Building a Cloud-Based Distributed System for Storing and Retrieving Audio Data Using Java and Oracle Cloud Infrastructure

Distributed Software Systems (COEN 6731)
Assignment-1

Pranav Jha
Student ID: 40081750
Department of Electrical and Computer Engineering
Concordia University
Montreal, Canada
jha_k.pranav@live.com

Vishruth Khatri
Student ID: 40241455
Department of Electrical and Computer Engineering
Concordia University
Montreal, Canada
itsvishruthkhatri@hotmail.com

***Abstract*—In this project, we developed a distributed software system that can store audio files and their relevant attributes such as artist name, track title, album title, year, number of reviews, etc. We chose to use Java as the programming language and Maven as the build tool, in line with project requirements. To ensure thread safety, the servlet has been implemented with two methods - get and post, and utilizes an executor and concurrent hash map. Our client test file contains JUnit 5 test cases that simulate various scenarios with different numbers of clients and ratios of get and post requests. The system is capable of executing the get method and returning the relevant audio item when a file is requested with an artist name. In cases where no attributes are provided, the system returns all audio data in JSON format to the client. The post method successfully uploads audio files when all attributes are provided. The system has been deployed on the Oracle Cloud Infrastructure and is functioning as intended. Additionally, we have created an OpenAPI specification on SwaggerHub to document the endpoints and operations of our distributed software system for storing audio items. This provides a clear and standardized way for developers to understand and interact with the system's API.**

***Keywords*—Cloud Deployment, Open API Design, GET and POST Methods, Servlet Implementation, Concurrent Clients, Thread Safety, Round-Trip Time.**

INTRODUCTION

This report presents the design, implementation, and evaluation of a distributed software system for storing and retrieving audio items. The system is designed to be deployed on oracle cloud infrastructure and is implemented using Java programming language and Maven as a build tool [1]. The system includes an open API design for handling GET and POST requests for audio items, and a servlet for handling requests from clients [2]. The servlet is implemented using a thread-safe, in-memory data structure to store and retrieve audio items. A test class is also implemented to simulate concurrent requests from multiple clients and to record the round-trip time for each request.

I. TASK 1: CLOUD DEPLOYMENT

The deployment of the client/server-based distributed architecture for our project is done on a free-tier oracle cloud virtual machine. The server is configured to run on Jetty. This is the link for our project: http://168.138.93.110:9090/coen6731/audio

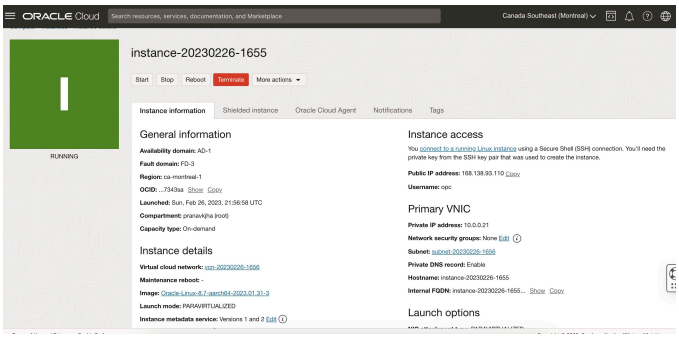

Figure 1. Oracle cloud instance.

II. TASK 2: RESOURCE OPEN API DESIGN

We have designed open APIs for the GET and POST methods to interact with the audio item properties. This OpenAPI specification defines a RESTful API for managing audio data. It includes two endpoints: /audio and /audio1.

The /audio endpoint allows clients to retrieve a single property value for a given audio item. The property name is provided as a query parameter, and the response includes the requested property value in JSON format as shown in Fig. 3. Additionally, the endpoint allows clients to create new audio items by sending a POST request with the required fields included in the query parameters.

The /audio1 endpoint provides clients with the ability to retrieve all audio items stored in the database, as well as to create new audio items by sending a POST request with the

required fields included in the query parameters as shown in Fig. 4 and Fig. 5, respectively.

The specification includes a description for each endpoint, as well as a list of the required parameters for each API operation. It also defines the expected responses for each API operation and provides examples of the expected response format.

