# **MuleSoft Final Project Functional Requirement specification document (FRS)**

**Implementation project of craft note management api**



**Developed by: Group B**

**Date:5-23-2024**

# **Table of contents**

# **Introduction**……………………………………………………………………………………...3

### 1.1. Project Summary……………………………………………………………………….……3

### 1.2. System OverviewFunctional Requirements Specification (FRS)………………………....3

1.3 Mule flows………………………………………………………………………………....3

## 2. High-Level Design (HLD)……………………………………………………………………..5

### 2.1. System Components……………………………………………………………………....5

2.2 Data flow…………………………………………………………………………………..5

## 3. Low-Level Design (LLD)……………………………………………………………………...6

### 3.1. Functional Requirements………………………………………………………………...6

#### 3.1.1. Note Management ……………………………………………………………………..6

3.1.1.1. GET /api/notes……………………………………………………………………….6

3.1.1.2. GET /api/notes/{noteId}……………………………………………………………..7

3.1.1.3. POST /api/notes……………………………………………………………………...7

3.1.1.4. PUT /api/notes/{noteId}……………………………………………………………..8

3.1.1.5. DELETE /api/notes/{noteId}………………………………………………………...8

### 3.2. Database Specifications………………………………………………………………….…..9

### 3.3. Security and Best Practices…………………………………………………………….….....9

## 3.4. Deployment and Integration………………………………………………………………...10

### 3.5. Documentation and Support………………………………………………………………....10

### 4. Outro………………………………………………………………………………………...…10

## **1. Introduction**

### **1.1. Project Summary**

This FRS document provides a comprehensive overview of the MuleSoft API project. The goal is to develop a robust and secure API that manages notes within a database and integrates with an external Zip Code API to fetch zip code details.

### **1.2. System Overview**

The API will provide the following functionalities:

Note Management:

* Listing all notes stored in the database
* Retrieving a single note by its ID
* Creating new notes
* Updating existing notes
* Deleting notes

Zip Code Integration (Optional):

* Fetching details from an external Zip Code API based on a provided zip code

### **1.3. Mule Flows**

Separate Mule flows will be implemented to handle each API endpoint:

* GET /api/notes: Retrieve all notes from the database, ensuring the updatedAt field is populated.
* GET /api/notes/{noteId}: Retrieve a specific note by its ID, validating the updatedAt field.
* POST /api/notes: Validate the request body against a defined JSON schema, insert a new note into the database, and set the updatedAt field.
* PUT /api/notes/{noteId}: Check if a note exists, update its content and updatedAt field, or insert a new note if it doesn't exist.
* DELETE /api/notes/{noteId}: Delete the note associated with the provided ID.
* Optional Zip Code Integration: Use the zip URI parameter to fetch details from an external Zip Code API.

## **2. High-Level -Design (HLD)**

### **2.1. System Components**

* API Gateway: Exposes the API endpoints to external consumers.
* RAML Definition: Defines the API contract, including endpoints, request/response structures, and validation rules.
* Flow Orchestrator: Routes requests to appropriate Mule flows based on the target endpoint.
* Database Connector: Interacts with the database to perform CRUD operations on notes.
* External Zip Code API Connector (Optional): Fetches details from the external Zip Code API based on the provided zip code.
* Error Handling: Manages exceptions gracefully and provides informative error messages.

### **2.2. Data Flow**

**GET /api/notes:**

1. API Gateway receives the request.
2. Flow Orchestrator routes the request to the dedicated flow.
3. Database Connector retrieves all notes, ensuring the updatedAt field is not null.
4. The flow transforms the retrieved data into the desired JSON response format.
5. The response is sent back through the API Gateway.

**GET /api/notes/{noteId}:**

1. API Gateway receives the request with the noteId parameter.
2. Flow Orchestrator routes the request to the dedicated flow.
3. Database Connector retrieves the specific note using the provided noteId.
4. The flow validates if the retrieved note has a populated updatedAt field.
5. If the note exists and is valid, it's transformed into JSON format for the response.
6. If the note doesn't exist or is invalid, an appropriate error response is sent back.

**POST /api/notes:**

1. API Gateway receives the request with the note data in the request body.
2. Flow Orchestrator routes the request to the dedicated flow.
3. The flow validates the request body against the defined JSON schema for note data.
4. If valid, the flow uses the Database Connector to insert the new note.
5. The newly created note is retrieved and transformed into JSON format for the response.
6. If the request is invalid, an appropriate error response is sent back.

**PUT /api/notes/{noteId}:**

1. API Gateway receives the request with the noteId parameter and optional update data in the request body.
2. Flow Orchestrator routes the request to the dedicated flow.
3. Database Connector checks if a note exists with the provided noteId.
4. If the note exists, the flow updates the note content and sets updatedAt to the current timestamp.
5. If the note doesn't exist, the flow inserts a new note with the provided data, setting updatedAt to the current timestamp.
6. The updated or newly created note is retrieved and transformed into JSON format for the response.
7. If an error occurs during update or insertion, an appropriate error response is sent back.

**DELETE /api/notes/{noteId}:**

1. API Gateway receives the request with the noteId parameter.
2. Flow Orchestrator routes the request to the dedicated flow.
3. Database Connector deletes the note with the provided noteId.
4. If the deletion is successful, a 204 No Content response is sent back.
5. If the note doesn't exist or an error occurs, an appropriate error response is sent back.

