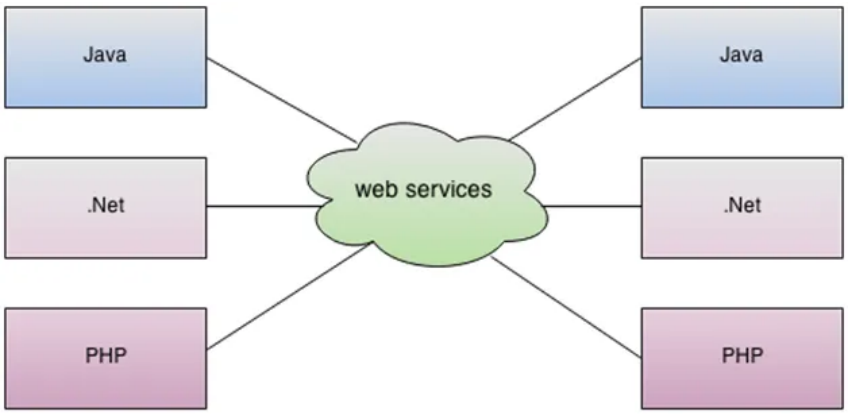
Introduction to Flask REST API

What is a Web Service?

A **Web Service** is a system that enables communication between applications over a network. It can be defined as:

- A client-server application or component that facilitates communication.
- A method of exchanging information between two devices over the internet.
- A software system designed for **machine-to-machine communication**.
- A collection of standards and protocols that enable applications to interact with each other, regardless of their underlying programming languages.

For example, a **Java application** can communicate with applications built using **.NET, PHP, or Python** through a web service. This makes web services a **language-independent** way of communication.



A web service is used to create an API that allows different applications, developed in different programming languages, to communicate with each other.

What is an API?

An **API (Application Programming Interface)** is a set of rules that allows different applications to communicate with each other. APIs enable one system to access data or functionality from another system, even if they are built using different technologies.

Key Features of an API:

- **Interoperability** – Enables communication between different applications (e.g., a weather app fetching data from a government API).
- **Data Exchange** – APIs allow applications to send and receive data in a structured format.
- **Automation** – Enables programmatic access to functionalities, reducing the need for manual work.
- **Security** – APIs can implement authentication and authorization mechanisms to control access.

Real-World Example of an API

Imagine you are ordering food through a food delivery app:

1. You select a restaurant and place an order.
2. The app sends your order to the restaurant's system.
3. The restaurant prepares the food and updates the app when it's ready.
4. The delivery person picks it up and delivers it to you.

Here, the food delivery app and the restaurant's system **communicate** through an API. Similarly, APIs allow different software applications to exchange data and perform tasks efficiently.

✅ **Google Maps API** – Used by ride-sharing apps to display routes.
✅ **Payment Gateway APIs** – Websites use Stripe or PayPal APIs to process payments.
✅ **Weather API** – Applications retrieve weather forecasts from external services.

APIs allow applications to interact **without knowing the internal details** of each other, making development **faster and more efficient**.

What is a REST API?

A **REST API (Representational State Transfer API)** is a type of web service that follows REST principles. REST APIs use HTTP methods to interact with data in a structured way.

Key HTTP Methods Used in REST APIs:

Action	HTTP Methods	Description
Create	**POST**	Add a new resource (e.g., create a new user)
Read	**GET**	Retrieve data (e.g., fetch user details)
Update	**PUT**	Modify existing data (e.g., update user profile)
Delete	**DELETE**	Remove a resource (e.g., delete a user account)

💡 **Example:** Think of a **library system**:

- **GET** → A student views a list of available books.
- **POST** → A librarian adds a new book to the system.
- **PUT** → A librarian updates book details (e.g., changes the price).
- **DELETE** → A book is removed when it's no longer available.

Why Do We Need a REST API in Flask?

So far, we have built **Flask web applications** where users interact with a web page and see data displayed visually using HTML templates. But what if we want **another application** (not just a human) to access the data?

Real-Life Examples of Why We Need REST APIs

✅ E-commerce Website (Amazon, Flipkart, etc.)

- Initially, users shop online through a website.
- Later, a **mobile app** was developed.
- Instead of rewriting the logic, a **REST API** is created so both the website and app can access the same data.

✅ **Bus Ticket Booking System**

- A travel company wants users to check available bus seats on multiple booking platforms.
- Instead of updating each platform manually, a **REST API** allows them to fetch real-time seat availability.