Finally, the specification includes a components section that defines the schema for the audio data stored in the database, including the fields that should be included for each audio item. This section also includes an Audios schema that defines an array of audio data.

We have implemented these APIs on SwaggerHub as shown in Fig. 2, a platform for designing, documenting, and testing APIs. Our design has been validated to create an open API web page, and we have made the API public so that it can be accessed by anyone with the link.

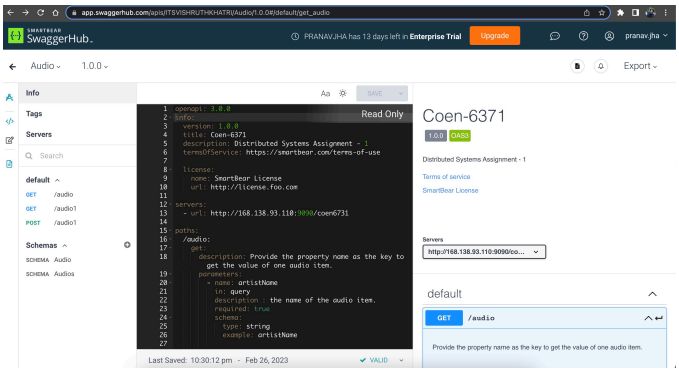

Figure 2. Open API design on SwaggerHub.

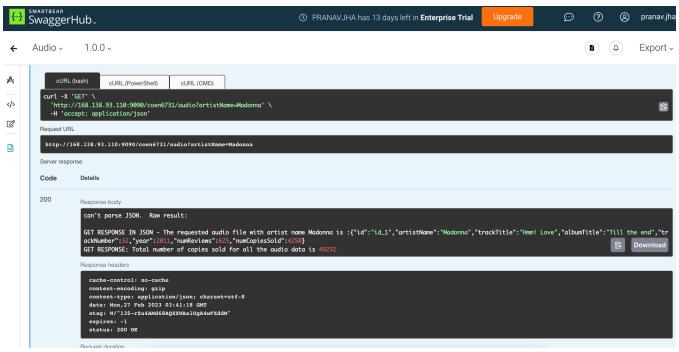

Figure 3. GET audio with an artist name.

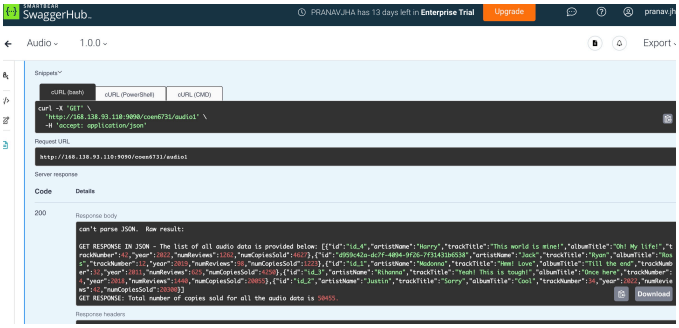

Figure 4. GET all the audio data.

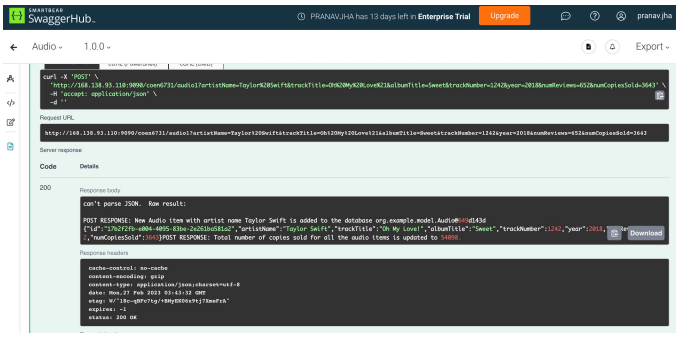

Figure 5. POST an audio item with all the attributes.

The link for our open API is https://app.swaggerhub.com/apis/ITSVISHRUTHKHATRI/Audio/1.0.0

III. Task 3: APIs Handling in the Server Side using Servlet

This section describes the implementation of a servlet for handling GET and POST requests from clients. The servlet is multithreaded and uses a thread-safe, in-memory data structure for storing and retrieving audio items.

