**Question Categorization System-Design**

**Project Group-65**

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

**About Project:**

Question categorisation system aims to categorise questions based on subject and keyword input.

Topic and keyword wise question categorization helps students to identify important topics which are most frequently asked in exam and prioritise their study accordingly.

In this system design user can perform below actions,

* Register and login
* Add new exam year
* Add new exam type, subject, and keyword.
* Add questions
* Categorise questions based on subject and keyword.

This project is build using python code,SQLite database and RESTful APIs.

Also it is implemented using Microservice and Monolithic architecture.

**Prerequisites**

- Python 3.x

- Flask

- Flask-SQLAlchemy

-requests

- werkzeug

- PyJWT

- SQLite

-Git bash cmd

-Github repository

-Docker desktop

-WSL and virtualization should be enabled for Docker

**Microservice Components:**

**User Management Service**

- Manages user registration and authentication.

- Interacts with a SQLite database (`users.db`).

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)

**Year Management Service**

- Manages academic years.

- Interacts with a SQLite database (`years.db`).

class Year(db.Model):

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

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

**Exam Type Management Service**

- Manages exam types and their related subjects and keywords.

- Interacts with a SQLite database (`exam\_types.db`).

class ExamType(db.Model):

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

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

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'))

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'))

**Question Management Service**

- Manages questions associated with different exam types and years.

- Interacts with a SQLite database (`questions.db`).

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)

**Categorization Service**

- Fetches questions and categorizes them based on subjects and keywords.

- Interacts with the Question Service and Exam Type Service to retrieve necessary data.

The services communicate over HTTP using RESTful APIs. The Categorization Service fetches data from both the Question Service and the Exam Type Service to categorize questions based on their text content.

**Advantages:**

Agility – Promote agile ways of working with small teams that deploy frequently.

Flexible scaling – If a microservice reaches its load capacity, new instances of that service can rapidly be deployed to the accompanying cluster to help relieve pressure. We are now multi-tenanant and stateless with customers spread across multiple instances. Now we can support much larger instance sizes.

Continuous deployment – We now have frequent and faster release cycles. Before we would push out updates once a week and now we can do so about two to three times a day.

Highly maintainable and testable – Teams can experiment with new features and roll back if something doesn’t work. This makes it easier to update code and accelerates time-to-market for new features. Plus, it is easy to isolate and fix faults and bugs in individual services.

Independently deployable – Since microservices are individual units they allow for fast and easy independent deployment of individual features.

Technology flexibility – Microservice architectures allow teams the freedom to select the tools they desire.

High reliability – You can deploy changes for a specific service, without the threat of bringing down the entire application.

**Disadvantages:**

Development sprawl – Microservices add more complexity compared to a monolith architecture, since there are more services in more places created by multiple teams. If development sprawl isn’t properly managed, it results in slower development speed and poor operational performance.

Exponential infrastructure costs – Each new microservice can have its own cost for test suite, deployment playbooks, hosting infrastructure, monitoring tools, and more.

Added organizational overhead – Teams need to add another level of communication and collaboration to coordinate updates and interfaces.

Debugging challenges – Each microservice has its own set of logs, which makes debugging more complicated. Plus, a single business process can run across multiple machines, further complicating debugging.

Lack of standardization – Without a common platform, there can be a proliferation of languages, logging standards, and monitoring.

Lack of clear ownership – As more services are introduced, so are the number of teams running those services. Over time it becomes difficult to know the available services a team can leverage and who to contact for support.

**Monolithic Structure**

Single Codebase: All components (user management, year management, exam type management, question management, and categorization) are integrated into a single application.

Database: The application uses SQLite as the database management system, with separate tables for users, years, exam types, subjects, keywords, and questions.

Routing: Flask's routing capabilities are used to define various endpoints for the application.

**Advantages:**

Easy deployment – One executable file or directory makes deployment easier.

Development – When an application is built with one code base, it is easier to develop.

Performance – In a centralized code base and repository, one API can often perform the same function that numerous APIs perform with microservices.

Simplified testing – Since a monolithic application is a single, centralized unit, end-to-end testing can be performed faster than with a distributed application.   
Easy debugging – With all code located in one place, it’s easier to follow a request and find an issue.

**Disadvantages:**

Slower development speed – A large, monolithic application makes development more complex and slower.

Scalability – You can’t scale individual components.

Reliability – If there’s an error in any module, it could affect the entire application’s availability.

Barrier to technology adoption – Any changes in the framework or language affects the entire application, making changes often expensive and time-consuming.

Lack of flexibility – A monolith is constrained by the technologies already used in the monolith.

Deployment – A small change to a monolithic application requires the redeployment of the entire monolith.

**Security**

Passwords are hashed using `werkzeug.security` to ensure user data is stored securely.

- The system can be enhanced with additional security measures, such as token-based authentication for API endpoints.

**Error Handling**

Each service includes basic error handling, returning appropriate status codes and messages for successful operations and error situations.

**Application Execution steps:**

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>

**Conclusion**

The Question Categorization System provides a structured way to manage users, years, exam types, questions, and their categorization.