Paypal payment API

Overview

During this coursework the team and I implemented a flight aggregator software that can be interacted with through the command-line. I worked on the payment-API where I implemented features to create a simple payment account, pay for an order and create a transaction out of that order, store different credit cards along with the billing address and transactions for each account on the payment system.

Database Models

To get the code functioning as intended these were the database models set up. The models comply with the spec introduced in CW1 to allow for other team-members to interact with my models and api-endpoints with ease. The models created are

1.) User model
2.) Card Model
3.) Billing Address Model
4.) Transaction Model
5.) Order Model

User Model

```
class testUser(models.Model):
    userId = models.BigAutoField(primary_key=True, unique=True)
    username = models.CharField(max_length=16)
    def __str__(self):
        return self.username
```

Figure 1.1

Firstly, a very simple user model was created, this user model **(Fig 1.1)** was used to link cards and transactions to each user. The user model could be made more secure with a password or via making the user ID a UUID4 instead of the Django's ID default BigAutoField, but for ease of interaction and testing the default BAF was used.

Order Model

```
class TestOrder(models.Model):
    itemId = models.IntegerField(primary_key=True,unique=True)
    itemName = models.CharField(max_length=30)
    itemCode = models.CharField(max_length=26)
    itemPrice = models.CharField(max_length=22)
```

Figure 1.2

The order model **(Fig 1.2)** was implemented in order to integrate and test my APIs interaction with orders until the team members set up their APIs. The order model was explicitly used to set up the transaction

Card Model

```
class Card(models.Model):
    card_number_format_validator = RegexValidator(
        regex=r'^\d{4}\-\d{4}\-\d{4}\-\d{4}$',
        message='Card number must be in the format "xxxx-xxxx-xxxx-xxxx" where x is a number'
    )

    cvv_number_format_validator = RegexValidator(
        regex=r'^\d{3,4}$',
        message='CVV number must be either 3 or 4 digits.'
    )

    expiry_date_format_validator = RegexValidator(
        regex=r'^\d{2}/\d{4}$',
        message='Expiry date must be in the format "MM/YYYY".'
    )
    #add card ID
    card_number = models.CharField(max_length=19,
                                   null=False, blank = False,
                                   primary_key=True, unique=True,
                                   validators=[card_number_format_validator],)
    cvv = models.CharField(validators=[MinLengthValidator(3), cvv_number_format_validator], max_length=4, null=False, blank=False)
    expiry_date = models.CharField(null=False, blank = False, max_length=16)
    name_on_card = models.CharField(max_length=26)
    user_card = models.ForeignKey(testUser, on_delete=models.CASCADE, blank=True,null=True)
    last_4_digits = models.CharField(max_length=4, null=True, blank=True, editable=False)


    def save(self, *args, **kwargs):
        if not self.last_4_digits:
            self.last_4_digits = self.card_number[-4:]
        super().save(*args, **kwargs)


    def __str__(self):
        return self.name_on_card
```

Figure 1.3

The card model **(Fig 1.3)** was the most complicated model to set up. Regex validators were used to ensure that card details are consistent and imitate real credit card credentials. For the card number, the regex ensures that users enter 4 digits followed by a '-' except for the last 4 digits, there is no '-' at the end. A check on the cvv was done to ensure that the cvv is between 3-4 digits. Finally, another check on the date to make sure it is entered in this format where d stands for digit, 'dd/dddd'. The card number was used as the primary key for

the card model. A function is defined to get the last 4 digits of a given credit card, the function checks for the last 4 digits of the credit card and saves it to the credit card on creation.

Billing Address Model

```
class BillingAddress(models.Model):
    card = models.OneToOneField(
        Card,
        on_delete=models.CASCADE,
        primary_key=True)
    full_name = models.CharField(max_length=36, null=False, blank=False)
    address_1 = models.CharField(max_length=50, null= False, blank=False)
    address_2 = models.CharField(max_length=50, blank= True)
    city = models.CharField(max_length=20, null = False , blank=False)
    postcode = models.CharField(max_length=10, null=False, blank= False)
    region = models.CharField(max_length=26, null=False, blank= False)
    country = models.CharField(max_length=40, null=False, blank= False)
```

Figure 1.4

The billing address **(Fig 1.4)** is set up as a one to one relation with the card model, meaning that each registered card will have a billing address associated with it. The format checks are done in the main view instead of the actual model and are explained in the next chapter.