A. *Audio.java*

The Audio.java file is an important part of our audio storage system. It defines the class for audio items and their relevant attributes, including artist name, track title, album title, track number, year, number of reviews, and number of copies sold. This class provides the structure for creating and manipulating audio items in our system.

The class contains private data members for each attribute, with getter and setter methods to access and modify these values. For example, the getArtistName() method returns the artist name of an audio item, while setTrackNumber() sets the track number for the item.

Additionally, the class includes a unique identifier for each audio item, as represented by the id member variable. This identifier is used to differentiate between different audio items in the system.

The Audio.java file plays an important role in our system's functionality, as it is used to create and manipulate audio items in our database. It is utilized by the servlets and other components of our system to handle requests from clients and store data in a thread-safe manner.

B. ResurceServlet.java

This is a Java servlet [3] that handles GET and POST requests for audio data. It uses a ConcurrentHashMap to store audio objects, which can be searched by artist name, track title, album title, track number, year, number of reviews, and number of copies sold. If no search parameters are provided, all audio objects in the database are returned as a JSON object as shown in Fig. 6 and Fig. 7. If any search parameters are provided, the servlet searches the database for matching audio objects and returns them as a JSON object as shown in Fig. 8 and Fig. 9. The servlet also initializes the audio database with some example audio objects and keeps track of the total number of copies sold. It will always return the number of total copies sold for all the files in the database as shown in Fig. 8.

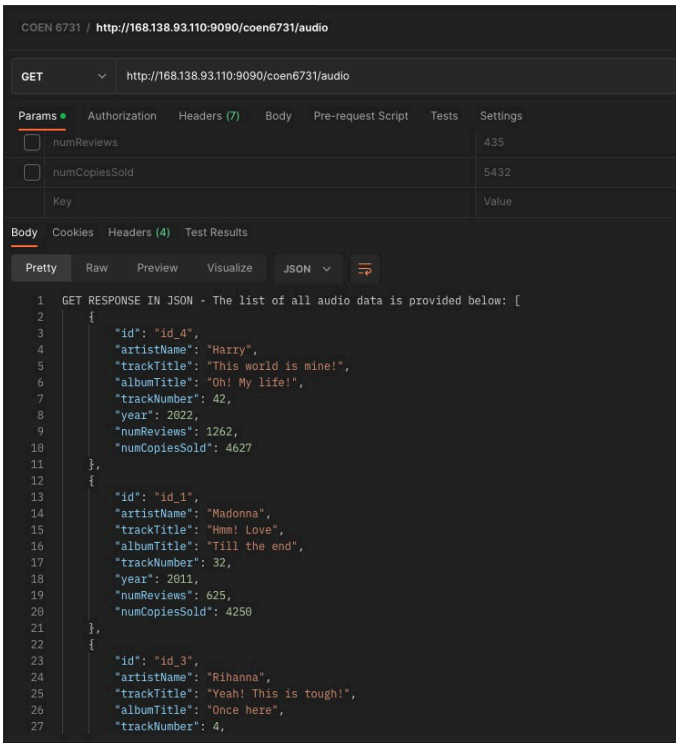

Figure 6. GET response in postman when no search parameter is provided.

Figure 7. GET response in the Oracle cloud console when no search parameter is provided.

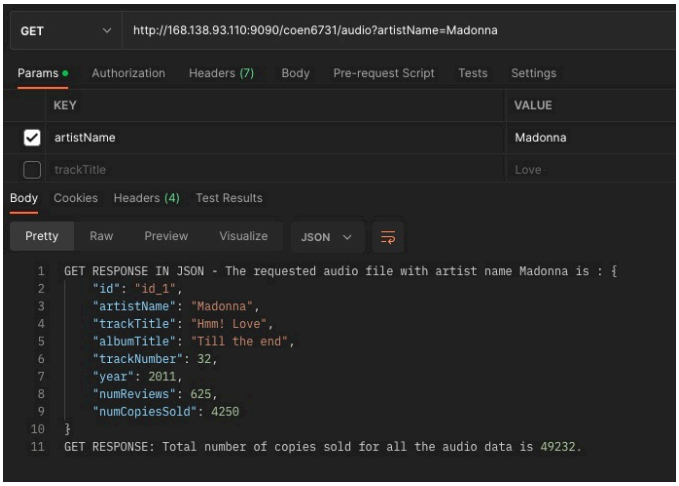

Figure 8. GET response in postman when the audio item is available in the database.

