COMP3011 Coursework 2 Report

Razvan-Antonio Berbece (sc19rab)

PayFriend Payment Provider Service

Table of Contents

Introduction

The payment provider service that I implemented (PayFriend) is a PayPal-like service which processes transactions from payer to payee using simple user accounts. It supports user registration with an email and password, user authentication, transaction processing, transaction storing and transaction deletion.

Solution Architecture

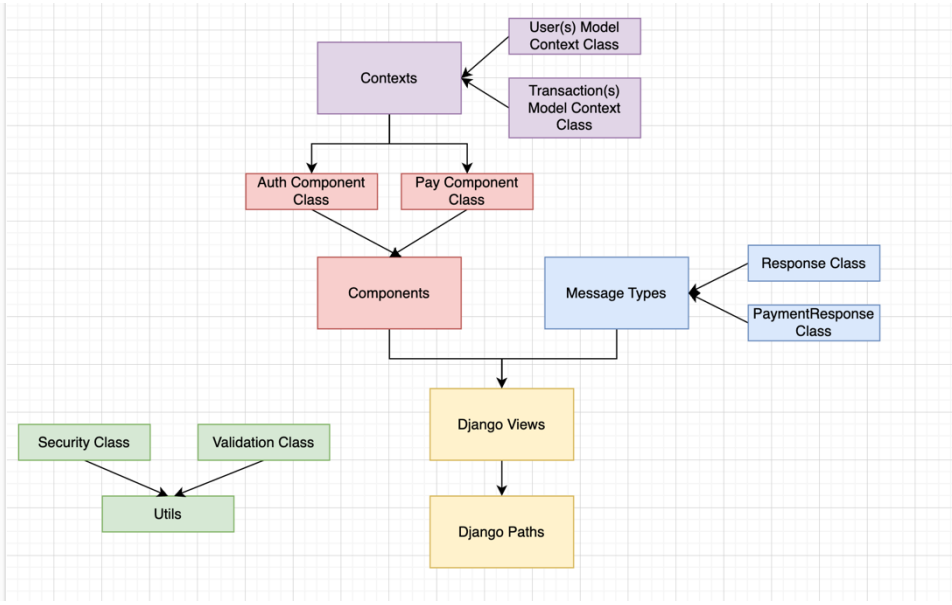

Figure 1 - The architecture of the Django project (I used OOP practices and employed separation of concerns).

To keep the project codebase clean and tidy and to employ separation of concerns in the program logic, I had to reason about what each project scope really cares about. After careful consideration, I decided that:

- The Django Views only make use of:
    - Response message type classes
        - Response Class which is used as a return value for account or transaction management requests.
        - PaymentResponse Class which is used for transaction processing requests.
    - Components (Custom logic)
        - Auth component which provides the heavy logic methods for the authentication feature of the service (registration and signing in).
        - Pay component which provides the heavy logic methods for the payment processing feature of the service (recording transactions, retrieving, and deleting them).
- The Auth and Pay components only make use of:
    - The DB Context classes
        - User(s) model context class interfaces with the Users sqlite3 database which holds all the service's registered accounts.
        - Transaction(s) model context class which interfaces with the Transactions sqlite3 database which holds all the transactions processed by the service.
- The Utils classes are used throughout the project whenever required:
    - Security Class which provides methods for cryptographic operations like hashing, salting.
    - Validation Class which provides user input validation methods.

Django Views

There are certain similarities between all the view methods that I implemented, specifically:

- They all return JsonResponse(s) rather than HttpResponse(s). This is so that the service matches the initial design spec from CW1 and to allow clients to parse the responses into language-native objects.
- They all handle error scenarios inside the view method accounting for the return values of the Auth and Pay components function calls. By doing this I made sure that clients know exactly what went wrong and where, providing them with meaningful error messages when needed.
- The view methods first parse any request parameters or form data available and then instantiate Auth or Pay components according to the view's needs.
- After calling the component methods, the return codes are then analysed and a Response Message type is built using the current state of the service (operation result, error message, resource path, timestamp etc.).
- They all handle a very specific part of the system (e.g.: a view method which only handles registration, a view method which only handles payment processing; this helps further in keeping the code clean and emphasizing the separation of concerns)

An example of a view method's execution flow can be seen in the code snippet below, where the similarities listed above can be observed.

```python
# Get POST body data
email = request.POST.get('email')
password = request.POST.get('password')
# Inject component instances
auth_component = AuthComponent()
# Process
status = auth_component.register_user(email, password)
# Send response
timestamp = datetime.now(timezone.utc).timestamp() * 1000 # in milliseconds since Unix epoch
response = Response("/signup/", None, {}, timestamp, 1)
if status < 0:
    if status == -1:
        # User already registered
        response._error = { "message": f"A user with the {email} email address already exists." }
        response.success = 0
        return JsonResponse(response.get_json(), safe = False)
    elif status == -2:
        # Invalid email address
        response._error = { "message": f"The provided email address {email} is invalid to use for registration." }
        response.success = 0
        return JsonResponse(response.get_json(), safe = False)
    elif status == -3:
        # Invalid password
        response._error = { "message": f"The provided password is too weak to use for registration." }
        response.success = 0
        return JsonResponse(response.get_json(), safe = False)
return JsonResponse(response.get_json(), safe = False)
```

Figure 2 – Code snippet from the Register view method emphasizing the workflow that each view method follows.

