MODERN APPROACH FOR MAINTAINABLE AND SCALABLE BACKEND ARCHITECTURE

A Project
Presented to the
Faculty of
California State Polytechnic University, Pomona

In Partial Fulfillment
Of the Requirements for the Degree
Master of Science
In
Computer Science

By
Hao Zheng
2020
PROJECT: MODERN APPROACH FOR MAINTAINABLE AND SCALABLE BACKEND ARCHITECTURE

AUTHOR: Hao Zheng

DATE SUBMITTED: Fall 2020

Department of Science

Dr. Lan Yang
Project Committee Chair
Computer Science

Dr. Yu Sun
Computer Science
# TABLE OF CONTENTS

- SIGNATURE PAGE ...................................................... ii
- ABSTRACT ........................................................................ iv
- LIST OF TABLES ........................................................... v
- LIST OF FIGURES .......................................................... vi
- INTRODUCTION ............................................................. 1
- SECTION 1: MICROSERVICE ARCHITECTURE ......................... 3
  - Microservices Introduction ............................................. 3
  - Techniques to Implement Microservices ........................... 4
  - Microservice Architecture Implementation ..................... 6
- SECTION 2: DATA STORAGE ARCHITECTURE ....................... 9
  - SQL vs NoSQL .......................................................... 9
  - ER Diagrams and Database Tables ................................. 10
  - Database Controller Package ........................................ 12
- SECTION 3: BACKEND SERVICE ARCHITECTURE .................. 16
  - Python Language ...................................................... 16
  - UML Diagrams ........................................................ 16
  - Code Implementation .................................................. 18
- SECTION 4: RESTful ARCHITECTURE .................................. 20
  - RESTful Architecture Introduction ................................ 20
  - REST APIs Designing ............................................... 20
  - REST APIs Implementation ........................................ 21
  - Postman Validation and Unit Testing .............................. 23
- CONCLUSION .................................................................... 25
- REFERENCES ...................................................................... 26
ABSTRACT

In the field of web application development, there are three main parts in code implementation: backend architecture, frontend implementation, CI/CI pipeline. Since backend architecture is extremely important, therefore it is crucial to design a maintainable and extensible backend architecture for further development. The paper is to demonstrate building a solid backend architecture for an online library website.

The complete implementation of the architecture is split into four sections: microservices designing, database architecture, backend architecture, and REST API designing. In the introduction, the problems of traditional software development methodology are pointed out. The concepts of microservice and Docker are then introduced in Section 1. Section 2 demonstrates how to design the database architecture by illustrating ER diagrams and database controller packages. Section 3 demonstrates the specific steps of implementing backend logic for microservices using Python. REST APIs, the bridge connecting backend services and frontend clients, are introduced in Section 4. The verification of REST APIs using unit testing and postman is also introduced at the end of the paper. Each section explains related concepts and demonstrates specific steps of the implementation. The complete picture of the architecture is clear to illustrate the maintainability and extensibility of the system.
LIST OF TABLES

Table 1: REST API Documentation Table ........................................ 21
LIST OF FIGURES

Figure 1: Use Case Diagram ................................................................. 7
Figure 2: Backend Architecture Diagram ............................................. 7
Figure 3: CI/CD Pipeline Architecture .................................................. 8
Figure 4: ER Diagram ........................................................................ 11
Figure 5: Database Tables ................................................................. 12
Figure 6: Database Package UML Diagram .......................................... 13
Figure 7: Backend Services UML Diagram ............................................ 17
Figure 8: Postman Screenshot ............................................................. 23
INTRODUCTION

Backend architecture is the key component for web application development. However, due to the developments of new services and techniques, backend architecture varies every year. In frontend development, three main frameworks (React, Vue, Angular) gain the most popularity as jQuery still owns a large portion of the market. Backend frameworks are overwhelming, such as Python Django, Java Spring, NodeJS, Ruby on rails, and PHP. Nowadays, to satisfy specific requirements of data storage, different types of databases bloom up: Oracle, MySQL, PostgreSQL, MongoDB, Firebase, Redis, Neo4J, and etc. As different tech stacks serve specific purposes, it is important to pick the right tools so that the architecture can be implemented effectively.

A good backend architecture should be maintainable by more developers as the team grows, and it should be extensible to integrate with more components. Therefore, the architecture should leave space for future modification and be flexible to handle future modification.

