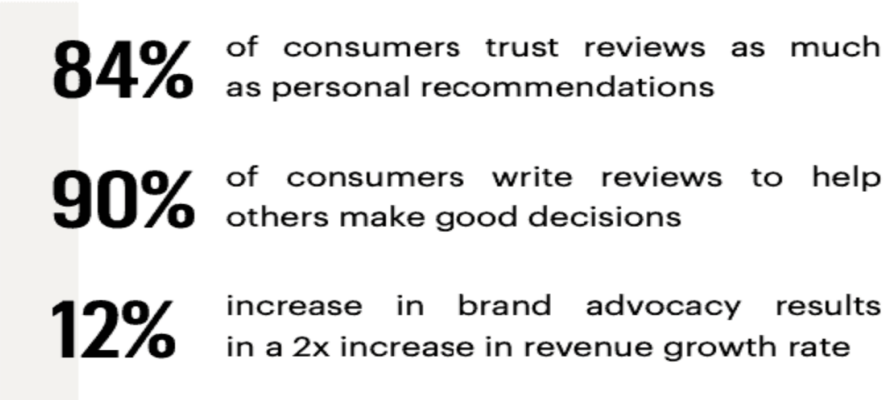
R1. ***Explain the problem that this app will solve and explain how this app solves or addresses the problem.***

Brands and consumers often face challenges when trying to make informed decisions when buying quality products, as not all products are created equally. With an overwhelming amount of products to choose from both online and instore, it can be difficult to discern which products are indeed high-quality and worth the price tag and the investment. Consumers frequently encounter some of the following concerns when purchasing products.

Overwhelming choices – The sheer volume of products available on the market can lead to decision stalemate.
Inconsistent review – Not all review are trustworthy, biased, fake or bought reviews can skew people perceptions, and people are unsure if the reviews are legitimate.
Lack of Personalisation – Reviews that don't tailor to customers' needs and individual preferences.
Difficulty Comparing Products – Consumers struggle to compare the features, prices and user experiences, as again consumers are unsure of trustworthy reviews.

This app solves the problem of legitimacy, as it is an independent app where people can post their honest reviews, good bad or indifferent, it can be customized into categories. This app will incorporate customer reviews and/or buying guidelines to enhance the credibility of the product being reviewed, whereby helping consumers make more informed decisions. It will foster a community of users who can ask questions, share their experiences, and provide real-time feedback, creating a more interactive environment.



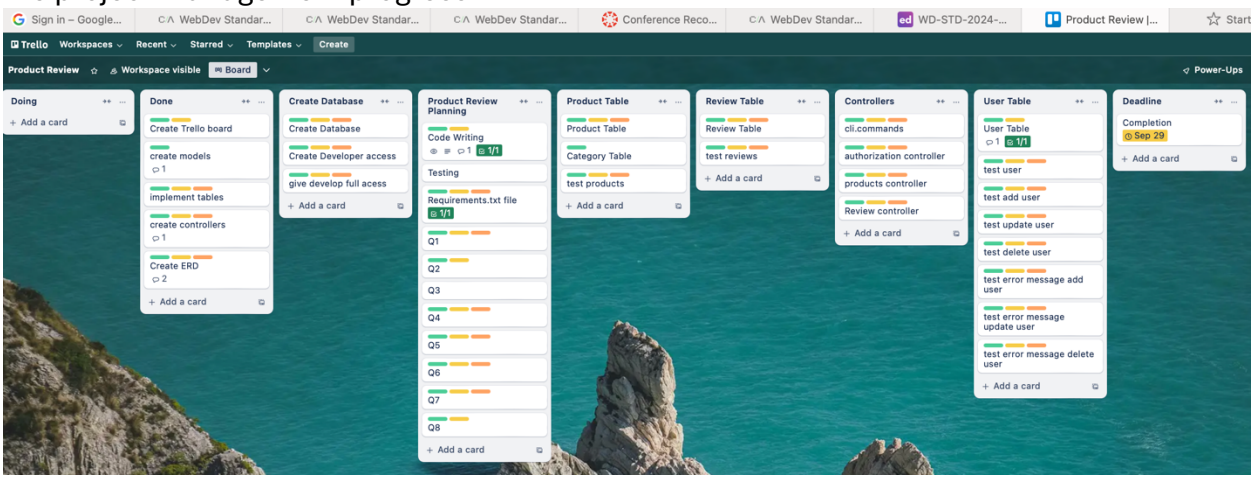
According to surveys, about 84% of consumers trust online review as much as personal recommendations. This highlights the significance of reviews in consumers decision making. 90% of consumers write reviews to help others make good decisions. Research indicates that 70% of consumers advise they research a product before making a purchase and read on average 7 reviews before trusting a product. Products with positive reviews can increase sales but up to 18% while a negative review can deter around 22% of potential consumers. While 12% in brand advocacy results in an increase of two times in revenue.

With the introduction of iPhones, mobile phone shopping is accounted for more than 50% of consumers purchases, with a strong lean to those looking at product reviews while in store before purchasing.

This emphasises the need for accessible review platforms. These statistics illustrate the critical role product review apps play in shaping consumer behaviour and driving sales, thus highlighting their value in the ecommerce landscape

R2. ***Describe the way tasks are allocated and tracked in your project.***

Task Creation and Allocation
Tasks are defined and loaded into Trello, where each task is allocated a status indicator, (Started, In Progress, Completed) that gets updated as each task is worked on to update the project management progress.



I utilise the dashboard for the task status to track progress, completions, deadlines. This visualisation of tasks helps me quickly assess where I am up to with the project and what needs to be addressed next. This structured approach helps maintain clarity and efficiency in managing all the tasks associated with project managing an API project and helps to effectively meet the goals with Trello's tracking methods

Trello Board Set Up
A Trello board is created specifically for the project, organizing the different work stages and all task required to complete an API.

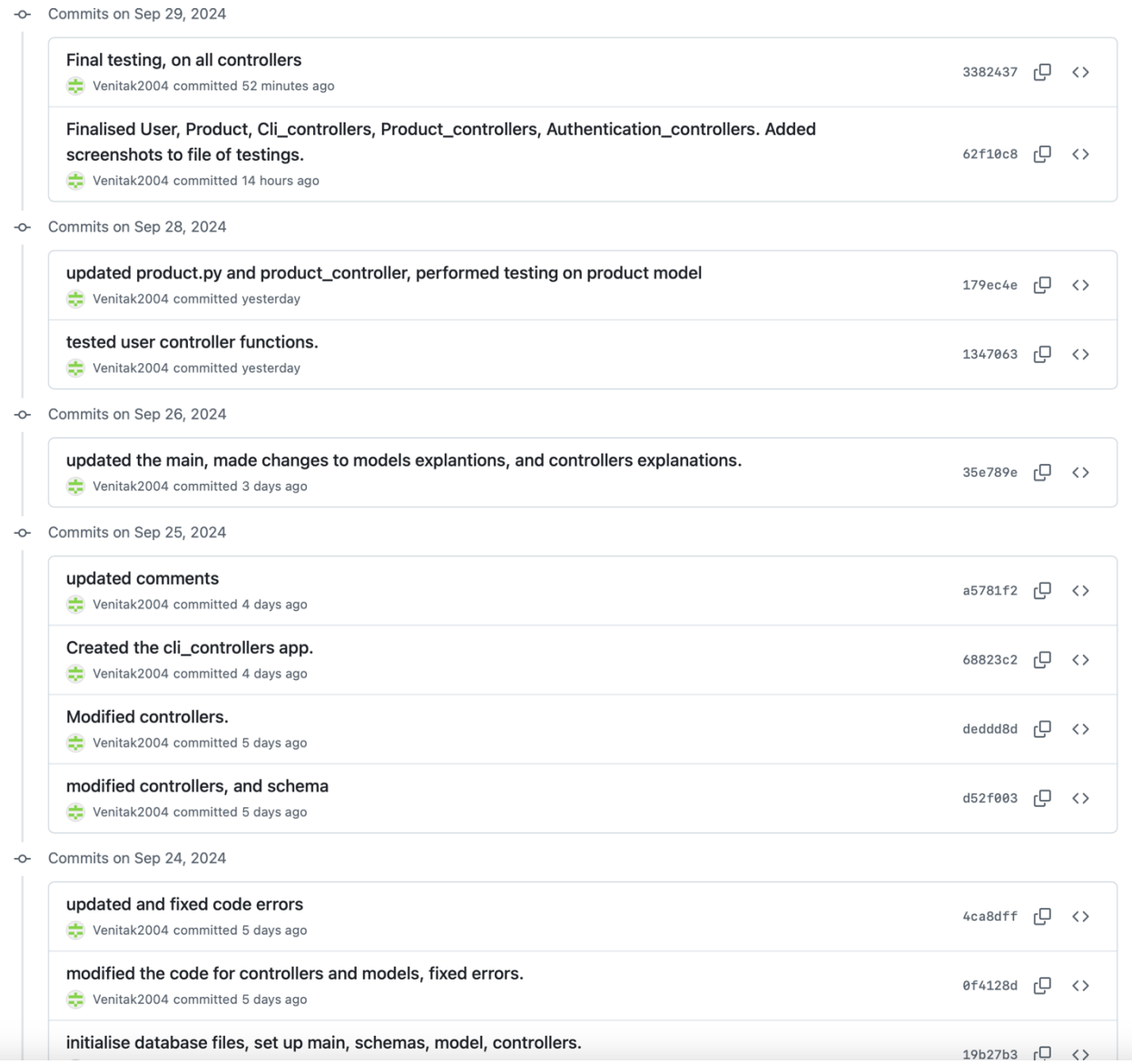
Tracking of Progress
As tasks are started the Trello board is updated with the progress of each task, with Started, In Progress, and Completion, each task us updated notes with regards to the progress of each stage.

Labels and Due Dates
Each task is labeled for cleared task management, and due dates are applied to make sure timelines are adhered to. Also checklists to help ascertain if items are ticked off.

Report
Trello has a built if feature to help report performance and identify areas of improvement.

GitHub

Github was used to track all changes to the webserver project which allowed recovery of data from previous commits if necessary. Each file was updated and committed with a git init, git status, git add ., git commit -m"meaningful message", and git push origin main.



R3. ***List and explain the third-party services, packages and dependencies used in this app.***

In our API we had leveraged several third party services, packages and libraries to enhance the functionality and app development, below is a breakdown of utilised packages. By utilising SQL's base-classes for all models called db.Model

Flask

The Flask application library enhances database functionality and streamlines development, it is utilised for lightweight web applications where it serves as the backbone of managing routing, requests and responses

```
@app.route('/')
def home():
   return 'Hello, World!'`
```

Bcrypt

pip3 install flask_bcrypt

Bcrypt is utilised for secure password hashing to protect valuable information. It is a lightweight WSGI web application framework for Python. It serves as the foundation for building the web app by handling the routing, requests and responses and is an extension for Flask. Bcrypt enables secure password storage by hashing passwords with the bcrypt algorithm making them resistant to brute force cyber-attacks. It also allows the developer to assign user management by defining user models that include hashed passwords.

```
if user and bcrypt.check_password_hash(user.password, request_body_data.get("password")):
    #Create the JWT security Token with a 1 day validity time limit
    token = create_access_token(identity=str(user.id), expires_delta=timedelta(days=1))
```

JWT Extended

pip3 install flask_jwt_extended

Is one of the ways you can implement security with a token-based authentication into your API Is standard for creating tokens that get used for authentication and authorisation of web applications for your APIs. JWT is a type of toke that has a structure that can be decrypted by a server and allows authenticating of the users' identity of the application. Unlike other token types, JWT tokens largely contain necessary information without the use of an external database. It consists of three parts H P S the Header, the Payload, and the Signature.

Header – This stores the token encryption algorithm
Payload – This contains the data identifiers for the user like ID or username
Signature – This is the digital signature which is generated with the Header and Payload to verify the content.

```python
# Creating a decorator for authorising an administrator for delete functions

def auth_as_admin(fnc):
    @wraps(fnc)
    def wrapper(*args, **kwargs):
        # get the user's identity from get_jwt_identity
        user_id = get_jwt_identity()

        #Retrieve the user from the database, filter by user id
        stmt = db.select(User).filter_by(id=user_id)
        user = db.session.scalar(stmt)

        # Check if user is admin, before execution
        if user and user.is_admin:
            # The decoration function will execute
            return fnc(*args, **kwargs)

        else:
            #Else return error 403 Forbidden
            return {"error": "Only admin can perform this delete function"}, 403

    return wrapper
```

SQLAlchemy

pip3 install flask_sqlalchemy

For Python SQLAlchemy is a flexible, powerful toolkit for ORM. It is designed for efficiency and high performance database access and is an extension that simplifies database interactions by providing an Object Relational Mapping (ORM) layer making it easier to define the models and query the database directly.

```python
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True, nullable=False)
    email = db.Column(db.String(120), unique=True, nullable=False)

    def __repr__(self):
        return '<User %r>' % self.username
```

You use **Column** to define your database columns. The name that you assign to your column becomes the column name. It utilises the table relationships with ONE-TO-ONE, ONE-TO-MANY, and MANY-TO-MANY relationships.

Is not just an adapter but a more like a comprehensive toolkit for ORM systems and Python applications. Its complexities for database communications allows developers to engage and interact with the database using Python expressions. SQLAlchemy enables you to map python objects to database tables. It automatically translates Python expressions into SQL code reducing the need for writing SQL queries.

```python
#GET all products in the Database
@product_bp.route("/")
def get_all_products():
    stmt = db.select(Product)
    products = db.session.scalars(stmt)
    #If products to return from the database return to user 200 for successful
    return products_schema.dump(products), 200
```

Psycopg2-binary

pip3 install psycopg2-binary

Is the most popular PostgreSQL adapter for Python its purpose is to allow the application to connect and interact with the database. Its binary version simplifies installation by including the necessary libraries and thread safety. It serves as a bridge to enable you to directly interact with your PostgreSQL database.

Marshmallow

pip3 install flask_marshmallow
pip3 install marshmallow_sqlalchemy

This application has been used to create schemas to help to seralise and deserialise objects in python into readable objects for the viewing panel. It is used inside the model for validating user input and nested attributes, also used fields, and validate

```python
class Product(db.Model):
    __tablename__ = "products"

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String, nullable=False)
    description = db.Column(db.String)
    category = db.Column(db.String)

    user_id = db.Column(db.Integer, db.ForeignKey("users.id"), nullable=False)

    user = db.relationship('User', back_populates='products')
    reviews = db.relationship('Review', back_populates='product', cascade="all, de
```

Dot-env

Is a Python Flask extension that loads and environment variable from a .env file, it manages sensitive configurations securely, for example database URL's and secret keys. It helps with modularization and separates the application logic from the configuration data.

pip3 install flask-dotenv

R4. ***Explain the benefits and drawbacks of this app's underlying database system.***

Flask is a standard Python web structure that is used to craft RESTful APIs for data engineering tasks. It provides a flexible and simple way to build your API's that can handle HTTP requests and responses. Flask API projects commonly use PostgreSQL database systems. PostgreSQL is an open-source relational database management system which is known for its extensibility, reliability, data integrity, robustness, and extensive feature set. It can handle advanced data types, complex queries, foreign keys, triggers, and views as well as procedural languages for stored procedures. Its high expandability, allowing users to add new functions and data types. Its strong SQL compliance and combined support for ACID (Atomicity, Consistency, Isolation, Durability) properties, make it an efficient, scalable and secure database choice for developers.

Below are some of the examples of using Flask.

1. **Data Retrieval**: Flask can be used to construct APIs to retrieve data from databases, files or external API's. See example below; When you access the http address in your browser it will retrieve data from the PostgreSQL database and return it as a JSON response.

```
#GET all products in the Database
@product_bp.route("/")
def get_all_products():
    stmt = db.select(Product)
    products = db.session.scalars(stmt)
    #If products to return from the database return to user 200 for successful
    return products_schema.dump(products), 200
```

```
from flask import Flask, jsonify
import psycopg2

app = Flask(__name__)

@app.route('/data', methods=['GET'])
def get_data():
    conn = psycopg2.connect(host='localhost', dbname='mydb', user='myuser', pass
    cursor = conn.cursor()
    cursor.execute('SELECT * FROM mytable')
    data = cursor.fetchall()
    conn.close()
    return jsonify(data)

if __name__ == '__main__':
    app.run()
```

Data Transformation: Flask can be utilised to construct API's to perform data calculations. Here is an example that calculates the sum of two numbers.

```
from flask import Flask, request

app = Flask(__name__)

@app.route('/sum', methods=['POST'])
def calculate_sum():
    data = request.get_json()
    num1 = data['num1']
    num2 = data['num2']
    result = num1 + num2
    return str(result)

if __name__ == '__main__':
    app.run()
```

2. **Data Ingestion**: Flask can be used to construct API's that accept data uploads or input data from external sources. Example below of accepting a file upload and saves to the server.

```
from flask import Flask, request

app = Flask(__name__)

@app.route('/upload', methods=['POST'])
def upload_file():
    file = request.files['file']
    file.save('uploads/' + file.filename)
    return 'File uploaded successfully!'

if __name__ == '__main__':
    app.run()
```

These are just a few examples of the ways in which Flask can be used in data engineering tasks

Look at the Statistics
"StackOverflow statistics show that 26% of developers preferred it in 2017, 34% in 2019, and 40% in 2021. Most recently, in StackOverflow's 2022 Stack Developer Survey, PostgreSQL took a slight lead over MySQL (46.48% to 45.68%) as the most popular

database platform among professional developers.“ Author -Pete Scott – Percona Feb 2, 2024.