Response Message Types

The response message classes (Response and ResponsePayment) are simple Python classes with constructors which take in the necessary values for a suitable response (e.g.: operation result, error message, timestamp, IDs, etc.) to be returned to a client.

To make life easier, I added to the response classes methods which return a Python dictionary populated with the class instance field data. This helped me when returning JsonResponse(s) in the view methods, as now all that is necessary to "send an object" back to the client is to call JsonResponse(responseObject.get_json(), safe=False). This also helped in keeping the view methods short and tidy.

Components

Auth Component

The Auth component provides logic for clients to register new accounts and authenticate using credentials. Because this implies database connection and queries, I integrated the Transaction(s) Model Context class within the component to be able to store new accounts in the Users table and to query the table for a given email and password for authentication.

The register_user(email: string, password: string) method takes in an email and a plaintext password which are provided by the view method function (which are obtained from the POST form-body). The method then uses the Utils Validation methods to validate the email (i.e., make sure it is a valid format email address) and the password (i.e., make sure it is longer than 8 characters). If any of the validation checks fail, different return codes are returned to the view method, each with a specific meaning (e.g.: if the code is -2, it means that the email address that the client provided is in an invalid format).
Then, for user account storing, I decided to hash and salt the plaintext password to provide extra layers of security. I used the utility methods that I implemented in the Security class to get a random salt and a hashed version of the user password in the context of register_user. If no user exists with the given email, then I store a user data row in the Users table using the email, hashed password, and the generated salt.

The authenticate_user(email: string, password: string) method takes in an email and a plaintext password which are provided by the view method function (which are obtained from the POST form-body). The method then goes through the same validation process described above.
Then, to determine whether the credentials are correct, and that the user can log in, I retrieve the salt associated to the given email from the Users table. If no salt is found, it means that the account does not exist, at which point the authenticate returns and the view method handles the error. If a salt is successfully retrieved from the table, a hash is computed using the found salt and the plaintext password. The Users table is then queried for an entry with a matching email and hash. If a matching entry is found, a specific return code (0) is returned to the view.

Pay Component

The Pay component provides logic for clients to create new transactions in the system and delete them if necessary. The Pay component makes use of the Auth component, specifically in the process_payment() method. By implementing it like this, I made the client's job easier and adhered to the CW1 spec as users should be able to book a flight without specifically filling in a login form. By doing this, only 1 request is processed by the PayFriend service. As a result, a payment request with an invalid account will cascade fail (both the authentication and the payment fail).

The process_payment(email: str, password: str, value: float, company: str) method takes in account credentials and transaction details like the value of the transaction and the receiving company name. The method first attempts to authenticate a user through the Auth component using authenticate_user(). Based on the result of this operation, the method may return an error code to the view for further processing. If the account credentials match an account, the payment details are validated (transaction value is positive and greater than 0 and that the receiving company name is not an empty string). Provided that authentication and validation passed, and the method did not return, a transaction Python dictionary is built using the available transaction details, a newly generated timestamp and a UUID4 for the transaction identifier. The built transaction dictionary is then stored in the Transactions table using the Transaction(s) Model context.

The delete_payment(transactionId: str) method takes in a transaction ID and attempts to delete the transaction with the given ID from the Transactions table. The method first validates the passed in id (making sure it is not an empty string) and returns a specific error code if necessary to the view to instruct it that the input is not valid. If the input is valid, the method then calls the delete transaction method from the Transaction(s) Model context and passes it the given transaction id. If the status returned by the context method is -1, it means that no transaction was found with the given id and thus it was not deleted, at which point the delete_payment() method returns an error code to the view. If everything went well and the transaction got deleted, then the method returns a value of 0 to the view.

Contexts

User(s) Model Context Class

The User(s) Model Context class interfaces with the Django models and implements methods to add a new user to the table, check whether a user is registered and reading the salt value for a specific user account.

The add_user_to_table(email: str, hash: str, salt: str) method takes in three strings representing the fields for one user account (email, hashed password and the salt used before hashing). It first creates a User model using the passed in data and then calls the .save() method on it to insert it in the database. I decided not to do any other validation in this method because all the necessary validation was done beforehand in the previous scopes of the program execution.

