Ukrainian Catholic University

Faculty of Applied Sciences

Computer Science Program

Notes API project

Software Architecture Course

Authors:
Petro Mozil, Artur Pelcharskyi, Iryna Kokhan, Anna Yaremko

11 May 2025

Abstract

This report presents a detailed analysis of the "Notes API" project - a document management system built using microservice architecture. The system leverages such technologies as FastAPI, PostgreSQL, and Consul for service discovery to create an efficient solution for document storage and management. This analysis explores the architectural design, data model, implementation details, and potential extensions of the system.

1 Introduction

Modern document management systems must be reliable, scalable, and able to handle multiple requests simultaneously without data loss. The Notes API project is an example of such a system, built on the principles of microservices architecture and using asynchronous processing. This paper examines the structure, components, and functionality of the project in detail.

The Notes API system allows users to create, edit, delete, and organize documents, attach files to them, and establish links between different documents. It is designed with a focus on high performance, flexibility, and scalability, making it suitable for both personal note-taking and business needs.

2 System Architecture

2.1 General overview

The system is built on a modular microservices architecture, consisting of two independent services: public-api and notes-api. Each service is designed with a clear separation of concerns, utilizes its own dedicated database, and communicates over HTTP. Consul is employed for service discovery and health monitoring, allowing dynamic address resolution and removing the need for hardcoded service endpoints.

The notes-api microservice is responsible for managing internal data structures such as documents, attachments, topics, tags, and document links. It exposes unauthenticated RESTful endpoints and directly interacts with a PostgreSQL database via SQLAlchemy using async support through the asyncpg driver. Upon startup, the service registers itself with Consul, making it discoverable by other internal services.

The public-api service acts as the system's public-facing gateway. It handles user registration, authentication (using JSON Web Tokens), and proxies secure requests to notes-api over HTTP. All user-related data, including login credentials, is stored in MongoDB. Asynchronous communication with the database is implemented using Motor, an async MongoDB client. Before forwarding any request that targets the notes-api, the service queries Consul to dynamically resolve its current address, ensuring reliable service-to-service communication in dynamic environments.

Consul plays a central role as the service registry. It allows each service to register and deregister itself dynamically and provides real-time health monitoring and address discovery, facilitating smooth communication across a distributed system.