The Pros and Cons of using PostgreSQL in Flask;

Pros of Using PostgreSQL

Advanced Features

Acid Compliance – it ensures data integrity is maintained by supporting transactions and rollback capabilities.

Extensibility

It allows you to define custom data types, operators and index types. It also supports extensions like PostGIS for geographical/geospatial data, which is useful for location-based applications.

Scalability

PostgreSQL performs well on large scale datasets and is capable of handling high workloads by vertical scaling (adding more powerful hardware) With tools like Citus, PostgreSQL can also scale horizontally distribution data across many nodes.

SQL Compliance

PostgreSQL adheres closely with standards compliance making it easier for developers to work with who are familiar with SQL. It also supports advanced SQL window functions, these are valuable when using analytical queries.

Community and Available Documentation

PostgreSQL has a strong active community, who provide support and contribute to further developing. It’s highly detailed documentation assists developers utilise its systems effectively.

Cons of Using PostgreSQL

Complexity

PostgreSQL can be complex to configure and tune for optimal performance, especially on large scale projects. New users may find it hard to learn, particularly those unfamiliar with the advanced database features and functions.

Performance Overhead

While PostgreSQL excels in read heavy scenarios, write heavy scenarios might encounter performance overhead due to its consistency and transactional features. Also, its indexing and query optimization, PostgreSQL complex queries and indexing can sometimes create performance issues when not properly optimized.

Resource Usage

Disk and memory usage can be resource intensive, it uses significant amounts of disk space and memory, which may be a concern for certain constrained environments.

Tooling and Ecosystem

Compared to other databases there may be limited tools and third-party options, integrations specifically designed for PostgreSQL are continuously improving.

Replication

Complex replication setups and high availability may be involved and require additional configurations or tools.

For many Flask applications, PostgreSQL is a robust choice, because of its strong feature set, community support and complex data requirements, its advanced features like full-text search and custom data types, couple that with high data integrity and consistency, it's understandable why its popular. However, it is still essential to base your applications specifics on whether PostgreSQL is suitable for your project's needs.

R5. ***Explain the features, purpose and functionalities of the object-relational mapping system (ORM) used in this app.***

The primary purpose of the ORM system in the product review app is to streamline the database interactions, improve developer productivity and enhance maintainability and quicker feature development for easier updates. This is a brief list of features below.

- ORM provides a high-level construct over the database interactions and allows developers to work with the objects rather that the raw SQL queries by simplifying the database operations.
- It forces you to write MVC code, which makes your code a little cleaner.
- By mapping classes to database tables, this makes it easier to manage and manipulate the data.
- Sanitizing prepared statements or transactions are as easy as calling a method.
- Each class represents a table, and each instance of the class represents the row of each table.
- ORM automatically generates SQL queries based on the object operations, this then reduces the need for manual SQL coding which minimises errors.
- It keeps track of changes made to objects, ensuring only modified data is sent to the database.
- It handles relationships between the objects (1 to many, many to many) using associations, to simplify and manage the complex data relationships.
- ORM has built in support for transactions management to ensure data consistency and integrity.

By incorporating an ORM system the product review app can efficiently manage the data, streamline development processes and ensure that the application is scalable and maintainable, some of ORM's functionalities below.

- CRUD operations like CREATE, READ, UPDATE, DELETE through object manipulation.

2. Create a function to add (CREATE), read (READ), update(UPDATE), and delete data (DELETE operation).

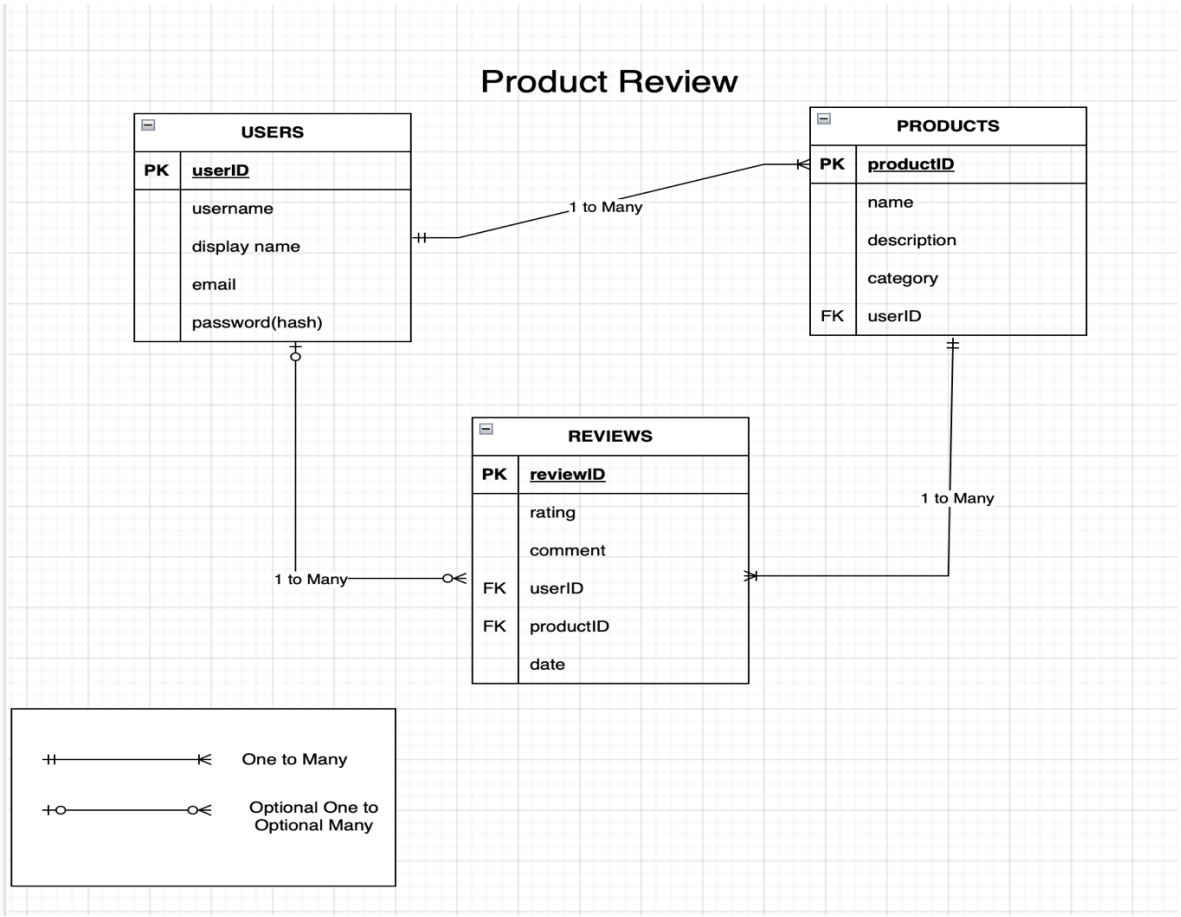
```
function createData(data) {
  connection.query('INSERT INTO your_table SET ?', data, (err, res)
=> {
    if (err) throw err;
    console.log('New data added:', res.insertId);
  });
}
function retrieveData(id) {
  connection.query('SELECT * FROM your_table WHERE id = ?', [id],
(err, res) => {
    if (err) throw err;
    console.log(res[0]);
  });
}
function updateData(id, data) {
  connection.query('UPDATE your_table SET ? WHERE id = ?', [data,
id], (err, res) => {
    if (err) throw err;
    console.log('Data updated:', res.affectedRows);
  });
}
```

- Data Validation which provides built in validation features to ensure data integrity.
- Supports complex queries for filtering through object-oriented interfaces to allow developers to retrieve data efficiently.
- Implements caching to reduce database load, it stores frequently accessed data efficiently for improved application performance.
- Easy database schema migrations, make managing changes in the database structure easier over time.
- Offers a multi database system support, allowing the app to switch databases with minimal code changes.

The code example comes from udacity git.

R6. ***Design an entity relationship diagram (ERD) for this app's database and explain how the relations between the diagramed models will aid the database design.***

This should focus on the database design BEFORE coding has begun, eg. during the project planning or design phase.



An Entity Relationship diagram is a visual representation of the entities in a database system and their relationships to one another. For the product review app, the ER diagram typically includes entities such as Users, Products, Reviews and Categories. Below is the breakdown of each entity table.

Users

This entity represents the users of the app, storing their details, and passwords safely (hashed). Each user can submit multiple reviews.

- UserID (Primary Key) and not null
- Name
- Display Name
- Email, not null
- Password (hash)
- Admin_user

Products
This entity represents the products being reviewed in the app. Each product can have multiple reviews from different users.

- ProductID (Primary Key) and not null
- Name
- Description
- Category
- UserID (Foreign Key), not null

Reviews
This entity is the Junction table, it connects users and products together. Users can leave many reviews, and a products can have many reviews. It stores the rating and the comment for each product review.

- RatingID
- Rating – 1-5 rating
- Comment
- UserID (Foreign Key)
- Product ID (Foreign Key)
- Date – datetime delta

As you can see there are a few entities that have relationships to each other in the form of Foreign Key connections. These connects are important in the way you manage the data in the database.

Users table has a 1 mandatory relationship to many optional relationships with the Products table. Being that one user can list many products to review, and many products can be listed by one user.

Products table has a relationship with the User table, and the Review table. The relationship with the Users table has been discussed above.

Reviews table is where it all comes together as a Junction table, it is linked to the Users table via the Foreign Key and is linked with a one optional to many optional relationship. Reviews is also linked to Product table via its Foreign Key with its one mandatory product to many optional reviews relationship, it also pulls from the products relationship with categories to pull that data into its junction table.

This ER diagram has a foundational structure which allows for efficient data management and retrieval.

R7. *Explain the implemented models and their relationships, including how the relationships aid the database implementation.*

This should focus on the database implementation AFTER coding has begun, eg. during the project development phase.

The entity relationships help to shape and design the database.

Data Integrity with the use of Foreign Keys (FK) ensures that each product review is associated with a valid user and valid product. This helps maintain referential integrity in the database helping to prevent orphaned records. By implementing Foreign Keys, the database is ensured that every review has to correspond to a valid user and product.

Query efficiency is maintained by the relationships, for example you can retrieve all reviews for a specific product or find all reviews written by a certain user. The relationship helps to support the apps functionality by making data retrieval streamlined.

Scalability, with the one-to-many relationships enable the system to scale more easily. As more users and products get added to the database, the structure can accommodate the growth without requiring major redesigns. Each user can add multiple reviews, and it is designed so that each product can receive multiple reviews.

Flexibility for filtering reviews by the user or product enables better searching and sorting functions in the app. Future features may include updated user preferences which could alter review suggestions.

Normalisation helps minimise data redundancy and improves the datas' integrity. User information is stored once, and reviews do not need to be duplicated. By separating User, Product, and Review models/entities, the implementation adheres to normalization principles, reducing the redundancy
During the project development phase, implementing User, Product and Review entities with their respective relationships establishes a robust database structure. This design aids in efficient data handling, and prepares for future enhancements, ensuring long term scalability and maintainability. The clearly defined interconnections in the entities, ensures development teams can focus on building features that will leverage the structured data efficiently.

User Model

src > models > user.py > ...

```python
#import from the init.py file SQLAlchemy and Marshmallow
from init import db, ma
#import marshmallow module and fields
from marshmallow import fields
#import validate module and regexp
from marshmallow.validate import Regexp

  #install the user table attributes
    #define the primary key for data serialisation
class User(db.Model):
    __tablename__ = 'users'
    #Creating the user table column values
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(100), nullable=False)
    display_name = db.Column(db.String, nullable=False)
    email = db.Column(db.String, nullable=False, unique=True)
    password = db.Column(db.String(128), nullable=False)
    is_admin = db.Column(db.Boolean, default=False)

    #Relationships between products and review to share access from users model
    products = db.relationship('Product', back_populates='user')
    reviews = db.relationship('Review', back_populates='user')

#Create UserSchema to de/serialise objects
class UserSchema(ma.Schema):

    #creating the attribute specifics which interact with other models
    products = fields.List(fields.Nested('ProductSchema', exclude=["user"]))
    reviews = fields.List(fields.Nested('ReviewSchema', exclude=["user"]))
    #advises user of the valid email format
    email = fields.String(required=True, validate=Regexp("^\S+@\S+\.\S+$", error="Invalid Email Format, must be in proper email format."))
    username = fields.String(required=True)
    display_name = fields.String(required=True)
    class Meta:
        fields = ("id", "username", "display_name", "email", "password", "is_admin", "products", "reviews", "comment")
        ordered = True
# to handle a single user object
user_schema = UserSchema(exclude=["password"])

# to handle a list of user objects
users_schema = UserSchema(many=True, exclude=["password"])

```

User Controllers – Authentication_Controller

Create User

```python
#define the route to register users
@auth_bp.route('/register', methods=['POST'])
def create_user():
    try:
        #GET the data from the body of the request
        request_body_data = UserSchema().load(request.get_json())

        #GET the username from the User - Front End
        user = User(
            username = request_body_data.get('username'),
            email = request_body_data.get("email"),
            display_name = request_body_data.get("display_name")
        )
        #GET the password from the User - Front End and hash the password
        password = request_body_data.get('password')

        #Hash protect the password and store it with the user attribute
        if password:
            hashed_password = bcrypt.generate_password_hash(password).decode('utf-8')
            user.password = hashed_password or user.password

        is_admin = request_body_data.get("is_admin")
        if is_admin:
            user.is_admin = is_admin or user.is_admin

        #Add and commit the session to the Database
        db.session.add(user)
        db.session.commit()

        #return confirmation to the user
        return user_schema.dump(user), 201#"{"message": "User registered successfully!"}), 201

    #except error handling - 400 Bad Request File Not Found
    except IntegrityError as err:
        if err.orig.pgcode == errorcodes.NOT_NULL_VIOLATION:
            return {"error": f"The column {err.orig.diag.column_name} is required"}, 400
        if err.orig.pgcode == errorcodes.UNIQUE_VIOLATION:
            # unique violation - 400 Bad Request File Not Found
            return {"error": "Email address must be unique"}, 400
```

User Controllers – Authentication_Controller
Login_user

```python
#create the login user route
@auth_bp.route('/login', methods=['POST'])
def login_user():
    try:

        #Get the data from the body of the request
        request_body_data = request.get_json()

        email_request = request_body_data.get("email")
        password_request = request_body_data.get("password")

        if not email_request or not password_request:
            return{"error": "You must enter a valid email and password."}

        #Search for the user with the specified user input from the front end
        stmt = db.select(User).filter_by(email=request_body_data.get("email"))
        user = db.session.scalar(stmt)

        if user and bcrypt.check_password_hash(user.password, request_body_data.get("password")):
            #Create the JWT security Token with a 1 day validity time limit
            token = create_access_token(identity=str(user.id), expires_delta=timedelta(days=1))
            #Return back to the user
            return {
                    "email": user.email,
                    "is_admin": user.is_admin,
                    "token": token,},200
        else:
            #Reply back to user with an error message, 401 UnAuthorised
            return {"error": "Invalid user, email or password is incorrect."}, 401
    except Exception as e:
        return{"error": "An unexpected error has occured", "info":str(e)}, 500
```

