A blue and black logo

Description automatically generated

Atypon Java and DevOps (May 2024)

***Docker Assignment***

*Done by Bisher Alqudah*

[*bishr.alqudah@gmail.com*](mailto:bishr.alqudah@gmail.com)

*Section 1*

# Table of Contents

[Table of Contents 2](#_Toc175885582)

[Introduction 3](#_Toc175885583)

[Problem Definition 3](#_Toc175885584)

[1. Accessing My AWS Account 4](#_Toc175885585)

[1.1. Create a User Group with Appropriate Permissions 4](#_Toc175885586)

[1.2. Attach Policies to the User Group 4](#_Toc175885587)

[1.3. Create a User and Add It to the Group 5](#_Toc175885588)

[1.4. Create an Access Key for the User 5](#_Toc175885589)

[1.5. Configure AWS CLI Using Access Key 6](#_Toc175885590)

[2. Implementation of the Network 7](#_Toc175885591)

[2.1. Create a VPC named "MyVPC" with IPv4 CIDR 10.0.0.0/16 7](#_Toc175885592)

[2.2 Create Subnets 8](#_Toc175885593)

[2.3 Create and Attach an Internet Gateway 12](#_Toc175885594)

[2.4 Allocate Elastic IP Address 14](#_Toc175885595)

[2.5 Create a NAT Gateway and Assign an Elastic IP 15](#_Toc175885596)

[2.6 Create and Configure Route Tables 16](#_Toc175885597)

[2.7 Create Security Groups in MyVPC 20](#_Toc175885598)

[2.8 Create Key Pairs 25](#_Toc175885599)

[2.9 Create EC2 Instances 26](#_Toc175885600)

[2.10 Create Target Groups and Register Targets 30](#_Toc175885601)

[2.11 Create Load Balancer 33](#_Toc175885602)

[3. Testing the Network 37](#_Toc175885603)

[3.1 Testing SSH Access to the SSHBash-EC2 Instance 37](#_Toc175885604)

[3.2 Testing the Connection Between the Bastion SSHBash-EC2 and the Frontend Subnet 39](#_Toc175885605)

[3.3 Verifying NAT Gateway Functionality 40](#_Toc175885606)

[3.4 Testing Web Server Access via the Load Balancer 42](#_Toc175885607)

[3.5 Testing Connection Between WebApp1-EC2 and MySQL-EC2 45](#_Toc175885608)

[3.6 Connect the WebApp Server to the DB and get data 46](#_Toc175885609)

[Challenges and Solutions in Using AWS 50](#_Toc175885610)

# Introduction

In today’s digital landscape, cloud computing has revolutionized how organizations deploy and manage their IT infrastructure. Amazon Web Services (AWS) is a leading cloud service provider that offers a wide range of services to build, deploy, and scale applications efficiently. With its global network of data centers, AWS provides reliable, scalable, and cost-effective cloud solutions, enabling businesses to innovate faster without the heavy lifting of traditional IT infrastructure.

This assignment explores key AWS services and concepts such as Virtual Private Cloud (VPC), Elastic Load Balancing, EC2 instances, NAT Gateways, and security configurations. By leveraging these services, we can build a secure and robust network architecture that mimics real-world production environments. The steps outlined in this report demonstrate how to set up and configure a network infrastructure on AWS, highlighting the power and flexibility of cloud-based solutions.

# Problem Definition

This assignment focuses on setting up a secure, scalable, and highly available network architecture on AWS. The objective is to design and implement a Virtual Private Cloud (VPC) that supports a multi-tier web application, ensuring proper security configurations, efficient traffic management, and robust network connectivity.

The implementation involves creating a custom VPC with multiple subnets, including public and private subnets, to host different components of the application. Key elements of the network include an Internet Gateway (IGW) for Internet access, a NAT Gateway for secure outbound traffic from private subnets, and a bastion host for secure SSH access to private resources.

In addition, we deploy and configure essential AWS services, including EC2 instances, security groups, an Application Load Balancer (ALB), and a MySQL database, to simulate a real-world environment. The ALB ensures load distribution across multiple web servers, enhancing the application's availability and fault tolerance.

Throughout this assignment, each step is carefully documented, with detailed explanations of the commands and configurations used. The network is thoroughly tested to validate the setup, ensuring all components communicate as expected and that security measures are properly enforced.

The completion of this assignment demonstrates the ability to design and implement a secure, efficient, and resilient network architecture on AWS, laying the groundwork for deploying scalable and secure applications in a cloud environment.

# Theoretical Solution

The problem is about building a microservices-based system for uploading and streaming videos. Each service is containerized using Docker, which allows you to isolate each microservice while ensuring ease of deployment, scaling, and management. The main services include:

1. **Upload Video Service**: Responsible for allowing authenticated users to upload videos, with metadata stored in a MySQL database and the actual video stored via a File System Service.
2. **Authentication Service**: Validates user credentials.
3. **MySQL DB Service**: Stores video metadata such as video names and paths.
4. **File System Service**: Manages reading and writing video files to storage.
5. **Video Streaming Service**: Streams the uploaded videos to authenticated users.

Each of these services runs independently in its own container, allowing for better separation of concerns, flexibility in scaling, and clear communication between services using HTTP requests.

# 2. Practical Solution with Code and Explanation

## 2.1. Upload-Video-Service Explanation

To effectively solve the problem of building a containerized video streaming system with an **Upload Video Service**, I took a modular approach, which ensures each service is isolated and has its own responsibility. In this case, the **Upload Video Service** focuses solely on allowing users to log in, upload videos, and store metadata in a MySQL database. Here’s how I tackled this assignment step by step, explaining the theoretical aspects, code details, and defending why this is the best approach.

**Problem Breakdown:**

The task requires creating a microservice that allows users to upload videos after logging in. We need the following functionalities:

1. **Login system**: Ensures only authenticated users can upload videos.
2. **File handling**: Allows users to upload video files.
3. **Communication with other services**: Communicates with the **Authentication Service** for user validation and the **File System Service** for storing video files.
4. **Database interaction**: Stores metadata (like video file names and paths) in a MySQL database.

This system is implemented in a containerized environment using Docker, ensuring scalability, flexibility, and ease of deployment.

The **Upload Video Service** is the web service that interacts with users to enable video uploads. The service performs multiple responsibilities such as authenticating users, handling file uploads, and storing video metadata in the database.

### 2.1.1 Dockerfile

The Dockerfile for the Upload Video Service ensures that the necessary environment is created for the service to run.

**Code:**

A screenshot of a computer

Description automatically generated

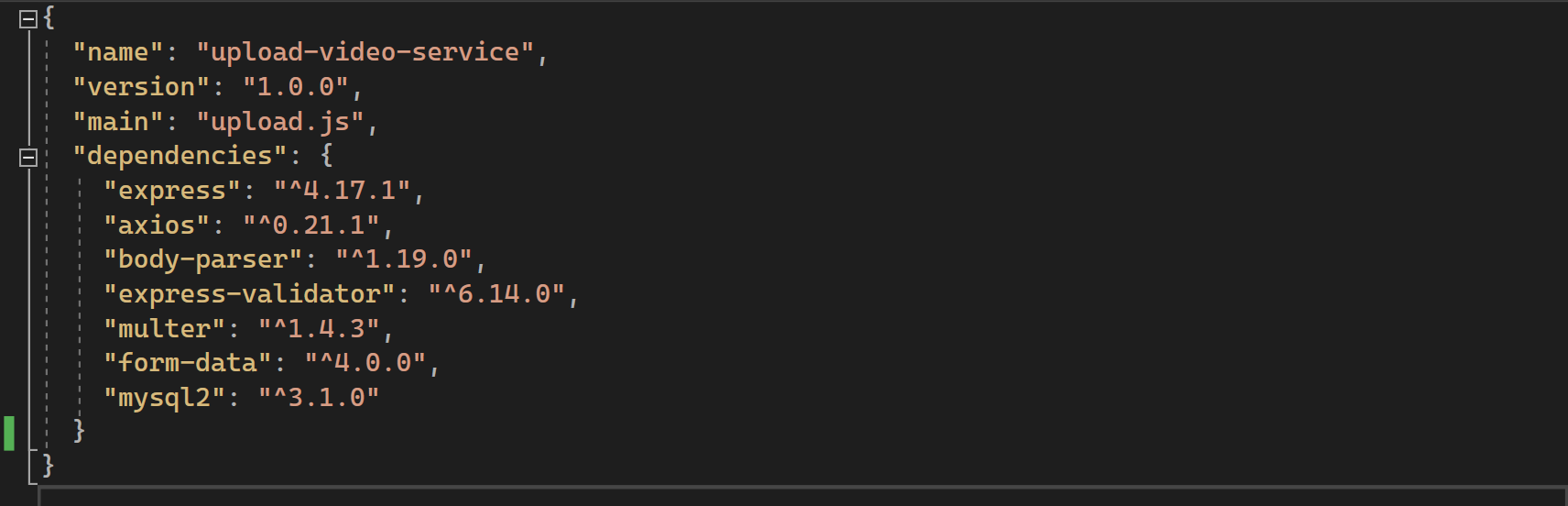
**Explanation**:

* **FROM node:14**: The base image is a lightweight Node.js environment that includes everything needed to run a Node.js application.
* **WORKDIR /usr/src/app**: Sets the working directory inside the container. This helps with organizing files and making the Docker image more maintainable.
* ***COPY package.json ./*\*** and **RUN npm install:** These commands ensure that the Node.js dependencies (like express, multer, etc.) are installed inside the container.
* **EXPOSE 3001**: Opens port 3001 for the **Upload Video Service** to listen for incoming HTTP requests.
* **CMD ["node", "upload.js"]**: Runs the main application logic, which is defined in the upload.js file.

### 2.1.2 package.json

The package.json file specifies the service's dependencies:

**Code:**



**Explanation**:

* The service depends on several libraries:
  + **Express**: To handle routing and server logic.
  + **Axios**: To make HTTP requests to the Authentication and File System services.
  + **Multer**: To handle file uploads.
  + **MySQL2**: To interact with the MySQL database.

### 2.1.3 upload.js

This file is the main application logic for the **Upload Video Service**.

**Serving Static Pages**:

* /: Serves the main welcome page where users are prompted to log in.
* /login: Serves the login page where users enter their credentials.

A screen shot of a computer code

Description automatically generated

**Login Route**:

* Handles user login by sending the entered credentials to the **Authentication Service**.
* If authentication succeeds, the user is redirected to the upload page. If not, an error message is displayed.

A screen shot of a computer screen

Description automatically generated

**Explanation**:

* The GET and POST routes handle the login and redirection to the upload page.
* After a user submits login credentials, they are sent to the Authentication Service (http://auth:3002/authenticate) via a POST request using Axios.
* If authentication is successful, the user is redirected to the upload page; otherwise, an error message is displayed.

Next, the video upload functionality:

**Handling Video Upload**:

* This route is responsible for receiving the video file, sending it to the **File System Service**, and storing video metadata in a MySQL database.

A screen shot of a computer program

Description automatically generated

A screen shot of a computer program

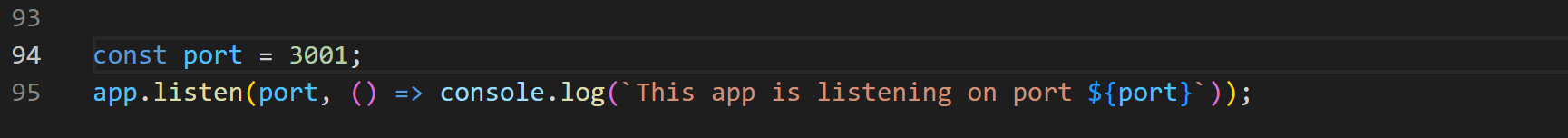
Description automatically generated

A screen shot of a computer code

Description automatically generated

A screen shot of a computer code

Description automatically generated



**Explanation**:

* This code handles the video upload process. It uses Multer to handle file uploads and temporarily stores the uploaded files in the temp directory.
* It then sends the video to the **File System Service** for storage using Axios.
* Once the file is successfully uploaded, the video's metadata (name and storage path) is inserted into the MySQL database using the mysql2 library.
* This service communicates effectively with other microservices (Authentication and File System) to complete its tasks.

