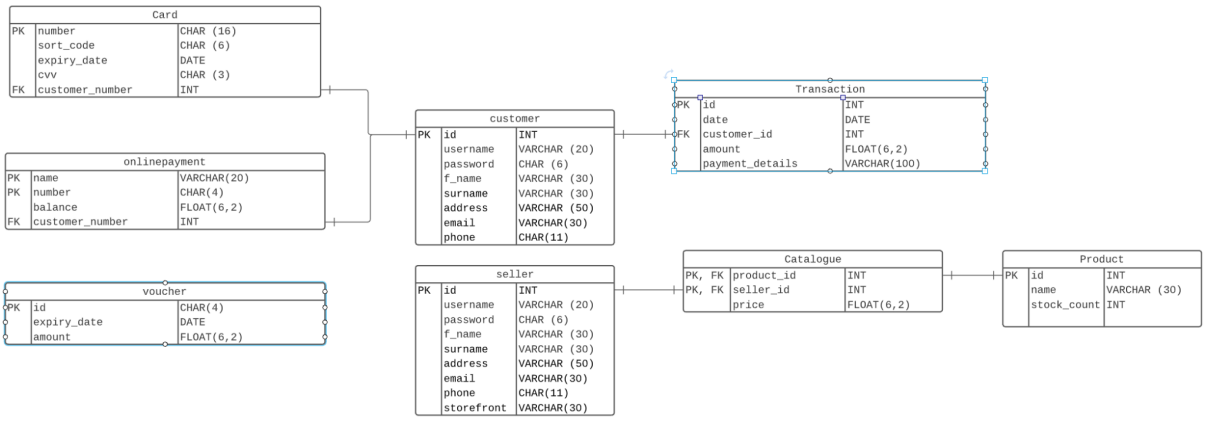
NOTE: If the customer enters any incorrect details, e.g. manually changes the number of the online payment system so that it doesn't match that which is connected to their account, an error message will be displayed. Similarly, if a customer tries to use a voucher of insufficient value to cover their purchase, an error will be displayed.

Upon successful payment, their basket is emptied and the customer is returned to the initial page displaying all of the products for sale.

4 Testing

Many of the classes used by the application have associated unit tests, Basket, Catalogue, Customer, External, PaymentDetails, OnlinePayment, Product, Card, Voucher. There are some exceptions, due to time constraints, but my hope is that the included tests sufficiently demonstrate my understanding of the concept. The tests are in `unit_tests.py`
Extensive functional and integration testing was carried out to test the validity of results being written to the database and the error handling ability of the application.

5 Database Structure



6 Project Structure

The project's files are arranged as thus:

- `application.py` : the main code for the running of the application and for the flask routes.
- `helpers.py` : common methods such as those for accessing the database.
- `errors.py` : declarations of error classes which are used to handle exceptions.
- `constants.py` : commonly used regex patterns and strings.
- `classes.py` : the main classes used for data storage
- `unit_tests.py` : the unit tests for those classes.
- database folder : `schema.sql` to build the database and populate with test data.
- static folder: `styles.css` styling for the html pages
- templates folder: All `html` files

6 Limitations

1. Though there is a registration page for new users, this has not been implemented and the page doesn't write to the database, or save new users.
2. When a customer pays for an order, a transaction is written to the database. With further time, an order should also be produced and stored in the db for warehouse operatives to access/fulfil.
3. Though there are stock levels for each of the products in the database, this number is not modified after the customer has made a purchase. As proof that, given more time, I could have implemented this, the `onlinepayment` table is updated, and the user's paypal balance is debited effectively when purchases are made.

6 Conclusion

The use of objects and object oriented principles to define classes used for data manipulation definitely made some of the logic within the application easier to implement. It also meant that functions such as process_payment and create_transaction could be easily tested using Python's unit testing module.