User Controllers – Authentication_Controller
Update_user

```python
#Create the update function for the specific user based on their login details
@auth_bp.route('/users', methods=["PUT", "PATCH"])
@jwt_required()
def update_user():
    try:
        # get the fields from the body of the request
        request_body_data = UserSchema().load(request.get_json(), partial=True)

        #create the user object to get the user data input
        request_password = request_body_data.get("password")
        request_name = request_body_data.get("username")
        #Update user and or password fields if user is updating
        # GET the user from where it is stored in the database
        # SELECT * FROM user WHERE id= get_jwt_identity()
        stmt = db.select(User).filter_by(id=get_jwt_identity())
        user = db.session.scalar(stmt)
        # if there is a relevant user in the database GET it and return to user
        if user.id == user.id:
        # then update the fields as required by the user
            user.username = request_name or user.username
            if request_password:
                user.password = bcrypt.generate_password_hash(request_password).decode("utf-8") or user.

            # commit the changes to the database
            db.session.commit()

            # return a response to the logged in user, succesfull, 200 ok
            return {"successful": "Updated changes sucessfully"}, 200

        elif not user.id:
            return{"error": f"Correct user token not supplied with user id {user.id}"}, 401
        # else:
        else:
        # return an error response 404 not found
            return {"error": "User does not exist."}, 404
    except ValidationError as e:
        return{"error": f"{e}"}, 400
```

User Controllers – Authentication_Controller
Delete_user

```python
# Create delete user so you can delte a specific user based on supplied user_id
@auth_bp.route("/<int:user_id>", methods=["DELETE"])
# JSON web token is required as a bearer token and check for is_authorised to use the endpoint
@jwt_required()
@auth_as_admin
def delete_user(user_id):
    try:
        # Retrieve the specific user from the database with supplied user_id
        stmt = db.select(User).filter_by(id=user_id)
        user = db.session.scalar(stmt)

        # check if there is such a user with the specific user_id:
        if user:
            # Delete the user if found and commit to the database session
            db.session.delete(user)
            db.session.commit()

            # Return a message to the that the user_id has been deleted and a success code 200
            return{"message": f"User with user_id {user_id} has been successfully deleted."}, 200

        else:
            # Return a message showing the user is not in the database
            return{"error": f"User with user_id {user_id} has not been found."}, 404

    # Error handling to handle any unexpected errors that may occur
    except Exception as e:
        return{"error": "An unexpected error occurred", "details": str(e)}, 500


@auth_bp.route("/", methods=["GET"])
```

User Controllers – Authentication_Controller

```python

@auth_bp.route("/", methods=["GET"])
def get_users():
    try:
        stmt = db.Select(User)
        users = db.session.scalars(stmt)

        if users:
            return users_schema.dump(users)
        else:
            return {"error": "No users to view"}
    except Exception as e:
        return {"error": f"{e}"}

@auth_bp.route("/<int:user_id>")
def get_a_user(user_id):
    try:
        stmt = db.Select(User).filter_by(id=user_id)
        user = db.session.scalar(stmt)

        if user:
            return user_schema.dump(user), 200
        else:
            return {"error": "No users to view"}
    except Exception as e:
        return {"error": f"{e}"}
```

Product Model

models > product.py > ...

```python
from init import db, ma
from marshmallow import fields, validates
from marshmallow.validate import Length, And, Regexp, OneOf
from marshmallow.exceptions import ValidationError

VALID_STATUSES = ("Beauty", "Technology", "Toys", "Furniture", "Sport", "Household Goods", "Electrical", "Other")


class Product(db.Model):
    __tablename__ = "products"

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String, nullable=False)
    description = db.Column(db.String)
    category = db.Column(db.String)

    user_id = db.Column(db.Integer, db.ForeignKey("users.id"), nullable=False)

    user = db.relationship('User', back_populates='products')
    reviews = db.relationship('Review', back_populates='product', cascade="all, delete")


class ProductSchema(ma.Schema):

    user = fields.Nested('UserSchema', only=["username", "email", "comment"])
    reviews = fields.List(fields.Nested('ReviewSchema', exclude=['product']))
    description = fields.String(required=True)
    name = fields.String(required=True, validate=And(Length(min=1, error="Title must be at least 4 characters in length.
    #select one from the valid statuses selection
    category = fields.String(validate=OneOf(VALID_STATUSES))

    @validates("category")
    def validate_category(self, value):
        # if trying to see the value of category
        for category in value:
            #if value == VALID_STATUSES[1]:
            # check whether an existing category exists or not
            #SELECT COUNT(*) FROM table_name WHERE category="VALID_STATUSES"
            stmt = db.select(db.func.count()).select_from(Product).filter_by(category=category)
            count = db.session.scalar(stmt)
            # if it exists
            if count > 0:
                # send error message
                raise ValidationError(f"You already have a Product in the Category {category}.")

class Meta:
        fields = ("id", "name", "description", "category", "users", "email", "reviews", "comment")
        ordered = True

product_schema = ProductSchema()
products_schema = ProductSchema(many=True)
```

Product Controllers – Get all products, Get a Product(<product_id>

controllers > product_controller.py > ...

```python
from flask import Blueprint, request, jsonify

from models.product import Product, product_schema, products_schema
from flask import Blueprint, request
from marshmallow.exceptions import ValidationError
from flask_jwt_extended import jwt_required, get_jwt_identity
from controllers.review_controller import review_bp
from auth import auth_as_admin
from init import db

product_bp = Blueprint('products', __name__,url_prefix="/products")
product_bp.register_blueprint(review_bp)

#GET all products in the Database
@product_bp.route("/")
def get_all_products():
    stmt = db.select(Product)
    products = db.session.scalars(stmt)
    #If products to return from the database return to user 200 for successful
    return products_schema.dump(products), 200

#Retrieve one specific product from the Database
@product_bp.route("/<int:product_id>", methods=['GET'])
def get_a_product(product_id):
    stmt = db.select(Product).filter_by(id=product_id)
    #Select the user input search for a specific product
    product = db.session.scalar(stmt)
    #if product exists return product
    if product:
        return product_schema.dump(product)
    else:
        #Return error code to user if product can not be found in database
        return {"error": f"Product with product_id: '{product_id}' has not been found"}, 404

```

Product Controllers – Add a Product, Delete a Product

```python
#Create a new product instance and add it into the Database
@product_bp.route("/", methods=['POST'])
@jwt_required()
def add_product():
    try:
        #Request the body data from the user input from the front end
        request_body_data = product_schema.load(request.get_json())


        #create the new product model instance
        product = Product(
            name = request_body_data.get('name'),
            description = request_body_data.get('description'),
            category = request_body_data.get('category'),
            user_id = get_jwt_identity()
        )
        #add product and commit to the database
        db.session.add(product)
        db.session.commit()

        #Return to the user success code 201
        return product_schema.dump(product), 201

    except ValidationError as err:
        return jsonify(err.messages), 400


#Delete a product from the database
@product_bp.route("/<int:product_id>", methods=["DELETE"])
@jwt_required()
@auth_as_admin
#check if the user is admin before allowing access to admin functions
def delete_product(product_id):

    # Retrieve the product from the database for the user
    stmt = db.select(Product).filter_by(id=product_id)
    product = db.session.scalar(stmt)
    # if the product is in the database,
    if product:
        # delete the product from the database
        db.session.delete(product)
        #commit the changes to the database
        db.session.commit()
        return {"message": f"Product {product.name} has been deleted!"}
    # else
    else:
        # return the error message that the product could not be found, 400 Not Found
        return {"error": f"Product with product_id {product_id} has not been found"}, 404


```

Product Controllers – Update Product

```python
#Make changes to an exisitng product- Authorised admin only
@product_bp.route("/<int:product_id>", methods=["PUT", "PATCH"])
@jwt_required()
#@auth_as_admin
def update_product(product_id):
    # Retrieve the data from the user body of the request
    request_body_data = product_schema.load(request.get_json(), partial=True)
    # Retrieve the product from the database
    stmt = db.select(Product).filter_by(id=product_id)
    product = db.session.scalar(stmt)

    # check whether the user is admin or not, if not return error message
    if product:
    #     # if the user is not admin or the owner, then return error

    #         return {"error": "Cannot perform this operation. Only owners are allowed to execute this ope

        # update the fields as required
        product.name = request_body_data.get("name") or product.name
        product.description = request_body_data.get("description") or product.description
        #product.category = request_body_data.get("category") or product.category

        # commit to changes to the database
        db.session.commit()
        # return acknowledgement, to the admin that the product has been updated
        return product_schema.dump(product)
    # else
    else:
        # return error message
        return {"error": f"Product with product_id: {product_id} could not be found."}, 404
```

Review Model

src > models > review.py > ...

```python
#import from init.py SQLAlchemy & Marshmallow modules
from init import db, ma
#import datetime module to use date function
from datetime import date
#import marshmallow module and utilise fields function and validate function
from marshmallow import fields, validate


class Review(db.Model):
    __tablename__ = "reviews"
    #Creating the review table column values
    id = db.Column(db.Integer, primary_key=True)
    rating = db.Column(db.Integer, nullable=False)
    comment = db.Column(db.String, nullable=False)
    date = db.Column(db.Date, default=date.today) #default to todays date

    #Attaching the foreign key elements to the table
    user_id = db.Column(db.Integer, db.ForeignKey('users.id'), nullable=False)
    product_id = db.Column(db.Integer, db.ForeignKey('products.id'), nullable=False)


    #Relationships between products and user tables to share access from review model
    user = db.relationship("User", back_populates="reviews", cascade="all, delete")
    product = db.relationship("Product", back_populates="reviews")


class ReviewSchema(ma.Schema):
    #which user made the review
    user = fields.Nested("UserSchema", only=["username", "email"])
    #which product is the review about
    product = fields.Nested("ProductSchema", exclude=["reviews"])

    #this will ensure any data inserted will have to adhere to the listed specifications
    id = fields.Int(required=True)
    rating = fields.Int(required=True, validate=validate.Range(min=1, max=5))
    comment = fields.Str(required=True, validate=validate.Length(max=255))
    user_id = fields.Int(required=True)
    product_id = fields.Int(required=True)

class Meta:
    fields = ("id", "rating", "comment", "date", "user","product")
    ordered = True


review_schema = ReviewSchema()
reviews_schema = ReviewSchema(many=True)
```

Review_Controllers – Add Review

src > controllers > review_controller.py > ...

```python
#import the datetime module and utilie date function to apply a time expiry
from datetime import date
#import flask to utilise Blueprints and request function
from flask import Blueprint, request
#import jwt_extended to create user access tokens and retrievew user id's tokens
from flask_jwt_extended import jwt_required, get_jwt_identity
#import review model to to create object and serialising/deserialse with schemas
from models.review import Review, review_schema, reviews_schema
from init import db
#import product model module
from models.product import Product
#auth_as_admin imported module from auth.py for authentication
from auth import auth_as_admin

#import from init.py SQLAlchemy
from init import db

review_bp = Blueprint('reviews', __name__, url_prefix="/<int:product_id>/reviews")

#create the route for Adding a review
@review_bp.route('/', methods=['POST'])
@jwt_required()
def add_review(product_id):
    # get the user review from the request body
    request_body_data = request.get_json()
    # fetch the product with the matching product_id
    stmt = db.select(Product).filter_by(id=product_id)
    product = db.session.scalar(stmt)
    # if the product exists in the database system
    if product:
        # Add a review for the product
        review = Review (
            comment = request_body_data.get("comment"),
            rating = request_body_data.get("rating"),
            date = date.today(),
            product_id = product_id,
            user_id = get_jwt_identity()
        )
        # add the review to the database and commit the session
        db.session.add(review)
        db.session.commit()
        # return successfull review added, 201 added successfully
        return review_schema.dump(review), 201
    else:
        # Else return error, 404 Not Found
        return {"error": f"Product with product_id {product_id} has not been found."}, 404

```

Review_Controller – Delete Review

```python
# Delete Comment authorised user only
@review_bp.route("/<int:review_id>", methods=["DELETE"])
@jwt_required()
@auth_as_admin
def delete_review(product_id, review_id):
    try:
        # Retrieve the review_id from the database where id=review_id matches user input
        stmt = db.select(Review).filter_by(id=review_id)
        #Retrieves the review from the database
        review = db.session.scalar(stmt)
        # if review exists in the database:
        if review:
            # delete
            db.session.delete(review)
            #commit the changes to the database
            db.session.commit()
            # return confirmation message that review has been deleted
            return {"Message": f"Review '{review.comment}' has been deleted successfully."}

        else:
        #Else return error message, 404 Not found
            return {"error": f"Review with id {review_id} has not been found"}, 404

    except Exception as e:
        return{"error": "Unexpected Error", "details": str(e)}, 500
```

Review_Controller – Get_a_Review, Update_Review

```python
#GET a review in the database for a specific product
@review_bp.route('/', methods=['GET'])
def get_a_review(product_id):
    stmt = db.select(Review).filter_by(id=product_id)
    reviews = db.session.scalars(stmt).all()
    if reviews:
        #If product id matches search, return from the database, 200 for successful
        return reviews_schema.dump(reviews), 200
    else:
        #return to user error message 400 Products not found
        return {"error": f"There were no products in the database"}, 404



# Update and exisiting review and update details
@review_bp.route("/<int:review_id>", methods=["PUT", "PATCH"])
@jwt_required()
def update_review(product_id, review_id):
    # Retrieve the  user input value from the body of the request
    request_body_data = request.get_json()
    # find the review with id that matches id = review_id
    stmt = db.select(Review).filter_by(id=review_id)
    review = db.session.scalar(stmt)
    #If the review exists
    if review: #if there is a review, then update the review
        review.comment = request_body_data.get("comment") or review.comment
        # commit the changes to the database
        db.session.commit()
        # return the updated review, with a confirmation
        return review_schema.dump(review)

    else:
        # Else return error message, 404 Not Found
        return {"error": f"Review with review_id {review_id} has not been found."}, 404
```

R8. ***Explain how to use this application's API endpoints. Each endpoint should be explained, including the following data for each endpoint:***

- **HTTP verb**
- **Path or route**
- **Any required body or header data**
- **Response**

User Controller
The product model is set up with specific attributes for the tables, and the controller actions these from the front end user input. Id is autogenerated, username can not be null, display_name can not be null, email can not be null, and password can not be null.

CREATE a User

```
  #install the user table attributes
    #define the primary key for data serialisation
class User(db.Model):
    __tablename__ = 'users'
    #Creating the user table column values
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(100), nullable=False)
    display_name = db.Column(db.String, nullable=False)
    email = db.Column(db.String, nullable=False, unique=True)
    password = db.Column(db.String(128), nullable=False)
    is_admin = db.Column(db.Boolean, default=False)

    #Relationships between products and review to share access from users model
    products = db.relationship('Product', back_populates='user')
    reviews = db.relationship('Review', back_populates='user')
```

HTTP Verb – methods=[‘POST’]
Path or Route: http://localhost:8000/auth/register

```python
#define the route to register users
@auth_bp.route('/register', methods=['POST'])
def create_user():
    try:
        #GET the data from the body of the request
        request_body_data = UserSchema().load(request.get_json())

        #GET the username from the User - Front End
        user = User(
            username = request_body_data.get('username'),
            email = request_body_data.get("email"),
            display_name = request_body_data.get("display_name")
        )
        #GET the password from the User - Front End and hash the password
        password = request_body_data.get('password')

        #Hash protect the password and store it with the user attribute
        if password:
            hashed_password = bcrypt.generate_password_hash(password).decode('utf-8')
            user.password = hashed_password or user.password

        is_admin = request_body_data.get("is_admin")
        if is_admin:
            user.is_admin = is_admin or user.is_admin

        #Add and commit the session to the Database
        db.session.add(user)
        db.session.commit()

        #return confirmation to the user
        return user_schema.dump(user), 201#"{"message": "User registered successfully!"}), 201

    #except error handling - 400 Bad Request File Not Found
    except IntegrityError as err:
        if err.orig.pgcode == errorcodes.NOT_NULL_VIOLATION:
            return {"error": f"The column {err.orig.diag.column_name} is required"}, 400
        if err.orig.pgcode == errorcodes.UNIQUE_VIOLATION:
            # unique violation - 400 Bad Request File Not Found
            return {"error": "Email address must be unique"}, 400
```