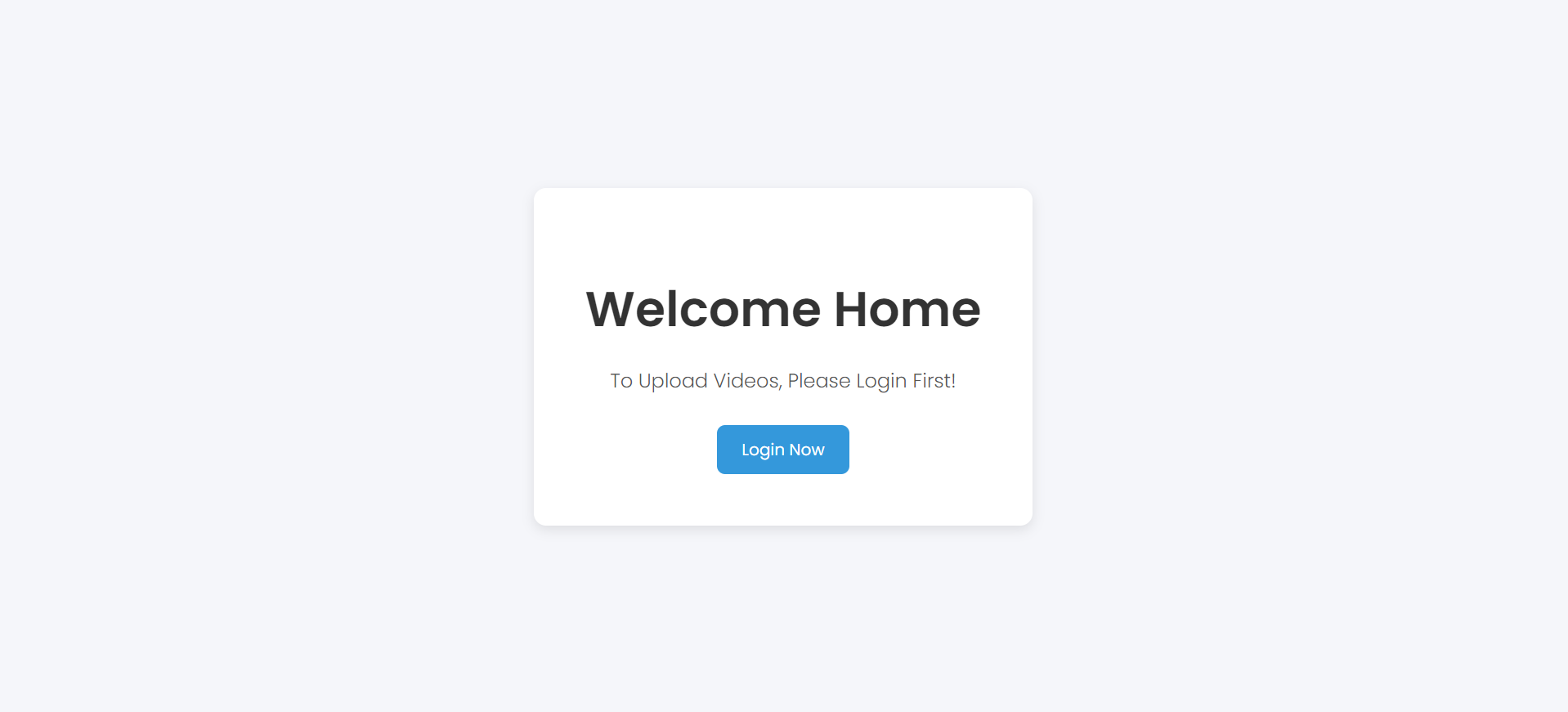
### 2.1.4 index.html:

* This is the home page that welcomes users and directs them to the login page.

A black screen with white text

Description automatically generated

* It's simple but effective in directing user flow.



**Justification:**

The landing page establishes a clear call to action, prompting users to log in. This approach adds a layer of security by ensuring only authorized users can access the upload functionality.

### 2.1.5 login.html:

* This form allows users to enter their credentials.

A login screen shot

Description automatically generated

* The form sends a POST request to /login on the **Upload Video Service**.

A screen shot of a computer

Description automatically generated

**Justification:**

A simple form-based authentication approach makes the system intuitive and easy to use. Additionally, using form validation prevents incorrect or unsafe data from being submitted.

### 2.1.6 upload.html:

* After logging in, users are redirected to this page, where they can upload their video files.

A screenshot of a video

Description automatically generated

* The form sends the video file to /upload-video.

A screen shot of a computer program

Description automatically generated

**Justification:**

This interface provides a straightforward method for video submission. Since file uploads can be large, this service would integrate with a back-end storage solution (like the File System Service) to handle large files securely.

### 2.1.7 Why My Solution Works:

1. **Separation of Concerns**: Each microservice in this system (authentication, file handling, MySQL DB, etc.) has a well-defined role, making the system modular, easier to debug, and maintain.
2. **Security**: By placing authentication before file upload, I ensure that only authorized users can access the upload functionality, reducing the risk of abuse.
3. **Reliability**: Docker ensures that the environment remains consistent, regardless of where it's deployed. The services can be scaled independently, and if one fails, it doesn't necessarily bring the others down.
4. **Code Simplicity**: The use of tools like Express.js and Multer keeps the code simple yet effective for handling HTTP requests and file uploads, respectively.

### 2.1.8 Why This is the Best Solution:

* **Scalability**: Using Docker, each service can be scaled independently. If we need more instances of the upload service, we can easily run multiple containers.
* **Flexibility**: By separating the logic into different microservices, this solution can easily adapt to new features or updates in the future without affecting the entire system.
* **Security**: Only authenticated users can upload videos. This ensures that unauthorized users are blocked from accessing the service.
* **Reliability**: The use of error handling and external services like MySQL and file storage ensures that even if one part of the system fails, it won’t crash the entire service.

## 2.2. Authentication-Service

The Authentication Service is responsible for handling user login and credential validation. I chose Node.js for its speed and scalability in handling asynchronous operations, which is particularly beneficial for services like authentication.

The Authentication-Service folder contains the Docker setup, the package manifest, and the logic for the authentication microservice. Here's a breakdown of how everything works together:

### 2.2.1 Dockerfile

The Dockerfile is a blueprint for containerizing the authentication service. Here's a step-by-step breakdown of the key components:

A computer code with black text

Description automatically generated

* **FROM node:14**: This pulls the Node.js version 14 base image to build the container.
* **WORKDIR /usr/src/app**: This sets the working directory inside the container.
* \**COPY package*.json ./\*\*: Copies the package.json and package-lock.json (if available) to the working directory.
* **RUN npm install**: Installs all dependencies listed in the package.json file.
* **COPY . .**: Copies the rest of the application source code into the container.
* **EXPOSE 3002**: Exposes port 3002, where the authentication service will be available.
* **CMD ["node", "authService.js"]**: Starts the Node.js application by running the authService.js file.

**Explanation:**

The Dockerfile starts by using the official Node.js image (node:14). It then sets up the working directory and installs the necessary dependencies specified in package.json. After copying the source code, the service is set to run on port 3002, exposing it for communication with other services.

**Justification:**

Using Docker ensures that the authentication service is isolated, meaning changes to the environment, dependencies, or code will not affect the other services. The Node.js base image (node:14) ensures compatibility with the code and provides an optimal runtime environment.

### 2.2.2 package.json

This is the configuration file for managing the Node.js project and its dependencies:

A screenshot of a computer

Description automatically generated

* **name**: The name of the project, authentication-service.
* **version**: Project version (1.0.0).
* **main**: The entry point file (authService.js).
* **dependencies**: Lists the required Node.js modules for the project:
  + express: A web framework for Node.js, used to build the API.
  + axios: A promise-based HTTP client, which may be used to make requests (though not in use here).
  + body-parser: Middleware to parse the request body.
  + express-validator: A set of middleware for validating and sanitizing request inputs.

**Explanation:**

The package.json lists the project dependencies, including express (for server handling), axios (for HTTP requests), body-parser (for parsing incoming request bodies), and express-validator (for input validation).

**Justification:**

This minimal set of dependencies allows for lightweight, fast service while maintaining good security practices, especially through validation with express-validator to prevent injection attacks or malformed input.

### 2.2.3 authService.js

This is the core authentication logic built using the Express.js framework:

* **Express and Body-Parser Setup**:
  + express() creates an Express app.
  + body-parser middleware is used to parse URL-encoded and JSON request bodies.

A computer screen shot of text

Description automatically generated

* **Validation Middleware**:
  + loginValidate: Validates that the username field is a properly formatted email and that the password contains at least 8 characters, a number, and an uppercase letter. This uses express-validator.

A screen shot of a computer screen

Description automatically generated

* **Simple Credentials Check**:
  + Hardcoded credentials:
    - **Username**: user@example.com
    - **Password**: Password123

A screen shot of a computer code

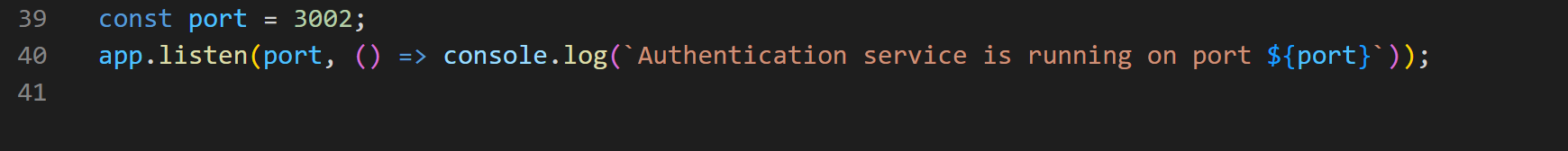
Description automatically generated

* + The /authenticate route processes POST requests. It first checks for validation errors (e.g., malformed email or weak password). If validation passes, it compares the request's credentials with the predefined ones. If they match, it responds with a success message ({ success: true }), otherwise, it returns an error.

A screen shot of a computer code

Description automatically generated

* **Error Handling**:
  + If the request contains validation errors, a 422 status code is returned with a list of error messages.
* **Port**:
  + The app listens on **port 3002**, which should match the port you configured elsewhere in your system (like in Docker Compose).



**Explanation:**

The authService.js file defines the logic for handling user authentication. It uses express-validator to validate inputs, ensuring the username is a valid email address and the password follows the required criteria (e.g., length, inclusion of numbers and uppercase letters). If the provided credentials match the predefined valid credentials (validUsername and validPassword), it returns a success message.

**Justification:**

This method offers a simple, yet secure, approach to authentication for demonstration purposes. The use of validators ensures that only well-formed and safe data is processed. The static validation against predefined credentials simplifies the system for this project without needing a more complex database lookup.

**Defense:**

I chose not to implement token-based authentication (JWT or OAuth) to keep the service lightweight and easy to understand, which aligns with the simplicity of the project. This design can easily be extended to a database-driven authentication mechanism in a production scenario.

**How It Works:**

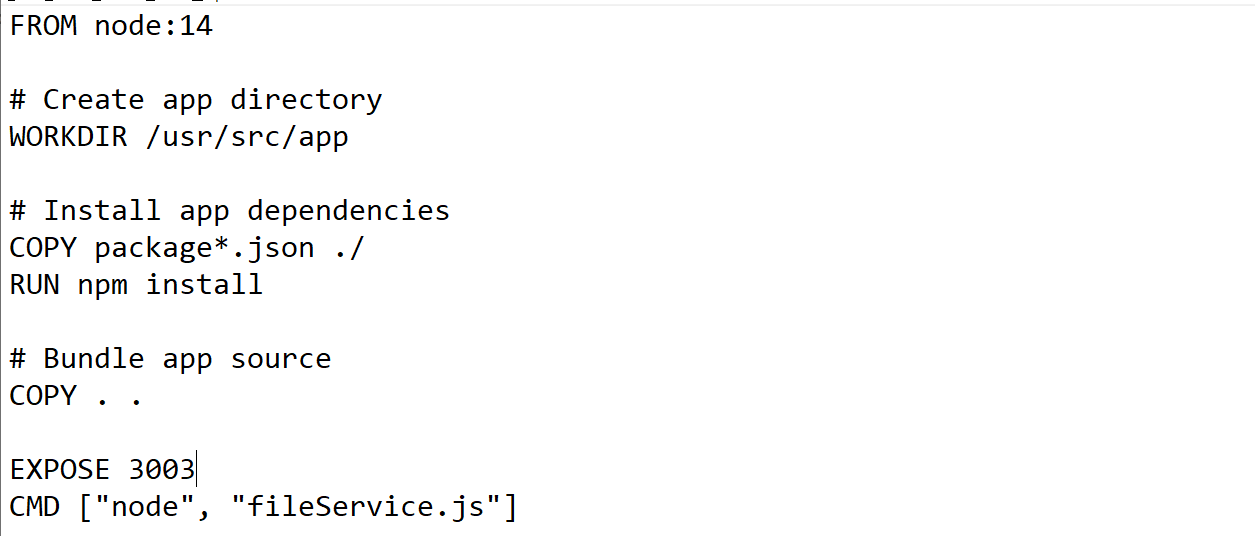
1. When a client sends a POST request to /authenticate with username and password fields, the service first checks if the input conforms to the validation rules (email format for username and strong password rules).
2. If the inputs are valid, the service compares them to the hardcoded credentials. If they match, the response confirms a successful login.
3. If the credentials are wrong, the service responds with a message indicating invalid login details.

This service is containerized using Docker, making it easy to deploy and run independently as part of your video streaming system.

## 2.3 File-System-Service

The **File System Service** is responsible for handling video uploads, storing them on the server, and allowing retrieval of the video files. This service is crucial as it ensures the system can manage video content efficiently. I used Node.js with the multer package for file uploads, along with basic file system (fs) operations to manage video files.