Without a REST API, these systems would have to **manually share data**, which is inefficient and error-prone.

How to Build a REST API in Flask?

Flask provides multiple ways to create REST APIs, but the easiest method is using **Flask-RESTful**, a simple extension that helps define API endpoints.

Steps to Learn REST API Development in Flask:

1. **Understand APIs and REST**
    - Learn what APIs are and how REST APIs work.
    - Understand HTTP methods (GET, POST, PUT, DELETE).
2. **Learn JSON Format**
    - Understand how data is exchanged using JSON format.
3. **Set Up Flask for API Development**
    - Install Flask and Flask-RESTful.
    - Set up a basic Flask application.
4. **Create Your First REST API in Flask**
    - Define API endpoints using Flask-RESTful.
    - Implement basic CRUD operations.
5. **Use Postman to Test Your API**
    - Learn to send requests and view responses using Postman.
6. **Implement Authentication & Authorization**
    - Secure your API endpoints with authentication (e.g., API keys, JWT).
7. **Connect Flask API to a Database**
    - Use SQLAlchemy to store and retrieve data from a database.
8. **Handle Errors & Responses Properly**

- Return proper HTTP status codes (200 OK, 404 Not Found, 400 Bad Request, etc.).
- Implement error handling for invalid requests.

Steps to develop the Flask REST API:

1. Install Flask & Flask-RESTful (Install the necessary libraries)

   pip install flask

   pip install flask-restful

2. Setup Flask Application:

```
from flask import Flask

app = Flask(__name__)
```

3. Create a Resource:

- The **Resource** class is where you define the functionality of your API. You define different methods (e.g., GET, POST, PUT, DELETE) that correspond to the HTTP actions.

  For example, creating a resource for student data:

```
from flask_restful import Resource,Api

api = Api(app)

class Student(Resource):

    def get(self):

        return {"name": "Raj", "address": "delhi", "marks": 850}



# Add resource to the API and set its endpoint

api.add_resource(Student, '/')
```

4. Run the Flask Application:

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

5. Test the API
    - Use Postman or a browser to test the endpoints (e.g., `http://127.0.0.1:5000/`).

Understanding Resources and Functionalities

In Flask-RESTful, a **Resource** is a Python class that represents an entity in your application (e.g., a user, product, or post). A resource handles HTTP methods like **GET**, **POST**, **PUT**, and **DELETE**. Each of these methods corresponds to the standard CRUD operations:

- **GET**: Retrieve the resource (data) from the server. Typically used to fetch data.
- **POST**: Create a new resource. It's often used to add data to the server.
- **PUT**: Update an existing resource. Used when modifying data on the server.
- **DELETE**: Remove the resource. Used to delete data from the server.

For each resource, you can define multiple methods depending on what actions you want to perform on that resource. You can have just one or several of these methods, depending on your use case.

Example:

```
class Student(Resource):

    def get(self, student_id):

        # Retrieve a student's details by ID

        student = get_student_from_db(student_id)

        if student:

            return student # Return student in json


        return {"message": "Student not found"}, 404
```

```
def post(self):
    # Add a new student to the database
    data = request.get_json()
    add_student_to_db(data)
    return {"message": "Student created successfully"}, 201

def put(self, student_id):
    # Update student information
    data = request.get_json()
    update_student_in_db(student_id, data)
    return {"message": "Student updated successfully"}

def delete(self, student_id):
    # Remove student from database
    delete_student_from_db(student_id)
    return {"message": "Student deleted successfully"}
```

Using Postman for API Testing

Postman is a popular API testing tool that allows developers to make HTTP requests to an API and inspect the responses. It's an essential tool for testing and debugging REST APIs without needing to write front-end code.

Here's a quick overview of how to use **Postman**:

Basic Steps to Use Postman:

1. **Install Postman**
   Download and install Postman from Postman's official website
   https://www.postman.com/downloads/

2. **Create a Request**
   - Open Postman and click on the **New Request** button.
   - Enter the **URL** of the API endpoint (e.g., `http://127.0.0.1:5000/student`).
3. **Choose HTTP Method**
   Select the **HTTP method** you want to test from the dropdown (GET, POST, PUT, DELETE).
4. **Send the Request**
   Click **Send** to send the request to your API.
5. **Inspect the Response**
   - Check the response in the **Body** tab (which should show the data returned by the API).
   - You'll also see **status codes** (200 OK, 201 Created, 404 Not Found) in the **Status** section, which helps determine if the request was successful.
6. **Testing Different Methods**
   - **GET**: Fetch data from the server (no body needed).
   - **POST**: Include data in the **Body** to create a new resource. You can choose the format (usually JSON).
   - **PUT**: Similar to POST but used for updating existing data.
   - **DELETE**: Send a request to delete data by providing an ID or parameter in the URL.
7. **Save Requests**
   Postman allows you to save requests for later use. You can create **collections** of API requests and organize them.