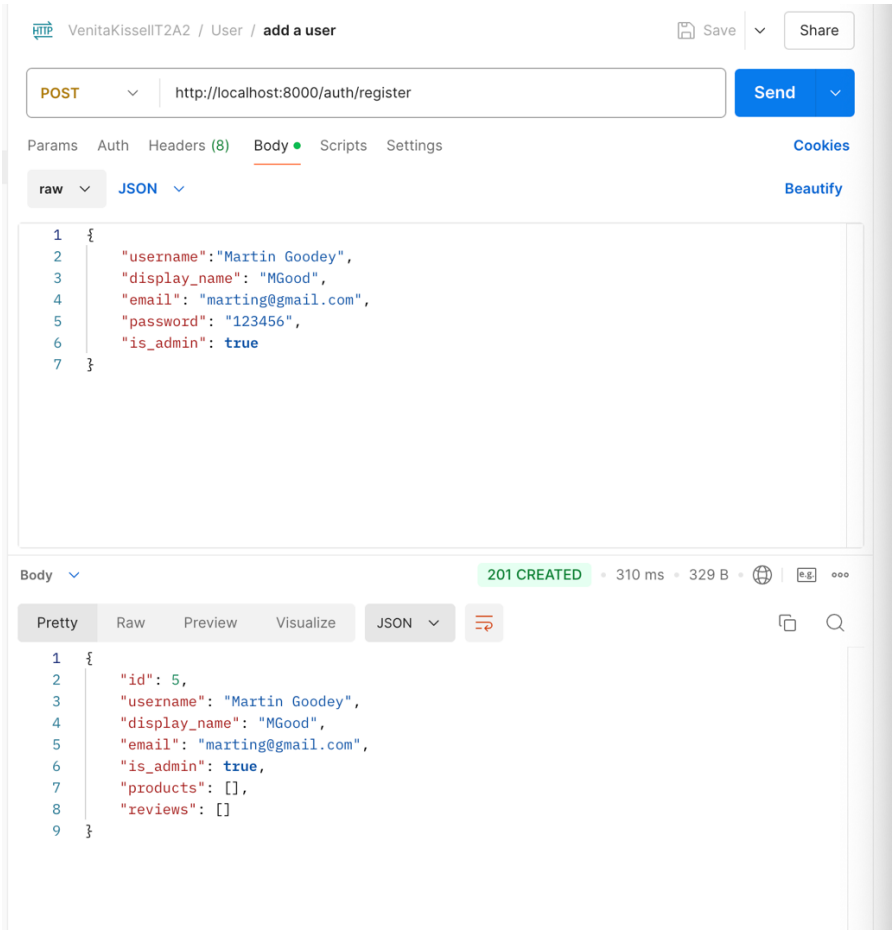
Required body/header –username, email, display_name, password

raw JSON Beau

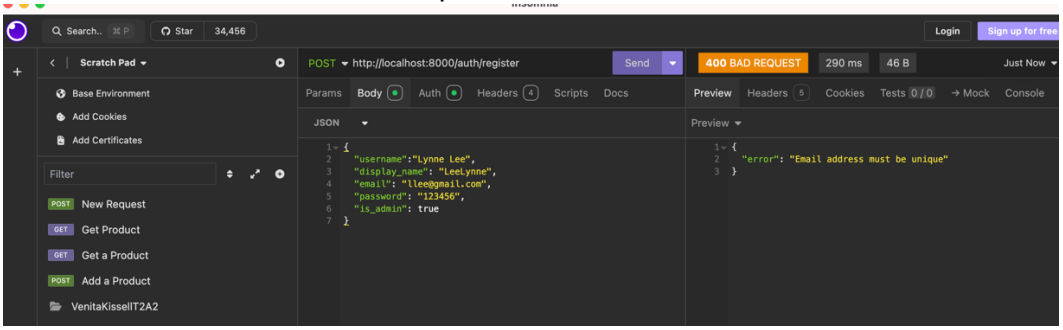
```
{
    "username":"Martin Goodey",
    "display_name": "MGood",
    "email": "marting@gmail.com",
    "password": "123456",
    "is_admin": true
}
```

Response –

Success:201 Created



Error: “Email address is not unique” 400



Login a User

The product model is set up with specific attributes for the tables, and the controller actions these from the front end user input. To login in a user, the jwt_token needs to be sourced from the database which matches the username and password.

HTTP Verb – POST

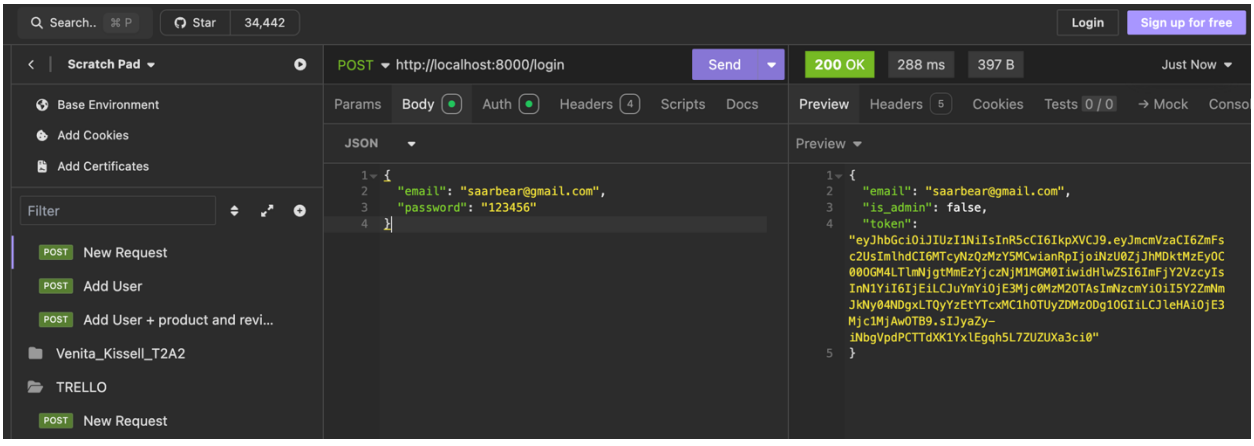
Path/Route: http://localhost:8000/login

```python
#create the login user route
@auth_bp.route('/login', methods=['POST'])
def login_user():
    try:

        #Get the data from the body of the request
        request_body_data = request.get_json()

        email_request = request_body_data.get("email")
        password_request = request_body_data.get("password")

        if not email_request or not password_request:
            return{"error": "You must enter a valid email and password."}

        #Search for the user with the specified user input from the front end
        stmt = db.select(User).filter_by(email=request_body_data.get("email"))
        user = db.session.scalar(stmt)

        if user and bcrypt.check_password_hash(user.password, request_body_data.get("password")):
            #Create the JWT security Token with a 1 day validity time limit
            token = create_access_token(identity=str(user.id), expires_delta=timedelta(days=1))
            #Return back to the user
            return {
                    "email": user.email,
                    "is_admin": user.is_admin,
                    "token": token,},200
        else:
            #Reply back to user with an error message, 401 UnAuthorised
            return {"error": "Invalid user, email or password is incorrect."}, 401
    except Exception as e:
        return{"error": "An unexpected error has occured", "info":str(e)}, 500

```

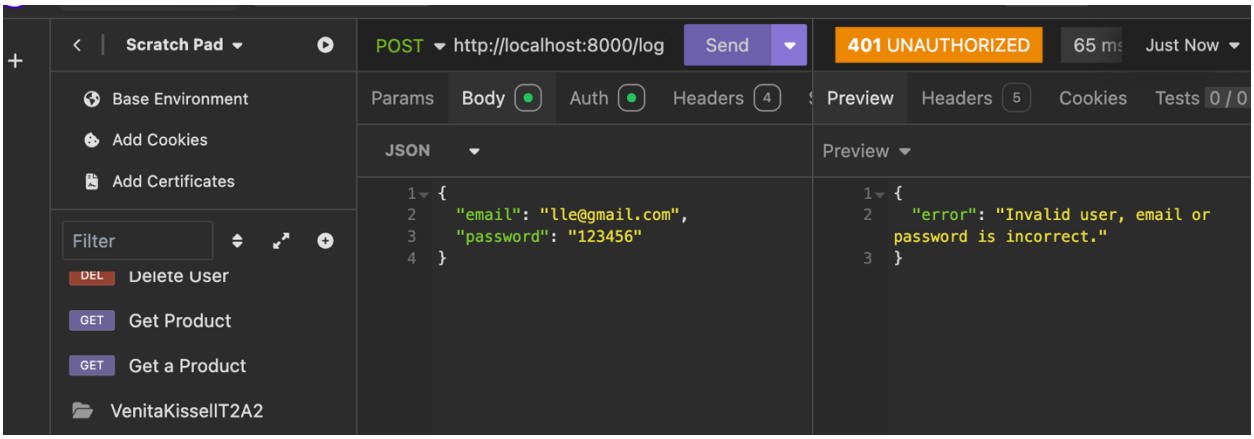
Required body/header – email, password, jwt token is then retrieved.

Response –

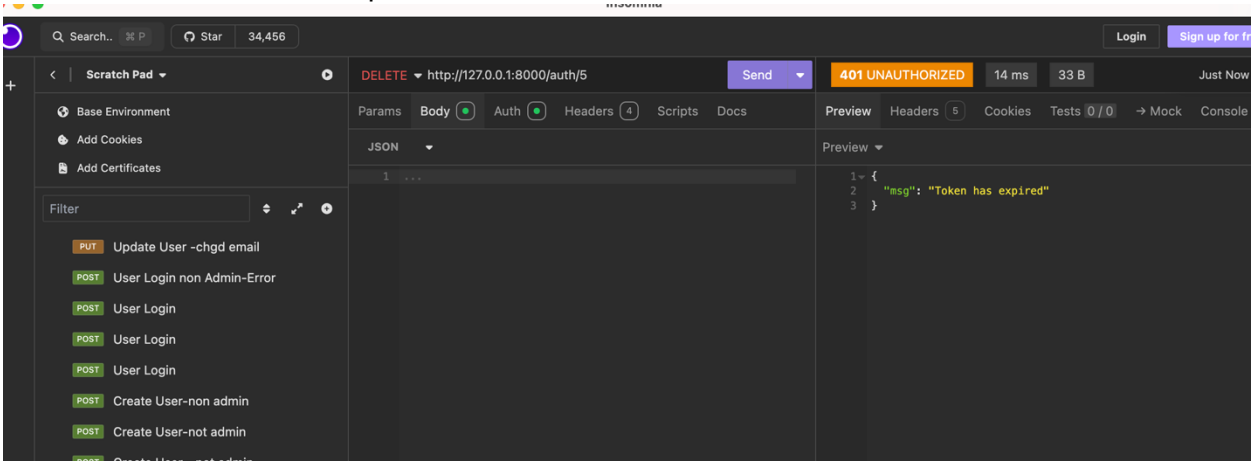
Success: 200 Ok



Error: “Invalid user, email or password is incorrect.” 401 Unauthorised.



Error:"Token has expired" 401 Unauthorised



UPDATE a User

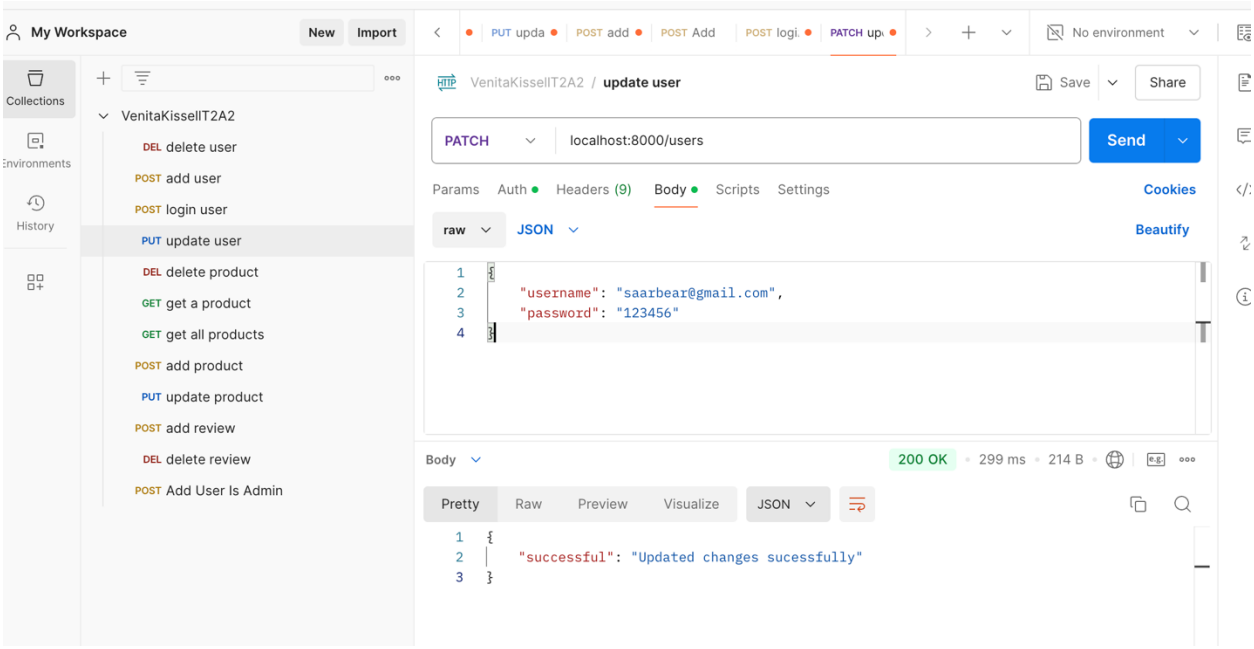
The product model is set up with specific attributes for the tables, and the controller actions these from the front end user input. With user update function, the user needs to supply their email address, and their password, the jwt token will be retrieved from the

database if the email and password matches.

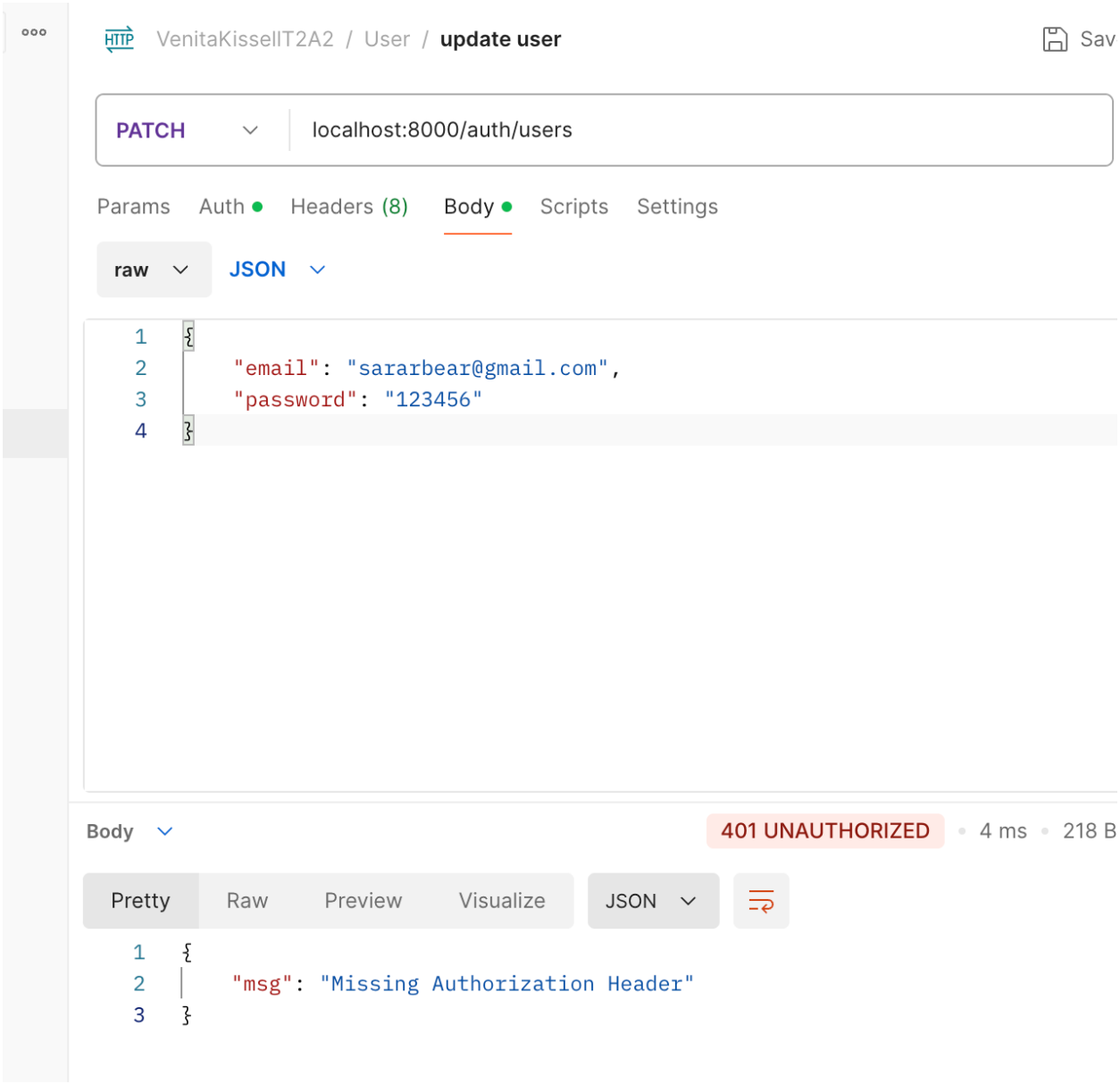
```python
#Create the update function for the specific user based on their login details
@auth_bp.route('/users', methods=["PUT", "PATCH"])
@jwt_required()
def update_user():
    try:
        # get the fields from the body of the request
        request_body_data = UserSchema().load(request.get_json(), partial=True)

        #create the user object to get the user data input
        request_password = request_body_data.get("password")
        request_name = request_body_data.get("username")
        #Update user and or password fields if user is updating
        # GET the user from where it is stored in the database
        # SELECT * FROM user WHERE id= get_jwt_identity()
        stmt = db.select(User).filter_by(id=get_jwt_identity())
        user = db.session.scalar(stmt)
        # if there is a relevant user in the database GET it and return to user
        if user.id == user.id:
        # then update the fields as required by the user
            user.username = request_name or user.username
            if request_password:
                user.password = bcrypt.generate_password_hash(request_password).decode("utf-8") or user.password

            # commit the changes to the database
            db.session.commit()

            # return a response to the logged in user, succesfull, 200 ok
            return {"successful": "Updated changes sucessfully"}, 200

        elif not user.id:
            return{"error": f"Correct user token not supplied with user id {user.id}"}, 401
        # else:
        else:
        # return an error response 404 not found
            return {"error": "User does not exist."}, 404
    except ValidationError as e:
        return{"error": f"{e}"}, 400
```

