**Question Categorization System-Code**

**Project Group-65**

|  |  |
| --- | --- |
| **Name** | **ID** |
| Aparna Phundir | M23AID006 |
| Bhuvaneswari J | M23AID053 |

**Microservice:**

**User Application**

The User Application provides user registration and login functionalities using Flask and SQLAlchemy. It stores user credentials in a SQLite database.

Database Model

- User: Represents a user in the database.

- `id`: Integer, primary key.

- `username`: String, unique and not nullable.

- `password`: String, not nullable.

Routes

- \*\*GET /\*\*: Returns a welcome message for the User API.

- \*\*POST /register\*\*: Registers a new user. Expects JSON data with `username` and `password`.

- Hashes the password using PBKDF2 before storing it in the database.

- Returns a success message and HTTP status code 201.

- \*\*POST /login\*\*: Logs in an existing user. Expects JSON data with `username` and `password`.

- Checks the provided credentials against the database.

- Returns a success message and HTTP status code 200 on successful login, or a failure message with HTTP status code 401 on failure.

Initialization

- Creates the `users.db` database and the User table when the application starts.

---

**Year Application**

The Year Application allows users to add new years to a SQLite database.

Database Model

- \*\*Year\*\*: Represents a year in the database.

- `id`: Integer, primary key.

- `year\_value`: String (4 characters), not nullable.

Routes

- \*\*GET /\*\*: Returns a welcome message for the Year API.

- \*\*POST /year\*\*: Adds a new year. Expects JSON data with `year\_value`.

- Returns a success message and HTTP status code 201.

Initialization

- Creates the `years.db` database and the Year table when the application starts.

---

**Exam Type Application**

The Exam Type Application manages exam types, subjects, and keywords in a SQLite database.

Database Models

- \*\*ExamType\*\*: Represents an exam type.

- `id`: Integer, primary key.

- `name`: String, not nullable.

- \*\*Subject\*\*: Represents a subject associated with an exam type.

- `id`: Integer, primary key.

- `name`: String, not nullable.

- `exam\_type\_id`: Integer, foreign key referencing `ExamType`.

- \*\*Keyword\*\*: Represents keywords related to subjects.

- `id`: Integer, primary key.

- `value`: String, not nullable.

- `subject\_id`: Integer, foreign key referencing `Subject`.

Routes

- \*\*GET /\*\*: Returns a welcome message for the Exam Type API.

- \*\*POST /exam-type\*\*: Adds a new exam type. Expects JSON data with `name`.

- Returns a success message and HTTP status code 201.

Initialization

- Creates the `exam\_types.db` database and the related tables when the application starts.

---

**Question Application**

The Question Application allows users to add questions to a SQLite database, associated with specific exam years and types.

Database Model

- \*\*Question\*\*: Represents a question.

- `id`: Integer, primary key.

- `text`: String (up to 200 characters), not nullable.

- `exam\_year\_id`: Integer, not nullable.

- `exam\_type\_id`: Integer, not nullable.

Routes

- \*\*GET /\*\*: Returns a welcome message for the Question API.

- \*\*POST /question\*\*: Adds a new question. Expects JSON data with `text`, `exam\_year\_id`, and `exam\_type\_id`.

- Returns a success message and HTTP status code 201.

Initialization

- Creates the `questions.db` database and the Question table when the application starts.

---

**Category Application**

The Category Application categorizes questions based on their text and associated subjects and keywords by fetching data from other microservices.

Routes

- \*\*GET /\*\*: Returns a welcome message for the Categorization API.

- \*\*GET /categorize/<int:question\_id>\*\*: Categorizes a question based on its text.

- Fetches the question text from the Question Service and subjects and keywords from the Exam Types Service.

- Searches for matches between the question text and subjects/keywords.

- Returns the question details and categories if found, or an error message if not.

Error Handling

- Handles exceptions for network requests to other services and returns appropriate error messages.

Initialization

- The application runs on port 5000.

---

**Monolithic:**

This monolithic application manages users, years, exam types, subjects, keywords, and questions. It integrates functionalities from multiple microservices into a single Flask application.

Project Structure

The application consists of the following main components:

- \*\*User Management\*\*: Handles user registration and login.

- \*\*Year Management\*\*: Allows for adding and managing years.

- \*\*Exam Type Management\*\*: Handles exam types, subjects, and keywords.

- \*\*Question Management\*\*: Manages questions and their associated metadata.

- \*\*Categorization\*\*: Categorizes questions based on their content, subjects, and keywords.

Imports

```python

from flask import Flask, request, jsonify

from flask\_sqlalchemy import SQLAlchemy

from werkzeug.security import generate\_password\_hash, check\_password\_hash

import os

import requests

```

- \*\*Flask\*\*: A web framework for Python that enables the creation of web applications.

- \*\*Flask-SQLAlchemy\*\*: An extension for Flask that adds support for SQLAlchemy, a SQL toolkit for Python.

- \*\*Werkzeug\*\*: A library for password hashing and verification.

- \*\*os\*\*: A standard library for interacting with the operating system.

- \*\*requests\*\*: A library for making HTTP requests to external services.

Configuration

```python

# Configure the database URI, ensure absolute path for consistency

db\_path = os.path.join(os.getcwd(), 'database\_name.db')

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///' + db\_path

app.config['SQLALCHEMY\_TRACK\_MODIFICATIONS'] = False

```

- Sets the database URI to use SQLite and specifies the database file path.

Models

1. \*\*User Model\*\*

```python

class User(db.Model):

id = db.Column(db.Integer, primary\_key=True)

username = db.Column(db.String(80), unique=True, nullable=False)

password = db.Column(db.String(120), nullable=False)

```

- Represents a user in the application.

