Braude project book – Phase 2

Submitted in partial fulfillment for the degree of

Bachelor of Science

in

Software Engineering

for

The Software Engineering Department

at

Braude - College of Engineering, Karmiel

By

Ward Zidani

Ahmad Bsese

Under guidance of

Zeev Frenkel

Project code

24-1-D-44

Summer 2024

Abstract

Within our project we develop AudioWave an audio sharing platform based on user-generated content. We look into solving some problems that, eventually, every successful platform faces as it grows its user base. These problems include the difficulties of serving users in times of dynamically changing traffic, such as traffic peaks, and the ability to control the flow of error detection and fault singulation. Another problem the project aims to solve is the complexity of the future of integrating new features. We implement a certain architectural pattern to the backend, an architecture based on the microservices architecture. The project's architecture is divided into separate microservices, each microservice has a main responsibility, with some minor ones. The main rule of implementing and dividing the system into microservices is that a single client request must require exactly one service to be running, meaning that if a service must not wait for an inter-service message to complete a client request. Our solution provides a fast and seamless experience of listening to audios by dividing the audios into smaller, more manageable chunks. Each chunk is a predetermined amount of seconds long, all the audios follow this predetermined chunk time-length. When a user requests to play an audio, even if not from the beginning, the backend returns the appropriate chunks of where the audio player is in the audio dynamically, doing this makes the platform faster and more performant.

1. Project Book

1.1. General Description

Project Name: AudioWave
Objective:
AudioWave is a microservices-based audio sharing platform that focuses on user-generated content. The primary goal of this platform is to allow users to upload, manage, and listen to audio files in an efficient and scalable way. The system is built to handle dynamic traffic loads and provide a seamless user experience by splitting audio files into manageable chunks for fast streaming.

1.2. **System Goals and Implementation**:

The main goals of the AudioWave platform are:

- **Scalability**: To accommodate varying traffic loads without sacrificing performance, especially during peak usage times.
- **Modular Architecture**: By utilizing a microservices approach, each service is dedicated to a specific functionality (such as user management, audio management, and playlists), ensuring the platform remains flexible and maintainable.
- **Efficient Audio Streaming**: To enhance user experience, audio files are divided into smaller segments that are dynamically loaded as the user listens, reducing waiting times and improving performance.
- **Error Handling and Fault Tolerance**: The architecture includes error detection and isolation mechanisms to ensure that one service failure does not affect the entire system.
- **Future-Ready Architecture**: The microservices architecture allows for easier integration of new features without disrupting existing services.

1.3. **System Structure**:

The system consists of various microservices, each containerized using Docker. Key services include:

- **User Management Service**: Handles user registration, login, and account management.
- **Audio Management Service**: Responsible for uploading, processing, and managing audio files.
- **Playlist Service**: Manages user-generated playlists, allowing users to create and share audio collections.
- **Audio Playback Service**: Delivers audio content to users by fetching and streaming audio chunks efficiently.

1.4. **Target Audience**:

- **Content Creators**: Individuals who produce audio content (musicians, lecturers, podcasters) and wish to share it with a broad audience.
- **Content Consumers**: Listeners who are interested in various types of audio content, from educational lectures to entertainment and music.

2. Solution Description

2.1. System Architecture (Architecture Overview):

The AudioWave platform is built using a microservices architecture, where each service is responsible for a specific business function. The high-level architecture consists of:

- **API Gateway**:- The reverse proxy is handled by Nginx, which manages service discovery, request routing, health checks, and error handling. Nginx ensures that client requests are correctly routed to the relevant microservice.
- **Microservices**:- Each microservice is dedicated to a specific function, ensuring scalability and fault tolerance. The major services include:
    - **User Management Service**: Handles all user-related functions, including registration, login, and profile management integrates with the Credential Encryption Service to securely store and retrieve user credentials, ensuring the safe handling of sensitive information.
    - **Audio Metadata Service**: Stores and manages metadata for audio files, such as titles, descriptions, and tags while interacting with the encryption service as needed for credential validation..
    - **Audio File Service**: Manages the actual audio files, breaking them into chunks for efficient streaming.
    - **Playlist Service**: Handles the creation, editing, and management of user playlists while interacting with the encryption service as needed for credential validation.
- **Databases**: We used a mix of SQL and NoSQL databases. SQL Server was used for user and metadata storage, while MongoDB was used for the playlist storage due to its flexible schema capabilities.
- **Message Broker**: We utilized RabbitMQ to facilitate communication between microservices, enabling asynchronous messaging and better fault tolerance.The system operates within a Virtual Private Cloud (VPC) to ensure security, and all microservices are containerized using Docker, allowing for easy deployment and scalability.
- **Encryption Service**: A new Credential Encryption Service is added to securely manage and store user credentials. The system utilizes MongoDB to store the encryption keys, which are generated by an external Credential Encryption Key Generator Application. This ensures secure handling of sensitive information such as user passwords or authentication tokens.
-