HTTP Verb – PUT
Path/Route: http://localhost:8000/auth/users
Required body/header – email, password, jwt token
Response –

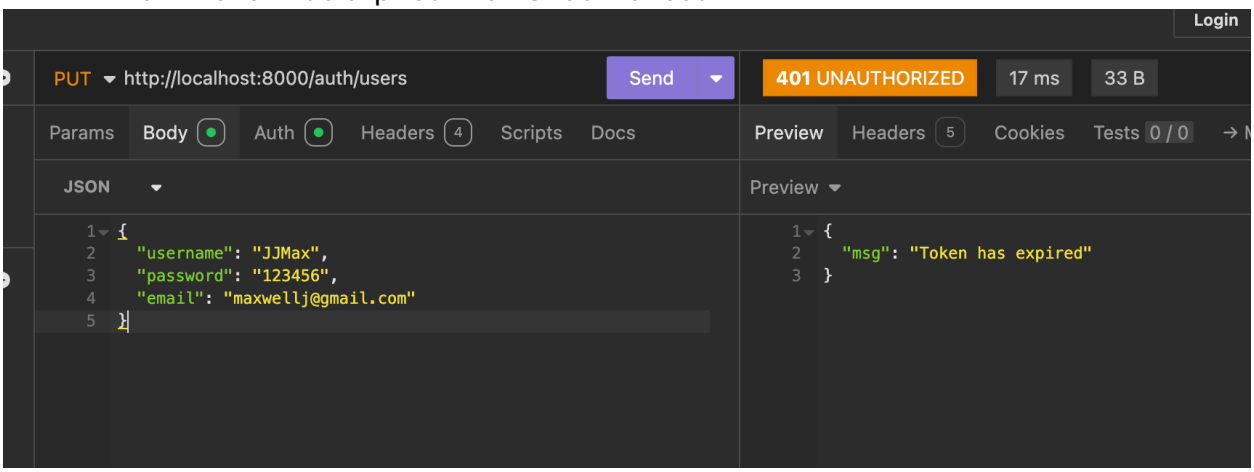
Success: “Updated changes successfully” 200 OK



Error: “Missing Authorisation Header, 401 Unauthorised



Error: “Token has expired” 401 Unauthorised



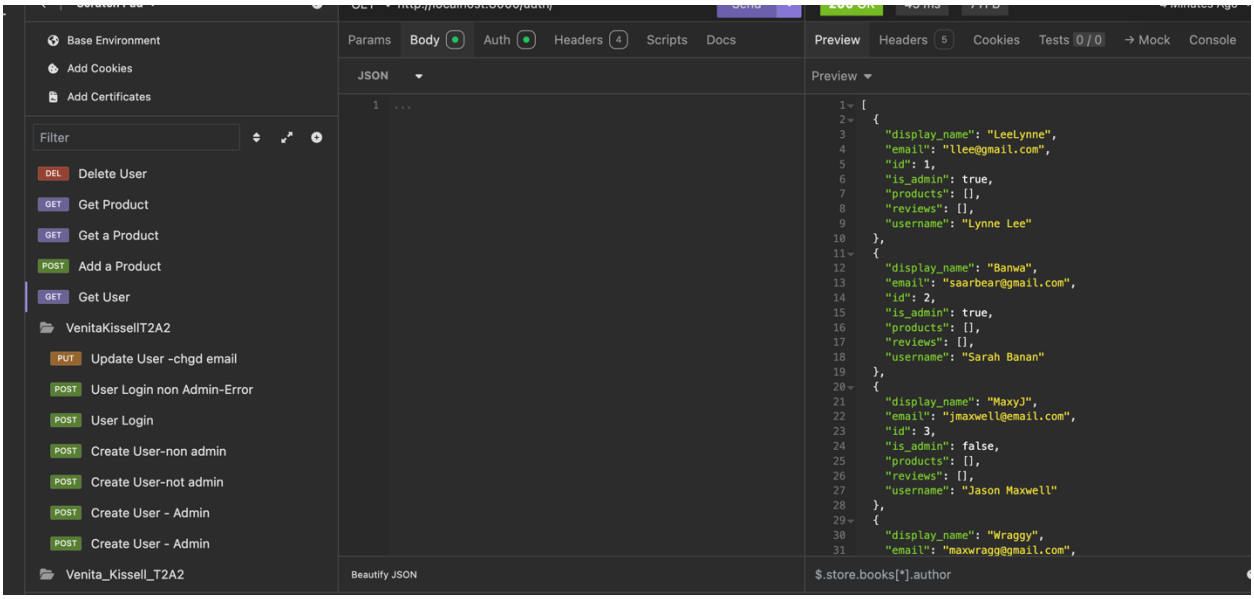
GET All Users

The user model is set up with specific attributes for the tables, and the controller actions these in the front end for the user. With the GET ALL user function the SQLAlchemy accesses the database and retrieves all users in the database.
HTTP Verb – GET

```python
@auth_bp.route("/", methods=["GET"])
def get_users():
    try:
        stmt = db.Select(User)
        users = db.session.scalars(stmt)

        if users:
            return users_schema.dump(users)
        else:
            return {"error": "No users to view"}
    except Exception as e:
        return {"error": f"{e}"}
```

Path/Route: http://localhost:8000/users
Required body/header –
Response –

Success: 200 Ok – returns all users in the database to the view



Error: “No users to view”

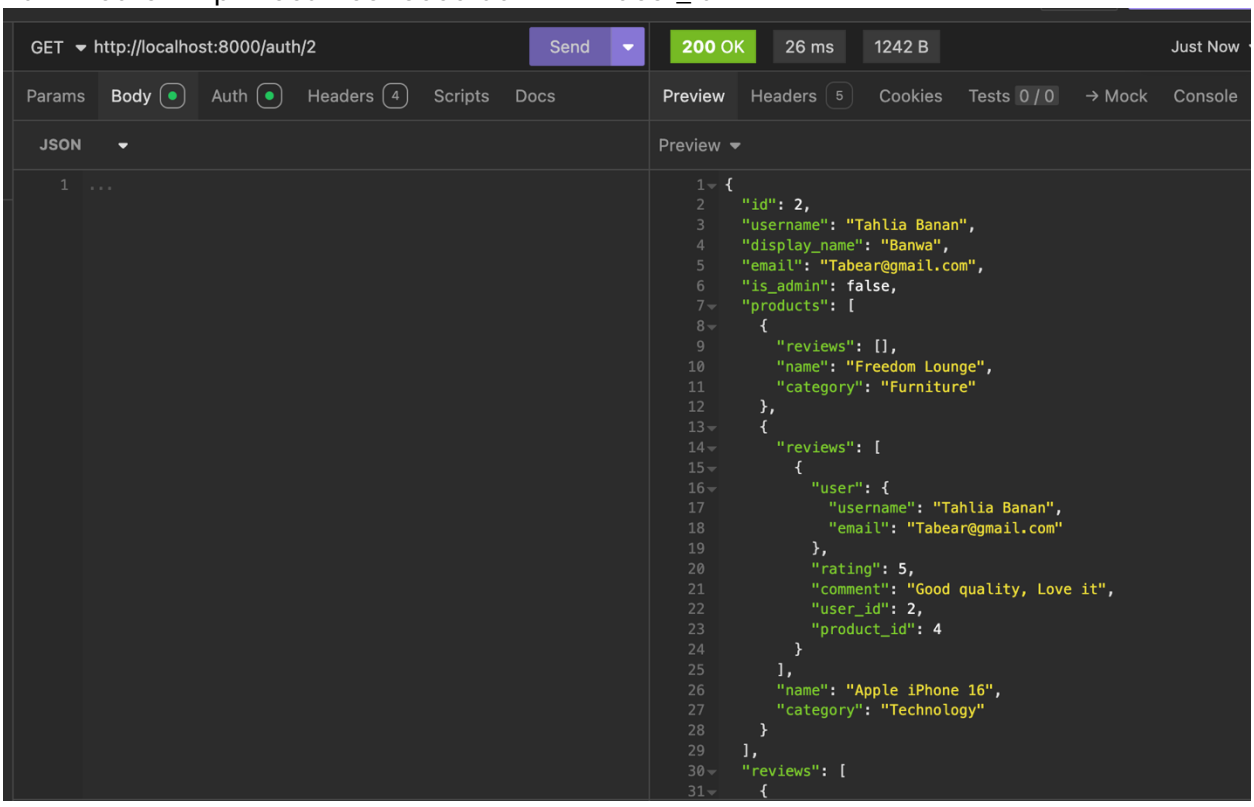
GET a Specific User

The user model is set up with specific attributes for the tables, and the controller actions these in the front end for the user. With the GET One user function the SQLAlchemy accesses the database and retrieves the specific user_id that matches the input data and

retrieves it from the database.

```python
@auth_bp.route("/<int:user_id>")
def get_a_user(user_id):
    try:
        stmt = db.Select(User).filter_by(id=user_id)
        user = db.session.scalar(stmt)

        if user:
            return user_schema.dump(user), 200
        else:
            return {"error": "No users to view"}
    except Exception as e:
        return {"error": f"{e}"}
```

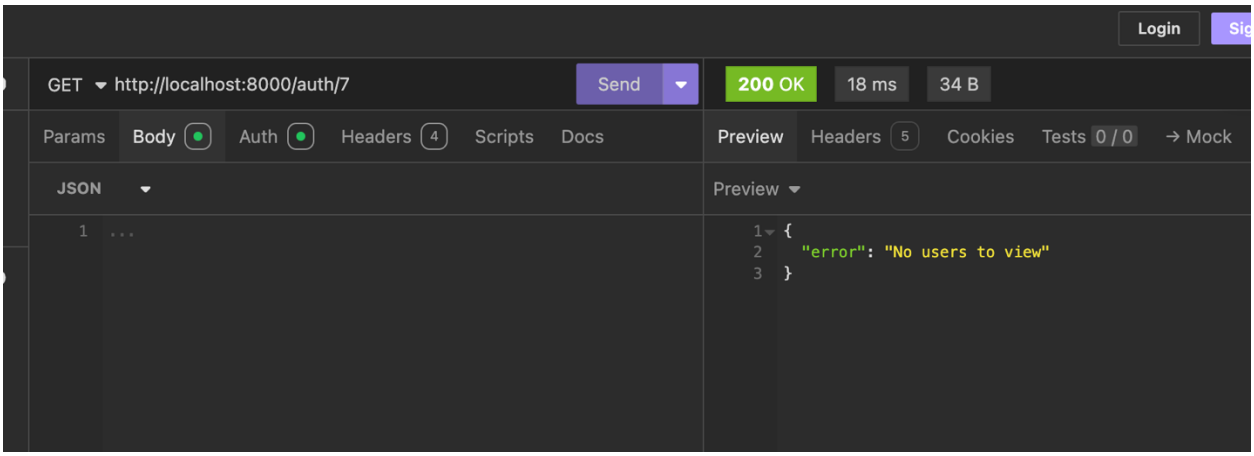
HTTP Verb – GET
Path/Route: http://localhost:8000/auth/<int:user_id>



Response –

Success: 200OK

Error: “No users to view, 200 OK

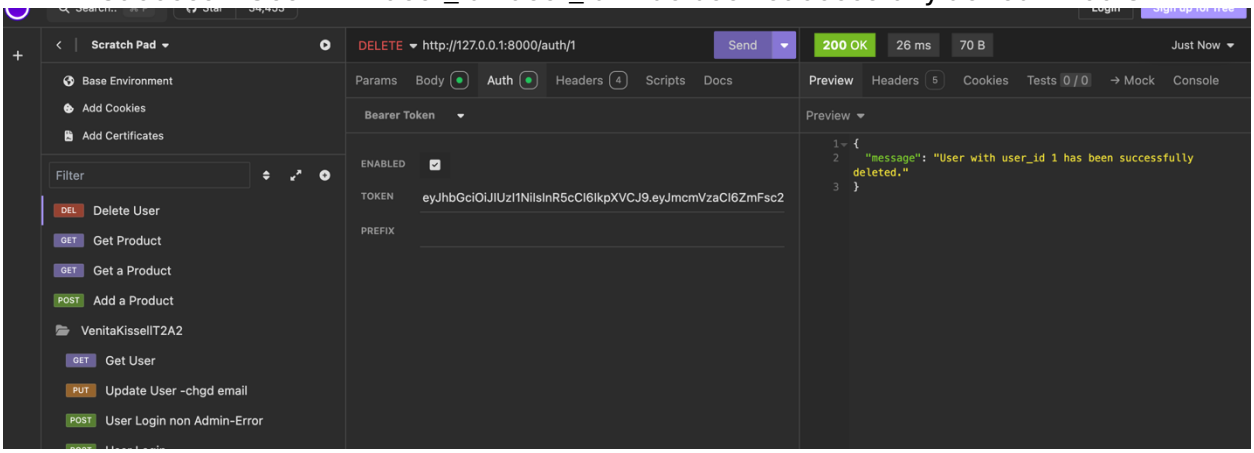


DELETE a USER

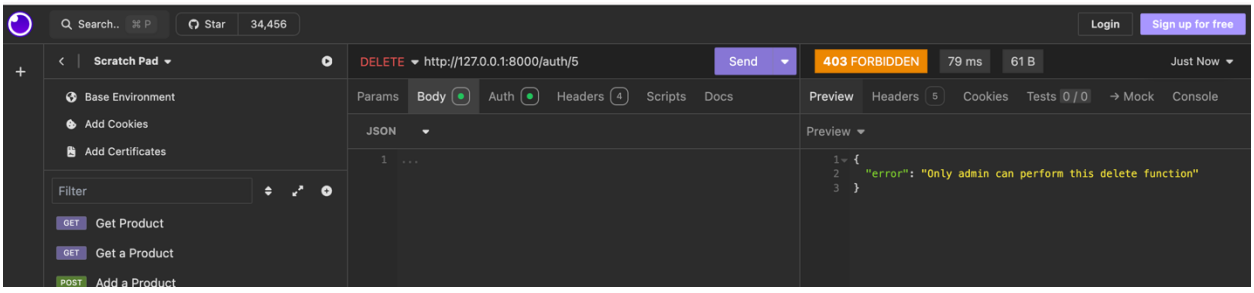
The user model is set up with specific attributes for the tables, and the controller actions these from the front end user input. The DELETE functions allows only an authorised admin user to Delete users from the table.