### 2.3.1 Dockerfile



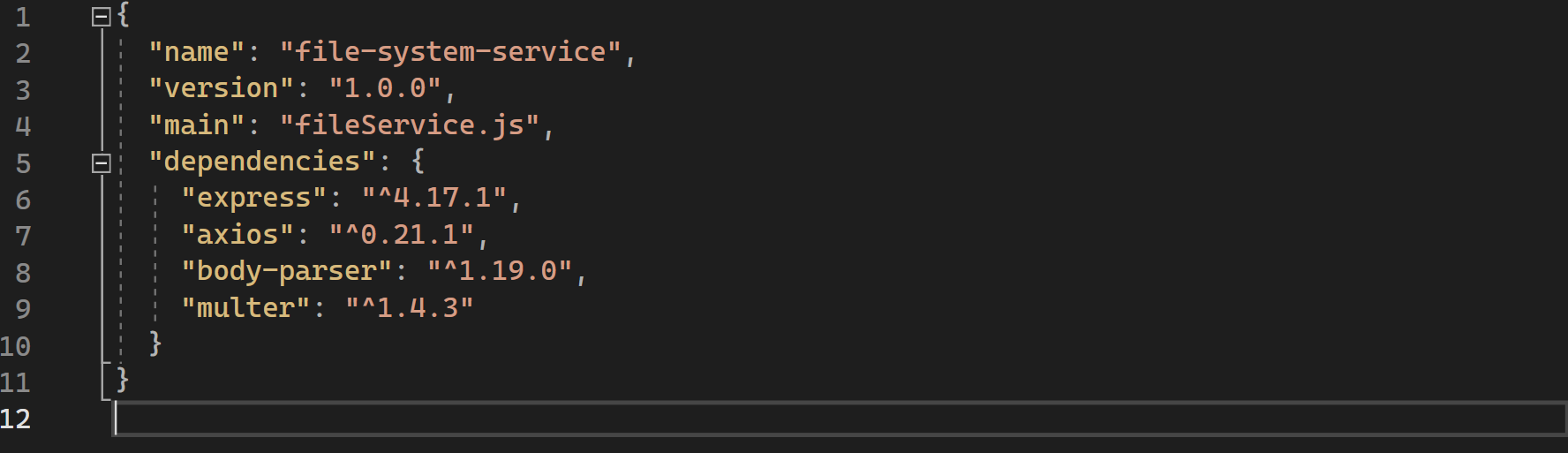
**Explanation:**

The Dockerfile builds the container image for the File System Service:

* The base image is node:14, which provides a stable Node.js environment for running the service.
* It sets the working directory to /usr/src/app.
* The package.json file is copied, and npm install is run to install the required dependencies.
* The source code is copied into the image.
* Port 3003 is exposed, allowing the service to communicate externally.
* The container runs the fileService.js file to start the service.

**Justification:** This containerization approach ensures consistency across different environments, whether development, staging, or production. By isolating the File System Service, updates or changes can be made independently without impacting the rest of the system.

### 2.3.2 package.json



**Explanation:** The package.json lists the service's dependencies:

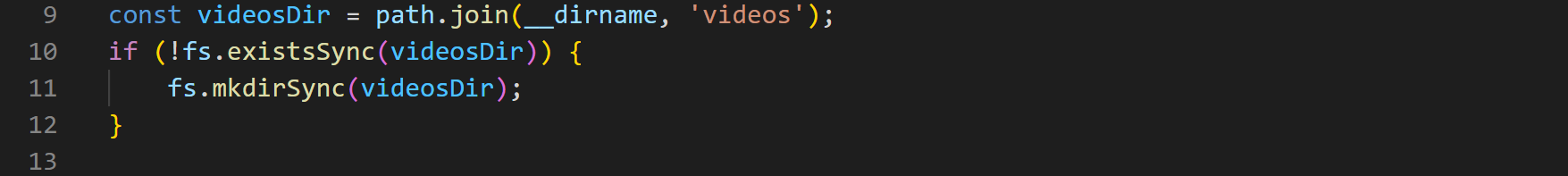
* **Express**: A minimal web framework for handling HTTP requests.
* **Axios**: For making HTTP requests if needed (though not used in this specific file).
* **Body-parser**: For parsing incoming request bodies, particularly in forms.
* **Multer**: A middleware for handling file uploads.

**Justification:** These dependencies are lightweight but powerful enough to handle the service's core functionality. Multer, in particular, simplifies the process of managing file uploads, ensuring videos are correctly handled and stored.

### 2.3.3 fileService.js

**Explanation:**

* **Directory Creation**: The service checks if the videos directory exists; if not, it creates it. This directory stores all uploaded video files.



* **File Upload**: Using Multer, the service accepts video uploads via a POST request to /upload-video. The video is stored in the videos directory under its original name.

A screen shot of a computer program

Description automatically generated

* **List Videos**: A GET request to /list-videos returns a list of all uploaded videos. The fs.readdir() function reads the contents of the videos directory and returns the filenames as a JSON response.

A screen shot of a computer code

Description automatically generated

* **Serve Video**: The service also allows users to access individual videos via a GET request to /video/:filename. The res.sendFile() method serves the video file to the client.

A screen shot of a computer code

Description automatically generated

**Justification:**

* **File Management**: By using the file system (fs) and Multer, I can easily manage video uploads and serve them without needing a more complex storage system like a database or external cloud storage (which could be added later for scalability).
* **Simplicity**: Keeping the files in a local directory ensures simplicity and ease of demonstration. For larger-scale systems, this can be extended to use AWS S3 or another cloud storage solution.

### 2.3.4 Defense of the File System Service Design

* **Multer for File Handling**: Multer is well-suited for handling multipart/form-data, especially in cases where files are being uploaded. It's lightweight and integrates seamlessly with Express, making it an ideal choice for this service.
* **Local File Storage**: I opted for local file storage (fs module) to simplify the demonstration. In a production environment, the file handling logic could be extended to store videos on a distributed file system or cloud storage, such as AWS S3. The design is modular, so this change could be implemented without much disruption to the existing codebase.
* **REST API Design**: The service exposes simple endpoints (/upload-video, /list-videos, /video/:filename) that make it easy to interact with from both the frontend and other services (such as the video streaming service). This ensures that the architecture remains modular and extensible.
* **Extendability**: This service is designed to handle simple file uploads and listings, but it can be extended to include more advanced features, such as file metadata storage in a database, video transcoding for different formats, or integration with content delivery networks (CDNs) for efficient streaming.

**Conclusion**

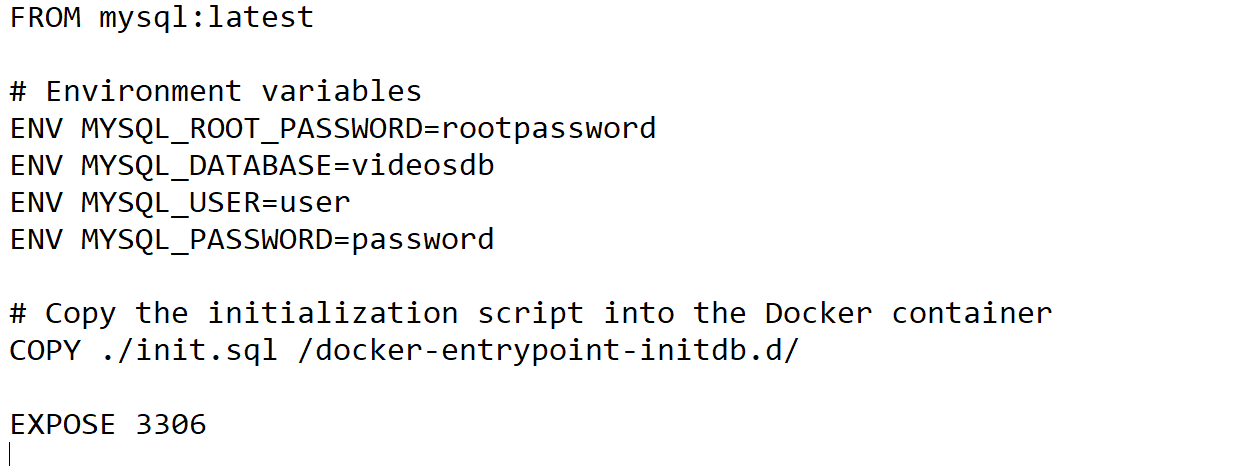
The File System Service is an integral part of the overall Dockerized Video Streaming System. It efficiently manages video uploads, stores them in a local directory, and allows users to retrieve or list videos through a simple API. The use of Docker containers ensures that this service is isolated and can be easily scaled or modified independently of other services.

In a real-world scenario, this service could be extended to integrate cloud storage or advanced file management features, but for the scope of this project, the current design meets the requirements while keeping the implementation straightforward and understandable.

## 2.4 MySQL-DB-Service

The **MySQL-DB Service** is responsible for managing and storing metadata about the uploaded video files. This includes storing the video's name and its path on the file system. By using a MySQL database, the system ensures that it can easily query and manage video-related data.

### 2.4.1 Dockerfile



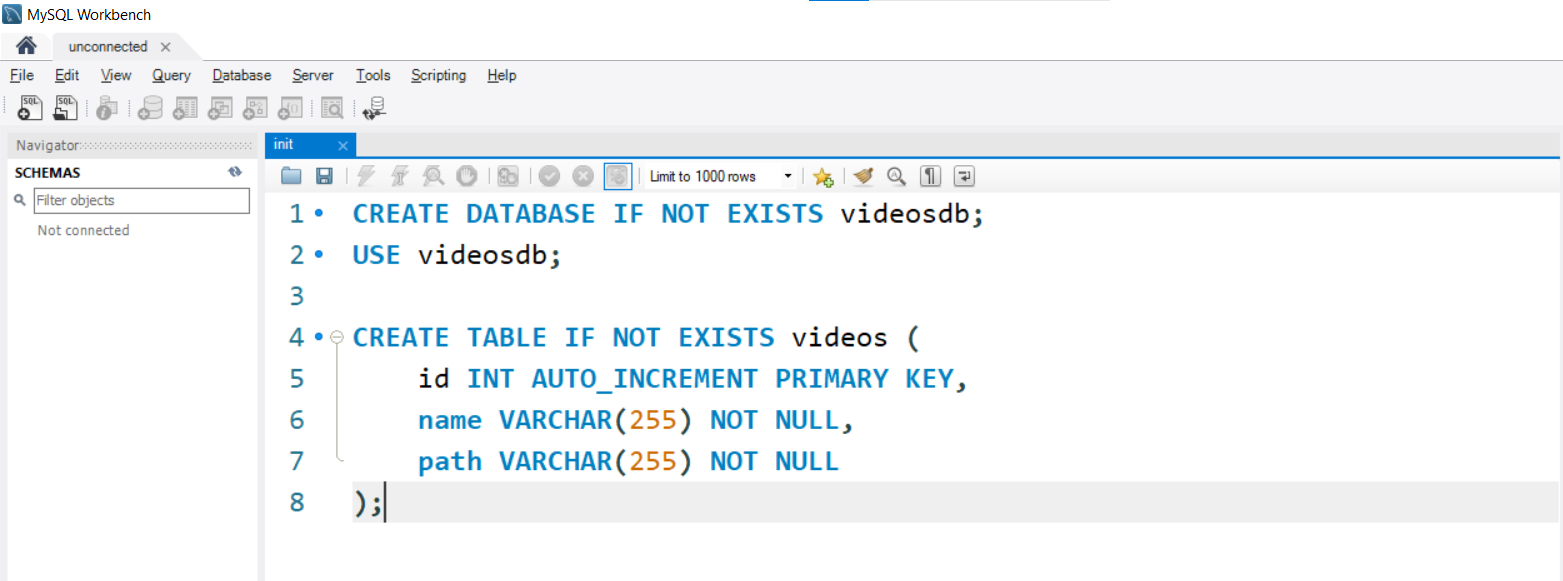
**Explanation:**

* **Base Image**: This service uses the official mysql:latest image as the base. This ensures that we are running a stable and up-to-date version of MySQL in the container.
* **Environment Variables**: The environment variables are used to configure the MySQL instance.
  + MYSQL\_ROOT\_PASSWORD: Sets the root user password.
  + MYSQL\_DATABASE: Defines the default database (videosdb) to be created at startup.
  + MYSQL\_USER and MYSQL\_PASSWORD: These define a regular user (user) with the provided password, which can be used to connect to the database.
* **Database Initialization**: The init.sql file is copied into the MySQL container's /docker-entrypoint-initdb.d/ directory. MySQL automatically runs any .sql scripts placed in this directory when the container is started, ensuring that the videosdb database and its table are created.
* **Expose Port**: The container exposes port 3306, the default port for MySQL, allowing other services to connect to the database.

**Justification:**

Using Docker for the database service ensures that the database is portable and consistent across environments. Environment variables provide flexibility, allowing credentials and database names to be customized without hard-coding them into the image. Copying the initialization SQL script into the container guarantees that the necessary tables and schema are set up automatically.

### 2.4.2 init.sql



**Explanation:**

* **Create Database**: The script starts by ensuring that the videosdb database exists. If it doesn't, it creates it.
* **Create Table**: The script then switches to the videosdb database and creates a videos table, if it doesn't already exist. This table has the following columns:
  + id: An auto-incrementing integer that serves as the primary key.
  + name: A string that stores the video's name.
  + path: A string that stores the video's file path in the file system.

**Justification:**

This schema is simple yet sufficient for the project. It stores basic metadata about each video, specifically the name and its file path. This design allows for easy expansion—additional fields (such as upload time, file size, or video format) can be added later if needed.

### 2.4.3 Defense of the MySQL-DB Service Design

* **Containerization**: Running the database in a Docker container isolates it from the rest of the system, ensuring portability and easy scalability. This allows the database to be managed independently of the application code.
* **Initialization Script**: Including the init.sql file ensures that the required database and table are always set up when the container starts. This is particularly useful for quickly spinning up new instances of the system, such as in CI/CD pipelines or when testing in different environments. There's no need for manual setup of the database schema.
* **Security**: The use of environment variables for the database credentials allows for flexibility and security. In production environments, these credentials can be securely injected using Docker secrets or environment variable management services.
* **Database Schema**: The table schema (videos table) is designed to store key video metadata in a relational manner. By storing the video path, this service ties in with the **File System Service**, which manages the actual video files. The relational database design ensures that this metadata can be efficiently queried and managed.
* **Scalability**: Although the current setup uses a single MySQL container, this design can easily be expanded. For example, in a more advanced production environment, the MySQL service could be replaced with a managed database like Amazon RDS, or replicated and scaled using MySQL replication or clustering.