Although backend architecture is a complex system combining different technologies and concepts. However, the main parts haven’t been changed: Development methodology, backend language, database choice, and middleware APIs. To build a sustainable architecture, it is important to implement each part in the right sequence and decompose the architecture in a logical way. In this paper, I come up with strategies to implement each component with different technologies, which help to deliver a web application quickly and make it can be integrated with other components. The complete project also can be easily maintained by more developers. In the following sections, I will introduce my strategies of implementation of each part (microservices architecture, database implementation, backend logic)
implementation, and REST architecture), related concepts are also explained in each section.
SECTION 1: MICROSERVICE ARCHITECTURE

1.1 Microservices Introduction

Back in the 2000s, the waterfall architecture model was the traditional choice for software development. In the traditional approach, developers design the complete application before implementation, and start to build the application from one component, then add more parts into that as the requirements grow. The benefit of this approach is that it is easier to deliver the first prototype since less thoughtfully designing ideas need to be considered. And the developing process using waterfall architecture is straightforward. After the foundation is set up, programmers implement each component and integrate every piece together.

However, the traditional approach is not suitable to develop web applications. The requirements usually grow rapidly as the web application gains more users and takes heavier traffic. As more services are needed, it is painful to update the application because of the high coupling between components. As the team size grows, it is almost impossible for developers to collaborate because each needs to understand the dependencies to contribute changes.

The low coupling development architecture is required for most web applications, the microservice architecture has sprung up over the last few years. It is a particular way of designing applications as collections of independently deployable services, it helps to arrange an application as a group of loosely coupled services. Services built from microservice architecture are highly cohesive and lightweight.

In a microservice architecture, developers design the core services of the applications, and then implement each service individually. Microservices architecture helps to produce more flexible components. On the other hand, it helps the team to split work evenly so that the application can be progressively finished by
adding up services. The microservice architecture is not perfect. The main
disadvantage is it increases the complexity of application designing, the developers
need to take more consideration to decouple the main services so that the application
can be extensible to integrate more services.
Yet such concern should not obscure the benefits that come from the microservice
architecture. It really makes the development process smoother and makes the
application easy to scale. Since the components can be deployed independently in a
microservice architecture, it becomes pretty popular in small development teams for
developing highly maintainable and testable prototypes.

1.2 Techniques to Implement Microservices

Microservices is a high-level designing methodology, and different
combinations of techniques can be actually used to implement the architecture. While
low-level implementations vary based on different techniques, the main strategies are
commonly used: data storage solution, independent deployment, services decoupling,
and agile development. All these approaches serve the same purpose: to decouple
components and ease the development process.

In terms of data storage, it is better to split data storage for each component.
For each microservice, developers should choose the database that best fits the
service. The benefit of using the same database reduces the duplication of work.
However, if one service changes the database structure for a new requirement, the
other services will also be affected since the same database is used. By creating
independent data storage for each service, it is easier to scale databases, and also helps
to make different databases better serve specific use cases. On the other hand,
technique problems like how to keep data consistent need to be thoughtfully
considered when designing the architecture.
In web applications, clients directly interact with interfaces built by frontend components, and it is extremely important to split frontend interfaces and backend services. If the backend architecture is one piece of glued components, whenever a small update or refactoring is needed, the application has to be terminated during the development period, which makes the web application inaccessible to users. This disconnection between users actually harms the user experience, and really hurts the profit in some areas (e.g., trading system, gaming). RESTful architecture helps to make this split possible, which focuses more on how to expose backend services to the frontend interface through REST APIs. And there are lots of libraries able to manage and handle that, which makes the implementation unpainful.

To actually implement the microservice architecture, different services should be deployed independently. Configuring different low-level dependencies between development and production environments is a painful job for DevOps. One way to solve this problem is containing services so that the same service can be launched regardless of different operating systems. Docker is a great tool to achieve this goal, and with Docker Hub, the cloud storage for Docker containers, further eases the management of different containers.

By decoupling components, the development tasks can be executed more effectively. Since the tasks can be easily split for each component, using the model software development methodology like Agile can further improve the speed of production, and the management process becomes easier through assigning tickets for different tasks.

Basically, microservice architecture breaks components into independent deployable components, and the backend services should be exposed to front-end components using RESTful API. Regardless of the different environment for
deployment, containerizing applications using Docker helps to ease the DevOps operations. With these techniques, the services are decoupled loosely, which further improves the development collaboration.