HTTP Verb – DELETE
Path or Route: http://localhost:8000/auth/1
Required body/header – Authorisation: bearer token for user authentication
Response –

Success: “User with user_id <user_id> has been successfully delted.” 200 OK



Error: “Only admin can perform this delete functions.” 403 Forbidden



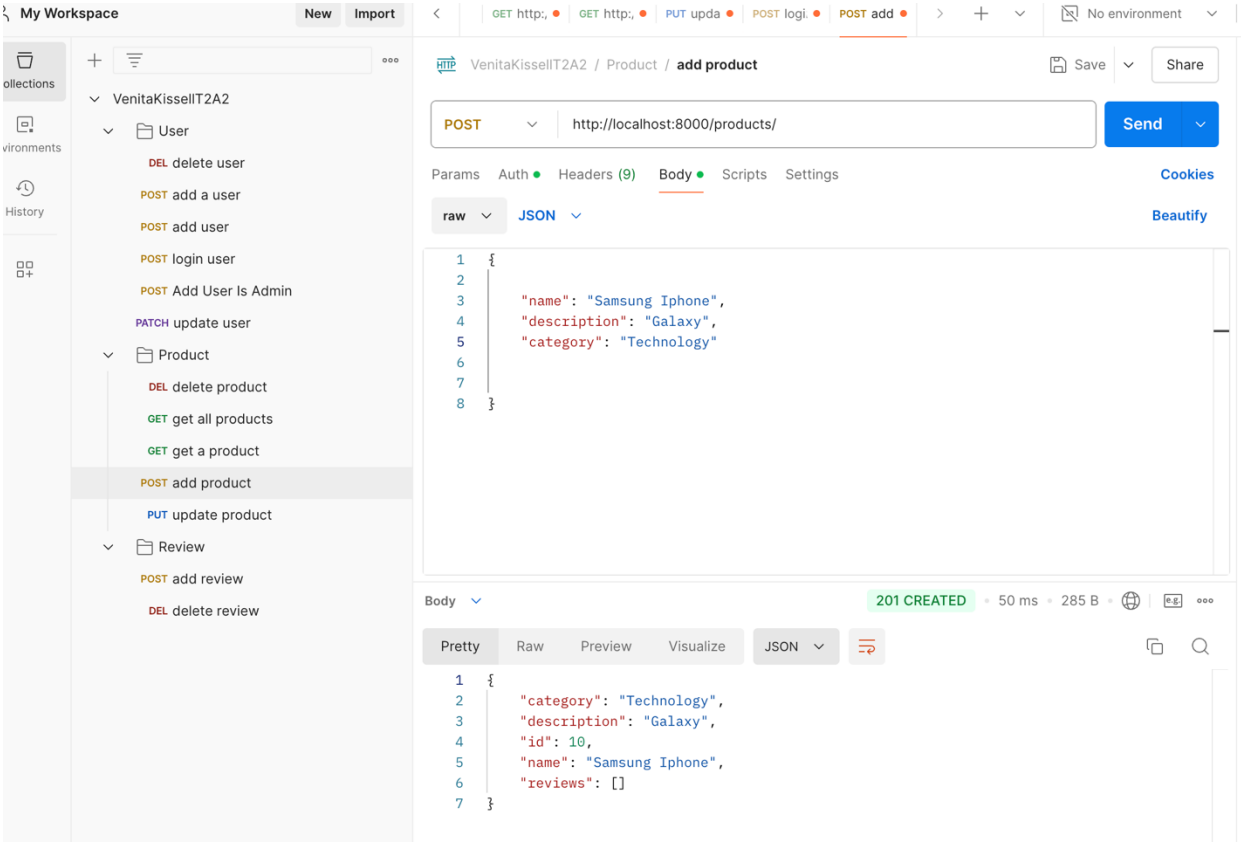
CREATE a Product

The product model is set up with specific attributes for the tables, and the controller actions these from the front end user input. The CREATE a product functions allows the user to input data into specific fields to add a product to the database. A JWT-Token is required to add a product to the database

```python
#Create a new product instance and add it into the Database
@product_bp.route("/", methods=['POST'])
@jwt_required()
def add_product():
    try:
        #Request the body data from the user input from the front end
        request_body_data = product_schema.load(request.get_json())


        #create the new product model instance
        product = Product(
            name = request_body_data.get('name'),
            description = request_body_data.get('description'),
            category = request_body_data.get('category'),
            user_id = get_jwt_identity()
        )
        #add product and commit to the database
        db.session.add(product)
        db.session.commit()

        #Return to the user success code 201
        return product_schema.dump(product), 201

    except ValidationError as err:
        return jsonify(err.messages), 400
```

HTTP Verb – POST
Path/Route: http://localhost:8000/products
Required body/header – name, description, category, user_id, JWT_Token
Response –

Success: 201 Created



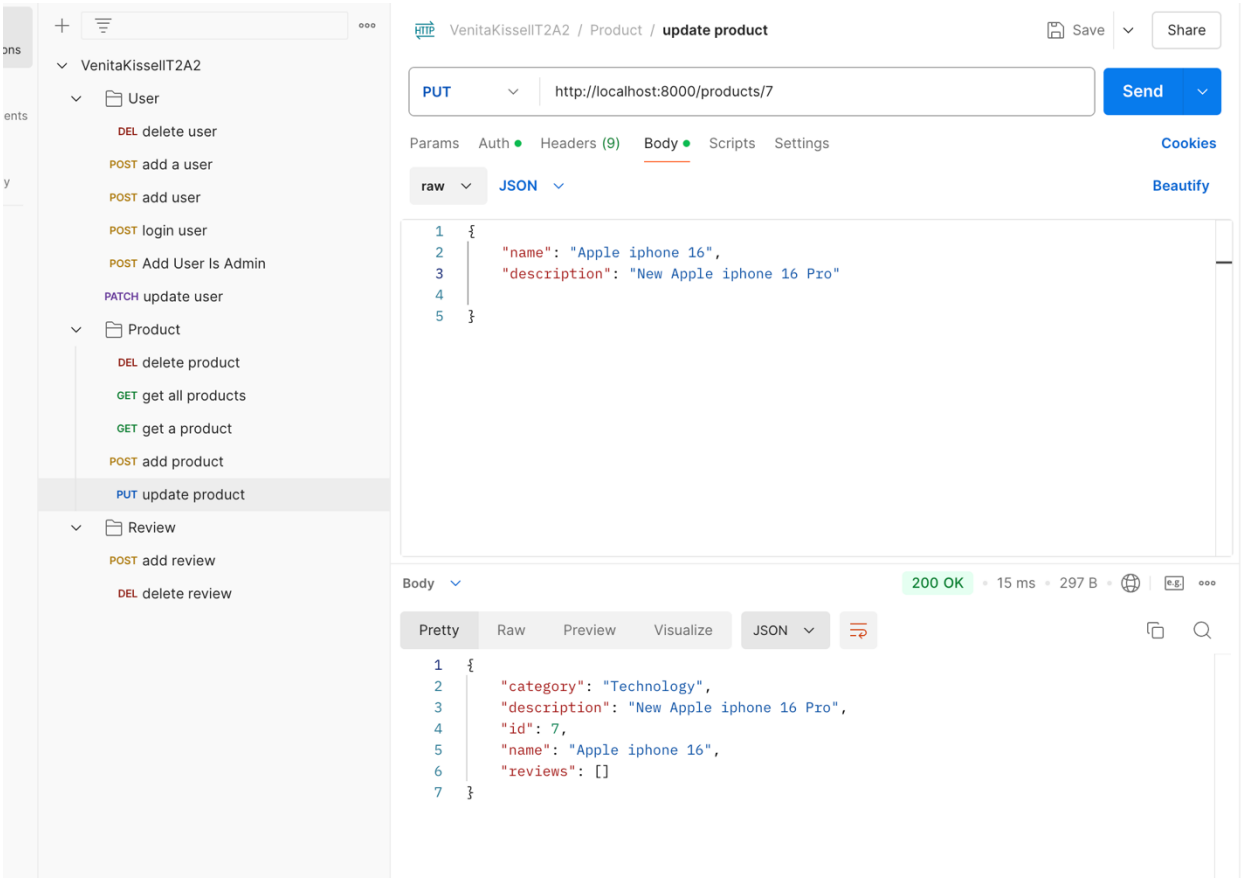
Error: “Invalid”, 400

UPDATE a PRODUCT

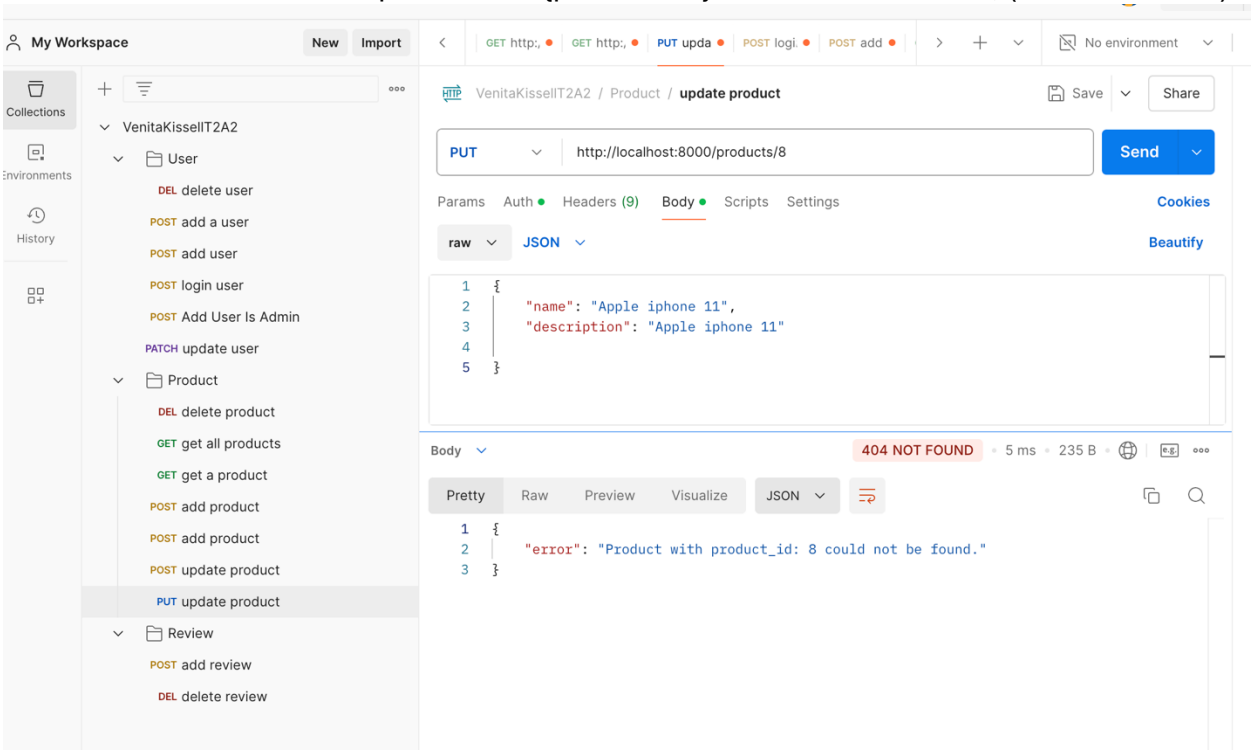
The product model is set up with specific attributes for the tables, and the controller actions these from the front end user input. The UPDATE function allows the user to input data into specific fields to update a product in the database, a JWT_Token is also required to update a product.

HTTP Verb –PUT,PATCH
Path or Route: http://localhost:8000/products/<product_id>
Required body/header – Authorisation: bearer token for user authentication, product_id, then the fields that are to updated
Response –

Success: “200 OK”



Error: “Product with product_id: {product_id} could not be found”, (404 Not Found)



GET a PRODUCT

The product model is set up with specific attributes for the tables, and the controller actions these from the front end user input. The GET a Product function allows the user to input <product_id> retrieve a specific product in the database.

```
#Retrieve one specific product from the Database
@product_bp.route("/<int:product_id>", methods=['GET'])
def get_a_product(product_id):
    stmt = db.select(Product).filter_by(id=product_id)
    #Select the user input search for a specific product
    product = db.session.scalar(stmt)
    #if product exists return product
    if product:
        return product_schema.dump(product)
    else:
        #Return error code to user if product can not be found in database
        return {"error": f"Product with product_id: '{product_id}' has not been found"}, 404
```

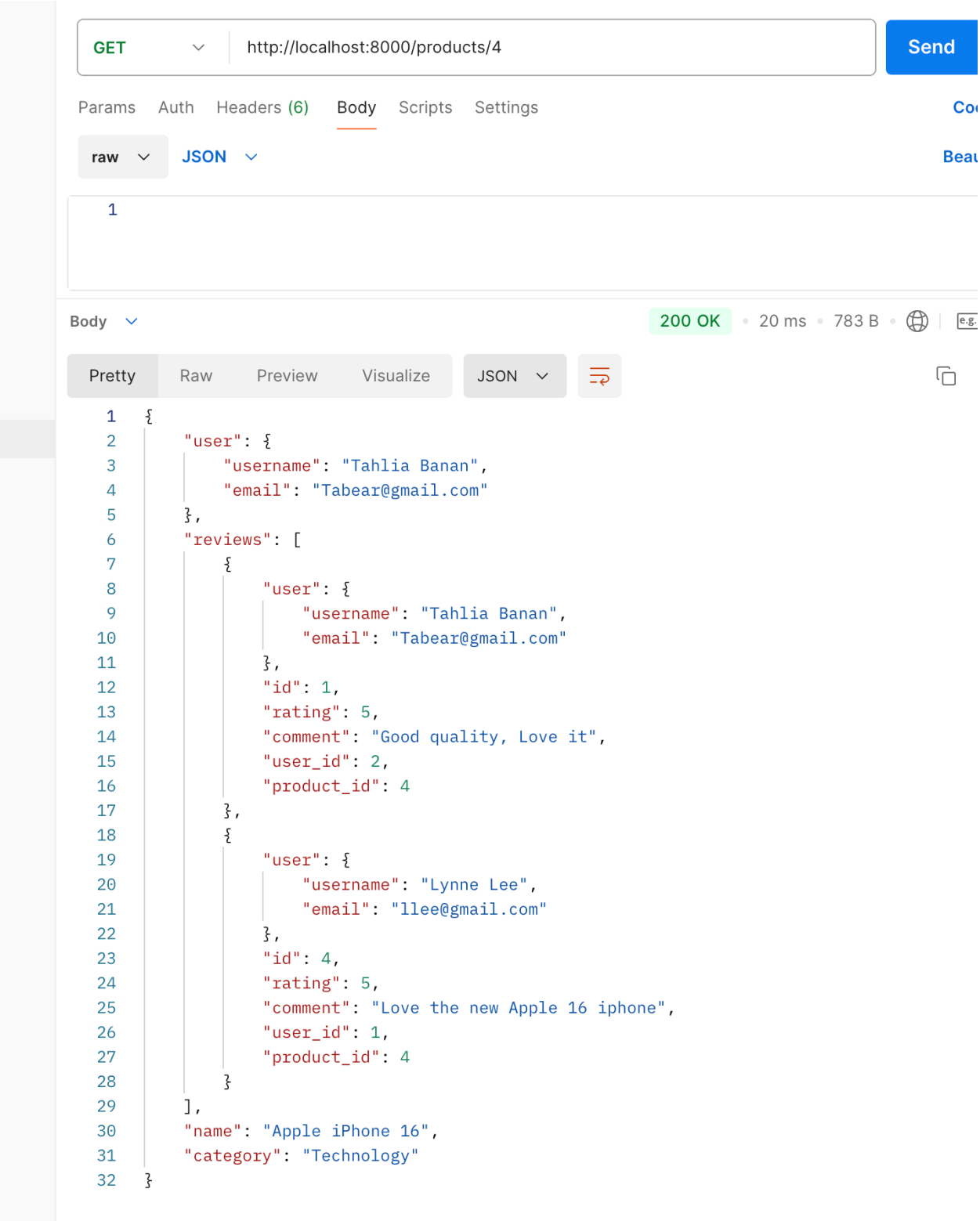
HTTP Verb – GET

Path or Route: http://localhost:8000/product/<product_id>

Required body/header – Authorisation: bearer token for user authentication

Response –

Success: “Review was successfully created”