**Conclusion:**

The **MySQL-DB Service** forms the backbone of the metadata storage for your Dockerized Video Streaming System. By using MySQL, the system benefits from a reliable and widely supported relational database system that can easily handle the video metadata. The use of Docker ensures that the database environment is consistent and easily deployable across different platforms.

In this solution, the MySQL container is seamlessly integrated with the other services in the system. The schema is simple and modular, allowing for future growth and additional features. This service plays a crucial role in enabling the system to manage the data associated with the video content in an efficient and organized manner.

## 2.5 Stream-Video-Service

**Purpose:**

The Stream-Video-Service is responsible for providing users with the ability to log in, fetch a list of videos, and stream a selected video from the file system service. It connects to the MySQL database to retrieve the metadata of the videos and interacts with the authentication service to validate users.

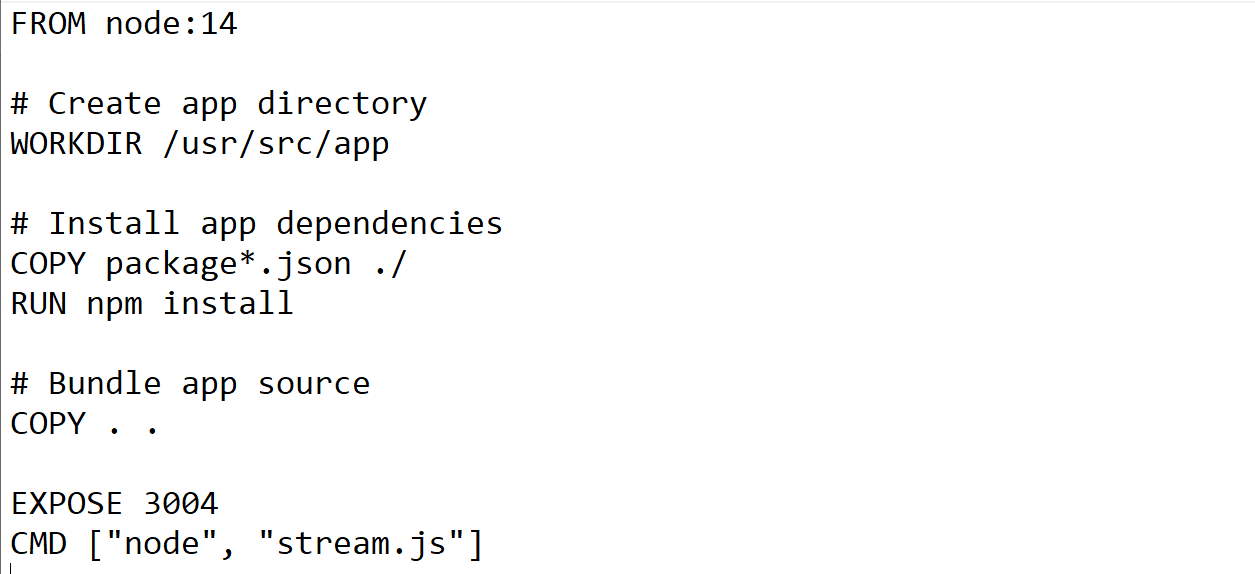
**Components:**

1. **Dockerfile**
2. **package.json**
3. **stream.js**
4. **HTML pages:**
   * index.html
   * login.html

### 2.5.1 Dockerfile

The Dockerfile defines the environment and necessary steps to build the Docker image for the **Stream-Video-Service**.

**Code:**



**Explanation:**

* **Base Image:** We use the official node:14 image as the base environment for running the Node.js application.
* **Working Directory:** The working directory is set to /usr/src/app inside the container, where the application code will live.
* **Install Dependencies:** Dependencies from package.json are installed via npm install.
* **Copy Source Code:** The code is copied into the working directory.
* **Expose Port 3004:** The container exposes port 3004, where the **Stream-Video-Service** listens.
* **Run Command:** The container runs the stream.js file when started.

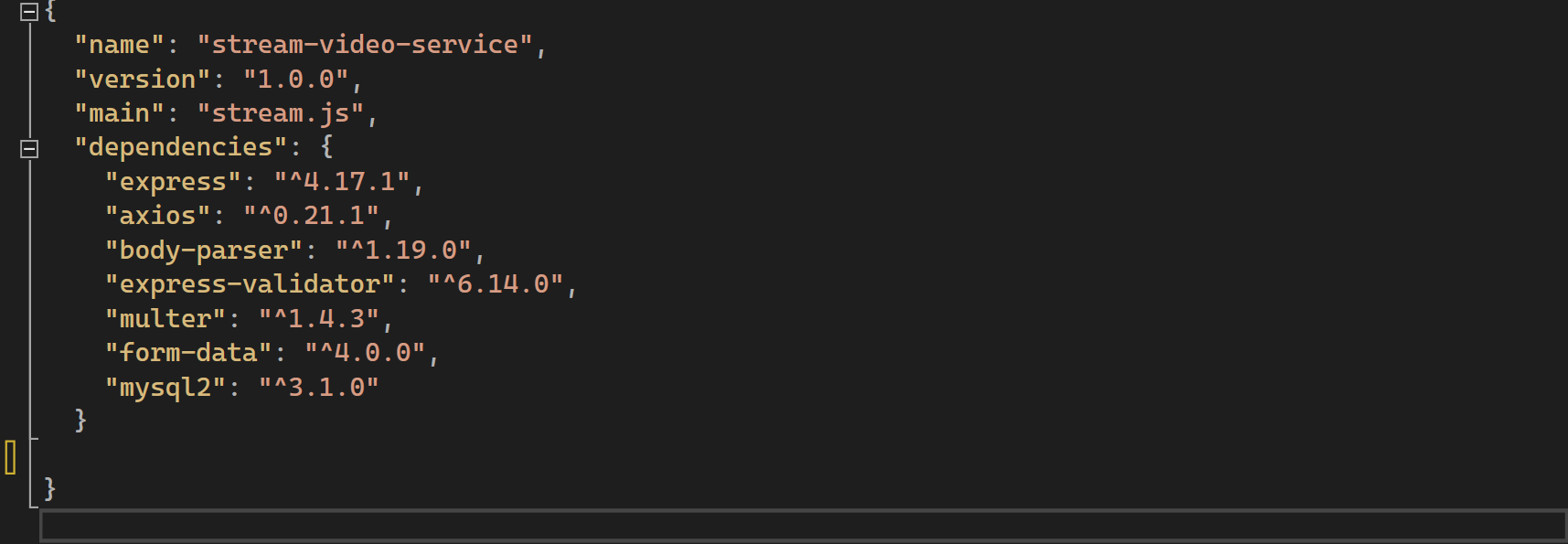
**Justification:**

* The node:14 image is lightweight and well-supported, making it suitable for a service that requires rapid HTTP response handling.
* Dependencies like express, mysql2, and axios are bundled with the application, ensuring that the service can handle HTTP requests, interact with the MySQL database, and communicate with other microservices.
* The use of EXPOSE 3004 ensures that the service is accessible on the correct port, facilitating easy communication between services.

### 2.5.2 package.json

The package.json defines the project's metadata and dependencies necessary for the service to run.

**Code:**



**Explanation:**

* **express:** A Node.js web framework for building the API endpoints and handling HTTP requests.
* **axios:** Allows the service to make HTTP requests to external services, like the authentication and file system services.
* **body-parser:** Parses incoming request bodies to extract form data (used for login credentials).
* **mysql2:** Connects to the MySQL database to fetch video metadata.
* **multer:** Though not currently in use, it handles file uploads, potentially useful if you extend the service to allow users to upload videos.
* **form-data:** Helps with handling form data, especially for file uploads (if needed).

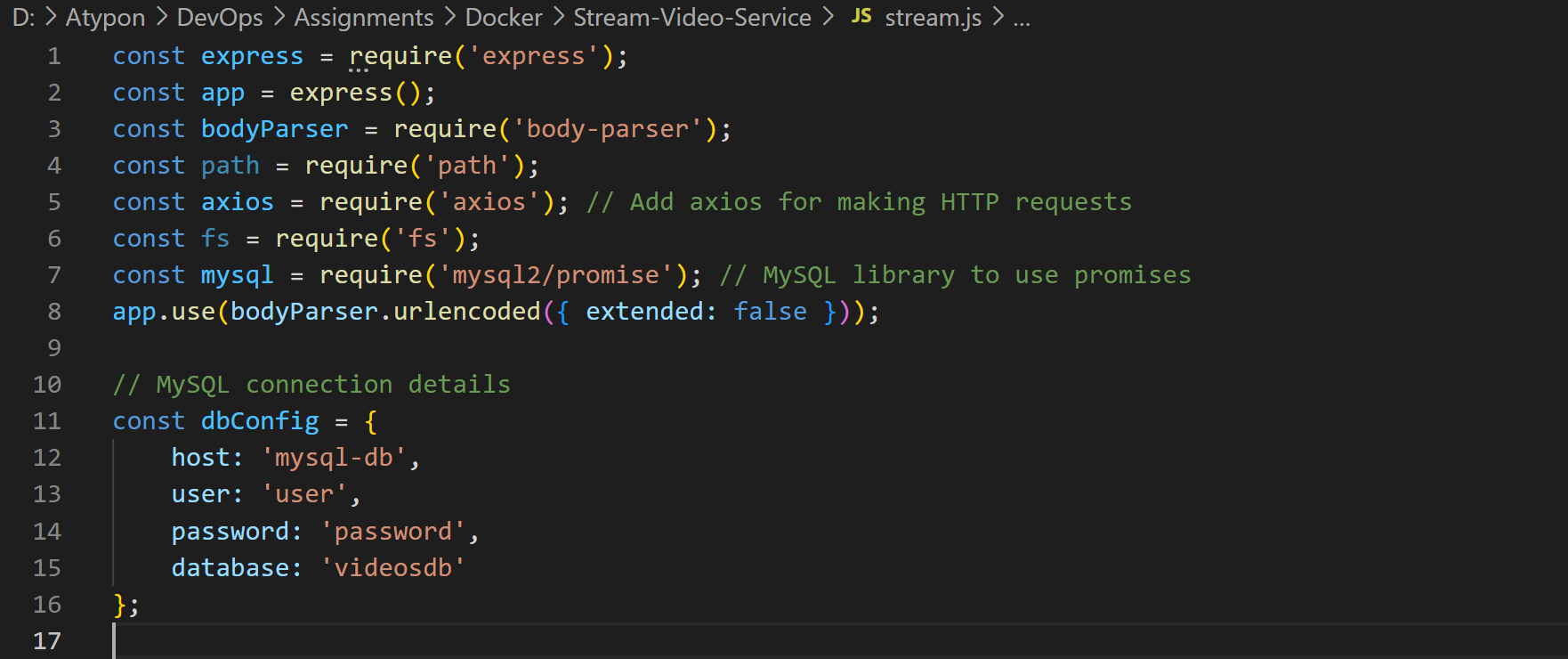
**Justification:**

* **express**: The backbone of our service, used for creating routes and handling HTTP requests.
* **axios**: Makes it easy to handle communication between microservices.
* **mysql2**: Provides a modern and promise-based way to interact with the MySQL database, ensuring smooth and asynchronous communication.

### 2.5.3 stream.js

This file contains the main logic for serving web pages, handling login, and streaming videos.

**Code:**



**Explanation:**

* The service sets up a simple express server with body-parser to handle form data.
* axios is used to send HTTP requests, particularly to the authentication and file services.
* MySQL configuration (dbConfig) allows the service to retrieve video metadata stored in the database.