2.2. Research/Development Process

Development Steps:

- **Planning:** We started with an architectural design phase where we outlined the high-level design of the system. This included selecting the technologies (Flutter, .NET Core, SQL Server, MongoDB, RabbitMQ) and designing the microservices.
- **Development:** The development was carried out in two parts:

    - **Frontend:** Built with Flutter for mobile and React with TypeScript for the web application.
    - **Backend:** Created as a series of microservices, each handling a specific domain.
- **Deployment:** Each microservice was containerized using Docker, enabling easy deployment and scaling. The entire system was deployed to a Virtual Private Cloud (VPC), and we used Kong Gateway for routing requests.
- Testing: Functional and performance tests were performed at each stage to ensure the system's responsiveness and scalability under different load conditions.

Tools Used:

- **Frontend:** Flutter (mobile),.
- **Backend:** .NET Core for the microservices, Docker for containerization.
- **Database:** SQL Server (user and metadata), MongoDB (playlists).
- **Communication:** RabbitMQ (asynchronous messaging).
- **API Gateway:** Kong Gateway for reverse proxy and request routing.
- **Version Control:** GitHub for version control and collaboration.

Customer Interaction: We gathered feedback from potential users (both content creators and listeners) to design a user-friendly interface and an efficient audio management system. Their feedback informed the design of features like playlists, audio segmentation, and streaming.

2.3. Challenges and Solutions

- **Challenge 1: Problem:** Streaming large audio files without performance lags during playback.
    - **Solution:** We implemented an audio chunking system where audio files are divided into small segments. This allows for quick loading and seamless playback as the next chunk is pre-loaded while the current one is playing.
- **Challenge 2: Problem:** Ensuring communication between services without causing bottlenecks.
    - **Solution:** We used RabbitMQ to enable asynchronous messaging between microservices. This allowed for better scaling and prevented any single service from becoming a bottleneck.
- **Challenge 3: Problem:** Handling peak traffic without compromising system performance.
    - **Solution:** The microservices architecture allowed us to scale individual services based on traffic demands. For example, we could scale the Audio File Service independently if there was a surge in playback requests.
- **Challenge 4: Problem:** Ensuring that a failure in one service doesn't bring down the entire system.
    - **Solution:** The microservices are loosely coupled, meaning that a failure in one service (e.g., Playlist Service) does not affect others. Additionally, the system is designed to reroute requests to other instances of the service if one instance fails.
- **Challenge 5:** When attempting to build the Flutter project in Android Studio, an error message appeared stating that the build was unsuccessful, suggesting the use of

flutter doctor. Additionally, the emulator was not launching properly, which prevented testing the app.

- **Solution**:
  The issue was resolved by opening Android Studio as a system administrator. This allowed the build to proceed successfully, and the emulator was able to launch and run the Flutter app without further issues.
-

2.4. Results and Conclusions

- **Project Objectives**: We successfully met the objectives by building a scalable, flexible, and fault-tolerant audio sharing platform. The microservices architecture allowed us to scale the system efficiently, even during high traffic periods.
- **Performance**: The system was able to stream audio efficiently, with minimal latency, thanks to the chunking system. This was tested under various loads, and the performance remained stable.
- **Decision-Making**: Key decisions, such as using microservices, asynchronous messaging, and audio chunking, were made to ensure the system's scalability and performance. These decisions were validated through testing and feedback from users.
- **Challenges Overcome**: The main challenge was ensuring smooth communication between microservices and efficient streaming. These were resolved by the use of RabbitMQ for messaging and chunking for audio files.

2.5. Lessons Learned

- **What Worked Well:** The decision to use microservices and Docker allowed for easy scaling and maintainability. This structure gave us the flexibility to make changes to individual services without affecting the whole system.
- **What Could Have Been Improved:** In hindsight, we could have optimized our caching strategy earlier, which would have further reduced the load on certain services during peak times.
- **Changes for Future Development:** We plan to add more detailed logging and monitoring systems to catch issues earlier and optimize performance in real-time.

2.6. Meeting Project Criteria

- We met the project's technical and functional goals. The system is scalable, modular, and provides a seamless user experience.
- The system handled high traffic loads during testing, and we were able to deploy it successfully in a Virtual Private Cloud environment.

References

[1] **Flutter Documentation**:

The Flutter team. (n.d.). *Flutter Documentation*. Retrieved from https://flutter.dev/docs

[2] **Docker Documentation**:

Docker, Inc. (n.d.). *Docker Documentation*. Retrieved from https://docs.docker.com

[3] **RabbitMQ Documentation**:

Pivotal Software. (n.d.). *RabbitMQ Documentation*. Retrieved from https://www.rabbitmq.com/documentation.html