### 1.3 Microservice Architecture Implementation

Microservices are organized around business capabilities, therefore the first step to implement microservices is determining the use cases of the application. The project is a platform storing learning resource info of different books and videos, users are able to search public resources and recommend new resource info.

To specify the use cases, user groups of the application should be first analyzed. There are two main groups in the platform: users and administrators. Users can search and recommend learning resources, and the administrators are able to evaluate users’ recommendations and manage public resources.

To further break the application into services, detailed use cases need to be specified. There are 6 main use cases for this web application:

1. A user searches public resources.
2. A user modifies public resources.
3. A user creates new recommendations.
4. An administrator evaluates users’ modifications.
5. An administrator evaluates users’ recommendations.
6. The platform recommends resources to users

The following is the use case diagram:
After analyzing use cases, 3 main services should be considered for supporting all use case:

1. Searching services: Get books, Get videos
2. Recommendation services: Recommend users related learning resources
3. Admin services: Evaluates (pass or reject) users’ contribution (modification, recommendation)

Each service contains its data storage and requires an independent backend service for CRUD operations. The whole backend architecture is like the following:

Figure 1: Use Case Diagram

Figure 2: Backend Architecture Diagram
To deploy each service independently, backend services should be able to build as containers so that each can be set up in different environments in the same way:

![CI/CD Pipeline Architecture](image)

*Figure 3: CI/CD Pipeline Architecture*

After the high-level microservices architecture is designed, individual components can be implemented sequentially and integrated together at the end. The rest sections demonstrate the specific steps of the specific implementation of each component:

1. Build the database architecture for each component.
2. Implement backend services connecting databases and rapped each as a container.
3. Expose the services as REST APIs to connect services and frontend interfaces.
SECTION 2: DATA STORAGE ARCHITECTURE

After the high-level microservices architecture has been designed, the next step is to implement the low-level data storage architecture. In this section, the reasoning for choosing the database is introduced, and specific steps of implementations are also demonstrated.

2.1 SQL vs NoSQL

Choose the right database is an important task for any application. However, choosing the right database is also a difficult decision to make since different databases serve very different purposes. The first decision to make is whether to choose a relational database or a non-relational database.

The relational database is the traditional choice that was first used in 1970. Data are stored in tables and rows in a relational database. Different tables are linked together through the use of unique keys. The significant benefit of using a relational database is referential integrity, which refers to the consistency and accuracy of stored data. Due to the strict relationship between tables, a relational database is also useful to reduce data duplication. Even though a relational database works well with structured data, not all data storage system requires constraints between data. Using a relational database for simple data storage actually increases the complexity of the application, and designing highly-structured tables can be a challenge for developers.

Nowadays, the non-relational database, or NoSQL database, gains more popularity among web developers for handling less-structured data and data scaling. There are no tables and rows in the non-relational database. Instead, the NoSQL database uses a specific storage model for different requirements based on the type of data. On the other hand, NoSQL databases are suitable to store large amounts of data due to the simplicity to scale less-structured data horizontally.
There are four popular NoSQL data models: document data store, column-oriented database, key-value store, and graph database. Document data are JSON-like data that can be easily integrated with data structures of popular programming languages like Python and Java. The mapping allows programmers to store the data in a similar way that developers use it in business code, which helps programmers to write less code, leading to reduce development time and fewer bugs. Therefore, document data storage is a good choice for my project which requires object-oriented designing for backend services.

Among different NoSQL databases using the document data model, MongoDB is the most popular choice. Due to the maturity of the company and the community, MongoDB provides a suite of easy-to-use tools and libraries for development. Therefore, for the convenience of the development, MongoDB had been chosen for my project.

2.2 ER Diagrams and Database Tables

The designing of databases can be flexible since MongoDB is a NoSQL database, yet logical relationships between data should be well-organized for further development. ER diagram is a tool helping to visualize relationships between tables, which helps to establish a logical database system. Based on the high-level use cases analyzed in section 1, the ER diagram is drawn in figure 3:
Databases in MongoDB associate with collections, which contain JSON-like documents. User collection contains the information of each user. Video and Book collections provide detailed information on learning resources, and each is associated with Comment collection. Recommendation and Modification services are associated with the following 4 collections: VideoRecommendation, BookRecommendation, VideoModification, and BookModification.

