**XML to JSON Processing System**

**Overview**

This system is designed to allow clients to upload XML files, which are then processed in the backend, converted to JSON, and stored in a remote microservice. The system is built using two separate microservices: FileUploadServiceAPI and ProcessedFilesServiceAPI.

**Components:**

* **FileUploadServiceAPI**: Responsible for uploading and converting XML files.
* **ProcessedFilesServiceAPI**: Responsible for storing and managing the processed JSON files.
* **SharedLibrary**: A library containing the shared model ProcessedFile.

**Requirements**

**System Requirements:**

* .NET 7.0 SDK
* SQL Server (or compatible database)
* Visual Studio 2022 (or any IDE supporting .NET)

**Required Packages:**

* Microsoft.EntityFrameworkCore
* Microsoft.EntityFrameworkCore.SqlServer
* Newtonsoft.Json
* Swashbuckle.AspNetCore (for Swagger)

**Installation and Configuration**

**1. Clone the Repository**

**2. Configure the Database**

Update the appsettings.json in both FileUploadServiceAPI and ProcessedFilesServiceAPI with the correct connection string to your SQL Server.

*{*

*"ConnectionStrings": {*

*"DefaultConnection": "Server=YOUR\_SERVER\_NAME;Database=ProcessedFilesDB;Trusted\_Connection=True;MultipleActiveResultSets=true"*

*}*

*}*

**3. Run Migrations and Setup Database**

In each project, run the following commands to create the database:

*dotnet run --project FileUploadServiceAPI*

*dotnet run --project ProcessedFilesServiceAPI*

**4. Running the Microservices**

You can start each microservice using Visual Studio or by running:

*dotnet run --project FileUploadServiceAPI*

*dotnet run --project ProcessedFilesServiceAPI*

**5. Swagger**

Each microservice has integrated Swagger UI, accessible at:

* **FileUploadServiceAPI**: https://localhost:{PORT}/swagger
* **ProcessedFilesServiceAPI**: https://localhost:{PORT}/swagger

**API Endpoints**

**FileUploadServiceAPI**

**1. Upload XML File**

* **URL:** POST /api/FileUpload/upload
* **Description:** Uploads an XML file, converts it to JSON, and sends the JSON to ProcessedFilesServiceAPI.
* **Parameters:**
  + file (form-data): XML file to be uploaded.
* **Sample Response:**

{

"message": "File processed and sent successfully"

}

**ProcessedFilesServiceAPI**

**1. Store Processed File**

* **URL:** POST /api/ProcessedFiles
* **Description:** Stores a processed JSON file in the database.
* **Request Body:**

{

"FileName": "example.xml",

"FileContent": "{\"root\": {\"element\": \"value\"}}"

}

* **Sample Response:**

{

"message": "File stored successfully",

"fileId": 1

}

**2. Get All Processed Files**

* **URL:** GET /api/ProcessedFiles
* **Description:** Returns a list of all processed files.

**3. Get Processed File by ID**

* **URL:** GET /api/ProcessedFiles/{id}
* **Description:** Returns information about a specific processed file.

**4. Download Processed File**

* **URL:** GET /api/ProcessedFiles/{id}/download
* **Description:** Downloads the JSON file for a specific processed file.

**5. Delete Processed File**

* **URL:** DELETE /api/ProcessedFiles/{id}
* **Description:** Deletes a specific processed file.

**Error Handling**

**General Error Handling:**

* Each microservice includes try-catch blocks to handle errors related to:
  + Invalid input data.
  + File processing errors.
  + Communication errors between microservices.
  + Database access errors.

**Possible Errors:**

* **400 Bad Request:** Invalid input data or missing file.
* **500 Internal Server Error:** Unexpected errors during processing or communication.

**Testing**

**Overview**

This section provides details on the testing strategy implemented for the XML to JSON Processing System. The tests ensure that both microservices, FileUploadServiceAPI and ProcessedFilesServiceAPI, work correctly, individually and together, to handle XML file uploads, conversion to JSON, and storage operations efficiently.

**Unit Tests**

**Purpose**

Unit tests have been implemented to validate the correctness of individual components within each microservice. These tests focus on ensuring that methods and classes perform as expected in isolation, without external dependencies.

**Implementation**

**FileUploadServiceAPI:**

Tests have been written to verify the correct handling and conversion of XML files to JSON format.

Error handling tests check that invalid or malformed XML files are handled gracefully with appropriate error messages.

**ProcessedFilesServiceAPI:**

Tests validate the CRUD operations (Create, Read, Update, Delete) on the processed JSON files within the database.

Tests ensure that data is correctly stored, retrieved, updated, and deleted according to the API specifications.