- Fields:

- `id`: Unique identifier for each user.

- `username`: Unique username for login.

- `password`: Hashed password for security.

2. \*\*Year Model\*\*

```python

class Year(db.Model):

id = db.Column(db.Integer, primary\_key=True)

year\_value = db.Column(db.String(4), nullable=False)

```

- Represents a year.

- Fields:

- `id`: Unique identifier for each year.

- `year\_value`: The value of the year (e.g., "2024").

3. \*\*ExamType Model\*\*

```python

class ExamType(db.Model):

id = db.Column(db.Integer, primary\_key=True)

name = db.Column(db.String(80), nullable=False)

```

- Represents an exam type.

- Fields:

- `id`: Unique identifier for each exam type.

- `name`: The name of the exam type (e.g., "SAT").

4. \*\*Subject Model\*\*

```python

class Subject(db.Model):

id = db.Column(db.Integer, primary\_key=True)

name = db.Column(db.String(80), nullable=False)

exam\_type\_id = db.Column(db.Integer, db.ForeignKey('exam\_type.id'))

```

- Represents a subject associated with an exam type.

- Fields:

- `id`: Unique identifier for each subject.

- `name`: The name of the subject.

- `exam\_type\_id`: Foreign key referencing the exam type.

5. \*\*Keyword Model\*\*

```python

class Keyword(db.Model):

id = db.Column(db.Integer, primary\_key=True)

value = db.Column(db.String(80), nullable=False)

subject\_id = db.Column(db.Integer, db.ForeignKey('subject.id'))

```

- Represents a keyword associated with a subject.

- Fields:

- `id`: Unique identifier for each keyword.

- `value`: The keyword value.

- `subject\_id`: Foreign key referencing the subject.

6. \*\*Question Model\*\*

```python

class Question(db.Model):

id = db.Column(db.Integer, primary\_key=True)

text = db.Column(db.String(200), nullable=False)

exam\_year\_id = db.Column(db.Integer, nullable=False)

exam\_type\_id = db.Column(db.Integer, nullable=False)

```

- Represents a question in the application.

- Fields:

- `id`: Unique identifier for each question.

- `text`: The content of the question.

- `exam\_year\_id`: ID of the exam year associated with the question.

- `exam\_type\_id`: ID of the exam type associated with the question.

API Endpoints

1. \*\*User API\*\*

- \*\*Register a new user\*\*:

- \*\*Endpoint\*\*: `/register`

- \*\*Method\*\*: `POST`

- \*\*Request Body\*\*:

```json

{

"username": "new\_user",

"password": "password123"

}

```

- \*\*Response\*\*: `201 Created` on successful registration.

- \*\*Login a user\*\*:

- \*\*Endpoint\*\*: `/login`

- \*\*Method\*\*: `POST`

- \*\*Request Body\*\*:

```json

{

"username": "existing\_user",

"password": "password123"

}

```

- \*\*Response\*\*: `200 OK` on successful login, `401 Unauthorized` on failed login.

2. \*\*Year API\*\*

- \*\*Add a new year\*\*:

- \*\*Endpoint\*\*: `/year`

- \*\*Method\*\*: `POST`

- \*\*Request Body\*\*:

```json

{

"year\_value": "2024"

}

```

- \*\*Response\*\*: `201 Created` on successful year addition.

3. \*\*Exam Type API\*\*

- \*\*Add a new exam type\*\*:

- \*\*Endpoint\*\*: `/exam-type`

- \*\*Method\*\*: `POST`

- \*\*Request Body\*\*:

```json

{

"name": "SAT"

}

```

- \*\*Response\*\*: `201 Created` on successful exam type addition.

4. \*\*Question API\*\*

- \*\*Add a new question\*\*:

- \*\*Endpoint\*\*: `/question`

- \*\*Method\*\*: `POST`

- \*\*Request Body\*\*:

```json

{

"text": "What is the capital of France?",

"exam\_year\_id": 1,

"exam\_type\_id": 1

}

```

- \*\*Response\*\*: `201 Created` on successful question addition.

5. \*\*Categorization API\*\*

- \*\*Categorize a question\*\*:

- \*\*Endpoint\*\*: `/categorize/<int:question\_id>`

- \*\*Method\*\*: `GET`

- \*\*Response\*\*: `200 OK` with categorized information if successful, `404 Not Found` if no categories found.

Database Initialization

The application attempts to create the necessary database tables upon startup:

```python

with app.app\_context():

db.create\_all()

```

**Running the Application**

Each application and service should have Dockerfile and requirements.txt to execute process in docker desktop using docker-compose yaml file.

Execute below command in git bash to run the process in docker.

docker-compose up –build

Use docker ps to check the container port

Add input and check the category value using curl command

$ curl -X POST http://localhost:5001/register -H "Content-Type: application/json" -d '{"username": "newuser11", "password": "password123"}'

{

"message": "User created"

}

$ curl -X POST http://localhost:5001/login -H "Content-Type: application/json" -d '{"username": "newuser11", "password": "password123"}'

{

"message": "Logged in successfully"

}

$ curl -X POST http://localhost:5002/year -H "Content-Type: application/json" -d '{"year\_value": "2024"}'

{

"message": "Year added"

}

$ curl -X POST http://localhost:5003/exam-type -H "Content-Type: application/json" -d '{"name": "UPSC"}'

{

"message": "Exam type created"

}

curl -X POST http://localhost:5003/subject -H "Content-Type: application/json" -d '{"name": "Geography", "exam\_type\_id": 1}'

curl -X POST http://localhost:5003/keyword -H "Content-Type: application/json" -d '{"value": "River", "subject\_id": 1}'

curl <http://localhost:5000/categorize/1>