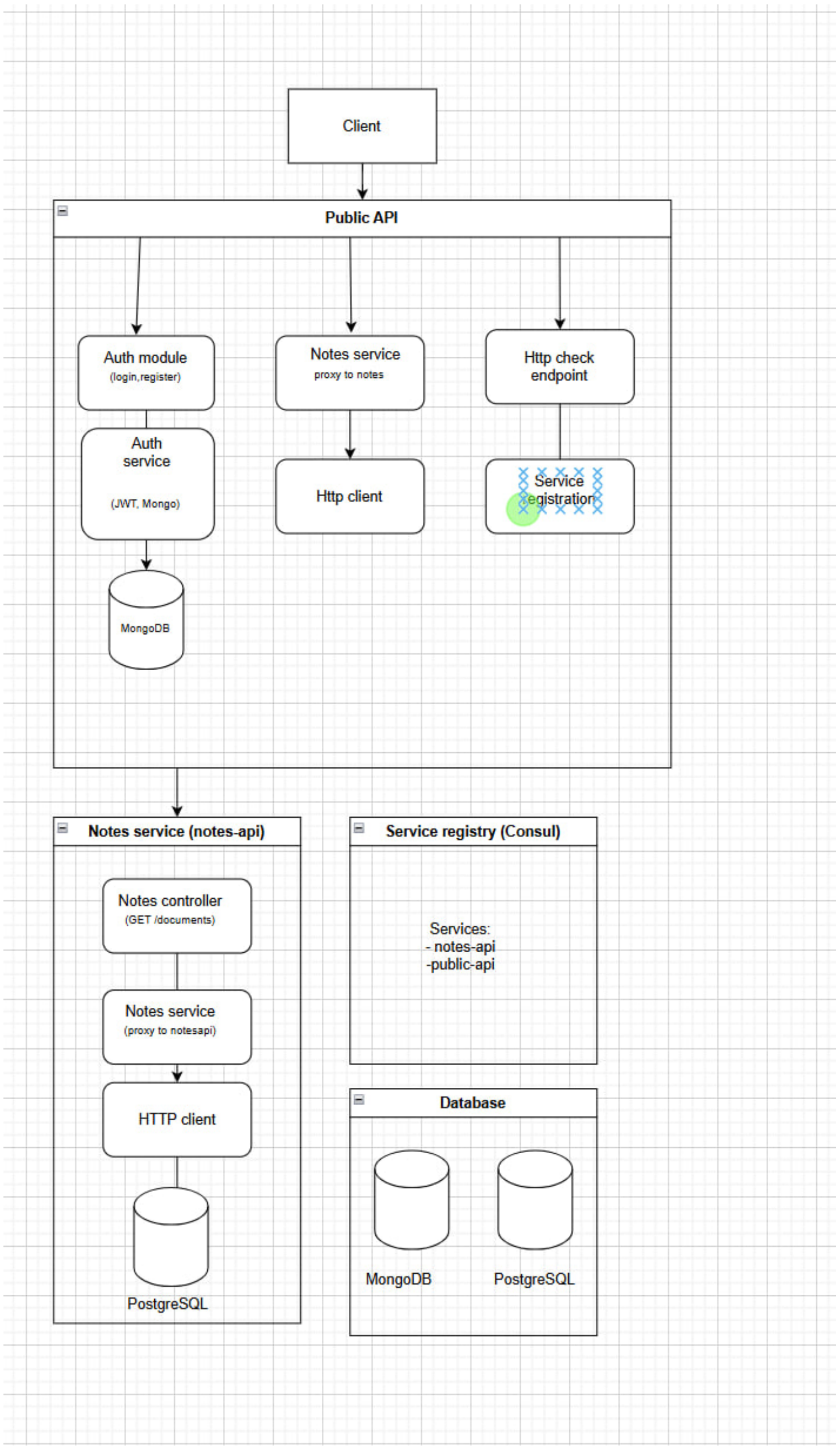


Figure 1: System architecture overview

2.2 Advantages

The architecture provides several significant benefits:

1. **Separation of responsibilities**: There is a clear split between internal data pro-

cessing (notes-api) and external user interaction (public-api), enhancing system maintainability and security.

2. **Microservice independence**: Each service is fully decoupled and independently deployable. By using isolated data stores (PostgreSQL and MongoDB), the system is more flexible, maintainable, and ready for future migrations or scaling needs.

3. **Dynamic service discovery**: Integration with Consul eliminates the need for hardcoded addresses and supports dynamic infrastructure, particularly useful in containerized or cloud-based environments.

4. **Asynchronous performance**: Both APIs use FastAPI's asynchronous capabilities, significantly improving performance under high load conditions by reducing I/O blocking.

5. **JWT-based security**: User authentication is stateless and scalable through the use of JSON Web Tokens, supporting secure and efficient access control.

6. **Clean project organization**: The domain-based folder structure (e.g., /models/, /schemas/, /services/, /api/) promotes clarity, maintainability, and faster onboarding of new developers.

2.3 Possible improvements

While the current architecture is robust and functional, several enhancements can further improve performance, security, and observability:

1. **Rate limiting and abuse prevention**: Implementing rate limiting (e.g., via Redis or an API Gateway) can protect the system from brute-force attacks or excessive load caused by misuse.

2. **Observability and monitoring**: Adding structured logging, distributed tracing (e.g., OpenTelemetry), and real-time metrics (e.g., Prometheus + Grafana) would significantly improve debugging, performance analysis, and operational insight.

3. **Connection robustness**: Adding connection pooling, retry policies, and exponential backoff in HTTP clients (httpx.AsyncClient) and database layers would improve resilience to temporary failures and latency spikes.

