Backend Refactoring

Motivation

The backend was refactored heavily during the first sprint of the project, since it had been handed over to us from a previous team and was deemed inextensible, poorly written and had highly incohesive components.

The goal of refactoring the backend API was to improve three aspects of the original codebase, namely:

1. Extensibility
2. Modularity
3. Maintainability

The general purpose of refactoring the codebase was also to improve the general quality of the codebase, stemming from observations within the pre-existing codebase including discernments such as poorly formatted code, poor variable/method naming conventions, and highly bloated classes etc.

Hence, a consensus decision was made by both teams to refactor the Python 3 and Flask backend to use a more pertinent Model-View-Controller (MVC) architecture.

This decision was not made lightly as this would place a sizable technical debt on our team, that would need to be completed before starting on the new features required by the client.

Initial Backend

The existing backend codebase, which was handed over to both teams, was devoid of a logical and extensible architecture. It used five main source code files to encapsulate all backend logic.

For example, all database operations were packaged into a single source file called *database.py*, of approximately 500 SLOC.

This meant that if we were to extend the web application, the single class responsible for handling database operations would become increasingly bloated and incohesive. In turn, this would make the application much harder to test, and less extensible.

Additionally, all code responsible for handling client requests and subsequent application flow logic were placed into two files, leading to low modularity.

Directory & File Structure

The old directory/file structure of the backend API can be seen below:

Old Directory Structure

```
.
 app
    auth.py
    __init__.py
    main.py
 datasources
     database.py
     iam.py
```

Refactored Backend

The refactored backend uses an MVC architecture, which is one of the most widely adopted industry-standard web development frameworks to create scalable and extensible projects.

The pattern separates an application into three main logical components: the model, the view, and the controller.

In the case of our project, since the backend is not responsible for serving web pages, the *view* component is not used in our architecture.

In addition to the *model* and *controller* components, we also employed the use of other common abstractions including the usage of a *service layer* and *resource* components.

A high level diagram displaying the backend architecture, and explanations of key architecture components are presented in the sections below.

Directory & File Structure

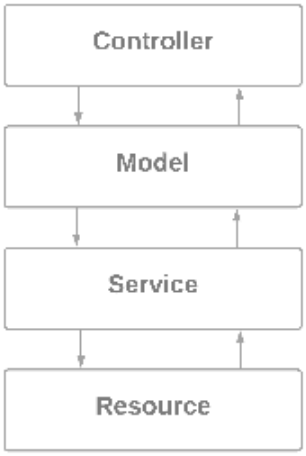
The refactored directory/file structure of the backend API can be seen below:

Refactored Directory Structure

```
.
 config.py
 Controller
    CustomerController.py
    OrderController.py
    ProductController.py
    UserController.py
 Exception
    exceptions.py
 Model
    Address.py
    BarcodeProduct.py
    Category.py
    CateProd.py
    Customer.py
    Image.py
    ModelMetadata.py
    Model.py
    OrderDetail.py
    Order.py
    Organization.py
    Price.py
    Product.py
    SellUnit.py
    Session.py
    User.py
 Resource
    CustomerResource.py
    DatabaseBase.py
    ImageResource.py
    ModelMetadataResource.py
    OrderResource.py
    ProductResource.py
    SessionResource.py
    SimpleModelResource.py
    UserResource.py
 Service
    CustomerService.py
    OrderService.py
    ProductService.py
    SquizzGatewayService.py
    UserService.py
 Util
     AuthUtil.py
     Validation.py
```

Layered Architecture

A diagram representing the layered nature of the architecture can be seen below:



As can be seen in this diagram, each layer provides an interface to the previous layer, for services including data transformation, database storage, fetching data etc.

Controllers

The controller is responsible for application flow control logic. It is responsible for handling frontend client requests and sending responses.

Models

The model component corresponds to all data-related logic that the user works with. It encapsulates business-logic related data.

Services

The service layer mediates communication between a controller and the resource layer, and/or other external public APIs.

The service layer contains business logic and validation logic.

Typically, the controller and service layers are decoupled via communication through a model state.

Resources

The resource layer provides a programmatic interface for handling database CRUD operations. It is a data-access layer.

Example: Refactored Code

This example succinctly displays how the new architecture provides the project with increased modularity, extensibility, and encapsulation, via *separati on of concerns*.

For example, in the initial codebase, the database logic was placed inside the controller actions.

This mixing of database and controller logic meant that the application would be more difficult to maintain over time.

The recommendation is that you place all of your database logic in a separate resource layer.

(Note: For brevity only one example is being shared, but one look at the source code available on GitHub paints a picture of the extent of work done in refactoring)

Initial

`main.py`

```python
@main.route('/updateproduct', methods=['GET'])
def update_product():
    if not auth.validate_login_session():
        return redirect(url_for('auth.login'))
    connection = auth.build_connection()
    data_type = 3
    json_response, json_values = connection.get_product_list(data_type)

    if json_response:
        result = db.update_product(json_values)
        return jsonify(result)
    else:
        result = {'status': "error", 'data': 'null', 'Message': "Error while retrieving product from server"
        return jsonify(result)
```

Refactored

ProductController.py

```python
# This method is not called from the front end. These are supposed to be called by the Postman or another similar tool tha
# allow you to make calls to the REST API.
# This method is repsonbile for getting the product update from SQUIZZ platform and updating the table in the local databa
@product.route('/updateProducts', methods=['GET'])
def update_products():
    if not authUtil.validate_login_session():
        return redirect(url_for('auth.login'))

    return jsonify(product_service.update_products())
```

ProductService.py

```python
def update_products() -> dict:
    connection = authUtil.build_connection()
    data_type = 3
    success, product_list = connection.retrieve_organisation_data(data_type)

    if success:
        return product_resource.update_products(product_list)

    return {
        'status': 'error',
        'data': 'null',
        'Message': 'Error while retrieving product from server'
    }
```

ProductResource.py

```python
    # This method is used to update the products that are stored in the database. Updated product infromation is fetche
    # from the SQUIZZ API.
    def update_products(self, product_list: List[Product]):
        """
        Takes as input the retrieved product data from SQUIZZ API, then
        synchronises the data with the current records stored in the database.
        If they are not identical, it will update the product information
        to the latest one.

        Args:
            product_list: list of Product objects created from data retrieved from SQUIZZ API
        """
        value_not_inserted = 0
        value_inserted = 0
```