Transaction Model

```
class Transaction(models.Model):
    transaction_id = models.UUIDField(default = uuid.uuid4, unique=True, editable = False, db_index=True)
    #merchant_id -- get from merchant
    date_created = models.DateTimeField(auto_now_add=True)
    amount_paid = models.DecimalField(max_digits = 9, decimal_places=2) #get from merchant
    user_transaction = models.ForeignKey(testUser, on_delete=models.CASCADE, blank = True, null = True)
    last_4_digits = models.CharField(max_length=4, null=True, blank=True)
    service = models.CharField(default = 'PayPal', editable=False, max_length=6)
    status = models.CharField(default = 'Outstanding', max_length=16)
```

Figure 1.5

The transaction mode **(Fig 1.5)** is set up as a many to one relation with the user model, as defined in the user_transaction field, where the user model is used as a foreign key. The transaction_id is stored as a uuid-4 (universally unique identifier), which is a 128 bit label; this was implemented as an attempt to imitate real-world transaction IDs **(Fig 1.6)**. The default status of a transaction is outstanding until the order is paid for. The date created of a transaction is automatically assigned on creation **(Fig 1.7).**

```
"transaction_id": "5419b882-2049-4b37-8e91-0cd9fce06c6b",
```

Figure 1.6

```
'date_created": "2023-05-08T18:08:41.469261Z",
```

Figure 1.7

Serialisers

Very simple serializers were created in order to deserialize the json objects being sent from the API to the client. (Fig 1.8) shows an example serializer.

```
class BillingAddressSerializer(serializers.ModelSerializer):
    class Meta:
        model = BillingAddress
        fields = "__all__"
```

Figure 1.8

Api-endpoints

API-endpoints were utilised in order to communicate with the client and other parts of the system. The table below shows what each endpoint does and the type of request handled. **(Table 2.1).**

Endpoint	Request	Function
/main		Command line interface
/newcard/	POST	Registers new card
/newuser/	POST	Registers new user
/usercards/<str:pk>/	GET	Gets given user registered cards
/usertransactions/<str:pk>/	GET	Gets given users transactions
/getuser/<str:pk>/	GET	Gets given user
/newBA/	POST	Registers billing address
/createTransaction/<str:order_id>	POST	Registers new transaction for given order, for flight companies
/canceltransaction/<str:pk>/	PATCH	Refunds given transaction
/createOrder	POST	Creates new order, called by flight company

Table 2.1

As specified in the CW1 spec, the end-points, createTransaction/{orderId} and /createOrder were both implemented to allow the flight companies to call them in order to create the order and transaction through this payment system. Once the transaction is created, the view calls the view, main, which prompts the user to login and choose their credit card and ask the user to refund/cancel any transactions, this is all done from within the command line interface discussed in the next section.

Command line interface (Main)

The command line interface is where all the functions are carried out, once paypal/main is called the the user interface loads up on the terminal where the user can interact with the system.

Grabbing the order

When create order is called by the flight system, the itemName, price and details are sent as a json and posted, once the order is created, the id of the order is sent back to the flight system. The flight system uses that order ID to create a transaction for given order, once that is done they are sent to the payment system command-line prompt in their terminal.

Checking for user

```
user = False
while user == False:
    curr = input("please enter your userID")
    response = requests.get(f"http://localhost:8000/PayApi/getuser/{curr}/")
    if response.status_code == 200:
        print("User Found")
        card_list = response.json()
        user = True
    else:
        print("User not found\n")
        new_user_choice = input('Would you like to create a new user? (y/n): ')
        if new_user_choice == 'y':
            username = input('Please enter your username: ')
            new_user = {
                'username': username,
            }
            response = requests.post("http://localhost:8000/PayApi/newuser/", data=new_user)
        else:
            print("\n")
```