**Code:**

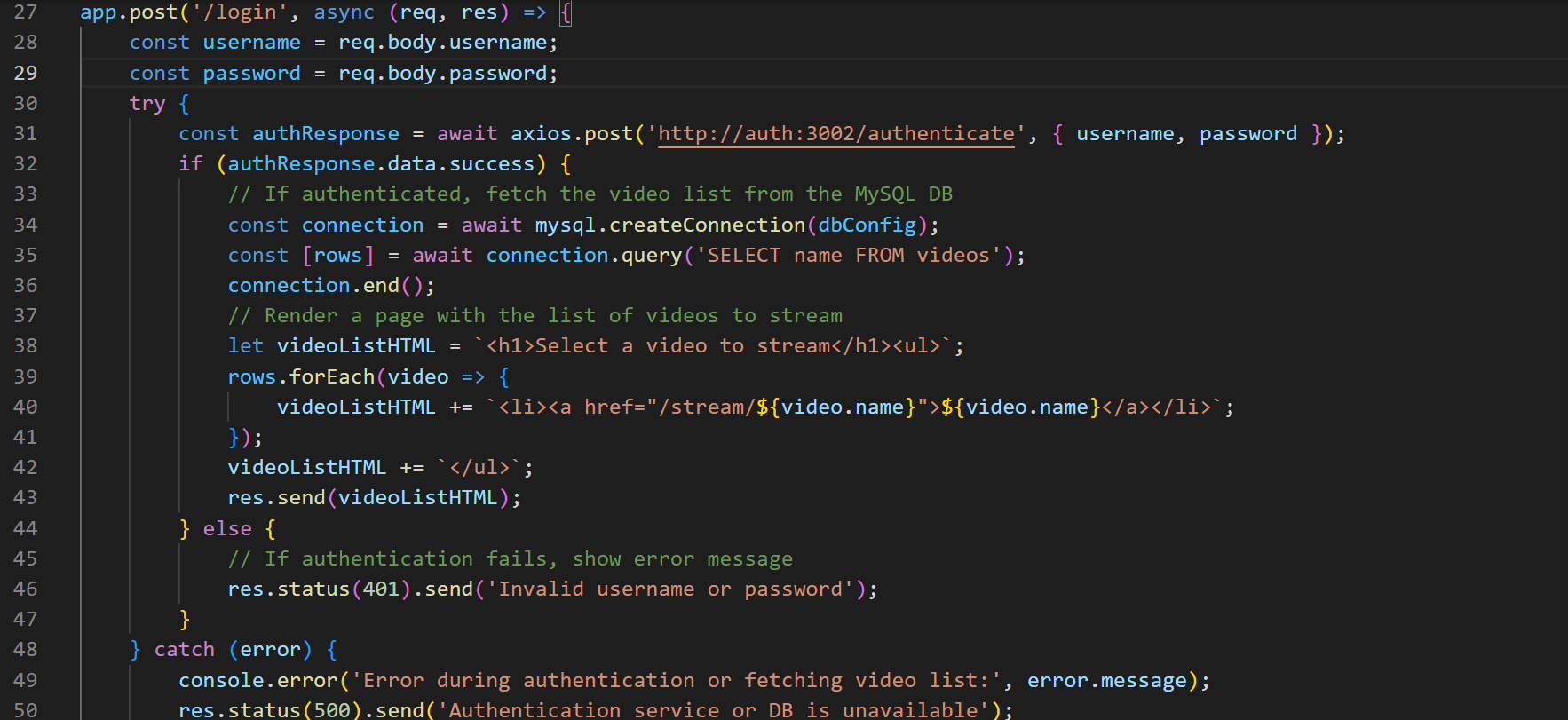
A computer screen shot of text

Description automatically generated

**Explanation:**

* These routes serve HTML files to the user. The index.html and login.html files provide a friendly user interface for the video streaming service.

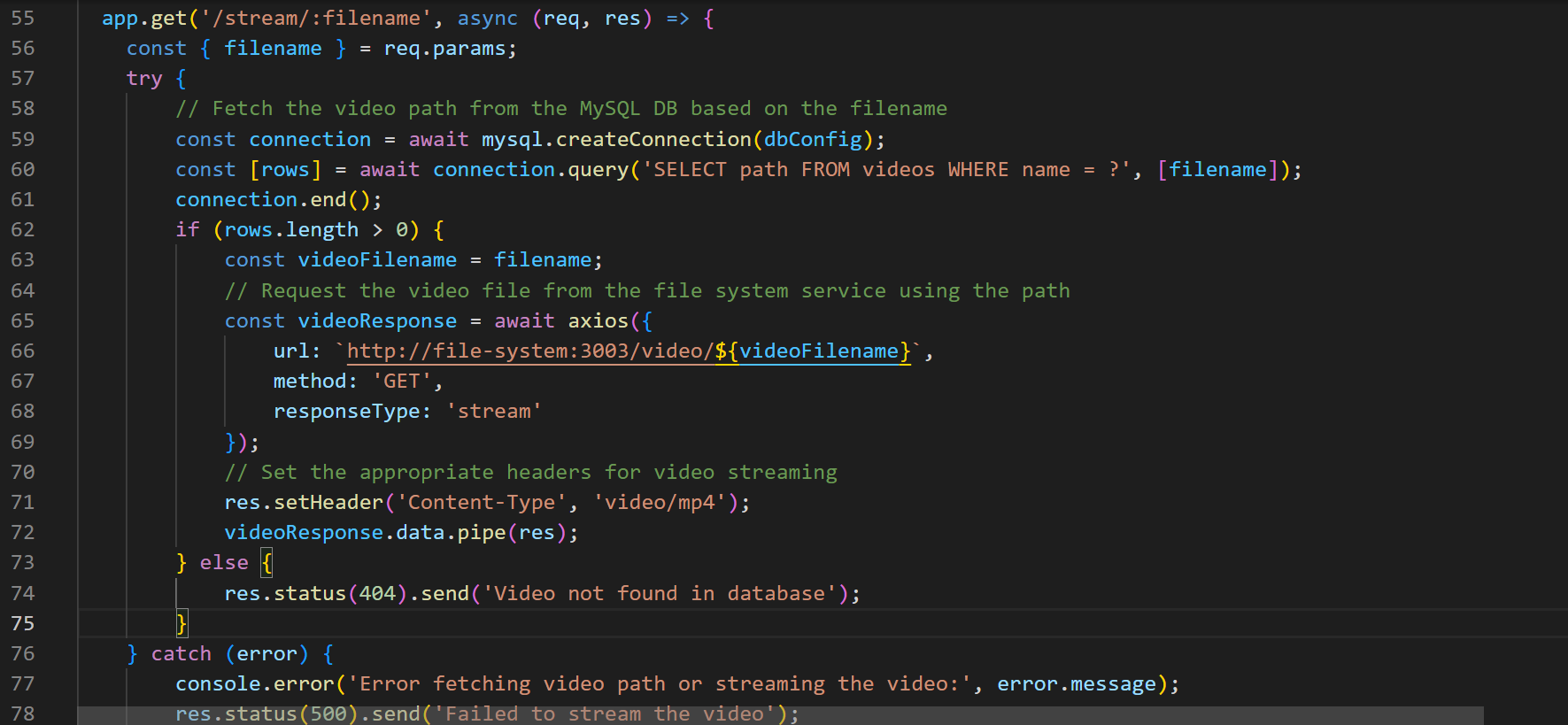
**Code:**

****

**Justification:**

* The login system is critical for security. After successfully authenticating against the **Authentication-Service**, the user is shown a list of available videos, which are retrieved from the **MySQL-DB-Service**.
* This approach ensures that only authorized users can view and stream videos, following the **Separation of Concerns** principle. By delegating authentication to a dedicated service, the overall system remains modular and maintainable.

**Code:**



**Justification:**

* The video stream process is handled efficiently. The video metadata (like the path) is stored in the **MySQL-DB-Service**, while the actual video files are stored and fetched from the **File-System-Service**.
* Using axios, the **Stream-Video-Service** fetches the file from the file service and streams it to the user. This distributed architecture enables scalability and separation of concerns, making it easier to manage large amounts of data (video files) separately from metadata (video information).
* This method ensures that the videos can be fetched and streamed efficiently without overloading the database with large file data.

**Overall Explanation:**

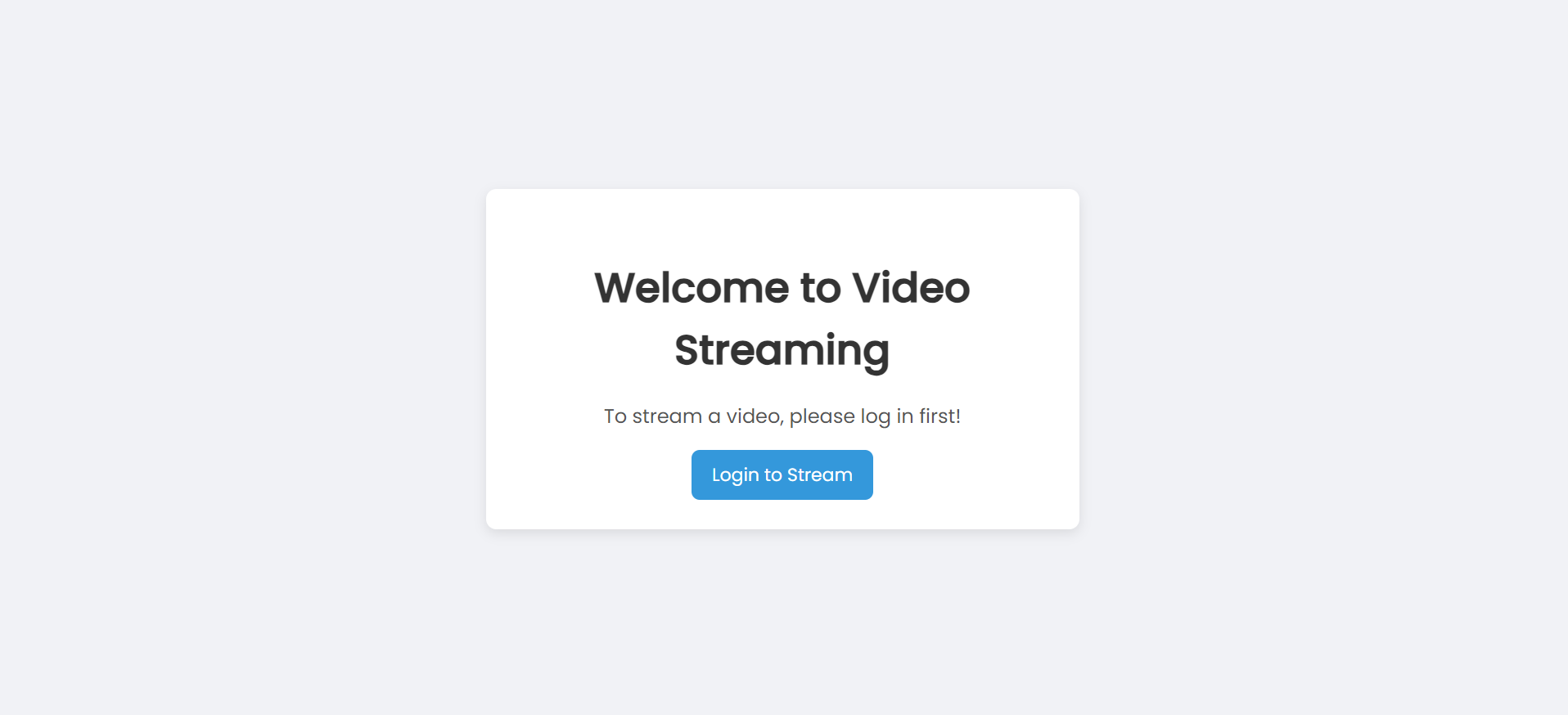
* **Login Handling:**
  + When the user submits their credentials, a POST request is made to the http://auth:3002/authenticate authentication service.
  + If the user is successfully authenticated, the service queries the MySQL database to fetch the list of available videos and displays them as links.
* **Video Streaming:**
  + When a video is selected, the service queries the MySQL database to fetch the file path, requests the video file from the **File-System-Service**, and streams the video to the user with the correct headers.
* **Error Handling:**
  + Both login and video streaming operations include error handling, such as failing to authenticate or stream videos from the file system.

### 2.5.4 HTML pages

There are two HTML pages, index.html and login.html, served to the users through **Stream-Video-Service**.

**index.html:**

This is the landing page that prompts users to log in before streaming videos.



**login.html:**

This page provides a form where users can input their username and password to log in.

A screenshot of a login screen

Description automatically generated

**Service Flow:**

1. **User visits the home page (/)**:
   * They are presented with a welcome page and a link to the login page.
2. **User navigates to the login page (/login)**:
   * They enter their credentials, which are sent to the authentication service.
3. **User authentication**:
   * If successful, a list of videos stored in the database is presented.
   * If unsuccessful, an error message is shown.
4. **User selects a video to stream (/stream/:filename)**:
   * The service queries the MySQL database for the file path.
   * The video file is streamed from the **File-System-Service**.

### 2.5.5 Defence of the Solution

* **Separation of Concerns**: Each service in the architecture has a clear and well-defined responsibility. The **Stream-Video-Service** focuses solely on user interaction and streaming logic, while the **File-System-Service** and **MySQL-DB-Service** handle file storage and database interactions, respectively.
* **Scalability**: By distributing tasks across multiple services (authentication, file storage, database management, and streaming), the system can easily be scaled by adding more instances of any service, depending on demand. For example, if streaming load increases, more instances of the **Stream-Video-Service** can be added.
* **Modularity**: The system's modular design allows for flexibility. Each service is containerized and can be developed, tested, and deployed independently. This ensures that future changes can be isolated to the service that needs modification without affecting the entire system.
* **Microservices Architecture**: The microservice-based architecture promotes independence, allowing for fault isolation. If one service (e.g., the authentication service) fails, the rest of the system can still function, reducing the risk of full system outages.
* **Security**: By keeping authentication as a separate service and requiring users to log in before streaming videos, we ensure that only authorized users can access the video content, following best practices for content delivery systems.

# 3. Testing the Network

In this section, I will demonstrate that the network configuration is working perfectly by performing a series of tests. These tests will validate the connectivity between different components of the network, ensuring that the EC2 instances, NAT gateway, and Load Balancer are functioning as expected.

## 3.1 Testing SSH Access to the SSHBash-EC2 Instance

**Objective**:  
To verify that the SSHBash-EC2 instance is correctly configured and accessible via SSH, we'll connect to it using the SSH protocol. This test ensures that the instance is accessible from the public internet and that the security group associated with the instance allows inbound SSH traffic.