4. **Automated testing and CI/CD pipelines**: Introducing automated integration tests between services and deploying CI/CD pipelines would help catch regressions early and ensure higher deployment reliability

3 Databases overview

3.0.1 Decomposition strategy

As a design for our database architecture, we used a deliberate entity decomposition strategy (vertical partitioning) to separate different aspects of documents and attachments into distinct tables. We implemented a clear separation between document metadata and content. Metadata is stored in the `documents` table, while the full document content is present in the `document_content` table. This allows the system to perform operations such as `listing` and `searching` documents without loading large content blocks. This helps us to improve the performance and avoid unnecessary additional loads. The same thing we have done with attachments, i.e., metadata about attachments and their binary data are stored separately (`attachments` and `attachment_content`). This is such an isolation to make querying efficient and reduce memory usage. Also, one more interesting thing is that we have relationships between documents externalized into a dedicated `document_relationships` table. Thus, we do not have a typical embedding relationship directly within document records. This is done for the data redundancy, to be able to support flexible relationship types and simplify bidirectional navigation.

3.1 Database architecture advantages

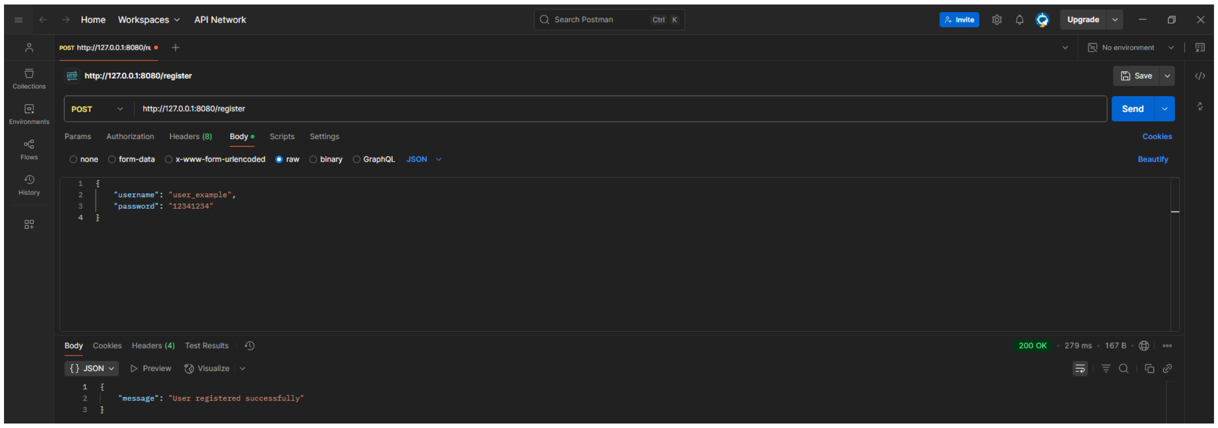
1. **efficient retrieval** - as it was mentioned before, such partitioning of the metadata and content helps us to avoid overloading (when there is no need to get large blocks of contents);
2. **optimized storage** - using binary format of content separately from a textual one is helpful during retrieval strategies appropriate to each data type. This is efficient during handling large files;
3. **automatic timestamp** - this step makes the application logic easier because there is less code needed in the application layer;
4. **flexible categorization/searchability** - there is a topic tagging system that makes easier relationship between documents and topics. And also it enables better document organization and filtering;
5. **normalization** - information stored in only one place and changes to one aspect don't affect others;
6. **ACID (atomicity, consistency, isolation, durability) principles** - this ensures consistency during concurrent operations and during the fail can recover.

3.2 Possible improvements

Of course, there can be some improvements, but for our point of view, they are more related to the security aspects. Such as encryption of some data - in order to make the data private. Or for such cases also dynamic data masking can be useful. This can be done in order to hide certain data for users who shouldn't see everything, by creating special views of the data.

4 Examples

Some basic(but not all) commands. Additional command examples can be found in the README.md file on our GitHub repository.