Creating Flask REST App with In-memory storage:

app.py:

```
from flask import Flask, request

from flask_restful import Api, Resource
```

```
app = Flask(__name__)

api = Api(app)

# Sample in-memory product list
products = [

    {"productid": 1, "name": "Speaker", "price": 5000,
        "quantity": 50, "category": "electronics"},
    {"productid": 2, "name": "Smartphone", "price": 25000,
        "quantity": 10, "category": "electronics"},
    {"productid": 3, "name": "Refrigrator", "price": 45000,
        "quantity": 5, "category": "homeapplience"},
    {"productid": 4, "name": "Notebook", "price": 50,
        "quantity": 500, "category": "stationary"}

]


class Product(Resource):

    # Helper function to find product by ID
    def find_product_by_id(self, productid):
```

```
        product = None
        for prod in products:
            if prod.get("productid") == productid:
                product = prod

        return product

    def get(self, productid):

        product = self.find_product_by_id(productid)

        if product:
            return product
        else:
            return {"message": "Product Not Found!"}, 404

    def post(self):
        data = request.get_json()

        # Validate request data
        if not data.get("name") or not data.get("price") or not
data.get("quantity") or not data.get("category"):
            return {"message": "All fields (name, price, quantity,
category) are required."}, 400
```

```
        # Create new product
        new_product = {
            "productid": len(products)+1,  # Auto incremented productid
            "name": data.get("name"),
            "price": data.get("price"),
            "quantity": data.get("quantity"),
            "category": data.get("category")
        }

        # Add new product to the list
        products.append(new_product)
        return {"message": "Product Added Successfully", "product":
new_product}, 201


    def put(self, productid):
        product = self.find_product_by_id(productid)

        if product:
            data = request.get_json()

            product["name"] = data.get("name", product.get("name"))
            product["price"] = data.get("price", product.get("price"))
```

```
            product["quantity"] = data.get("quantity",
product.get("quantity"))

            product["category"] = data.get("category",
product.get("category"))



            return {"message": "Product Updated Successfully", "Product":
product}, 200



        else:

            return {"message": "Product Not Found!"}, 404



    def delete(self, productid):

        product = self.find_product_by_id(productid)



        if product:

            products.remove(product)

            return {"message": "Product deleted successfully", "product":
product}



        else:

            return {"message": "Product Not Found!"}, 404




class ProductCategory(Resource):

    def get(self, category):
```

```
        category_products = []

        for prod in products:
            if (prod.get("category") == category):
                category_products.append(prod)

        if category_products:
            return {"products": category_products}
        else:
            return {"message": "No products found in this category"}, 404


class Products(Resource):
    def get(self):
        # Get all products
        return {"products": products}


# Routes
# /products for all products
api.add_resource(Products, "/products")
```

```
# POST: /products for adding Product
# /products/<int:productid> for getting,updating,deleting single product
api.add_resource(Product, "/products", "/products/<int:productid>")

# /products/category/<category> for filtering by category
api.add_resource(ProductCategory, "/products/category/<string:category>")

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

Test the above application:

- To get all the product list:

  GET http://127.0.0.1:5000/products

- To get a specific product:

  GET http://127.0.0.1:5000/products/1

- To Add a new Product:

  POST: http://127.0.0.1:5000/products

Payload:

```
{
    "name": "TV",
    "price": 8000,
    "quantity": 10,
    "category": "electronics"
}
```

- To Delete an existing Product:

  DELETE: http://127.0.0.1:5000/products/1

- To Update an existing product:

  PUT: http://127.0.0.1:5000/products/2

  Payload:

```
{
   "price": 30000,
   "quantity": 50
}
```

- To get all the products from the electronics category

  GET: http://127.0.0.1:5000/products/category/electronics

Modifying the above application using Database SQLite and Flask-sqlalchemy:

```
pip install flask-sqlalchemy
```

Note: Here we need to convert the product object to the json before returning it from the application. for this we will create a function as_dict() inside the ProductModel class.

app.py:

```
from flask import Flask, request
from flask_restful import Api, Resource
from flask_sqlalchemy import SQLAlchemy
import os


basedir = os.path.abspath(os.path.dirname(__file__))


app = Flask(__name__)


api = Api(app)


# Database setup
```

```
app.config['SQLALCHEMY_DATABASE_URI'] = "sqlite:///" + \
    os.path.join(basedir, "product.db")
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
db = SQLAlchemy(app)



# Product model for SQLAlchemy
class Product(db.Model):
    __tablename__ = 'products'

    productid = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(100), nullable=False)
    price = db.Column(db.Float, nullable=False)
    quantity = db.Column(db.Integer, nullable=False)
    category = db.Column(db.String(50), nullable=False)

    def __repr__(self):
        return f"ProductId: {self.productid} Name: {self.name} Price:
{self.price} Quantity: {self.quantity} Category: {self.category}"

     # Method to return a dictionary representation of the product

    def as_dict(self):
        return {
```

```
            "productid": self.productid,
            "name": self.name,
            "price": self.price,
            "quantity": self.quantity,
            "category": self.category
        }


with app.app_context():
    db.create_all()


class ProductResource(Resource):

    def get(self, productid):

        product = db.session.get(Product, productid)

        if product:
            return product.as_dict()
        else:
            return {"message": "Product Not Found!"}, 404
```

```
    def post(self):
        data = request.get_json()

        # Validate request data
        if not data.get("name") or not data.get("price") or not
data.get("quantity") or not data.get("category"):
            return {"message": "All fields (name, price, quantity,
category) are required."}, 400

        # Create new product
        new_product = Product(name=data.get("name"),
                              price=data.get("price"),
                              quantity=data.get("quantity"),
                              category=data.get("category")
                              )

        # Add new product to the Database
        db.session.add(new_product)
        db.session.commit()
        return {"message": "Product Added Successfully", "product":
new_product.as_dict()}, 201

    def put(self, productid):
        product = db.session.get(Product, productid)
```

```
        if product:
            data = request.get_json()

            product.name = data.get("name", product.name)
            product.price = data.get("price", product.price)
            product.quantity = data.get("quantity", product.quantity)
            product.category = data.get("category", product.category)

            db.session.add(product)
            db.session.commit()
            return {"message": "Product Updated Successfully", "Product":
product.as_dict()}, 200

        else:
            return {"message": "Product Not Found!"}, 404

    def delete(self, productid):
        product = db.session.get(Product, productid)

        if product:
            db.session.delete(product)
            db.session.commit()
```

```
            return {"message": "Product deleted successfully", "product":
product.as_dict()}


        else:

            return {"message": "Product Not Found!"}, 404




class ProductCategory(Resource):

    def get(self, category):


        all_category_products = Product.query.filter_by(

            category=category).all()


        # Converting it to the JSON

        # category_products = [product.as_dict() for product in
all_category_products]

        category_products = []

        for product in all_category_products:

            category_products.append(product.as_dict())


        if category_products:

            return {"products": category_products}

        else:

            return {"message": "No products found in this category"}, 404
```

```python
class ProductsResource(Resource):
    def get(self):
        # Get all products
        allproducts = Product.query.all()

        # Converting them to the list of json
        # products = [ product.as_dict() for product in allproducts]
        products = []
        for product in allproducts:
            products.append(product.as_dict())

        return {"products": products}



# Routes
# /products for all products
api.add_resource(ProductsResource, "/products")



# POST: /products for adding Product
# /products/<int:productid> for getting, updating and deleting a single product
```

```
api.add_resource(ProductResource, "/products",
"/products/<int:productid>")



# /products/category/<category> for filtering by category

api.add_resource(ProductCategory, "/products/category/<string:category>")



if __name__ == '__main__':

    app.run(debug=True)
```

Test Endpoints in Postman:

- To add a new product:

  POST: http://127.0.0.1:5000/products

  payload:

  {

  "name": "Notebook",

  "price": 50,

  "quantity": 50,

  "category": "Stationary"

  }

- To Get All products:

  GET: http://127.0.0.1:5000/products

- To Get a Specific Product:

  GET: http://127.0.0.1:5000/products/1

- To Update an existing product:

  PUT: http://127.0.0.1:5000/products/1

  Payload:

  ```
  {
     "price": 80,
     "quantity": 150,
  }
  ```

- To delete an existing product:

  DELETE: http://127.0.0.1:5000/products/1

Securing Your API:

Implementing Authentication & Authorization with API Keys and JWT:

Session-Based Authentication vs. Token-Based Authentication

1. Session-Based Authentication:

In session-based authentication, the server maintains the user's session in memory or a database. Upon successful login, a **session ID** is created and stored on the server. The session ID is then sent to the client and stored in a **cookie**. Every time the client sends a request, the server checks the session ID stored in the cookie to authenticate the user.

Example:

- The user logs in with a username and password.
- The server creates a session for the user and stores the session ID.
- The session ID is sent to the client as a cookie.
- In subsequent requests, the client sends the session ID in the cookie to authenticate.

Advantages:

- Simple to implement.
- Server-side control over sessions (can invalidate or expire sessions).

Disadvantages:

- The server must store session data, which can become inefficient for a large number of users.
- Scalability can be an issue as the server needs to store sessions, and in a distributed environment, it requires session synchronization.

2. Token-Based Authentication (JWT):

In token-based authentication, the server issues a **JWT** to the client upon login, and the client stores this token (typically in localStorage or memory). The token contains the user's identity and is sent as part of the HTTP request in the **Authorization header**. The server does not store the session; instead, it verifies the authenticity of the token for each request.

Example:

- The user logs in with a username and password.
- The server creates a JWT and sends it back to the client.

- The client stores the JWT (usually in localStorage or memory).
- In subsequent requests, the client sends the token as a Bearer token in the Authorization header.

Advantages:

- Stateless: No need to store sessions on the server; JWT contains all the necessary data.
- Scalable: Ideal for large applications, especially in microservices and distributed systems.
- The token can be used across different domains or services.

Disadvantages:

- Once the token is issued, it cannot be invalidated easily (unless additional measures like token blacklisting are implemented).
- If the token is compromised, it can be used until it expires.

Real-Time Example of Session-Based vs Token-Based Authentication:

Session-Based Authentication Example:

- **Banking Application**:
    - A user logs into their online banking app. The server creates a session for the user.
    - The session ID is stored in a cookie.
    - Every time the user performs a transaction, the server checks the session ID to verify the user's identity.
    - If the user logs out, the session is invalidated on the server.

Token-Based Authentication Example:

- **Social Media Application**:
    - A user logs into their social media account. The server generates a JWT containing the user's ID and role (admin, user).
    - The client stores the JWT and sends it in the Authorization header in subsequent requests (e.g., posting a status, uploading a photo).
    - The server verifies the JWT to ensure that the user is authenticated and authorized to perform the requested actions.
    - JWTs typically expire after a certain period, requiring the user to log in again.

Conclusion:

- **Session-based authentication** stores user information on the server, while **token-based authentication (JWT)** relies on tokens to store authentication information, making it stateless.
- **JWT** is ideal for scalable, distributed applications, while session-based authentication works well for simple, monolithic applications.

Introduction to JWT (JSON Web Tokens)

JWT (JSON Web Token) is a compact, URL-safe means of representing claims between two parties. JWTs are widely used for authentication and authorization in web applications, as they provide a way to securely transmit information between a client and a server. The information is encoded as a JSON object, which can be verified and trusted because it is digitally signed.

A typical JWT consists of three parts:

1. **Header**: This typically consists of the type of the token (JWT) and the signing algorithm (e.g., HMAC SHA256 or RSA).
2. **Payload**: This contains the claims (statements about an entity, typically the user, and additional data). The payload can contain information such as the user's ID, role, and other data necessary for the application.
3. **Signature**: The signature is used to verify that the sender of the JWT is who it says it is, and to ensure that the message wasn't changed along the way.

A JWT looks like this:

```
xxxxx.yyyyy.zzzzz
```

Where:

- **xxxxx**: Encoded header
- **yyyyy**: Encoded payload (claims)
- **zzzzz**: Encoded signature

refer: https://jwt.io/

How to Use JWT in a Flask Application

1. **Install the Required Packages:** You'll need the following libraries for JWT-based authentication:

```
pip install flask-jwt-extended
```

2. **Flask Setup for JWT:** In Flask, we can integrate JWT for secure user authentication. The `flask_jwt_extended` package provides a simple way to manage JWTs.

