**COMP3011 Coursework 2 Report**

Razvan-Antonio Berbece (sc19rab)

PayFriend Payment Provider Service

Table of Contents

[Introduction 3](#_Toc130942690)

[Solution Architecture 3](#_Toc130942691)

[Django Views 4](#_Toc130942692)

[Response Message Types 5](#_Toc130942693)

[Components 5](#_Toc130942694)

[Auth Component 5](#_Toc130942695)

[Pay Component 5](#_Toc130942696)

# 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.

The PayFriend service supports user registration with an email and password, user authentication, transaction processing and storing and transaction deletion.

I emphasized using sound OOP practices, I conducted TDD while developing the logic-heavy components of the service and created a CI workflow (continuous integration) on the GitHub repository to test the code on the fly when updated.

# Solution Architecture

Diagram

Description automatically generated

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, I had to think about what each Django module cares about. After 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
    - 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 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.).

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.

Text

Description automatically generated

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 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