**MuseumAPI Documentation**

**Table of Contents**

1. Overview
2. Backend Architecture
   1. Backend Structure
   2. Database Structure
3. Frontend Architecture
4. Deployment
   1. Deployment to Cloud
   2. Containerization
5. Testing
   1. Backend Testing
   2. Frontend Testing
   3. Performance Testing
6. Security and User Management
7. Communication Features
8. Machine Learning
9. Getting Started

**1. Overview**

MuseumAPI is a full stack application designed to efficiently manage a collection of museums with their artists, paintings, and exhibitions. This solution offers a comprehensive method for organizing the artworks of various artists within each museum. It also facilitates the creation of exhibitions designed to highlight the unique talents of all artists.

**2. Backend Architecture**

**Backend**

The backend structure of the application handles server-side operations. This part is developed in C# with .NET 7 and houses a REST API, operable through both Swagger UI and Postman. Implemented functionalities include CRUD operations, filters, and statistical reports.

**Database**

The backend API is connected to a Microsoft SQL Server database. The database is populated with an expansive dataset, with one million records each for artists, paintings, and museums, and ten million records for exhibitions. A Python script generates a SQL file that bulk inserts data into the database using the Faker library. Database migrations are used to manage changes to the database schema.

**3. Frontend Architecture**

The frontend is a Single Page Application (SPA) developed using the React framework in combination with Typescript, Vite, Node.js, and Material UI for user interface components. This component is deployed on Netlify and can be accessed at <https://museumapi-frontend.netlify.app/>.

The frontend design is responsive, adjusting seamlessly to any screen size. It doesn’t have scrollbars and guarantees proper component sizing and layout. Users can navigate through paginated data and personalize the page size via their profile settings.

**4. Deployment**

**Deployment to Cloud**

The entire application is deployed on a Google Cloud Ubuntu-based virtual machine, accessible at <https://museumapi.ddns.net/api>. The virtual machine contains Microsoft SQL Server, .NET 7 SDK, and nginx (configured with a reverse proxy to protect the application server from direct exposure to the internet).

A Kubernetes Cluster has been installed on cloud to scale resources dynamically. The resource scaling process can be visualized through JMeter with the Ultimate Thread Group plugin, showing how the system adapts to accommodate requests from JMeter.

***Containerization***

The application has been dockerized on another Ubuntu-based virtual machine. There are two distinct configurations available: one intended for development and another for production use. Depending on the user's selection, choosing 'Debug' will launch the development version of the application, whereas selecting 'Release' will initiate the production version.

**5. Testing**

**Backend Testing**

The backend is validated through unit tests constructed with the NUnit framework and the Moq library for mocking the data.

**Frontend Testing**

The frontend is subjected to End to End (E2E) testing with Cypress for intercepting the backend requests and mocking the data.

**Performance Testing**

Application performance is evaluated through stress testing with spikes using Apache JMeter. Resulting graphs illustrate how user numbers influence CPU usage and request response times. The test starts with few users: ~2 constant ones and ~10 for the spikes and gradually move up until 100% CPU usage to evaluate how the system handles sudden increases in load.

**6. Security and User Management**

The system implements a logging mechanism using JWT tokens. Passwords are securely hashed using the SHA-256 algorithm. The system defines distinct roles for users: anonymous users (no login) who can only read content, regular logged-in users who can add and edit their entries, moderator users who can add entries and edit all records, and admin users who can add entries, edit all records, alter user roles (including other admins), access the admin panel for bulk data operations, and modify the page preferences for all users.

**7. Communication Features**

The application features a chat page implemented with web sockets, allowing all visitors to see messages from others. Messages persist in the system, enabling a message history. Visitors can also select a nickname upon entering the chat.

**8. Machine Learning**

The application houses a Machine Learning model that estimates the price of a painting based on its age and height. This model, developed using the Fast Forest Regression algorithm, was trained for 600 seconds (10 minutes) on a custom dataset with one million rows. The machine learning model is incorporated into the application using the ML.NET framework.

**9. Getting Started**

To run the app, follow these steps:

1. Installation: Ensure Visual Studio 2022, .NET 7 core, and Entity Framework are installed on your machine.
2. Clone the Repository: Use the command *git clone https://github.com/UBB-SDI-23/lab-5x-923-Iliesi-Catrinel.git* to clone the repository.
3. Switch to the Development Branch: Navigate to the cloned repository and switch to the development branch using the command *git checkout development*.
4. Navigate to *DevOps/Docker* folder.
5. Build and Run: Execute *docker-compose up --build* to build and run the application.