```
from flask import Flask
from flask_jwt_extended import JWTManager

app = Flask(__name__)
app.config['JWT_SECRET_KEY'] = 'your_secret_key'  # This is used to encode/decode the JWT
jwt = JWTManager(app)
```

3. **User Authentication with JWT:** When a user logs in, you authenticate their credentials (usually with a username and password). If the credentials are correct, you generate a JWT access token and return it to the user.

```
from flask_jwt_extended import create_access_token

@app.route('/login', methods=['POST'])
def login():
    # Authenticate the user and generate token
    access_token = create_access_token(identity=user_id)
    return {"access_token": access_token}
```

4. **Protect Routes with JWT:** Once the user receives the JWT token, they need to send it in the Authorization header as `Bearer token_value` when making subsequent API requests. This is how you protect certain routes:

```
from flask_jwt_extended import jwt_required, get_jwt_identity

@app.route('/protected', methods=['GET'])

@jwt_required()

def protected():

    current_user = get_jwt_identity()

    return {"message": f"Hello, {current_user}"}
```

What is the "Identity" in JWT?

In the context of JWT, **identity** refers to the user's unique information (such as the user ID, email, or role) that is stored in the JWT. This is used to verify the identity of the user and authorize them to access certain resources.

When creating a JWT token, you can include identity-related information, such as the user's **role** or **user**

```
access_token = create_access_token(identity=user.id)  # Storing user ID as identity
```

When you decode the token and access the identity, you can check the user's role or any other data to authorize access to specific resources:

```
from flask_jwt_extended import get_jwt_identity

@jwt_required()

def some_protected_resource():

    current_user = get_jwt_identity()
```

```
    if current_user == 'admin':
        return "Admin Access Granted"
    else:
        return "Access Denied"
```

app.py:

```
from flask import Flask, request
from flask_restful import Api, Resource
from flask_sqlalchemy import SQLAlchemy
import os
from flask_bcrypt import Bcrypt
from flask_jwt_extended import JWTManager, create_access_token,
jwt_required, get_jwt_identity

basedir = os.path.abspath(os.path.dirname(__file__))

app = Flask(__name__)
api = Api(app)
bcrypt = Bcrypt(app)

# Database setup
app.config['SQLALCHEMY_DATABASE_URI'] = "sqlite:///" + \
    os.path.join(basedir, "product.db")
```

```
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

app.config['JWT_SECRET_KEY'] = 'supersecretkey'  # Change this in
production


db = SQLAlchemy(app)

jwt = JWTManager(app)


# Product Model




class Product(db.Model):

    __tablename__ = 'products'

    productid = db.Column(db.Integer, primary_key=True)

    name = db.Column(db.String(100), nullable=False)

    price = db.Column(db.Float, nullable=False)

    quantity = db.Column(db.Integer, nullable=False)

    category = db.Column(db.String(50), nullable=False)


    def as_dict(self):

        return {

            "productid": self.productid,

            "name": self.name,

            "price": self.price,

            "quantity": self.quantity,
```

```
            "category": self.category
        }

# User Model


class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(150), nullable=False)
    email = db.Column(db.String(150), unique=True, nullable=False)
    password_hash = db.Column(db.String(256), nullable=False)
    mobile = db.Column(db.String(15), nullable=False)
    role = db.Column(db.String(50), nullable=False, default='user')

    def set_password(self, password):
        self.password_hash = bcrypt.generate_password_hash(
            password)

    def check_password(self, password):
        return bcrypt.check_password_hash(self.password_hash, password)

    def as_dict(self):
        return {
```

```
            "id": self.id,
            "name": self.name,
            "email": self.email,
            "mobile": self.mobile,
            "role": self.role
        }


with app.app_context():
    db.create_all()

# User Registration Resource


class UserRegisterResource(Resource):
    def post(self):
        data = request.get_json()

        name = data.get("name")
        email = data.get("email")
        mobile = data.get("mobile")
        password = data.get("password")
        role = data.get("role", "user")
```

```
        if User.query.filter_by(email=email).first():
            return {"message": "Email already exists."}, 400
        user = User(name=name, email=email,
                    mobile=mobile, role=role)
        user.set_password(password)

        db.session.add(user)
        db.session.commit()
        return {"message": "User registered successfully."}, 201

# User Login Resource


class UserLoginResource(Resource):
    def post(self):
        data = request.get_json()

        username = data.get("username")
        password = data.get("password")

        user = User.query.filter_by(email=username).first()
        if user and user.check_password(password):
```