To further make the databases more scalable, eight collections can be split into four different databases:
The RecommendationDB contains VideoRecommendation and BookRecommendation collections to provide recommendation services, while ModificationDB contains VideoModification and BookModification for modification services. And Video, Book, Comment are stored in LearningResourceDB for public data exploring. User collection is stored in UserDB independently. By splitting services into different databases, each service can have its data storage and be further distributed.

### 2.3 Database Controller Package

To access databases more effectively and securely, a database controller package should be created for microservices to use. This database package plays a role like a middleware API between database and backend services. The main class of the package is designed as the following:
To ensure the users don’t insert the wrong data, schemas for each collection are defined in the following structure:

```javascript
book_modification_schema = {
    'bsonType': 'object',
    'required': ['modifier_id'],
    'properties': {
        'image_url': {'bsonType': 'string'},
        'name': {'bsonType': 'string'},
        'edition': {'bsonType': 'int', 'minimum': -1, 'maximum': 100},
        'authors': {'bsonType': 'array', 'minItems': 1, 'items': {'bsonType': 'string'}},
        'published_time': {'bsonType': 'string'},
        'isbn': {'bsonType': 'string'},
        'platforms': {'bsonType': 'array', 'items': {'bsonType': 'string'}},
        'link': {'bsonType': 'string'},
        'description': {'bsonType': 'string'},
        'related_knowledge_fields': {'bsonType': 'array', 'items': {'bsonType': 'string'}},
        'difficulty_level': {'$type': 'array', 'items': {'$type': 'int', 'minimum': 1, 'maximum': 3}},
        'modifier_id': {'bsonType': 'string'},
    }
}
```

And some default patterns can be also generated before the data can be inserted into the database.
With this package, once a user wants to create data, the object will be first filled in with default values, then the package starts to evaluate the validation of processed data based on the schema to make sure the document is in the right format.

Testing for the database package is also necessary to validate the functionality of the package.

To make the database more extensible helping to support CI/CD pipeline, the database package should be able to control different database environments. There are two environments commonly used, one is the testing environment, the other is the production environment. And the development can be a local environment. To use

```python
book_pattern = {
    'image_url': '',
    'edition': -1,
    'published_time': '',
    'isbn': '',
    'platforms': [],
    'link': '',
    'description': '',
    'related_knowledge_fields': [],
    'difficulty_level': -1,
    'rating': {
        '1': 0,
        '2': 0,
        '3': 0,
        '4': 0,
        '5': 0,
        'scores_num': 0,
        'rating_num': 0,
        'avg_score': float(0),
    },
    'bookmarked_num': 0,
    'comments': []
}
```
this package, just specific the environment type by passing the environment parameter, and then functionalities can be used for backend services development.

```python
class DB_Manager:
    def __init__(self, environment_type: str) -> None:
        if environment_type == 'local':
            self.db_client = pymongo.MongoClient('mongodb://localhost:27017/')
        elif (environment_type == 'test') or (environment_type == 'stage'):
            self.db_client = pymongo.MongoClient(MONGO_TEST)
        elif environment_type == 'prod':
            self.db_client = pymongo.MongoClient(MONGO_PROD)
```

After the database package is designed, the next step is to implement the actual microservices.
SECTION 3. BACKEND SERVICE ARCHITECTURE

3.1 Python Language

To implement the backend services and logic, the programming language is an essential choice to deliver the project more efficiently and faster. Many programming languages are suitable for this project, such as Java, NodeJS, and Python. Java is popular to develop an enterprise application, and the framework of Java such as Spring is suitable for a larger team to collaborate. NodeJS and Python are better candidates for small teams to quickly deliver a prototype, and both are suitable for my project. Since a recommender system may be a further feature to add on, a language with a community of machine learning related libraries is necessary. Due to the large community and simplicity of Python, Python is chosen for my project to implement backend services.

3.2 UML Diagrams

After the decision of choosing the programming language has been made, the next step is to write the code. Before writing the first line of the code, it is necessary to design the relationship between classes to come up with a logically-designed architecture. UML diagrams can help to specify, visualize, construct, and document the models of software systems (including structures and design), therefore it is a good tool for architecture designing. Even though it takes time for drawing UML diagrams, it benefits and speeds up further implementation processes.

The following are UML diagrams of the project:
There are three layers of different classes from bottom to top: low-level database management layer, main services layer, and high-level layer.