**Steps**:

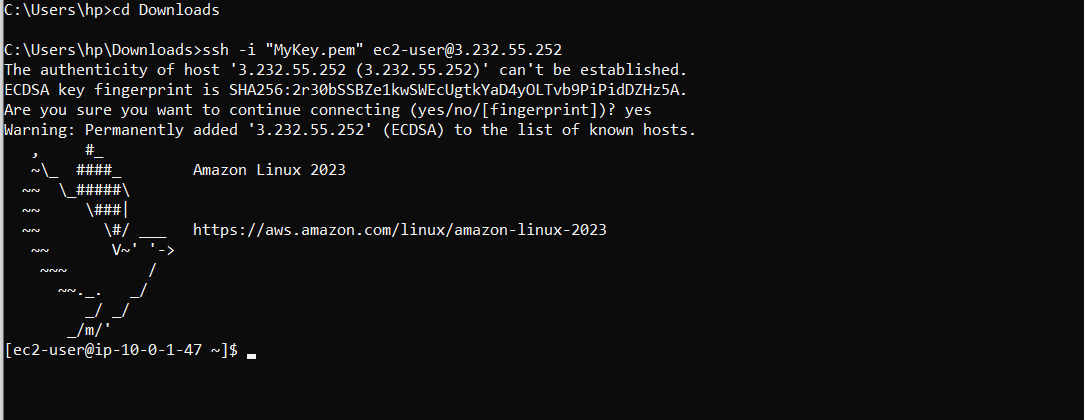
1. **Navigate to the Path Containing MyKey.pem**:
   * Before connecting to the instance, I ensure that I am in the directory where my SSH key pair (MyKey.pem) is stored. This key is required for authentication.
2. **Run the SSH Command**:
   * I will use the following SSH command to connect to the SSHBash-EC2 instance. This command specifies the SSH key to use and the public IP address of my instance.

ssh -i "MyKey.pem" ec2-user@3.232.55.252

* + **Explanation**:
    - **ssh:** The command to initiate an SSH session.
    - **-i "MyKey.pem":** Specifies the identity file, which in this case is my private key (MyKey.pem). This key is used for secure authentication.
    - **ec2-user@3.232.55.252:** The username (ec2-user) and the public IP address (3.232.55.252) of the SSHBash-EC2 instance.

1. **Expected Outcome**:
   * If the SSH connection is successful, I will be logged into the SSHBash-EC2 instance. The terminal will display a command prompt indicating that I am now operating within the instance.
   * If the connection fails, you may encounter issues such as incorrect key permissions, incorrect IP address, or misconfigured security groups. In such cases, check the SSH key permissions and ensure that the security group allows inbound SSH traffic on port 22.

Here we can see that the SSHBash-EC2 is working correctly:



**Reason for This Test**: This test is crucial because it confirms that the SSHBash-EC2 instance is accessible from the internet, that my security groups are properly configured, and that my SSH key is correctly set up. Successful access is the first step in managing and configuring my instance remotely.

## 3.2 Testing the Connection Between the Bastion SSHBash-EC2 and the Frontend Subnet

**Objective**:  
To verify that the bastion host (SSHBash-EC2) can successfully connect to instances in the frontend subnet (WebApp1-EC2) via SSH. This test ensures that the security groups and network configurations are correctly set up to allow such connections.

**Steps**:

1. **Ensure the Bastion Host (SSHBash-EC2) Has the SSH Key (MyKey.pem)**:
   * Verify that the MyKey.pem file is present on the SSHBash-EC2 instance. This key is necessary for SSH access to the WebApp-EC2 instances.
2. **Change the Permissions of the SSH Key**:
   * Secure the SSH key by setting the appropriate file permissions. The key file should be readable only by the owner to prevent unauthorized access.

sudo chmod 400 MyKey.pem

* + **Explanation**:
    - **chmod 400:** Changes the file permissions so that only the file owner has read access. This is a security best practice for SSH keys.

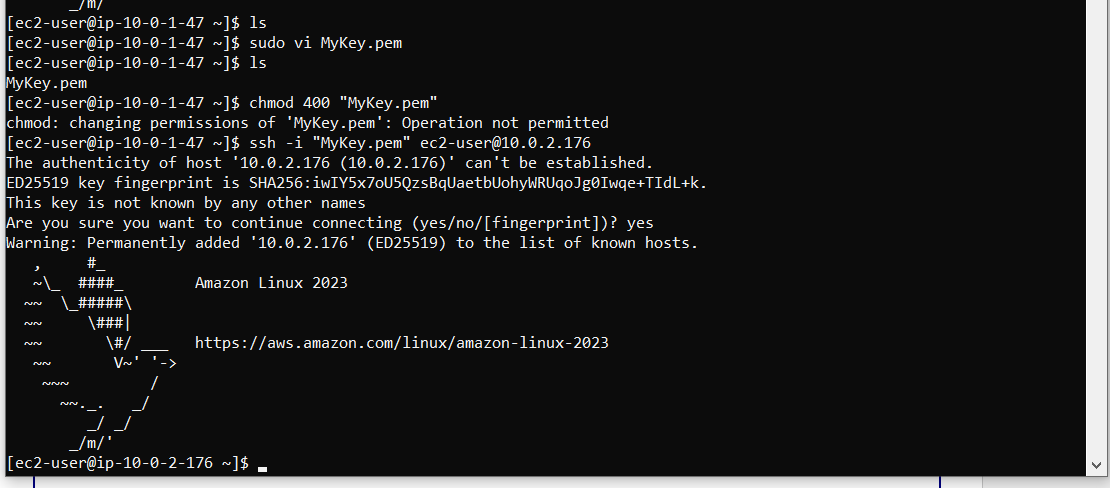
1. **Connect to WebApp1-EC2 via SSH**:
   * Use the SSH command to connect to the WebApp-EC2 instance from the SSHBash-EC2 instance.

ssh -i "MyKey.pem" ec2-user@10.0.2.176

* + **Explanation**:
    - ssh: Initiates the SSH session.
    - -i "MyKey.pem": Specifies the private key to use for authentication.
    - ec2-user@10.0.2.176: The username (ec2-user) and the private IP address (10.0.2.176) of the WebApp1-EC2 instance.

1. **Expected Outcome**:
   * If the connection is successful, I will be logged into the WebApp1-EC2 instance from the SSHBash-EC2 instance. This indicates that the SSH connection between the bastion host and the frontend subnet is correctly configured.
   * If I encounter connection issues, I should check the following:
     + Security group rules for both SSHBash-EC2 and WebApp-EC2 to ensure SSH traffic is allowed.
     + Network ACLs and route tables to ensure proper connectivity between subnets.
     + The private IP address and SSH key permissions.

Here we can see that the connection is correct as I succeeded in accessing WebApp1-EC2:



**Reason for This Test**: This test verifies that the bastion host (SSHBash-EC2) can access instances in the frontend subnet (WebApp-EC2) via SSH, which confirms that the network setup and security group configurations are correctly allowing SSH traffic within the VPC. Successful access is crucial for managing instances in private subnets from a bastion host.

## 3.3 Verifying NAT Gateway Functionality

**Objective**:  
To confirm that the NAT Gateway is functioning correctly by testing internet access from a private subnet instance (WebApp1-EC2). This ensures that instances in private subnets can reach the internet through the NAT Gateway for outbound traffic.

**Steps**:

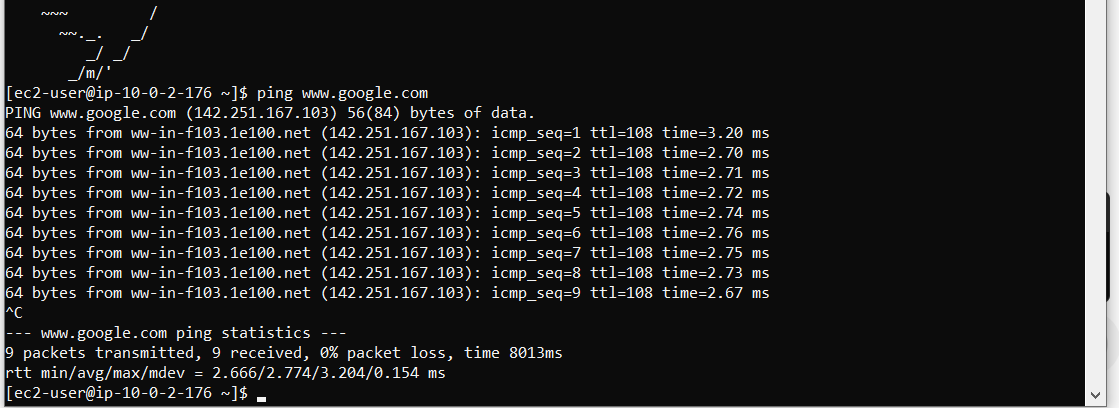
1. **Access WebApp1-EC2 Instance**:
   * I already have access to WebApp1-EC2 in the previous testing.
2. **Test Internet Connectivity**:
   * I will use the ping command to test connectivity to an external website, such as Google. This will help verify that the NAT Gateway is correctly routing traffic from the private subnet to the internet.

ping www.google.com

* + **Explanation**:
    - **ping:** A network utility used to test the reachability of a host on an IP network and measure the round-trip time for messages sent from the originating host to a destination computer.
    - **www.google.com:** The domain name of Google's website. A successful ping indicates that the WebApp1-EC2 instance can resolve domain names and reach the internet.

1. **Expected Outcome**:
   * **Successful Test**: If the NAT Gateway is functioning correctly, I should see responses from www.google.com, indicating that WebApp1-EC2 can successfully access the internet.
   * **Unsuccessful Test**: If the ping fails or times out, it indicates that the NAT Gateway may not be properly configured or there may be an issue with the route tables or security groups.
     + Troubleshoot by checking:
       - NAT Gateway configuration to ensure it's in the correct subnet and associated with the correct route table.
       - Route tables to verify that private subnets route outbound traffic to the NAT Gateway.
       - Security group and network ACL settings to ensure they allow outbound traffic.

Here we can see that the ping works, so the NAT Gateway is working correctly:



**Reason for This Test**: This test verifies that the NAT Gateway is correctly configured to provide internet access to instances in private subnets. Successful connectivity to an external website confirms that outbound traffic from private subnets is correctly routed through the NAT Gateway.

## 3.4 Testing Web Server Access via the Load Balancer

**Objective**:  
To verify that the Elastic Load Balancer (ELB) is properly distributing traffic between the web servers (WebApp1-EC2 and WebApp2-EC2). This ensures that the load balancer is functioning correctly and routing requests to multiple instances.

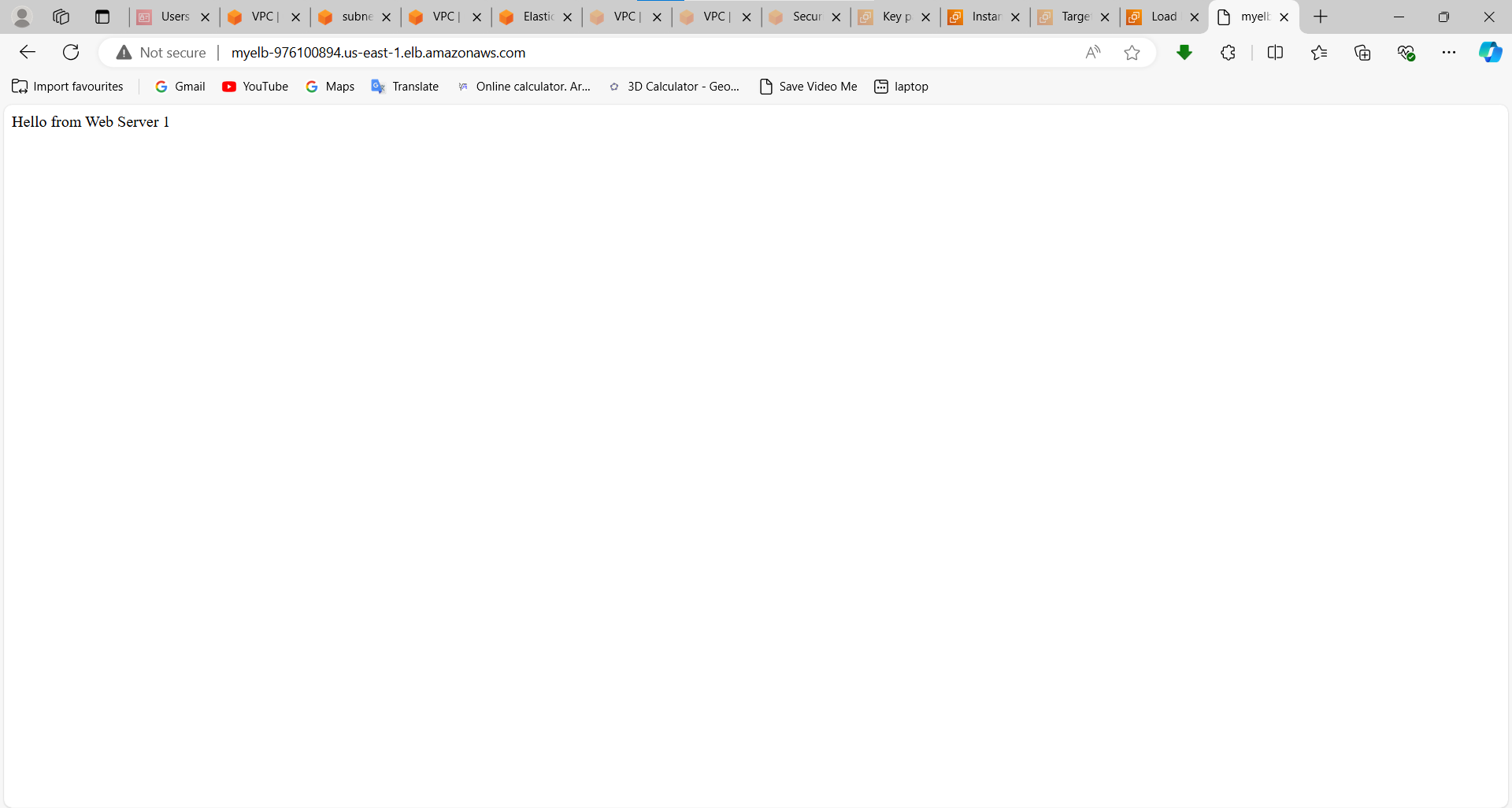
**Steps**:

1. **Access the Load Balancer DNS**:
   * I open a web browser and enter the DNS name of the Load Balancer. This is the URL provided by AWS for your Application Load Balancer.

