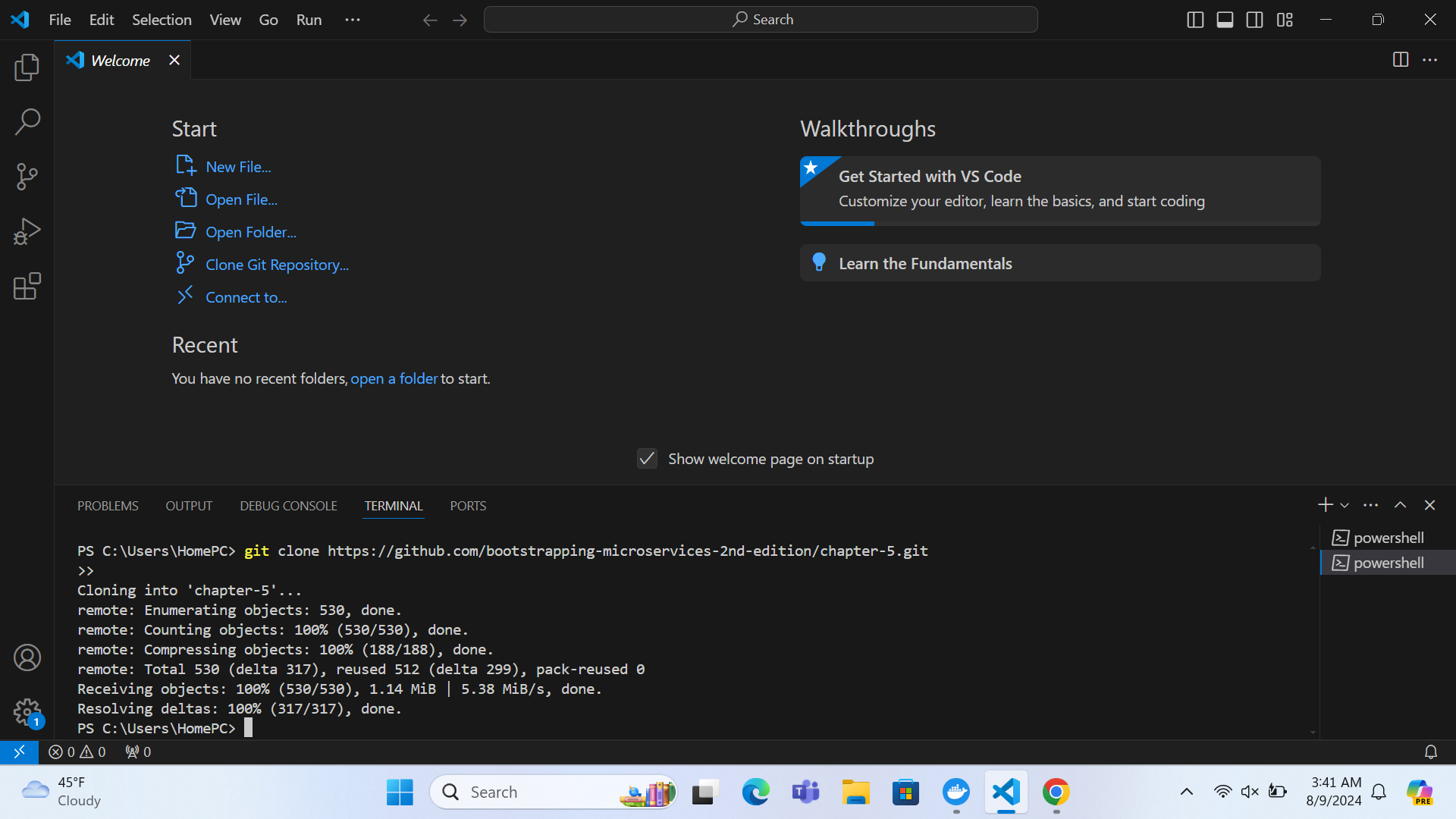
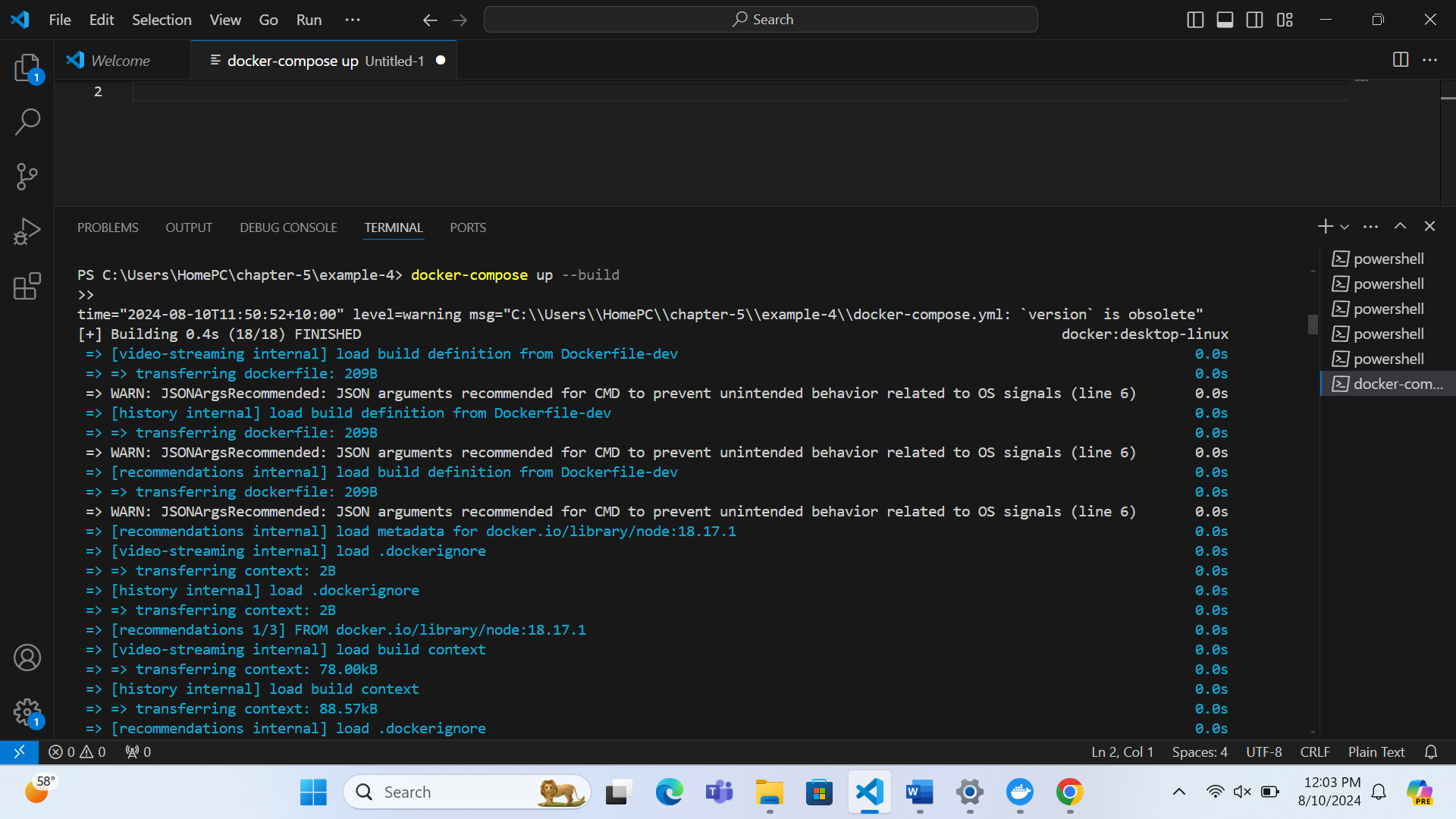