POST http://127.0.0.1:8080/register

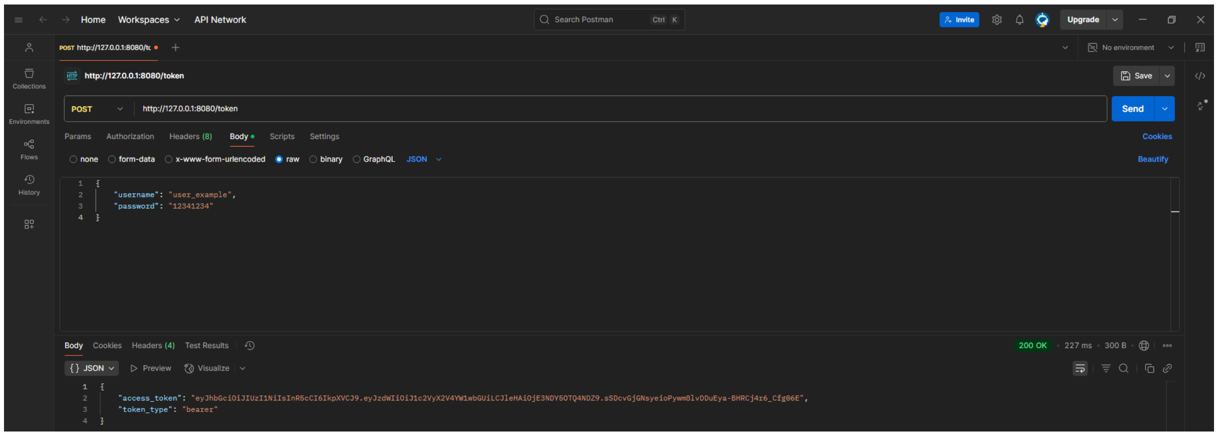
Body (raw, JSON):

```
{
    "username": "user_example",
    "password": "12341234"
}
```

200 OK · 279 ms · 167 B

```
{
    "message": "User registered successfully"
}
```

Figure 2: New user registration



POST http://127.0.0.1:8080/token

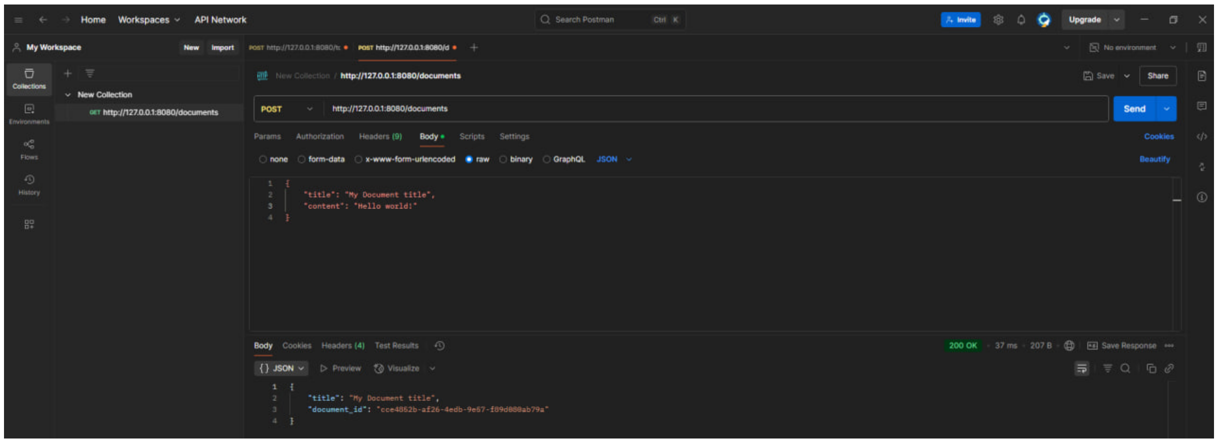
Body (raw, JSON):

```
{
    "username": "user_example",
    "password": "12341234"
}
```

200 OK · 227 ms · 300 B

```
{
    "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJ1c2VyX2V4YW1wbGUiLCJleHAiOjE3NDY5OTQ4NDZ9.sSDcvGjGNsyeioPywm8lvDDuEya-BHRCj4r6_Cfg06E",
    "token_type": "bearer"
}
```