The user_is_registered(email: str, hash: str) method is used specifically for logging into the service, and checks whether a record is retrieved from the Users table where a user has the given email and hash.

All logic under the methods in the User(s) Model Context class is encapsulated in try/except blocks to account for cases in which the table queries do not return anything, in which case values of None are returned to the callers for further error handling. I also marked the methods as static methods because I decided that there is no need for explicit instantiation for the context class, as there is no dependency on field data to query the tables.

Transaction(s) Model Context Class

The Transaction(s) Model Context class interfaces with the Django models and implements methods to add a new transaction to the table, retrieve a transaction with a given ID from the table and to delete a transaction with a given ID from the table.

The add_transaction_to_table(transaction: object) method takes in a Python object which holds the data relevant for a transaction (id, email, company, timestamp, value). It first creates a Transaction model using the passed in data and then calls the .save() method on it to insert it in the database. I decided not to do any other validation in this method because all the necessary validation was done beforehand in the previous scopes of the program execution.

The retrieval (selecting from the table) logic uses transaction IDs and does not require multiple transactions to be retrieved at the same time because of the decided design of both the airlines and the payment services. In the case of reading an entire transaction from the table, the Transaction.objects.get(TransactionId=id) method is used and if a transaction was indeed found, a Python object is built using the selected data. I build the Python object out of the Django model type so that the rest of the application can process it easier after it is returned.

All logic under the methods in the Transaction(s) Model Context class is encapsulated in try/except blocks to account for cases in which the table queries do not return anything, in which case tuples of None and an error message are returned to the callers for further error handling. I also marked the methods as static methods because I decided that there is no need for explicit instantiation for the context class, as there is no dependency on field data to query the tables.

Utils

Throughout the development of the PayFriend service, I had to use common logic in multiple parts of the codebase. As a result, to reduce code duplication I isolated the common code and put it under a “utilities” scope.

Security Class

The get_salted_and_hashed_password(plaintext: str) takes in a plaintext string and salts it with a random salt (generated using the Python secrets module). Then, the salted string is hashed using the hashlib Python library and a SHA256 algorithm. The hashed plaintext and the generated salt are then returned as a tuple for further processing (i.e.: storing in the database under a specific email address).

The get_hashed_with_salt(salt: bytes, plaintext: str) method returns the hash of a plaintext string salted with the given salt parameter. It uses the same hashlib methods and is mainly used for the sign in workflow of a client.

Validation Class

The is_valid_email_address(email: str) method takes in a string which represents an email and uses a regex pattern to validate that it is a valid email address. The regex expression is the standard RFC5322-compliant pattern which covers 99.99% of input email addresses[1]. This method is part of the validation that the Auth component uses for new account registration and authentication.

The is_valid_password(password: str) method takes in a string which represents a password in plaintext and returns false if it has less than 8 characters, in which case I considered it unsafe. More password validation could be added (like it needing to contain special characters, different cases for letters, etc.), but given the scope and the goal of the project, I decided that length is the only required validation that a password requires for the PayFriend service. This method is part of the validation that the Auth component uses for new account registration and authentication.

Testing

To assure that the service works as intended and as expected, I wrote an automated test harness using the Django testing framework Django.test. Each component and context are tested in isolation. The database contexts are unit tested whereas I wrote integration tests for the Auth and Pay components.

By writing these integration tests for the logic heavy components, I simulated real-life scenarios in which actual client data is passed to the components to process.
The tests setup the database in the desired state and then assertions are made on the tables and the return codes of the component methods.

For example, to test the behaviour of clients deleting transactions with a given id, the Users table is firstly configured to contain a registered user, then a valid transaction is processed, the id is recorded and then the transaction is deleted. The assertions are then made on the returned status codes and the error message returned by the component when trying to delete the transaction a second time after the first successful deletion.

Additionally, because I used GitHub for source control, I decided to make use of the GitHub Actions feature to run the Django tests as part of continuous integration on push events to the main branch.
I achieved this by creating a GitHub workflow file in which I specified the Python version to be installed on the GitHub worker which will run the tests, what other dependencies to install and what command to run to run the test harness.
As a result, when pushing to the main branch, I'd be able to check whether the tests are still passing after code updates, without needing to run the tests locally after every change to confirm that everything still works as expected.

Conclusion

By the end of this project, I managed to successfully develop and iterate on my PayFriend service. Last, but not least, the service works as expected and was successfully integrated by my other teammates in their services.

Finally, this project has given me a chance to emphasize the usage of sound OOP practices, to employ test-driven developing while working on the logic-heavy components of the service and to create a CI workflow (continuous integration) on the GitHub repository to test the code on the fly when updated. I had a lot of fun writing and testing it and learning new things about Django, especially because the scope was so open, and I had the freedom to express myself through this web service implementation.

References

[1] https://stackabuse.com/python-validate-email-address-with-regular-expressions-regex/