**.NET Core Microservices API Project**

**Project Overview**

This project is a .NET Core-based API system that includes four individual microservices. Each service is responsible for a specific domain functionality and is designed to work in a highly scalable, containerized environment on Azure. The system is designed with API endpoints that allow communication between various components of a healthcare system.

Migration is already been created for each project, you will need to change the server in the appsettings to your Db local server so you can create and test those API’s locally.

The microservices are structured as follows:

1. **MCG.Author.API** - Manages user authentication, registration, and role assignment.
2. **MCG.Profile.API** - Manages patient records, including CRUD operations and search by patient names.
3. **MCG.Attachments.API** - Handles patient attachments, allowing users to create, update, delete, and retrieve attachments for specific patients.
4. **MCG.MedicalHistory.API** - Manages medical history records for patients, including creation, updates, and deletions.

**System Architecture**

* **Microservices**: Each API project represents a separate microservice.
* **Database**: SQL Server is used as the database for all the services, with each service having its own schema and database tables.
* **API Gateway**: An API Gateway (likely using Azure Management API) is used to secure and control a single entrance to all the microservices.
* **Authentication**: JWT tokens are used to authenticate API calls and ensure security.
* **Containerization**: Docker or Kubernetes (K8) is used to containerize the services for easier scalability and deployment.

**Assumptions**

* The system assumes that the microservices will be deployed on Azure and managed using Azure API Management and Azure DevOps.
* Each microservice is assumed to be independently deployable and maintainable.
* The system assumes that all API communication will use HTTPS for security.
* The API Gateway will handle routing, authentication, and possibly rate limiting for all incoming API requests.
* Each service has its own SQL Server database instance for scalability and to avoid cross-service data access.
* Frontend applications (e.g., Angular or React) will communicate with the backend APIs via the API Gateway.
* The APIs will return data in JSON format and accept input in JSON format for consistency.

**Project Structure**

**1. MCG.Author.API**

* **Description**: Handles user authentication, registration, and role management.
* **API Endpoints**:
  + POST /register: Registers a new user.
  + POST /login: Authenticates a user and returns a JWT token.
  + POST /assign-role: Assigns a role to a user.
* **Authentication**: Uses .NET Core Membership Provider for user authentication.
* **JWT Token**: The API generates and validates JWT tokens for secure communication across APIs.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

**2. MCG.Profile.API**

* **Description**: Manages patient profiles.
* **API Endpoints**:
  + POST /create: Creates a new patient record.
  + DELETE /delete/{patientId}: Deletes a patient record.
  + PUT /update: Updates an existing patient record.
  + GET /search: Searches patients by last name or first name.
  + GET /{patientId}: Retrieves a specific patient profile by ID.
* **Database**: SQL Server database for storing patient data.

A white background with colorful lines

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

**3. MCG.Attachments.API**

* **Description**: Manages patient attachments (e.g., files or documents).
* **API Endpoints**:
  + GET /attachments: Retrieves all attachments for a patient.
  + GET /attachment/{attachmentId}: Retrieves a specific attachment.
  + POST /attachment: Creates a new attachment for a patient.
  + PUT /attachment/{attachmentId}: Updates an existing attachment.
  + DELETE /attachment/{attachmentId}: Deletes an attachment.
* **Database**: SQL Server database for storing attachment metadata.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

**4. MCG.MedicalHistory.API**

* **Description**: Manages the medical history of patients.
* **API Endpoints**:
  + POST /medical-history: Creates a new medical history entry for a patient.
  + PUT /medical-history/{historyId}: Updates an existing medical history entry.
  + DELETE /medical-history/{historyId}: Deletes a medical history entry.
  + GET /medical-history/{patientId}: Retrieves all medical history records for a patient.
* **Database**: SQL Server database for storing medical history data.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

**Limitations**

1. **Scalability**: While the project is designed to scale horizontally using Docker/Kubernetes, the overall scalability will depend on the infrastructure in place on Azure.
2. **Service Independence**: Each service is designed to operate independently. However, the system may experience issues if multiple services attempt to access the same data source simultaneously.
3. **Database Load**: Since each microservice uses its own SQL Server database, there may be challenges related to maintaining consistency across databases when data is updated across services.
4. **Limited Role Management**: Role-based access control (RBAC) is basic. Advanced features such as fine-grained access permissions are not implemented in this version.
5. **Authentication Flow**: Authentication is only based on JWT tokens; no support for OAuth or other complex authentication mechanisms in this version.
6. **Frontend Not Included**: The frontend (Angular/React) is not part of this repository but will interact with these APIs through the API Gateway.
7. **API Gateway Dependency**: The system depends on an API Gateway (Azure Management API) to route requests and handle authentication, so any failure in the gateway could cause an outage in the whole system.

**Future Improvements**

* Implement caching for frequently accessed data (e.g., patient profiles or medical history records) to improve performance.
* Enhance role-based access control (RBAC) with more granular permission management.
* Integrate a more advanced user authentication mechanism (e.g., OAuth).
* Add API versioning to support backward compatibility for future changes.
* Implement rate limiting and API throttling at the API Gateway to improve system performance and prevent abuse.
* Consider implementing logging and monitoring (e.g., with Azure Monitor) to track system health and performance.
* Add comprehensive logging and error handling across all microservices.

**Testing**

* Unit and integration tests are implemented for each microservice.
* A separate .NET Core project is created to handle user-defined testing and mock services.
* You can execute tests locally or in a CI/CD pipeline using Azure DevOps.

**Deployment**

* **Containers**: Docker/Kubernetes is used to containerize the services for easy deployment and scalability.
* **Azure**: The project is designed to be deployed on Azure using Azure services such as Azure App Services and Azure SQL Database.

Note: this current project a very small scale project that can grow quickly with more requirements and new demands. There are many tables need to be created in each DB and much more Micro to be added. I am looking forward to be on your team and take on this new project.

Thank you

Sincerely,

Georges Abdo

Ps: trying to set up one project in AZ for you to access , I will let you know when its done- updated 3/14/2025

A screenshot of a computer

AI-generated content may be incorrect.