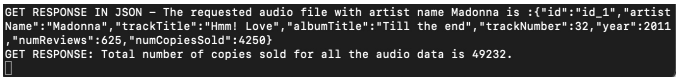

Figure 9. GET response in the cloud console when the audio item is available in the database.

If no audio parameter is provided, the servlet will return an error message indicating that the requested audio file is not available in the database. as shown in Fig. 10 and Fig. 11.

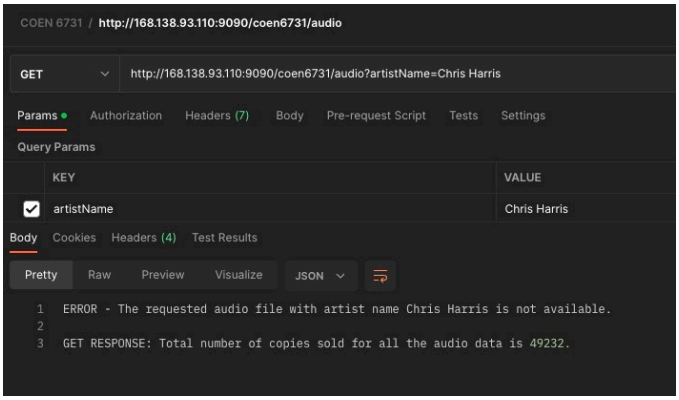

Figure 10. GET response in postman when the audio item is not available in the database.

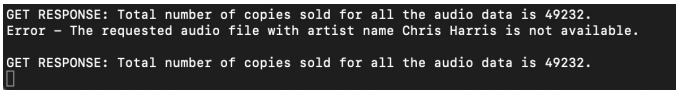

Figure 11. GET response in the cloud console when the audio item is not available in the database.

The POST request receives input parameters from the client and uses them to create a new audio item in the database, as depicted in Fig. 12 and Fig. 13 where the audio file with the name ”Chris” and other details are being added to the database.

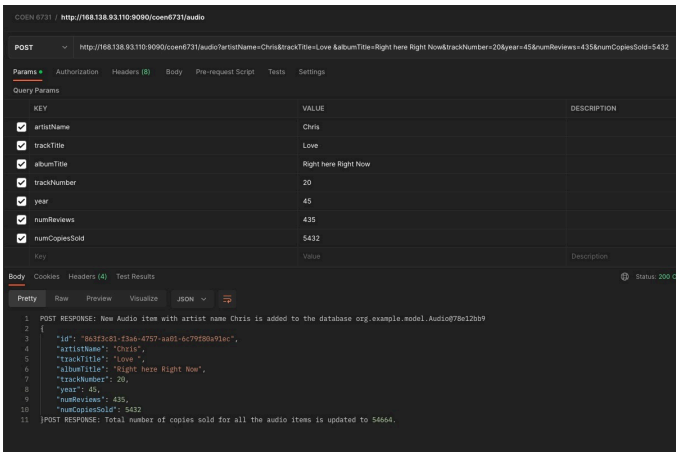

Figure 12. POST response in postman when an audio file is added.

```
POST RESPONSE: New Audio with artist name Chris is added to the database org.example.model.Audio
@78e12bb9

POST RESPONSE: Total number of copies sold for all the audio items is updated to 54664.
```

Figure 13. POST response in the cloud console when an audio file is added.