Figure 2.2

Once the user is transferred to the payment command-line, they are asked to enter their ID, if the user does not have an ID they are prompted to enter a user-name and they are assigned an ID which is presented to them on the screen. The user needs to remember their ID to access their cards and transactions. Once the user enters their ID, their credit cards are grabbed from the API and stored in the variable card list. If the user has no cards on their account they are prompted to create a new credit card with the same regex checks as in the database model. If they choose to create a credit card, they are prompted to enter the details for their credit card and the billing address associated with that credit card. **(Fig 2.3).** Once the details are entered they are used to create a card object and a billing address object that are posted to the following endpoints

```
please enter your userID
1
User Found
[10/May/2023 10:06:26] "GET /PayApi/getuser/1/ HTTP/1.1" 200 31
[10/May/2023 10:06:28] "GET /PayApi/usercards/1/ HTTP/1.1" 200 142
would you like to use a registered card, (y/n)? If there are no registered cards you will be prompted to create a new one:  n
enter the details of the new card
Enter card number (format "xxxx-xxxx-xxxx-xxxx"): 4516-5678-1234-1123


Enter CVV number (3 or 4 digits): 676


Enter expiry date (format "MM/YYYY"): 10/2036
Enter name on card: ahmed khaled
-------------------- BILLING ADDRESS DETAILS --------------------
Please enter your name: ahmed khaled
Please enter first address: 32 harold road
Please enter second address line, you can leave this empty:
Please enter city: leeds
Please enter post code: ls2 3es
Please enter region: west yorkshire
Please enter country: leeds
[10/May/2023 10:07:28] "POST /PayApi/newcard/ HTTP/1.1" 201 143
[10/May/2023 10:07:30] "POST /PayApi/newBA/ HTTP/1.1" 201 181
[10/May/2023 10:07:32] "POST /PayApi/newtransaction/ HTTP/1.1" 201 211
Would you like to refund/cancel any transaction made on your account (y/n)?n
[10/May/2023 10:07:35] "GET /PayApi/main HTTP/1.1" 200 39
```

Figure 2.3

Cancelling a transaction

The user can request to refund a given transaction at the end after paying for the current order, the user is presented with all their transactions they can then choose from the list which transaction they would like to refund. **(Fig 2.4)**

```
please enter your userID
1
[10/May/2023 07:44:51] "GET /PayApi/getuser/1/ HTTP/1.1" 200 31
User Found
[10/May/2023 07:44:53] "GET /PayApi/usercards/1/ HTTP/1.1" 200 142
would you like to use a registered card, (y/n)? If there are no registered cards you will be prompted to create a new one:  y
which card would you like to pay with

1 Card ending with ** 9999
enter the card you want to pay with 1
1
[10/May/2023 07:44:58] "POST /PayApi/newtransaction/ HTTP/1.1" 201 211
Would you like to refund/cancel any transaction made on your account (y/n)?y
[10/May/2023 07:45:02] "GET /PayApi/usertransactions/1/ HTTP/1.1" 200 1500
1 Transaction:  5419b882-2049-4b37-8e91-0cd9fce06c6b Amount:  200.00 Status:  Cancelled
2 Transaction:  2646ea80-dd61-4794-b042-71d221c0c7d4 Amount:  200.00 Status:  Paid
3 Transaction:  0110e11f-472a-4ca9-a582-e4831f3a811e Amount:  200.00 Status:  Paid
4 Transaction:  bcb21419-5a19-4052-9b1a-8b8ca78a51cd Amount:  200.00 Status:  Cancelled
5 Transaction:  ba834e79-0fb2-4af1-8162-421a4ed2c304 Amount:  200.00 Status:  Cancelled
6 Transaction:  917d7a3a-c5c8-4bb4-859a-7a137bf42b45 Amount:  200.00 Status:  Paid
7 Transaction:  01bb9836-0398-4360-9713-8b7e9925bcc3 Amount:  200.00 Status:  Paid
Which transaction do you wish to refund? 3
[10/May/2023 07:45:08] "PATCH /PayApi/canceltransaction/0110e11f-472a-4ca9-a582-e4831f3a811e/ HTTP/1.1" 200 216
Would you like to refund/cancel any transaction made on your account (y/n)?n
```

Figure 2.4

Once the user chooses the transaction they would like to refund, a patch request is sent along with the transaction id in order to refund it. **(Fig 2.5)**

```
transaction_cancel = {
    "status" : "Cancelled",
}

response = requests.patch(f"http://localhost:8000/PayApi/canceltransaction/{transaction['transaction_id']}/", data=transaction_cancel)
```

Figure 2.5

Once the user is satisfied with using the payment system they can opt out by entering ‘n’ in the command prompt when asked if they would like to refund any transactions. They are then redirected to this html message **(Fig 2.6)** and the terminal shuts down

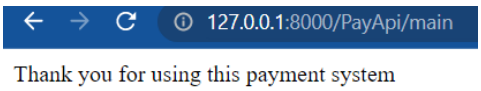

Figure 2.6

Conclusion

Overall, the service followed the spec in CW1, the api endpoints adhered to the ones specified, with a few extra additions such as registering new credit cards for accounts.Since the spec was well-made and communicated no changes were required for this service. All the api-endpoints were tested using postman to ensure that they work before sharing the code with the rest of the team.