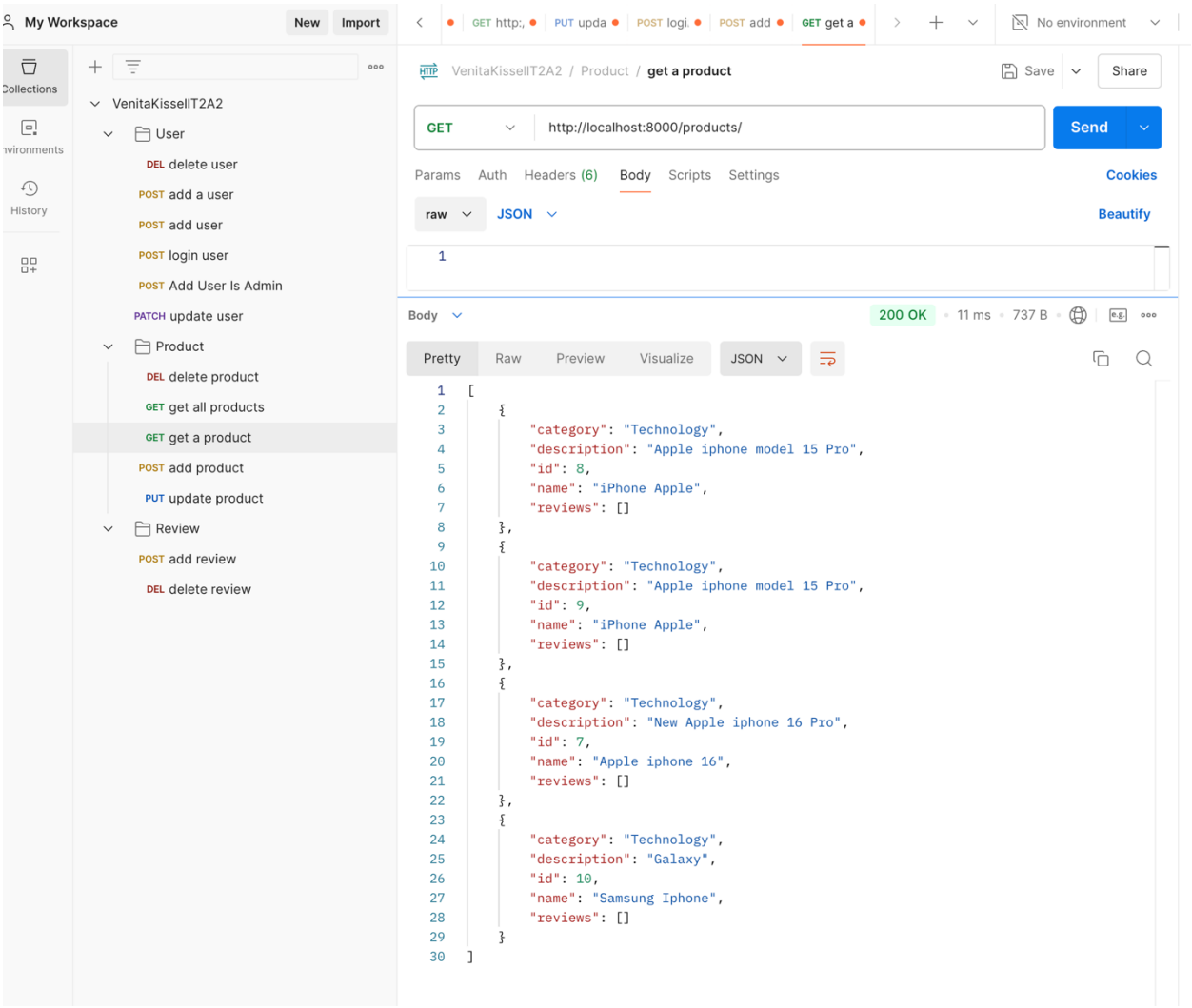
Error: “No product with {product_id} has been found” 404

GET all Products

The product model is set up with specific attributes for the tables, and the controller actions these from the front end user input. The GET ALL Products function allows the user to retrieve all products in the database.

```python
#GET all products in the Database
@product_bp.route("/")
def get_all_products():
    stmt = db.select(Product)
    products = db.session.scalars(stmt)
    #If products to return from the database return to user 200 for successful
    return products_schema.dump(products), 200
```

HTTP Verb – GET
Path/Route: http://localhost:8000/products
Required body/header – products
Response –
Success:

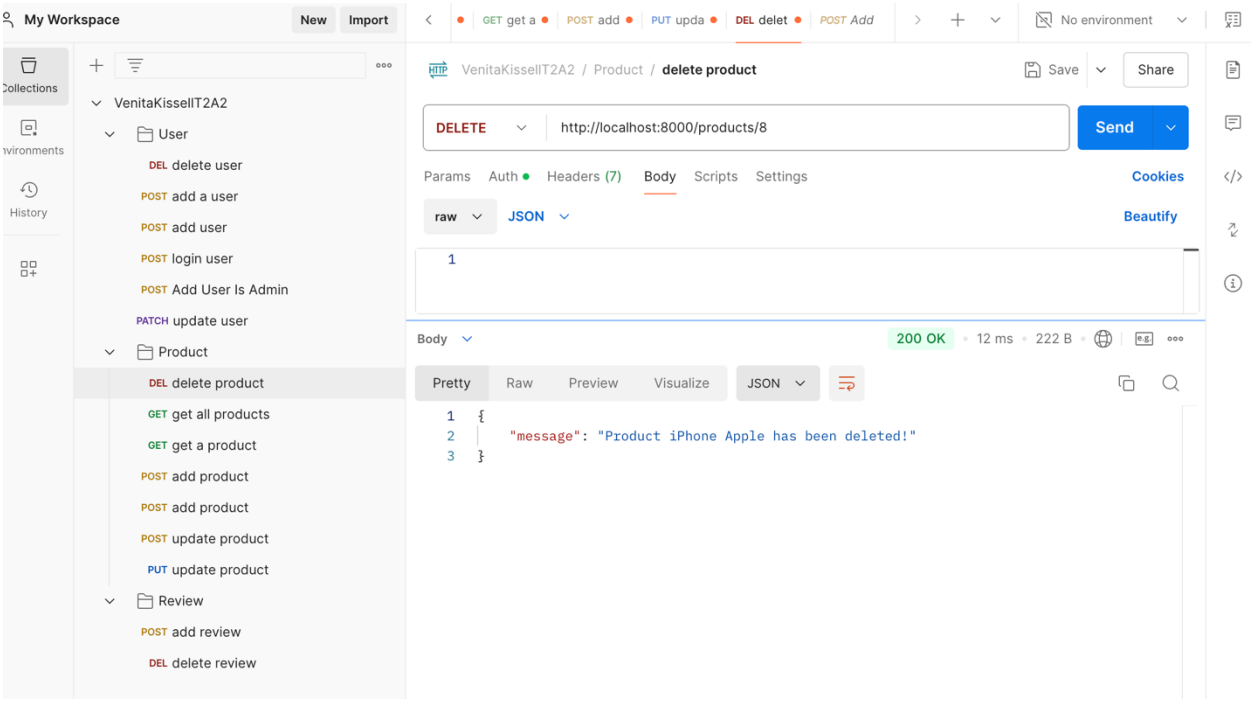


Error: “Invalid, products not found”, 404

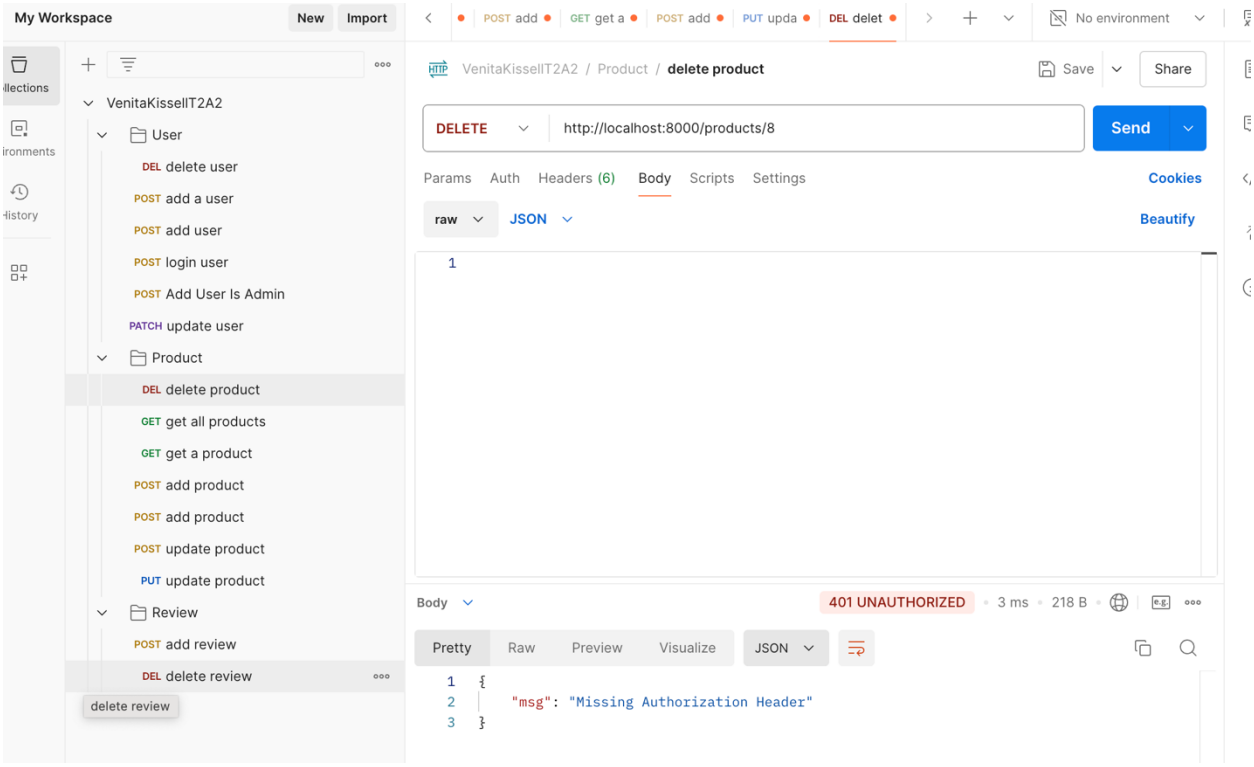
DELETE a PRODUCT

The product model is set up with specific attributes for the tables, and the controller actions these from the front end user input. The DELETE functions allows only an authorised admin user to Delete users from the table.

HTTP Verb – DELETE
Path or Route: http://localhost:8000/products
Required body/header – Authorisation: bearer token for user authentication/ <int:product_id>
Response –
Success: "Product {product_id} has been deleted!"



Error: “Missing Authorisation Header” 401 Unathorised



ADD a Review

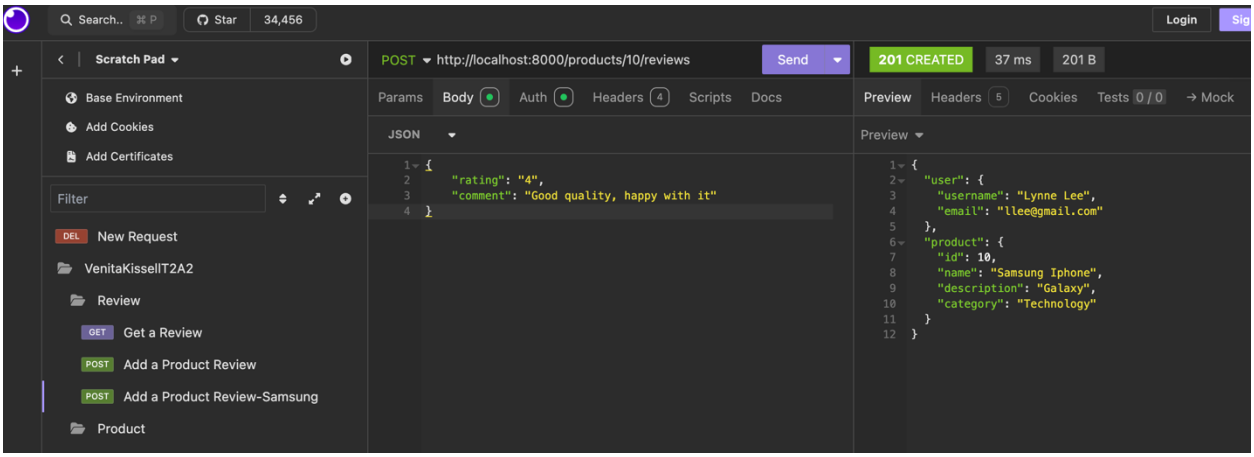
HTTP Verb – POST

The REVIEW model is set up with specific attributes for the tables, and the controller actions these from the front end user input. The ADD REVIEW function allows the user to add a review to a specific product in the database. The JWT_Token is required to add a review.

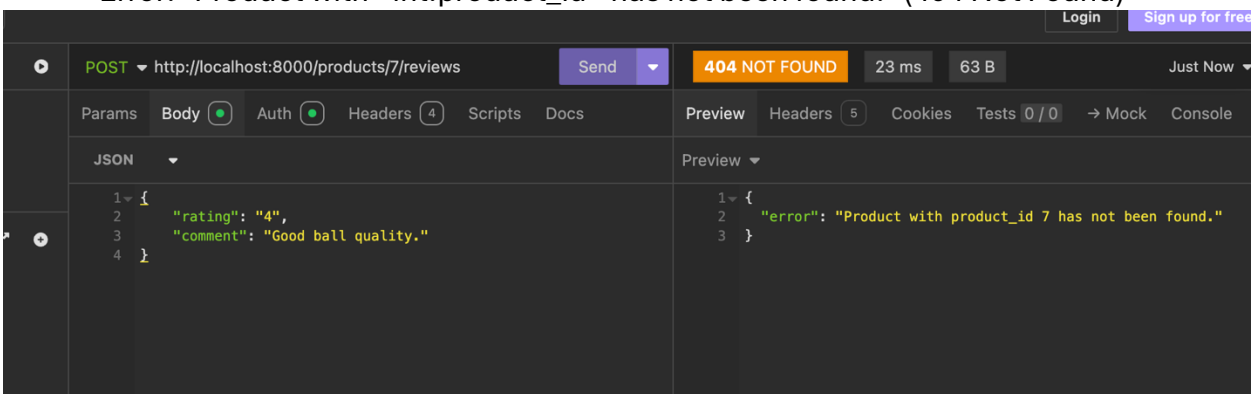
```python
#create the route for Adding a review
@review_bp.route('/', methods=['POST'])
@jwt_required()
def add_review(product_id):
    # get the user review from the request body
    request_body_data = request.get_json()
    # fetch the product with the matching product_id
    stmt = db.select(Product).filter_by(id=product_id)
    product = db.session.scalar(stmt)
    # if the product exists in the database system
    if product:
        # Add a review for the product
        review = Review (
            comment = request_body_data.get("comment"),
            rating = request_body_data.get("rating"),
            date = date.today(),
            product_id = product_id,
            user_id = get_jwt_identity()
        )
        # add the review to the database and commit the session
        db.session.add(review)
        db.session.commit()
        # return successfull review added, 201 added successfully
        return review_schema.dump(review), 201
    else:
        # Else return error, 404 Not Found
        return {"error": f"Product with product_id {product_id} has not been found."}, 404
```

Path/Route: http://localhost:8000/products/<int:product_id>/reviews
Required body/header – JWT Token, Rating, comment
Response –

Success: 201 Created OK



Error: "Product with <int:product_id> has not been found." (404 Not Found)



Error: “You are not authorised to update this review”, (403 Forbidden)

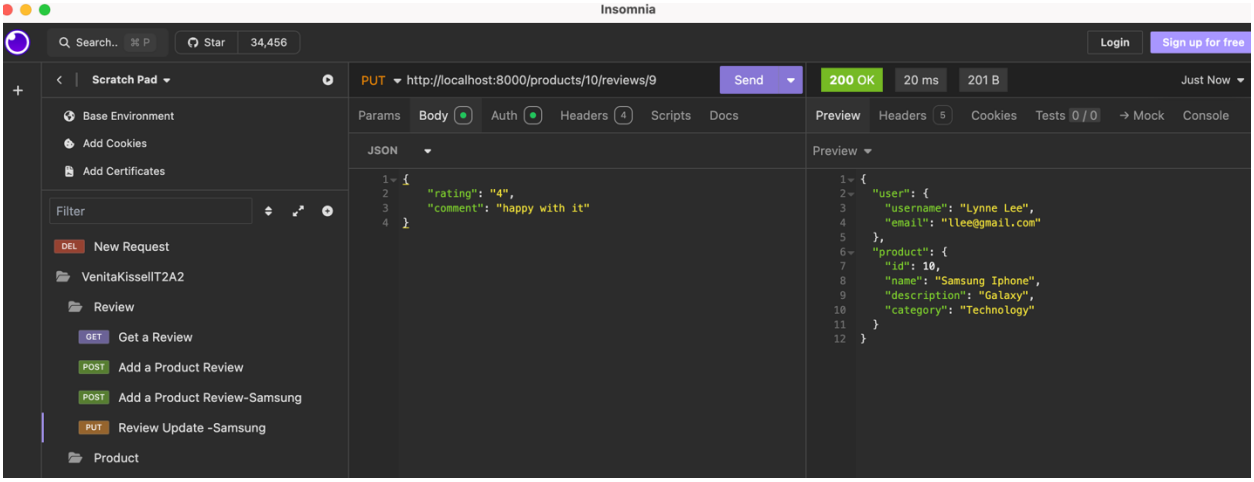
UPDATE a Review

The REVIEW model is set up with specific attributes for the tables, and the controller actions these from the front end user input. The UPDATE REVIEW function allows the user to update an existing review to a specific product in the database. Must have both product id and review id, must have JWT_Token to update