Low-level database management layer contains 3 classes: UserDataManager, PublicDataManager, and SubmissionDataManager. Each class interacts with independent databases. UserDataManager controls UserDB, PublicDataManager interacts LearningResourceDB, and SubmissionDataManager gets access to both RecommendationDB and ModificationDB.

The main services layer contains classes that actually implement the business logic of microservices, such as registering new users, searching public learning resources, writing comments, submitting recommendations, and modifications.

Classes in the main services layer are used by classes in the high-level layer. The high-level layer controls the main services and wraps use cases into classes for different user groups. PublicServices class provides necessary services for all users such as searching learning resources and registering. DeveloperServices controls access to all services classes to manage all resources. UserServices provides services such as recommendation and modification submission for registered users.
3.3 Code Implementation

There are 12 classes in total. Each class is written in Python and correlated with each other. Basically, each class exposes the APIs to the upper layer classes to use. The following 3 classes are shown in the figures for the demonstration.

The following class is public data services, which is used by users to get necessary learning resources:

```python
class PublicDataServices:
    def __init__(self):
        self.public_data_searching_services = PublicDataSearchingServices()
        self.public_data_interaction_services = PublicDataInteractionServices()

    def search_all(self, data_type, name, sortby):
        return self.public_data_searching_services.search_all(data_type, name, sortby)

    def get_collections(self, user_id, name, sort):
        return self.public_data_searching_services.get_collections(user_id, name, sort)

        return self.public_data_searching_services.get_book_info(book_id)

    def get_book_comments(self, book_id, sortby):
        return self.public_data_searching_services.get_book_comments(book_id, sortby)

    def get_video_info(self, video_id):
        return self.public_data_searching_services.get_video_info(video_id)

    def get_video_comments(self, video_id, sortby):
        return self.public_data_searching_services.get_video_comments(video_id, sortby)

    def get_knowledge_field_info(self, knowledge_field_id):
        return self.public_data_searching_services.get_knowledge_field_info(knowledge_field_id)

    def get_knowledge_field_related_books(self, knowledge_field_id):
        return self.public_data_searching_services.get_knowledge_field_related_books(knowledge_field_id)

    def get_platform_info(self, platform_id):
        return self.public_data_searching_services.get_platform_info(platform_id)

    def get_platform_related_books(self, platform_id):
        return self.public_data_searching_services.get_platform_related_books(platform_id)
```
The AdminServices class is used to get modifications and evaluations:

class AdminServices:
    def __init__(self):
        self.submission_services = SubmissionServices()
        ...

    def get_recommendations(self, user_id):
        return self.submission_services.get_recommendations(user_id)
    def get_all_book_recommendations(self, user_id):
        return self.submission_services.get_all_book_recommendations(user_id)
    def get_book_recommendation(self, user_id, recommendation_id):
        return self.submission_services.get_book_recommendation(user_id, recommendation_id)
    def get_all_video_recommendations(self, user_id):
        return self.submission_services.get_all_video_recommendations(user_id)
    def get_video_recommendation(self, user_id, recommendation_id):
        return self.submission_services.get_video_recommendation(user_id, recommendation_id)
    def get_knowledge_field_recommendations(self, user_id, filter_id):
        return self.submission_services.get_knowledge_field_recommendations(user_id, filter_id)
    def get_knowledge_field_recommendation(self, user_id, recommendation_id):
        return self.submission_services.get_knowledge_field_recommendation(user_id, recommendation_id)
    def get_platform_recommendations(self, user_id, recommendation_id):
        return self.submission_services.get_platform_recommendations(user_id, recommendation_id)
    def get_platform_recommendation(self, user_id, recommendation_id):
        return self.submission_services.get_platform_recommendation(user_id, recommendation_id)

    def get_modifications(self, user_id, filter_id):
        return self.submission_services.get_modifications(user_id, filter_id)
    def get_book_modifications(self, user_id, filter_id):
        return self.submission_services.get_book_modifications(user_id, filter_id)
    def get_book_modification(self, user_id, modification_id):
        return self.submission_services.get_book_modification(user_id, modification_id)
    def get_video_modifications(self, user_id):
        return self.submission_services.get_video_modifications(user_id)
    def get_video_modification(self, user_id, modification_id):
        return self.submission_services.get_video_modification(user_id, modification_id)
    def get_knowledge_field_modifications(self, user_id, filter_id):
        return self.submission_services.get_knowledge_field_modifications(user_id, filter_id)
    def get_knowledge_field_modification(self, user_id, modification_id):
        return self.submission_services.get_knowledge_field_modification(user_id, modification_id)
    def get_platform_modifications(self, user_id, filter_id):
        return self.submission_services.get_platform_modifications(user_id, filter_id)
    def get_platform_modification(self, user_id, modification_id):
        return self.submission_services.get_platform_modification(user_id, modification_id)