## **3. Low-Level Design (LLD)**

### **3.1. Functional Requirements**

#### **3.1.1. Note Management**

**3.1.1.1. GET /api/notes:**

Function: List all notes stored in the database.

Success Response:

* Status Code: 200 OK
* Response Body: An array of objects representing notes, each containing the following fields:
  + id: (integer) Unique identifier for the note (primary key)
  + title: (string) Title of the note
  + content: (string) Content of the note
  + createdAt: (datetime) Timestamp of note creation
  + updatedAt: (datetime) Timestamp of the last update to the note (must not be null)

Error Responses:

* Status Code: 400 Bad Request - Invalid request format
* Status Code: 500 Internal Server Error - Database access error

**3.1.1.2. GET /api/notes/{noteId}:**

Function: Retrieve a single note by its ID.

Request Parameter:

* {noteId}: (integer) Unique identifier for the note

Success Response:

* Status Code: 200 OK
* Response Body: An object representing the retrieved note with the same fields as in the GET /api/notes response

Error Responses:

* Status Code: 404 Not Found - No note found for the provided ID
* Status Code: 400 Bad Request - Invalid note ID format
* Status Code: 500 Internal Server Error - Database access error

**3.1.1.3. POST /api/notes:**

Function: Create a new note.

Request Body:

A JSON object containing the following fields (all required):

* id: (integer) Unique identifier for the note
* title: (string) Title of the note
* content: (string) Content of the note
* createdAt: (datetime) Timestamp of note creation (can be omitted, system will assign a default timestamp)
* updatedAt: (datetime) Timestamp of the last update (set to the current timestamp by the system)

Success Response:

* Status Code: 201 Created
* Response Body: An object representing the newly created note with the same fields as in the request body

Error Responses:

* Status Code: 400 Bad Request - Invalid request body format or missing required fields
* Status Code: 500 Internal Server Error - Database access error

**3.1.1.4. PUT /api/notes/{noteId}:**

Function: Update an existing note or create it if it doesn't exist.

Request Parameter:

* {noteId}: (integer) Unique identifier for the note

Request Body (Optional):

A JSON object containing any of the following fields to update:

* title: (string) Title of the note
* content: (string) Content of the note

Success Response:

* Status Code: 200 OK
* Response Body: An object representing the updated note with the same fields as in the GET /api/notes response, reflecting the changes

Error Responses:

* Status Code: 400 Bad Request - Invalid request body format
* Status Code: 404 Not Found - No note found for the provided ID
* Status Code: 500 Internal Server Error - Database access error

**3.1.1.5. DELETE /api/notes/{noteId}:**

Function: Delete a note by its ID.

Request Parameter:

* {noteId}: (integer) Unique identifier for the note

Success Response:

* Status Code: 204 No Content

Error Responses:

* Status Code: 404 Not Found - No note found for the provided ID
* Status Code: 500 Internal Server Error - Database access error
* API

### **3.2. Database Specifications**

The notes will be stored in a database table with the following schema:

Table Name: notes-{team\_name}

Columns

* id: INT, PRIMARY KEY, AUTO\_INCREMENT (starts at 101)
* title: VARCHAR
* content: VARCHAR
* createdAt: TIMESTAMP
* updatedAt: TIMESTAMP

(Note: The specific database configuration details (host, schema, username, password) will be provided separately for the development, testing, and production environments. Sensitive data like passwords will be encrypted and stored securely.)

### **3.3. Security and Best Practices**

* Implement data encryption methods, authentication, and authorization mechanisms (e.g., client credentials flow) to restrict API access.
* Store sensitive data like API keys securely using encrypted properties files.
* Configure access control lists (ACLs) within the database to limit data access.
* Avoid logging sensitive data in logs or runtime outputs.
* Follow MuleSoft's best practices for connector configuration and externalization.
* Implement environment-specific properties files (Dev, QA, Prod) to manage configuration details.
* Aim for at least 75% MUnit test coverage to ensure code quality.

### **3.4. Deployment and Integration**

* Set up separate environments for development, testing, and production.
* Ensure continuous integration and continuous deployment (CI/CD) practices are in place.
* Utilize MuleSoft's Anypoint Platform to manage the lifecycle of the API.
* Publish comprehensive API documentation, including examples and error codes.

### **3.5. Documentation and Support**

* Prepare and maintain a RAML file that includes detailed documentation of the API.

.

## **4. Outro**

This Functional Requirements Specification (FRS) document serves as a comprehensive guide for the development of the MuleSoft API project. It outlines the functional requirements, system design, and implementation details necessary to create an API that manages notes within a database and integrates with an external Zip Code API.

It's important to note that while this FRS document aims to provide a solid foundation for the project, it should be treated as a living document. As the development process progresses, and new insights or requirements emerge, the FRS may be updated and refined accordingly. This flexibility allows for the incorporation of feedback, best practices, and any necessary adjustments to ensure the final product meets the desired goals and specifications.

Our team should review and discuss the contents of this document thoroughly, addressing any placeholders or areas that require further clarification or decision-making. Additionally, it's recommended to involve relevant stakeholders and subject matter experts to ensure the FRS aligns with the project's objectives and industry best practices.

By adhering to the guidelines outlined in this FRS, the development team can ensure a consistent, scalable, and maintainable API implementation, while also fostering collaboration, communication, and adherence to industry standards.