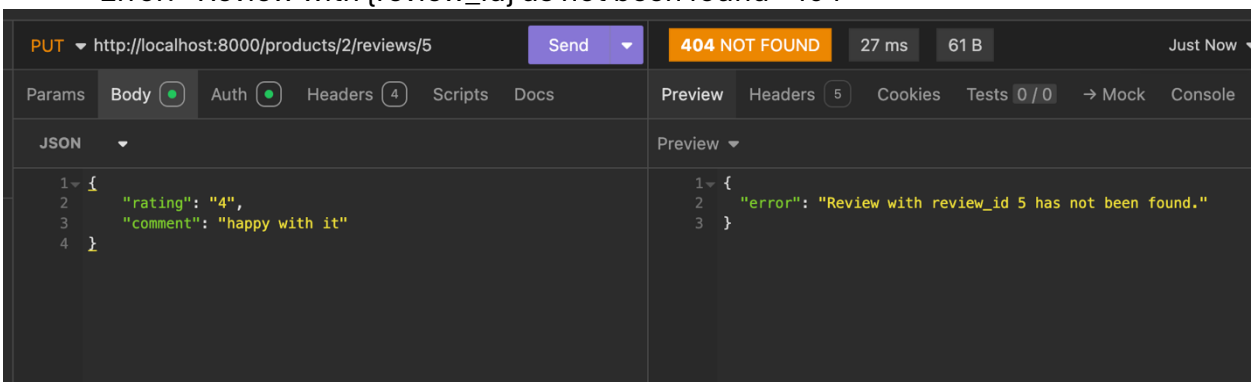
```
# Update and exisiting review and update details
@review_bp.route("/<int:review_id>", methods=["PUT", "PATCH"])
@jwt_required()
def update_review(product_id, review_id):
    # Retrieve the  user input value from the body of the request
    request_body_data = request.get_json()
    # find the review with id that matches id = review_id
    stmt = db.select(Review).filter_by(id=review_id)
    review = db.session.scalar(stmt)
    #If the review exists
    if review: #if there is a review, then update the review
        review.comment = request_body_data.get("comment") or review.comment
        # commit the changes to the database
        db.session.commit()
        # return the updated review, with a confirmation
        return review_schema.dump(review)

    else:
        # Else return error message, 404 Not Found
        return {"error": f"Review with review_id {review_id} has not been found."}, 404
```

HTTP Verb – PUT, PATCH
Path/Route: http://localhost:8000/products/<int:product_id>/reviews/<int:review_id>
Required body/header – product_id and review_id to update, JWT_Token
Response -

Success: “200 OK



Error: “Review with {review_id} as not been found” 404



GET a Review by PRODUCT ID

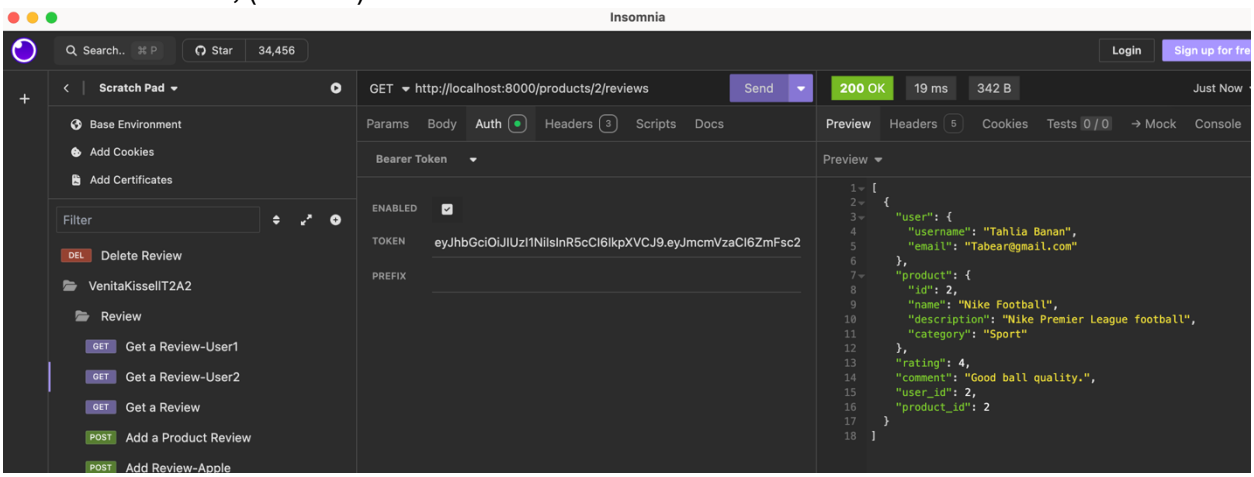
The REVIEW model is set up with specific attributes for the tables, and the controller actions these from the front end user input. The GET a REVIEW by Product function allows the user to fetch a review for a specific product in the database. Must have both product id and review id.

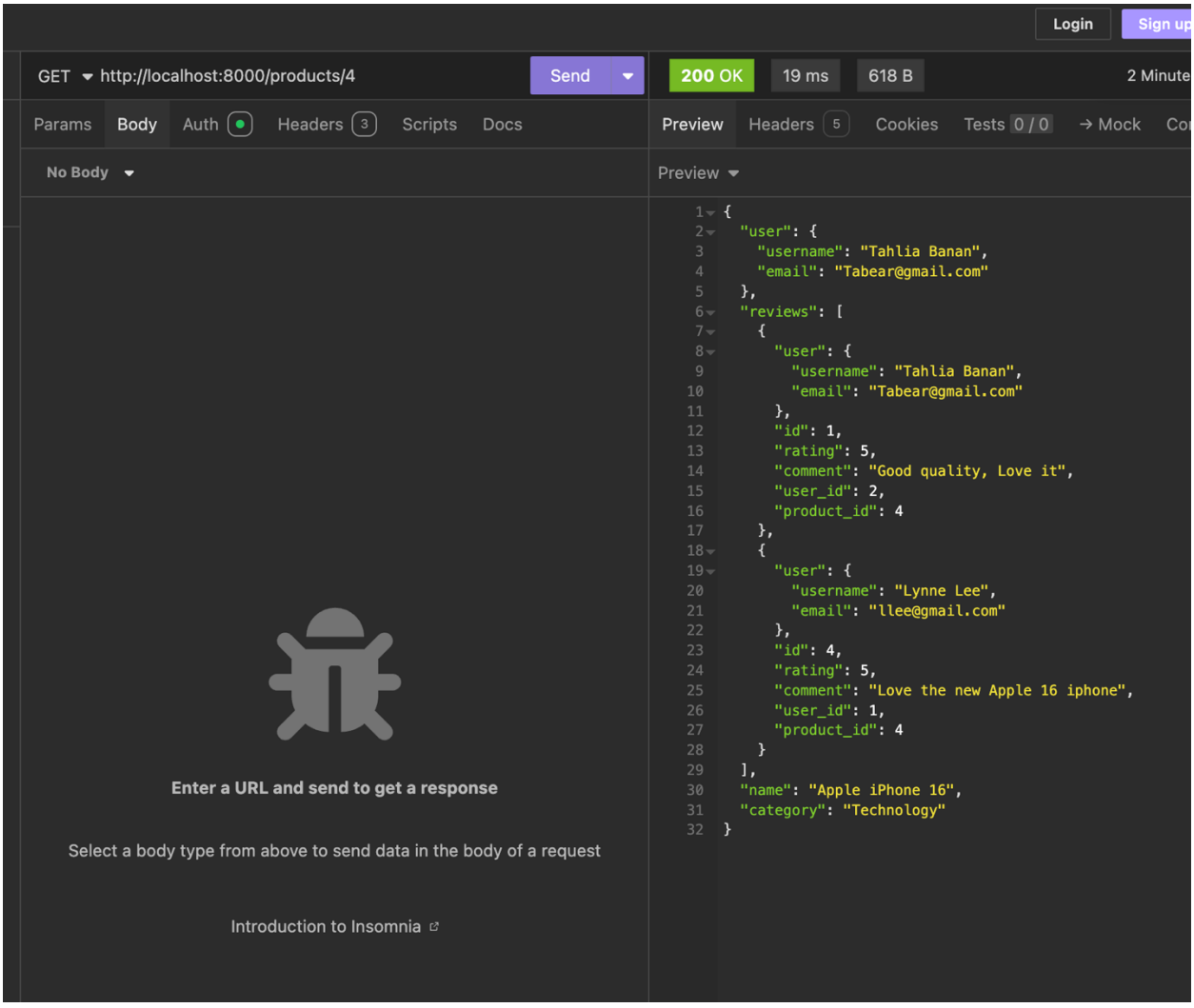
```python
#GET all reviews in the database for a specific product
@review_bp.route('/', methods=['GET'])
def get_a_review(product_id):
    stmt = db.select(Review).filter_by(id=product_id)
    reviews = db.session.scalars(stmt).all()
    if reviews:
        #If product id matches search, return from the database, 200 for successful
        return review_schema.dump(reviews), 200
    else:
        #return to user error message 400 Products not found
        return {"error": f"There were no products in the database"}, 404
```

HTTP Verb – GET
Path/Route: http://localhost:8000/products/<int:products_id>/reviews
Required body/header:, product_id,

Response –

Success: , (200 OK)





Error: “User not found”, (404 Not Found)

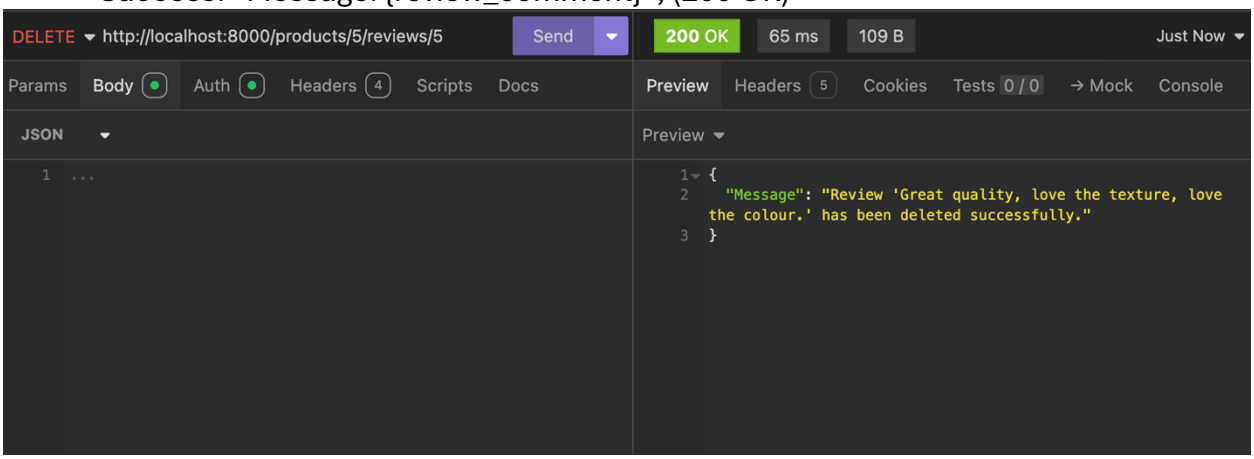
DELETE a Review

The REVIEW model is set up with specific attributes for the tables, and the controller actions these from the front end user input. The DELETE a REVIEW must have product_id and review_id for the function to allow the user to DELETE a review for a specific product in the database. Must have JWT_Token and auth_as_admin Admin user only can delete.

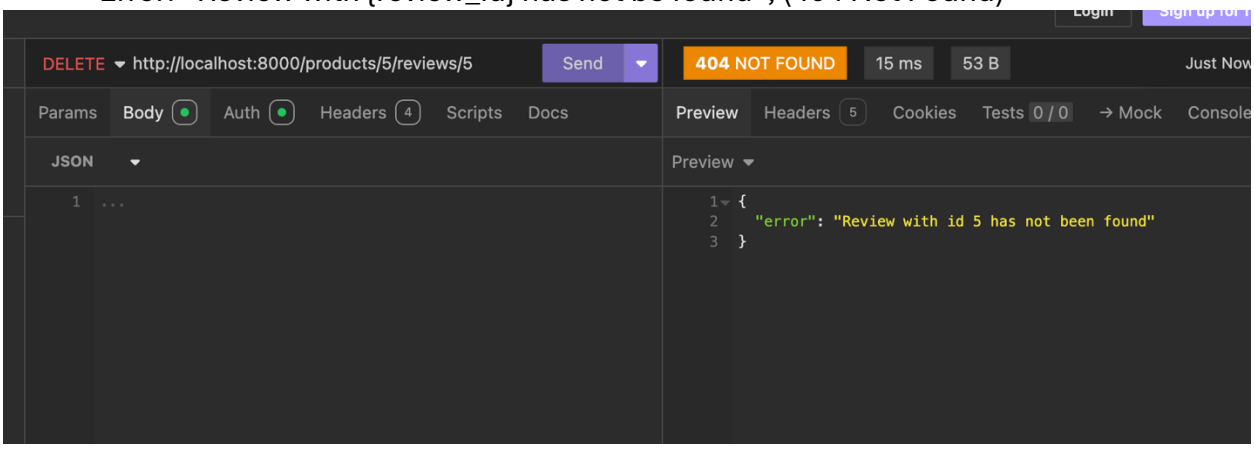
```python
# Delete Comment authorised user only
@review_bp.route("/<int:review_id>", methods=["DELETE"])
@jwt_required()
@auth_as_admin
def delete_review(product_id, review_id):
    try:
        # Retrieve the review_id from the database where id=review_id matches user input
        stmt = db.select(Review).filter_by(id=review_id)
        #Retrieves the review from the database
        review = db.session.scalar(stmt)
        # if review exists in the database:
        if review:
            # delete
            db.session.delete(review)
            #commit the changes to the database
            db.session.commit()
            # return confirmation message that review has been deleted
            return {"Message": f"Review '{review.comment}' has been deleted successfully."}

        else:
        #Else return error message, 404 Not found
            return {"error": f"Review with id {review_id} has not been found"}, 404

    except Exception as e:
        return{"error": "Unexpected Error", "details": str(e)}, 500
```

HTTP Verb – DELETE
Path/Route: http://localhost:8000/products/<int:products_id> / <int:review_id>
Required body/header –product_id, review_id plust JWT_Token & Admin Authorisation: Bearer token for user authentication.
Response -

Success: “Message: {review_comment}”, (200 OK)



Error: “Review with {review_id} has not be found”, (404 Not Found)



These API endpoints provide a comprehensive way to interact with the product review’s backend. It creates, retrieves, updates, and deletes user review while ensuring the data integrity and user authentication is maintained throughout the use of the app, the use of appropriate headers and error handling makes this a robust and scalable product review system.

References

4Geeks. (n.d.). *Understanding JWT and how to implement a simple JWT with Flask*. [online] Available at: https://4geeks.com/lesson/what-is-JWT-and-how-to-implement-with-Flask.

flask-sqlalchemy.palletsprojects.com. (n.d.). *Declaring Models — Flask-SQLAlchemy Documentation (2.x)*. [online] Available at: https://flask-sqlalchemy.palletsprojects.com/en/2.x/models/.

Gregorio, F.D. (n.d.). *psycopg2-binary: psycopg2 - Python-PostgreSQL Database Adapter*. [online] PyPI. Available at: https://pypi.org/project/psycopg2-binary/.

Paget, S. (2024). *Local Consumer Review Survey | Online Reviews Statistics & Trends*. [online] BrightLocal. Available at: https://www.brightlocal.com/research/local-consumer-review-survey/.

Podium.com.au. (2023). *Why Online Reviews Matter to Your Business (5 Reasons) - Podium*. [online] Available at: https://www.podium.com.au/guides/online-reviews-matter/.

Stackify. (2017). *What are CRUD Operations? Examples, Tutorials & More*. [online] Available at: https://stackify.com/what-are-crud-operations/.

ThinkDigits Inc (2024). *Learn how Flask-SQLAlchemy integrates SQLAlchemy to simplify database management in Flask apps, including setup, context handling, and*

convenience. [online] Linkedin.com. Available at: https://www.linkedin.com/pulse/relationship-between-sqlalchemy-flask-sqlalchemy-thinkdigits-hr1dc?trk=organization_guest_main-feed-card_feed-article-content [Accessed 29 Sep. 2024].

Timescale Blog. (2023). *When and How to Use Psycopg2*. [online] Available at: https://www.timescale.com/blog/when-and-how-to-use-psycopg2/.

Wei, H. (2019). *ORM For Python: SQLAlchemy 101 with Code Example*. [online] medium.com. Available at: https://medium.com/@haataa/orm-for-python-sqlalchemy-101-with-code-example-60868e65b0c.