Now, when the audio file with the name ”Chris” is added to the database, it can be requested by any client and the get response will be shown as given in the Fig. 14 and Fig. 15 below:

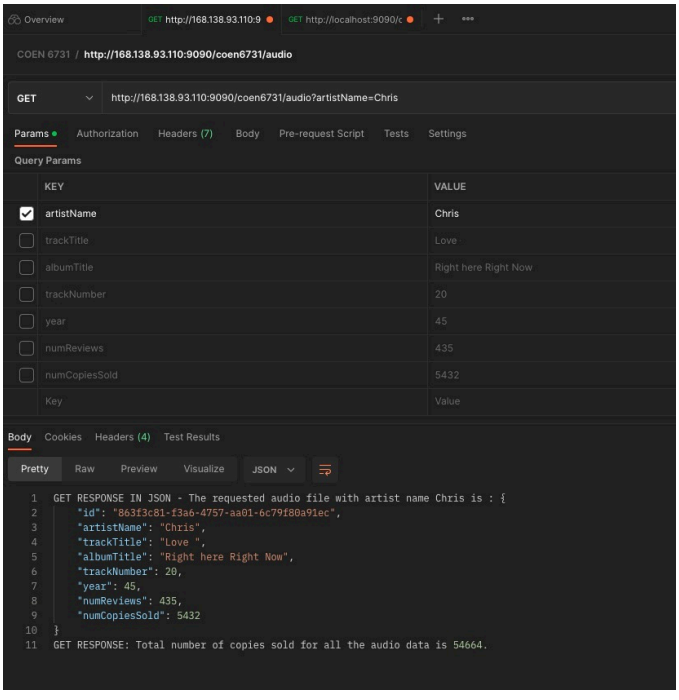

Figure 14. GET response output in postman after a POST response in console.

```
GET RESPONSE IN JSON - The requested audio file with artist name Chris is :{"id":"863f3c81-f3a6-4
757-aa01-6c79f80a91ec","artistName":"Chris","trackTitle":"Love ","albumTitle":"Right here Right N
ow","trackNumber":20,"year":45,"numReviews":435,"numCopiesSold":5432}
GET RESPONSE: Total number of copies sold for all the audio data is 54664.
```

Figure 15. GET response output in the cloud console after a POST response after the audio file is added.

IV. TASK 4: THE CONCURRENT CLIENTS

This section presents the AudioClientTest.java class that simulates concurrent requests from multiple clients. The purpose of this test class is to evaluate the system's ability to handle a variety of client loads and request types.

To achieve this, we have varied the ratio of clients sending GET requests and POST requests to 2:1, 5:1, and 10:1, and tested the system with a total of 10, 50, and 100 clients. The GET requests/responses have two types - one that returns a property value given an artist's name and another that returns all the artists' data in JSON format. On the other hand, the POST response is a message confirming that the audio item has been successfully stored in the database. The JUnit testing and the corresponding output for GET and POST can be seen in Fig. 16 and Fig. 20.

For each request, we have recorded the round-trip time taken and plotted the results on a line chart, where the y-axis represents the time in milliseconds or seconds, and the x-axis represents the number of clients. Each line on the chart represents a different ratio of client GET and POST requests as shown in Fig. 18 and Fig. 20.

The results of our tests indicate that the system can handle a significant amount of client requests while maintaining acceptable response times as shown in Fig. 18. For example, with a ratio of 2:1 and a total of 100 clients, the system recorded an average response time of 2.8 milliseconds for GET and POST requests.

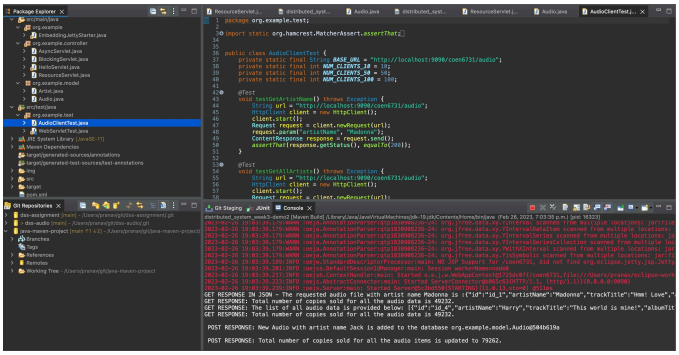

Figure 16. GET and POST response in the console.