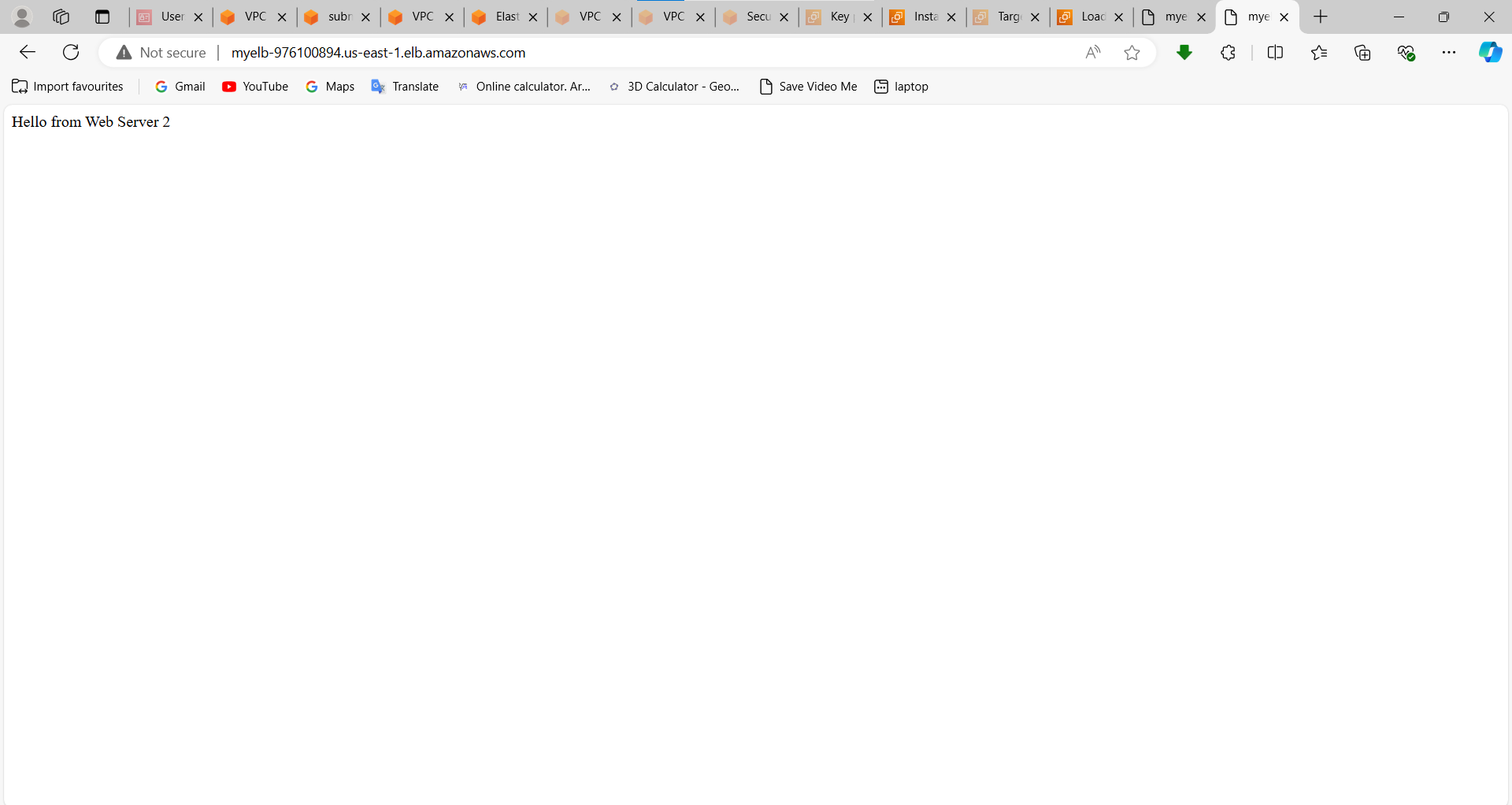
http://myelb-976100894.us-east-1.elb.amazonaws.com/

1. **Verify the Response**:
   * Upon visiting the URL, we should see a message indicating that the request has been handled by one of your web servers. The message might be "Hello from Web Server 1" or "Hello from Web Server 2," depending on which instance is handling the request at that time.
2. **Refresh the Page or Open in Another Browser/Tab**:
   * I could refresh the page or open the Load Balancer URL in a new browser tab or window.
   * I should see a different message from the other web server (i.e. if the first load shows "Hello from Web Server 1," refreshing might show "Hello from Web Server 2" or vice versa).
3. **Expected Outcome**:
   * **Successful Test**:
     + **Initial Access**: I see the message from one of the web servers (e.g., "Hello from Web Server 1").
     + **Refresh or New Tab**: The message changes to the other web server (e.g., "Hello from Web Server 2").
   * **Unsuccessful Test**:
     + If I only see one message consistently, or if there are errors loading the page, it indicates potential issues with the load balancer or target group configuration. This may involve checking:
       - **Target Group Health**: Ensure both WebApp1-EC2 and WebApp2-EC2 are marked as healthy in the target group.
       - **Load Balancer Configuration**: Verify that the load balancer is correctly configured with the target group and security groups.
       - **Web Server Status**: Ensure both web servers are running and correctly serving content.

Here we can see the first response when entering the DNS of the Load Balancer:

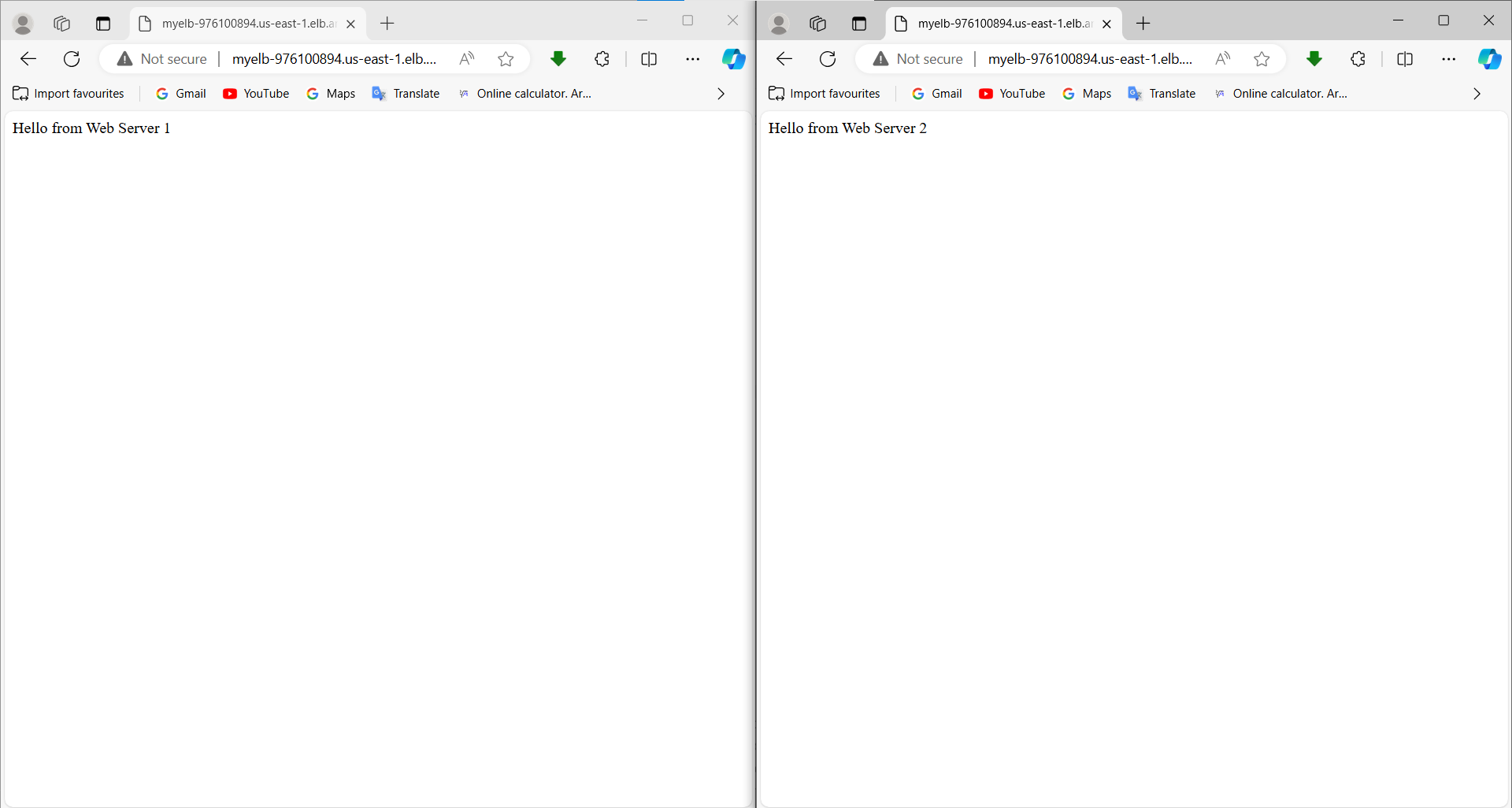


It indicates that it is Web Server 1.   
  
When opening a new tap and entering the same DNS we can see this message:



It indicates that it uses Web Server2.

Here is a better look at both servers working at the same time:



**Reason for This Test**: Testing the Load Balancer by accessing its DNS name in a browser confirms that the load balancer is properly routing traffic to the backend web servers. The rotation between different web servers on page refresh demonstrates that the load-balancing algorithm is working as expected.

## 3.5 Testing Connection Between WebApp1-EC2 and MySQL-EC2

**Objective:**

Verify that WebApp1-EC2 can successfully connect to MySQL-EC2 using the MySQL client with the updated password.

**Steps:**

1. **Update MySQL Password:**

**On MySQL-EC2:**

* + **Log in to MySQL:**

sudo mysql -u root -p

* + **Change the MySQL root Password:**

ALTER USER 'root'@'%' IDENTIFIED BY 'New@1Password';

FLUSH PRIVILEGES;

**Explanation:** The ALTER USER command updates the password for the root user. The FLUSH PRIVILEGES command is used to reload the grant tables, ensuring the new password is applied immediately.

1. **Test MySQL Connection from WebApp1-EC2:**
   * **Connect to MySQL from WebApp1-EC2 Using the New Password:**

mysql -h 10.0.3.27 -u root -p

**Explanation:** The mysql command connects to the MySQL server running on MySQL-EC2 (10.0.3.27). And I will enter the updated password (New@1Password) when prompted.

* + **Verify Successful Connection:** After entering the password, if the connection is successful, I will be logged into the MySQL shell.

And here we can see that from the WebApp2-EC2 I could access the MySQL-EC2:

A computer screen with a black background

Description automatically generated

## 3.6 Connect the WebApp Server to the DB and get data

**Objective:**

Verify that the PHP script on WebApp1-EC2 can successfully connect to the MySQL database on MySQL-EC2 and retrieve data.

**Steps:**

1. **Install PHP MySQL Extension:**

**Command:**

sudo yum update -y

sudo yum install -y php-mysqli

**Explanation:** The php-mysqli package is necessary for PHP to communicate with MySQL databases. This package includes the MySQLi extension, which provides an interface for interacting with MySQL databases.

1. **Restart Apache Web Server:**

**Command:**

sudo systemctl restart httpd

**Explanation:** Restarting Apache ensures that the new PHP extensions are loaded and applied. This step is crucial after installing or updating PHP extensions.