    def get_recommendation_evaluations(self, user_id, recommendation_id, reason):
        self.submission_services.get_recommendation_evaluations(user_id, recommendation_id, reason)
    def get_book_recommendation_evaluations(self, user_id, recommendation_id, reason):
        self.submission_services.get_book_recommendation_evaluations(user_id, recommendation_id, reason)
    def get_video_recommendation_evaluations(self, user_id, recommendation_id, reason):
        self.submission_services.get_video_recommendation_evaluations(user_id, recommendation_id, reason)
    def get_knowledge_field_recommendation_evaluations(self, user_id, recommendation_id, reason):
        self.submission_services.get_knowledge_field_recommendation_evaluations(user_id, recommendation_id, reason)
    def get_platform_recommendation_evaluations(self, user_id, recommendation_id, reason):
        self.submission_services.get_platform_recommendation_evaluations(user_id, recommendation_id, reason)
    def get_recommendation_reasons(self, user_id, recommendation_id):
        self.submission_services.get_recommendation_reasons(user_id, recommendation_id)
    def get_book_recommendation_reasons(self, user_id, recommendation_id):
        self.submission_services.get_book_recommendation_reasons(user_id, recommendation_id)
    def get_video_recommendation_reasons(self, user_id, recommendation_id):
        self.submission_services.get_video_recommendation_reasons(user_id, recommendation_id)
    def get_knowledge_field_recommendation_reasons(self, user_id, recommendation_id):
        self.submission_services.get_knowledge_field_recommendation_reasons(user_id, recommendation_id)
    def get_platform_recommendation_reasons(self, user_id, recommendation_id):
        self.submission_services.get_platform_recommendation_reasons(user_id, recommendation_id)
SECTION 4. RESTful ARCHITECTURE

4.1 RESTful Architecture Introduction

After the backend services are implemented, how to build a solid communication bridge between clients and the backend services needs to be considered. RESTful architecture eases the communication between the frontend and the backend services, it is a popular approach to transfer different resources (text, image, song, or service) using the HTTP protocol. In RESTful architecture, it is straightforward to control the state of resources using different HTTP requests (GET, PUT, POST, DELETE).

REST API is a convenient way to deliver the data to clients. After REST APIs are set up, if clients want to request details of a book, just send GET requests to the backend address /book/<book_id>, then a JSON file containing book details would be returned. As long as the frontend is connected to the Internet so that it can send HTTP requests, then the frontend gets consistent data from the backend.

Overall, there are two advantages of using RESTful architecture. REST APIs are highly reusable since they can be used in any form of application (mobile, browser, tablet, or desktop) once created, which benefits the scalability of the architecture. On the other hand, data providing by REST APIs is always accessible as long as the client is connected with the Internet, which provides convenience for both clients and developers.

4.2 REST APIs Designing

Before implementing the actual REST APIs, documents containing details of each API is extremely important to be first created. Since the REST APIs are used by both frontend and backend developers, shared documents can be a good bridge helping all developers keeping track of the details.
In the document, three main information of REST API is necessary: HTTP request type, the destination address, and a detailed description of usage. The following is an example table of the API table for book services.

Table 1: REST API Documentation Table

<table>
<thead>
<tr>
<th>HTTP Request</th>
<th>Destination</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/v1/books</td>
<td>Get all public book resources</td>
</tr>
<tr>
<td>GET</td>
<td>/v1/books?args</td>
<td>Search for books with specific name</td>
</tr>
<tr>
<td>GET</td>
<td>/v1/books?name=xxx&amp;sort=time</td>
<td>Advanced searching sort_by=time/popularity/rating</td>
</tr>
<tr>
<td>GET</td>
<td>/v1/books/&lt;id&gt;/info</td>
<td>Get detail info of a specific book</td>
</tr>
<tr>
<td>GET</td>
<td>/v1/books/&lt;id&gt;/comments?args</td>
<td>Get comments of a specific book</td>
</tr>
<tr>
<td>POST</td>
<td>/v1/books/&lt;id&gt;/rating?user_id=xxx&amp;score=1</td>
<td>Rate a book</td>
</tr>
<tr>
<td>POST</td>
<td>/v1/books/&lt;id&gt;/comments?user_id=xxx</td>
<td>Write a comment to a specific book</td>
</tr>
</tbody>
</table>

4.3 REST APIs Implementation

After finish documenting REST APIs, it is time to implement codes for APIs. There are many libraries and tools to manage and handle RESTful APIs, such as Python Django, Java Spring, Node ExpressJS, and etc. Among those, the Python Flask library gains more popularity due to its flexibility and simplicity.