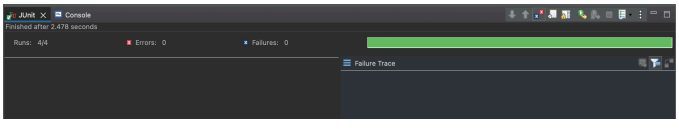

Figure 17. GET and POST Junit Test Response.

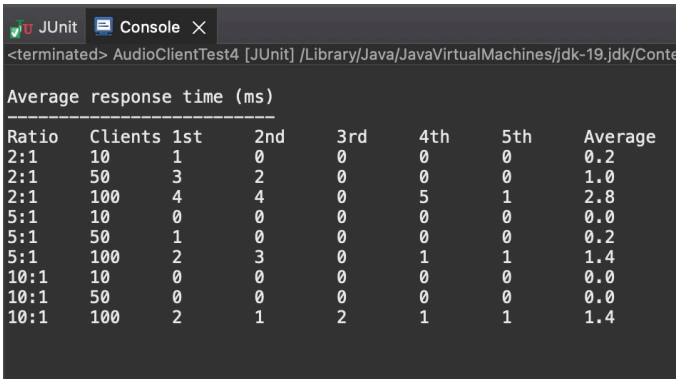

Figure 18. Round trip time for each request

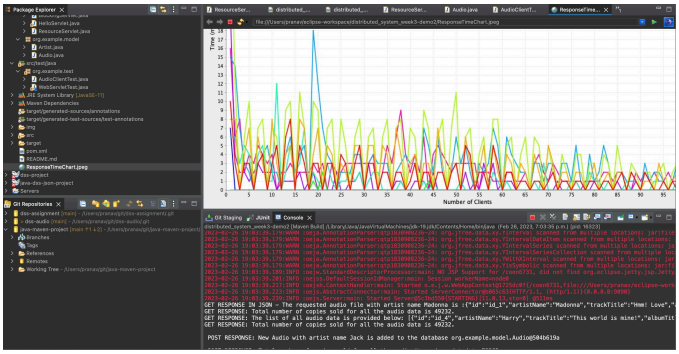

Figure 19. Generates a response time chart.

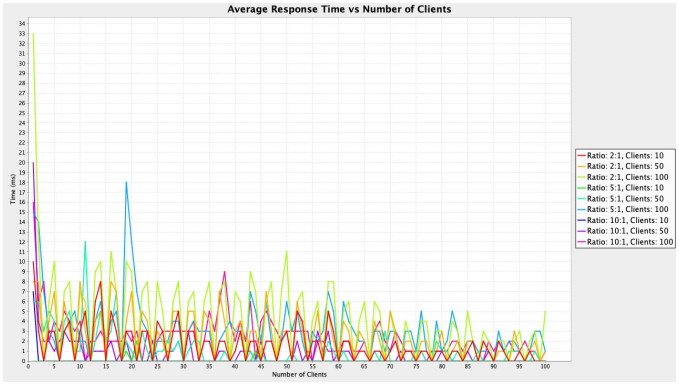

Figure 20. Response time chart.

V. Conclusion

In conclusion, the developed distributed software system provides a reliable and efficient solution for storing audio files and their relevant attributes. The implementation of thread-safe methods and the utilization of an executor and concurrent hash map ensures the system can handle concurrent requests from multiple clients. The successful implementation of both the get and post methods and the ability to return audio data in JSON format provides users with easy access to the stored audio files. The creation of an OpenAPI specification on SwaggerHub provides a standardized way for developers to interact with the system's API. The deployment of the system on the Oracle Cloud Infrastructure ensures scalability and availability for potential users. Overall, the developed system meets the project requirements and provides a practical solution for storing audio files in a distributed environment.

References

[1] Y. Innn. *Concordia University*. [Online]. Available: https://github.com/youyinnn/distributed_system_jetty_helloworld

[2] Y. Liu. *Concordia University*. [Online]. Available: https://app.swaggerhub.com/apis/yan.liu2/SkiRideAPI/1.0#/

[3] *Tutorials Point*. [Online]. Available: https://www.tutorialspoint.com/servlets/index.html