Figure 3: Token get



POST http://127.0.0.1:8080/documents

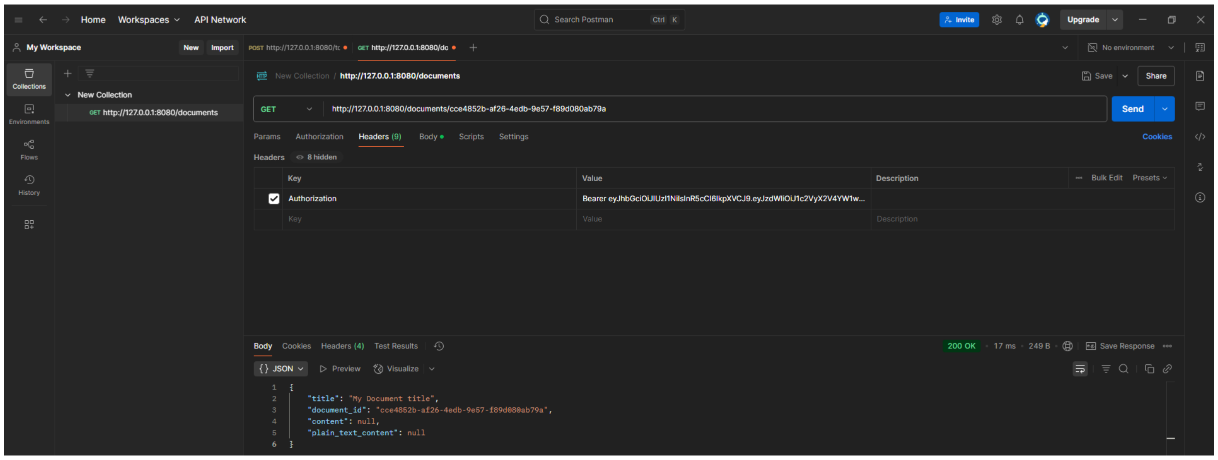
Body: raw JSON

```
{
    "title": "My Document title",
    "content": "Hello world!"
}
```

200 OK · 37 ms · 207 B

```
{
    "title": "My Document title",
    "document_id": "cce4852b-af26-4edb-9e57-f89d080ab79a"
}
```

Figure 4: Document add



GET http://127.0.0.1:8080/documents/cce4852b-af26-4edb-9e57-f89d080ab79a

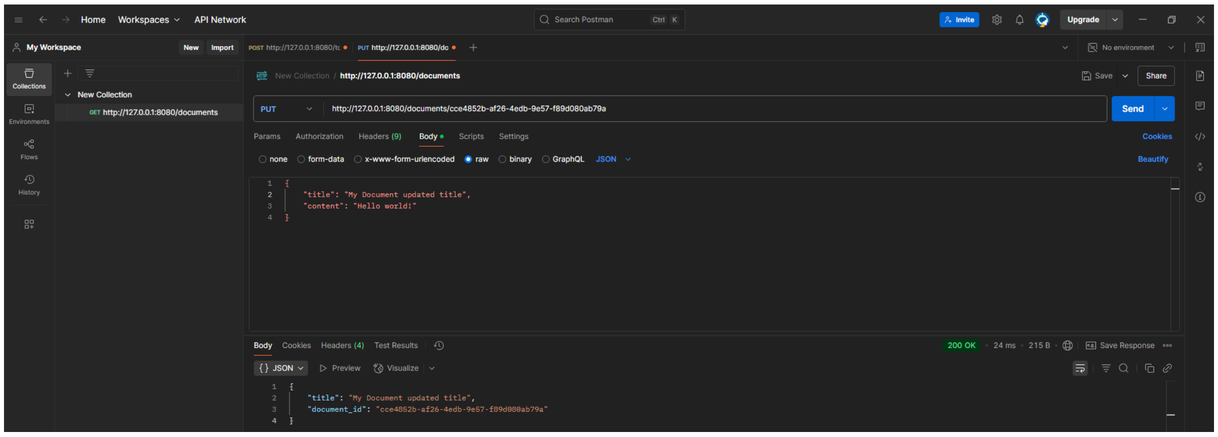
Headers (9)

Key	Value	Description
Authorization	Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJ1c2VyX2V4YW1w...	