**Dennis Kimutai Kimaiyo**

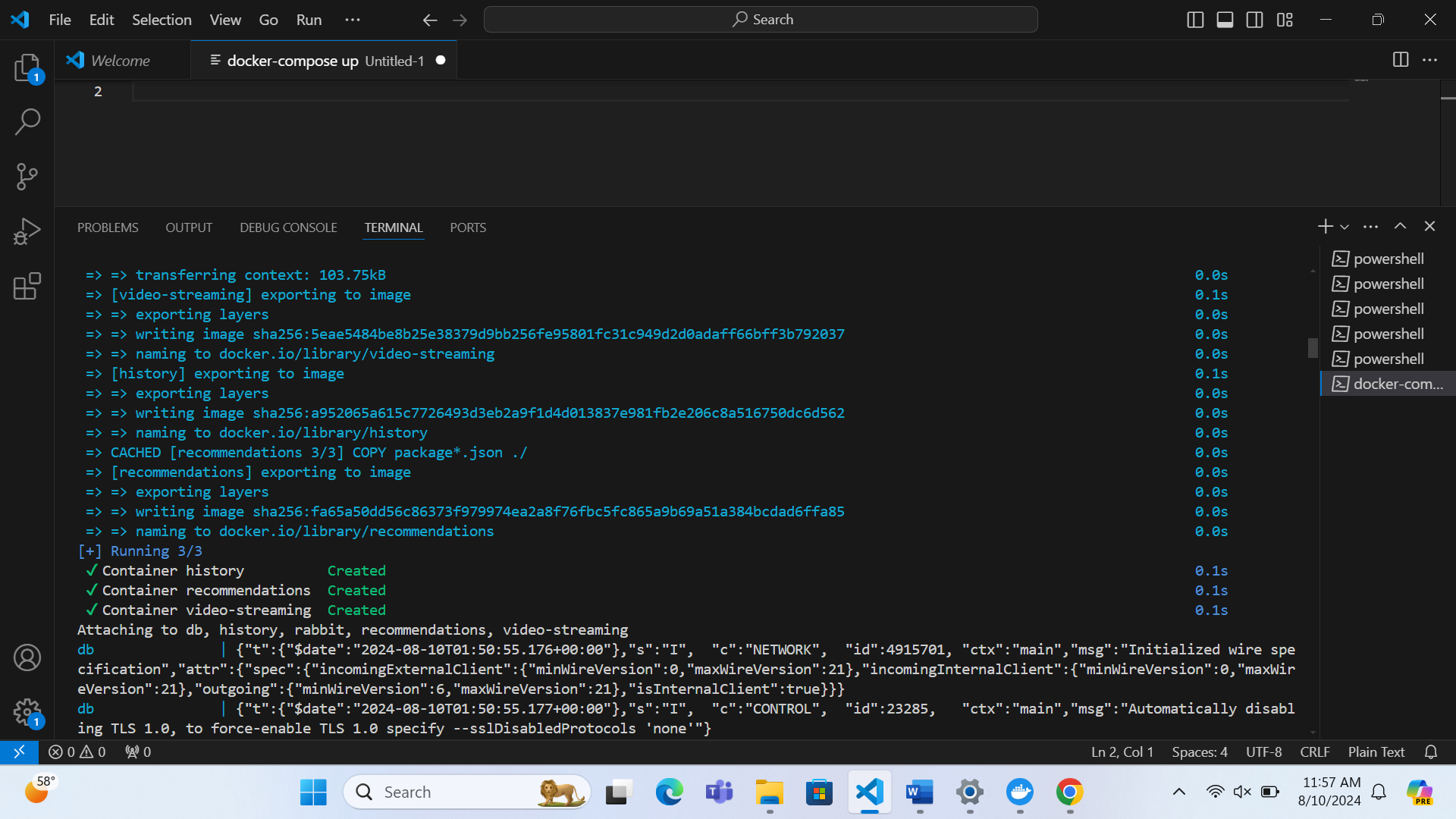