```
            access_token = create_access_token(
                identity=user.role)

            return {"access_token": access_token}, 200

        return {"message": "Invalid credentials."}, 401

# Product Resource


class ProductResource(Resource):
    @jwt_required()
    def get(self, productid):
        product = db.session.get(Product, productid)

        if product:
            return product.as_dict()
        else:
            return {"message": "Product Not Found!"}, 404

    @jwt_required()
    def post(self):
        data = request.get_json()
```

```
        name = data.get("name")
        price = data.get("price")
        quantity = data.get("quantity")
        category = data.get("category")

        if not all([name, price, quantity, category]):
            return {"message": "All fields (name, price, quantity,
category) are required."}, 400

        new_product = Product(name=name, price=price,
                              quantity=quantity, category=category)
        db.session.add(new_product)
        db.session.commit()
        return {"message": "Product Added Successfully", "product":
new_product.as_dict()}, 201

    @jwt_required()
    def put(self, productid):
        product = db.session.get(Product, productid)
        if product:
            data = request.get_json()
            product.name = data.get("name", product.name)
            product.price = data.get("price", product.price)
```

```
            product.quantity = data.get("quantity", product.quantity)

            product.category = data.get("category", product.category)


            db.session.add(product)

            db.session.commit()

            return {"message": "Product Updated Successfully", "Product":
product.as_dict()}, 200

        else:

            return {"message": "Product Not Found!"}, 404


    @jwt_required()

    def delete(self, productid):

        identity = get_jwt_identity()

        print(identity)

        if identity != "admin":

            return {"message": "Unauthorized: Admins only."}, 403

        product = db.session.get(Product, productid)

        if product:

            db.session.delete(product)

            db.session.commit()

            return {"message": "Product deleted successfully", "product":
product.as_dict()}

        return {"message": "Product Not Found!"}, 404
```

```python
# Products Resource



class ProductsResource(Resource):

    @jwt_required()

    def get(self):

        # Get all products

        allproducts = Product.query.all()


        # Converting them to the list of json

        # products = [ product.as_dict() for product in allproducts]

        products = []

        for product in allproducts:

            products.append(product.as_dict())


        if products:

            return {"products": products}

        else:

            return {"message": "No Products Available!"}



class ProductCategory(Resource):

    @jwt_required()
```

```
    def get(self, category):


        allproducts = Product.query.filter_by(

            category=category).all()


        # Converting it to the JSON

        # category_products = [product.as_dict() for product in
all_category_products]

        products = []

        for product in allproducts:

            products.append(product.as_dict())


        if products:

            return {"products": products}

        else:

            return {"message": "No products found in this category"}, 404




# Routes

api.add_resource(UserRegisterResource, "/register")

api.add_resource(UserLoginResource, "/login")

api.add_resource(ProductsResource, "/products")

api.add_resource(ProductResource, "/products",
"/products/<int:productid>")
```

```
# /products/category/<category> for filtering by category

api.add_resource(ProductCategory, "/products/category/<string:category>")




if __name__ == '__main__':

    app.run(debug=True)
```

To Test the endpoints in Postman:

Note: To perform any operation on the Products first user needs to register with his details and then need to login with valid credentials, to get an access token, with this access token only he can perform any operation on the products.

- To register a user:

  POST: http://127.0.0.1:5000/register

  Payload:

  ```
  {
    "name": "Raj",
    "email": "raj@gmail.com",
    "password": "1234",
    "mobile": 22222
  }
  ```

- To Login the registered user:

  POST: http://127.0.0.1:5000/login

  payload:

  {

  "username": "raj@gmail.com",

  "password": "1234"

  }

Here user will get the Access token (JWT token) in the response body

- To perform any operations on the Product pass the access token to the Headers as of the request

  Authorization: Bearer access_token

Now:

- To add a new product:

  POST: http://127.0.0.1:5000/products

  payload:

  {

  "name": "Notebook",

  "price": 50,

  "quantity": 50,

  "category": "Stationary"

  }

- To Get All products:

  GET: http://127.0.0.1:5000/products

- To Get a Specific Product:

  GET: http://127.0.0.1:5000/products/1

Note: To perform the delete functionality user need to register with the role admin:

Example:

- To register a user:

  POST: http://127.0.0.1:5000/register

  Payload:

```
{
  "name": "Rahul",
  "email": "rahul@gmail.com",
  "password": "1234",
  "mobile": 22222,
 "role": "admin",
}
```