200 OK · 17 ms · 249 B

```
{
    "title": "My Document title",
    "document_id": "cce4852b-af26-4edb-9e57-f89d080ab79a",
    "content": null,
    "plain_text_content": null
}
```

Figure 5: Document get



PUT http://127.0.0.1:8080/documents/cce4852b-af26-4edb-9e57-f89d080ab79a

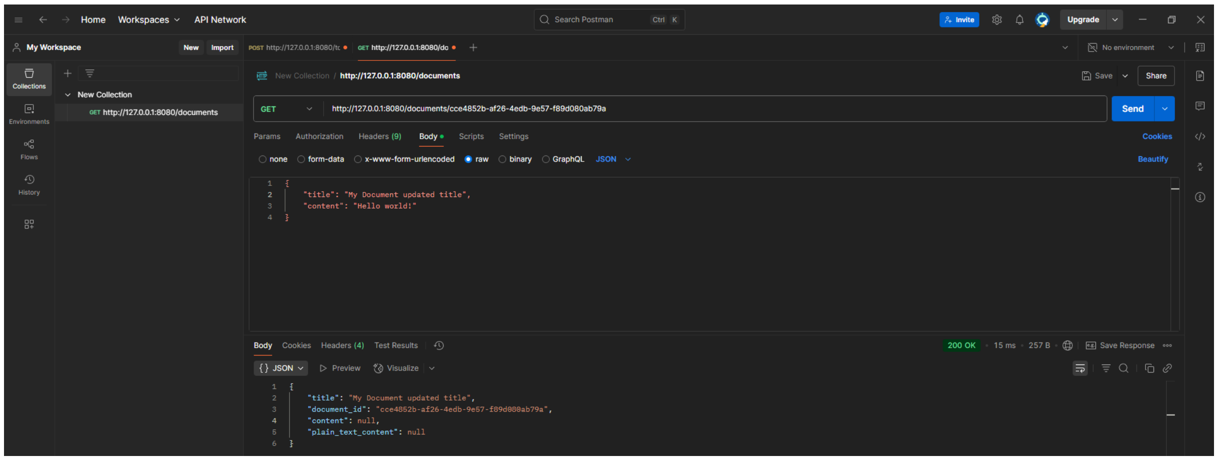
Body: raw JSON

```
{
    "title": "My Document updated title",
    "content": "Hello world!"
}
```

200 OK · 24 ms · 215 B

```
{
    "title": "My Document updated title",
    "document_id": "cce4852b-af26-4edb-9e57-f89d080ab79a"
}
```

Figure 6: Document update



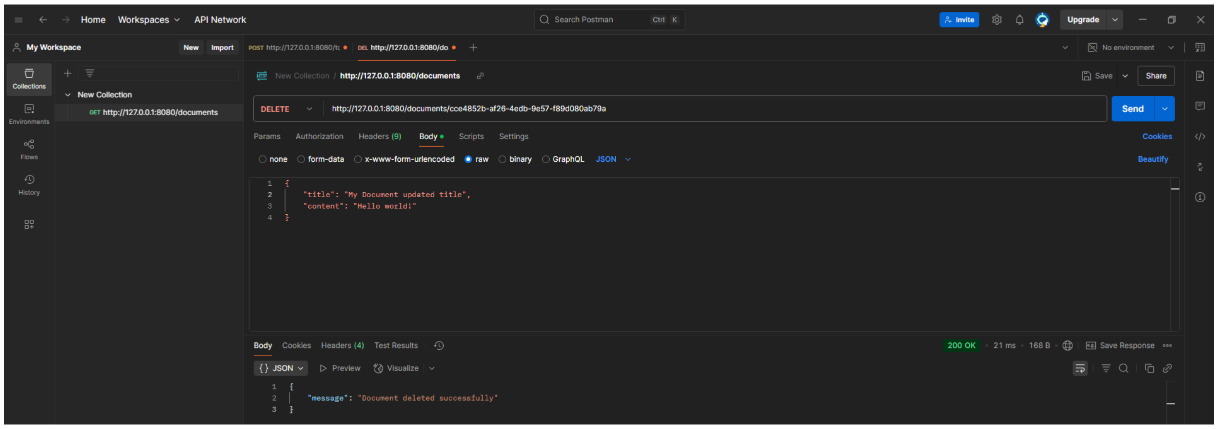
```
GET http://127.0.0.1:8080/documents/cce4852b-af26-4edb-9e57-f89d080ab79a
```

```
{
    "title": "My Document updated title",
    "content": "Hello world!"
}
```

```
200 OK · 15 ms · 257 B
```

```
{
    "title": "My Document updated title",
    "document_id": "cce4852b-af26-4edb-9e57-f89d080ab79a",
    "content": null,
    "plain_text_content": null
}
```