Flask is a WSGI web application library, it is created to implement RESTful architecture easily and quickly. It is a lightweight library that doesn’t require other libraries. Unlike the heavy framework Django, Flask has no form validation, database layer, or any other dependencies with pre-installed components that provide other common services. Flask only handles REST APIs and does it well, the application using Flask can be easily scaled up to a larger system. The other components such as
database controller, form validation, security authentication, and other common technologies can be easily integrated with flask. Therefore, I’ve chosen Flask to implemented the REST APIs of the project.

The following is the code fragment of Flask implementation:

```python
@app.route('/vl/books', methods=['GET'])
def search_for_public_books():
    try:
        name = request.args.get('name')
        sortby = request.args.get('sortby')
        name = None if (name is None) or (len(name) is 0) else name
        sortby = None if (sortby is None) or (len(sortby) is 0) else sortby
        books = main_services_manager.public_data_services.search_all('book', name, sortby)
        return dumps(books)
    except Exception as e:
        return response_factory.error_response(str(e))

@app.route('/vl/books/<id>/info', methods=['GET'])
def get_book_info(id):
    try:
        book_info = main_services_manager.public_data_services.get_book_info(id)
        return dumps(book_info)
    except Exception as e:
        return response_factory.error_response(str(e))

@app.route('/vl/books/<id>/comments', methods=['GET'])
def get_book_comments(id):
    try:
        sort = request.args.get('sort')
        book_comments = main_services_manager.public_data_services.get_book_comments(id, sort)
        return dumps(book_comments)
    except Exception as e:
        return response_factory.error_response(str(e))

@app.route('/vl/videos', methods=['GET'])
def search_public_videos():
    try:
        name = request.args.get('name')
        sortby = request.args.get('sortby')
        name = None if (name is None) or (len(name) is 0) else name
        sortby = None if (sortby is None) or (len(sortby) is 0) else sortby
        videos = main_services_manager.public_data_services.search_all('video', name, sortby)
        return dumps(videos)
    except Exception as e:
        return response_factory.error_response(str(e))
```

As you can see, it is very straight forward to implement REST API using route() decorator in Flask. After defining the end address and HTTP methods in route() decorator, a REST API can be finished followed by writing a python function to implement the actual business code.
4.4 Postman Validation and Unit Testing

After implementing REST APIs, it is necessary to use a tool to verify the correctness of each API. Postman is an easy tool to send HTTP requests and it is suitable to verify the validation of the REST APIs.

The following is a screenshot of using Postman:

![Postman Screenshot](image)

After defining the HTTP verb and destination in Postman, you can get the request by hitting the Send button. It is a user-friendly tool to use as well as a powerful tool to set up API testing. Therefore, it is a great tool to verify APIs.

To make testing automatic and more scalable, unit testing for REST APIs should be implemented. The following code shows the unit testing to search for books:

```java

```
In Python, a library called unittest can be used for unit testing. One thing to notice is corner case handling must be included to make the testing more robust so that we are able to change the code without fear. On the other hand, by running all unit tests, it is easy to debug the broken part. Therefore, unit testing is a required task after business implementation.
5. CONCLUSION

The content mentioned above introduced the technologies I used and the steps I took to implement the project. To summarize the approaches, my implementation of the backend architecture contains the following steps:

1. Design the microservices architecture by splitting use cases into different services.
2. Draw out ER diagrams for database architecture, and implement package to controller databases in different environments.
3. Draw UML diagrams for objected-oriented design, and start business code implementation.
4. Document and implement REST APIs, write unit testing for validation.

Overall, to implement the backend architecture, I first design the microservice structure, then implement the architecture by coding the database controller package, business code, and REST APIs, following by writing unit testing at the end. This architecture helps to produce my prototype of the web application quickly and easily, and the architecture can be easily extended to more developers and codes. My project yet confirms the importance of scalability of backend implementation.
REFERENCES