1. **Create and Verify PHP Info File:**

**Command to Create PHP Info File:**

sudo vi /var/www/html/info.php

**Add the Following Content:**

<?php

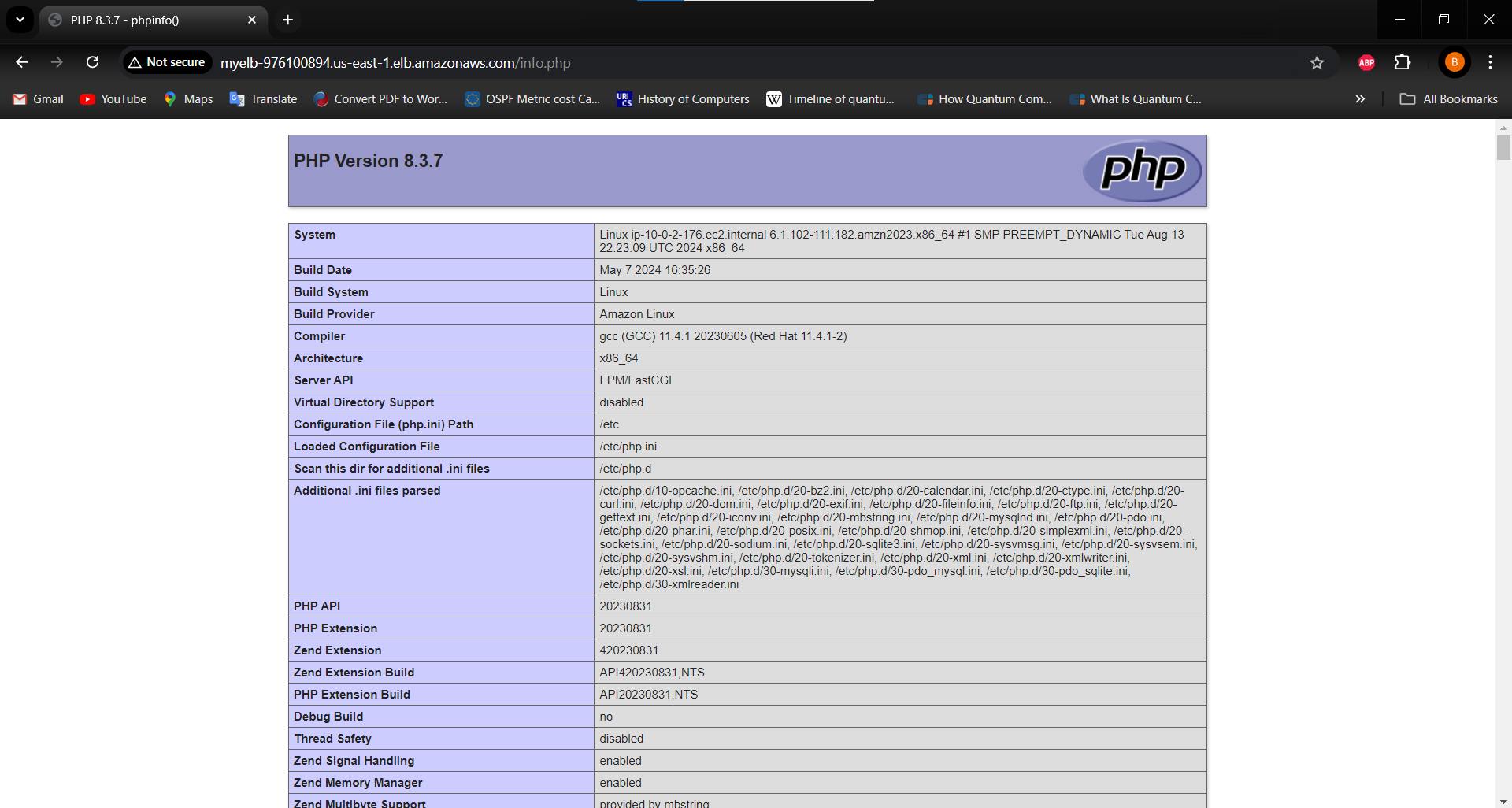
phpinfo();

?>

**Opening the browser and navigate to:**

<http://myelb-976100894.us-east-1.elb.amazonaws.com/info.php>

**Explanation:** Creating a phpinfo() file helps confirm that the mysqli extension is properly installed and loaded. This file provides detailed information about the PHP configuration and loaded extensions.



1. **Remove PHP Info File:**

**Command:**

sudo rm /var/www/html/info.php

**Explanation:** Removing the info.php file after verification is a good security practice. It prevents exposing sensitive configuration information to unauthorized users.

1. **Test PHP Script for MySQL Connection:**

**PHP Script:**

<?php

$servername = "10.0.3.27";

$username = "root";

$password = "New@1password";

$dbname = "TestDB";

$conn = new mysqli($servername, $username, $password, $dbname);

if ($conn->connect\_error) {

die("Connection failed: " . $conn->connect\_error);

}

$sql = "SELECT id, name, email FROM Users";

$result = $conn->query($sql);

if ($result->num\_rows > 0) {

while($row = $result->fetch\_assoc()) {

echo "id: " . $row["id"]. " - Name: " . $row["name"]. " - Email: " . $row["email"]. "<br>";

}

} else {

echo "0 results";

}

$conn->close();

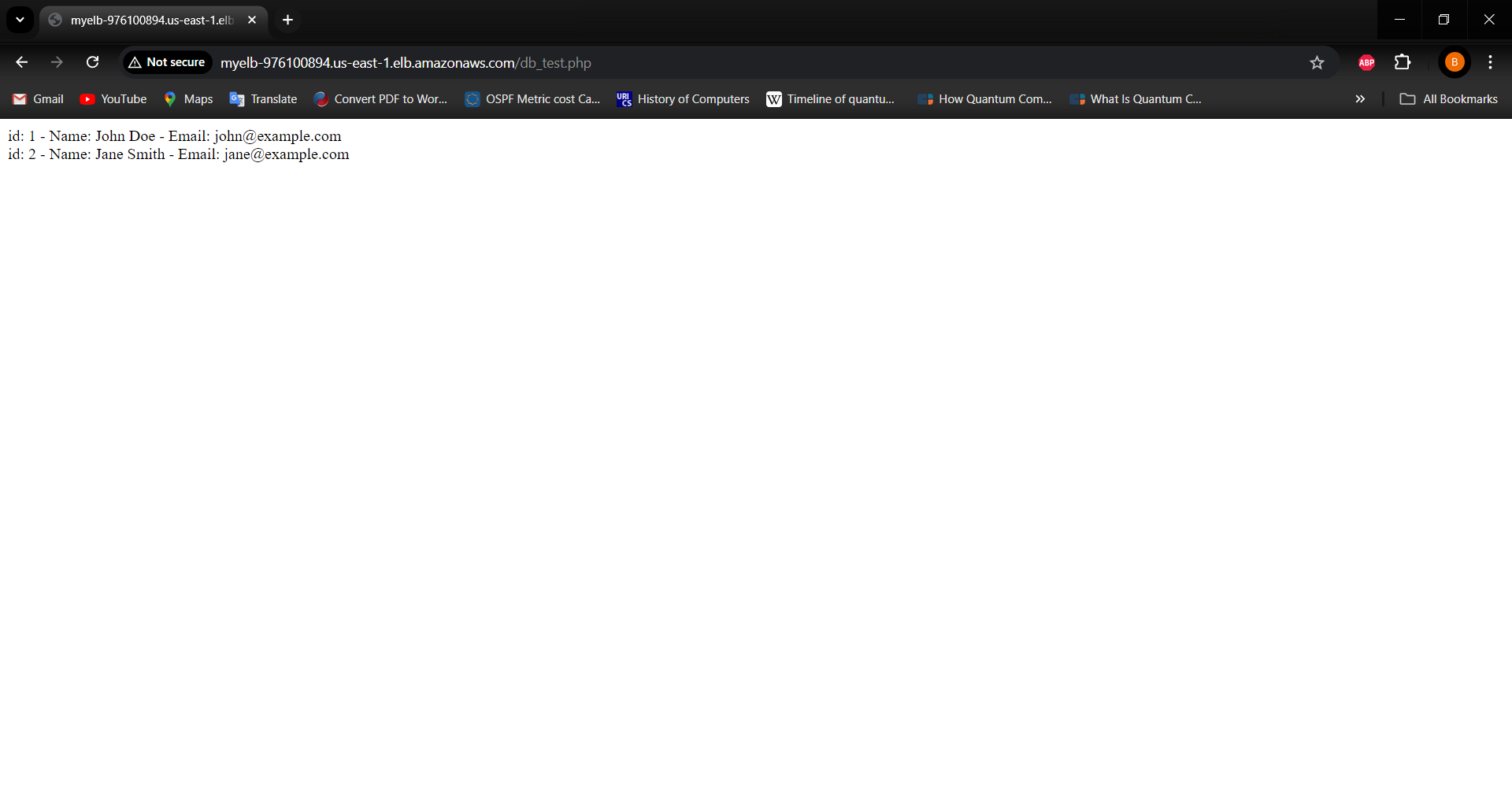
?>

**Explanation:** This PHP script connects to the MySQL database and retrieves data from the Users table. It checks the connection and displays the retrieved data or a message indicating no results.

**Opening the browser and navigating to db\_test.php through the Load Balancer DNS:**

<http://myelb-976100894.us-east-1.elb.amazonaws.com/db_test.php>

Here we can see the data retrieved from the database which indicates that our network and configurations are working correctly:



# Challenges and Solutions in Using AWS

Implementing and managing AWS services can present various challenges, particularly for those new to cloud computing or unfamiliar with AWS's extensive offerings. This section discusses some of the key challenges encountered during the setup of our network infrastructure and the solutions applied to overcome them.

**1. Configuring and Retrieving Data from Database Using WebApp Server**

**Problem:** Configuring a web application server (WebApp-EC2) to connect to a MySQL database server (MySQL-EC2) and retrieve data can be challenging. The issues might include configuring database access, ensuring correct permissions, and verifying connectivity between the web server and the database server.

**Detailed Steps and Solutions:**

1. **Database Access Configuration:**
   * **Issue:** The initial problem involved the MySQL server not allowing connections from the WebApp server due to incorrect permissions and authentication settings.
   * **Solution:**
     + **Adjust MySQL User Privileges:** The MySQL root user needed the appropriate privileges to accept connections from WebApp-EC2. This was resolved by executing:

GRANT ALL PRIVILEGES ON \*.\* TO 'root'@'%' IDENTIFIED BY 'New@1password';

FLUSH PRIVILEGES;

This command grants all privileges to the root user from any host and updates the password.

* + - **Update Password and Access Rules:** We updated the root password and verified the MySQL configuration to ensure that the web server could connect using the correct credentials.

1. **Web Server Configuration:**
   * **Issue:** The PHP script on WebApp-EC2 initially failed with an HTTP 500 error due to the absence of the MySQLi extension, which is necessary for PHP to interact with MySQL databases.
   * **Solution:**
     + **Install MySQLi Extension:** We installed the MySQLi PHP extension on WebApp-EC2 by running:

sudo yum install php-mysqli

sudo systemctl restart httpd

This command installs the required PHP module and restarts the Apache server to apply changes.

1. **Testing Database Connectivity:**
   * **Issue:** We needed to verify that WebApp-EC2 could connect to MySQL-EC2 and retrieve data.
   * **Solution:**
     + **Connect via MySQL Client:** We tested connectivity using the MySQL client from WebApp-EC2 with:

mysql -h 10.0.3.27 -u root -p

This command confirmed that the database was accessible and that the connection was successful.

* + - **Verify PHP Script:** Finally, we ensured that the PHP script was correctly retrieving data from the database and displaying it on the web page.

**2. Configuring VPC and Subnets**

**Challenge:** Understanding and configuring a Virtual Private Cloud (VPC) with appropriate subnets can be complex. It involves selecting the correct IP address ranges, creating multiple subnets for different purposes (public and private), and ensuring they are correctly associated with the appropriate route tables.

**Solution:** We carefully planned our IP addressing scheme and subnet configurations. By dividing the VPC CIDR block into smaller subnets, we ensured efficient IP address management. We also used AWS documentation and best practices to guide the creation of public and private subnets, ensuring proper network segmentation and security.

**3. Setting Up Internet and NAT Gateways**

**Challenge:** Configuring Internet Gateways and NAT Gateways to allow public and private subnet communication with the Internet can be challenging. It requires a clear understanding of how these gateways work and the associated routing configurations.

**Solution:** We created an Internet Gateway and attached it to our VPC to enable Internet access for public subnets. For private subnets, we set up a NAT Gateway in the public subnet, allocated an Elastic IP address, and updated route tables to route outbound traffic through the NAT Gateway. This configuration allows instances in private subnets to access the internet securely.

**4. Security Group and Network ACL Configurations**

**Challenge:** Defining and managing security groups and Network Access Control Lists (ACLs) to control inbound and outbound traffic can be intricate. Incorrect configurations can lead to security vulnerabilities or connectivity issues.

**Solution:** We meticulously defined security groups for each component of our network, specifying the required inbound and outbound rules. For example, we created specific security groups for SSH access, web server access, and MySQL database access. Additionally, we ensured that Network ACLs complemented our security groups, providing an extra layer of security.

**5. Ensuring High Availability**

**Challenge:** Achieving high availability involves setting up resources across multiple availability zones to avoid single points of failure. This includes load balancers, EC2 instances, and redundancy configurations.

**Solution:** We deployed critical resources such as our load balancer and NAT Gateway across multiple availability zones. By distributing resources, we enhanced the fault tolerance of our infrastructure. The use of Elastic Load Balancing ensured that traffic was evenly distributed across healthy instances, further improving availability and reliability.

**6. Key Management and Secure Access**

**Challenge:** Managing SSH keys and ensuring secure access to EC2 instances can be difficult, especially when multiple instances and users are involved.

**Solution:** We generated a key pair and associated it with our instances, allowing secure SSH access. The key file permissions were correctly set to prevent unauthorized access. By managing keys carefully and restricting access to only necessary users, we maintained a secure environment.

**7. Testing and Validation**

**Challenge:** Testing the network setup to ensure all components communicate correctly and perform as expected can be time-consuming and require meticulous verification.

**Solution:** We performed thorough testing at each stage of the setup. This included verifying SSH access, ensuring the functionality of the NAT Gateway, and checking web server access through the load balancer. Regular testing and validation helped identify and resolve issues promptly, ensuring a robust and functional network.