Figure 7: Document get(after update)



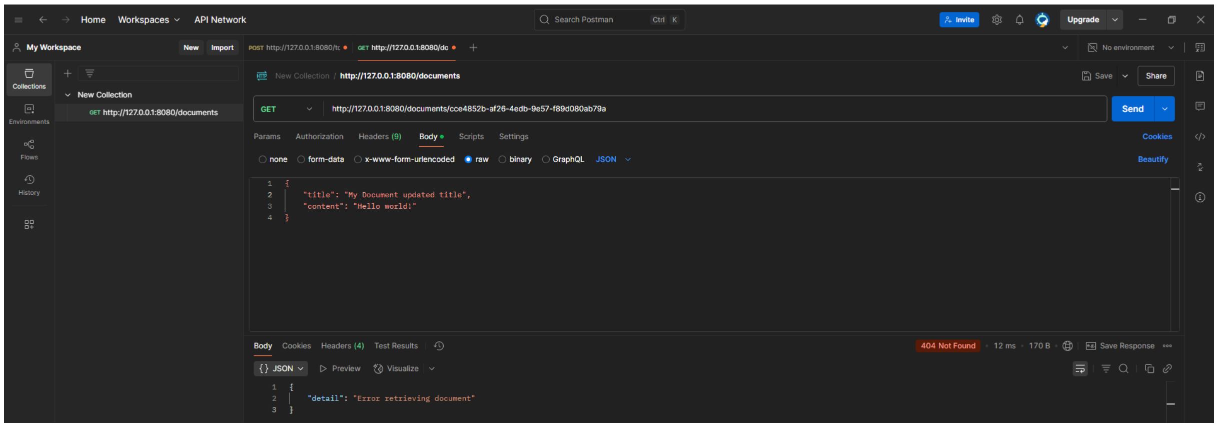
```
DELETE http://127.0.0.1:8080/documents/cce4852b-af26-4edb-9e57-f89d080ab79a
```

```
{
    "title": "My Document updated title",
    "content": "Hello world!"
}
```

```
200 OK · 21 ms · 168 B
```

```
{
    "message": "Document deleted successfully"
}
```

Figure 8: Document delete



```
GET http://127.0.0.1:8080/documents/cce4852b-af26-4edb-9e57-f89d080ab79a
```

```
{
    "title": "My Document updated title",
    "content": "Hello world!"
}
```

```
404 Not Found · 12 ms · 170 B
```

```
{
    "detail": "Error retrieving document"
}
```

Figure 9: Document get(after delete)

5 Conclusion

In this project, we were able to work with FastAPI, PostgreSQL, and Consul in order to try to provide reliability and extensibility at the same time. This architecture of the document management system is based on the principles of microservices and asynchronous processing, which allowed us to build a solution that can efficiently handle concurrent operations, scale as needed, and remain maintainable over time. By registering the service in Consul, we prepared the system for future scaling and integration with other services. Generally, this project made us to understand better how to combine modern tools and important principles to make a reliable but flexible system.