**ID: S223224152**

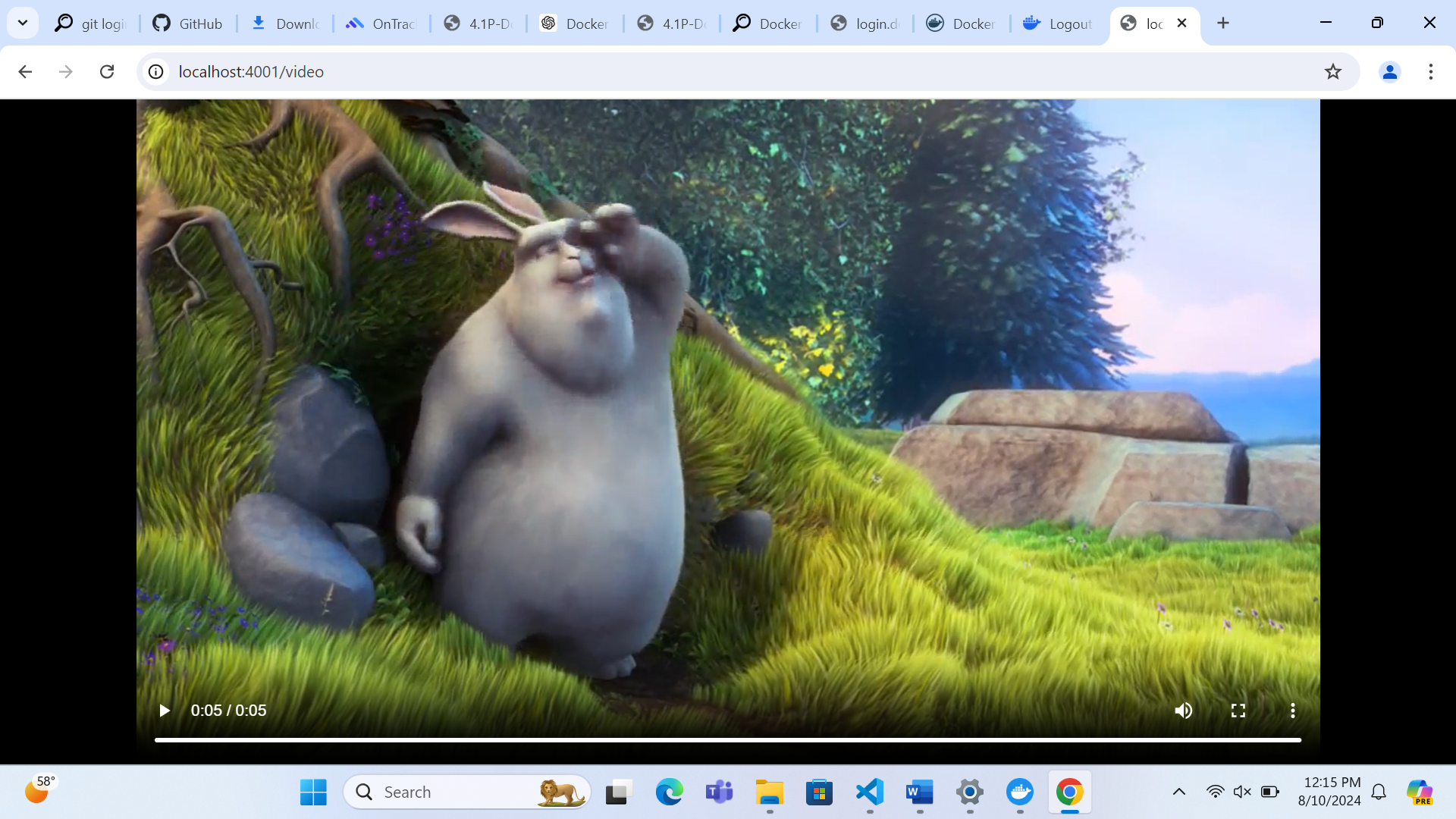
**Course: SIT722 Software Deployment and Operation**

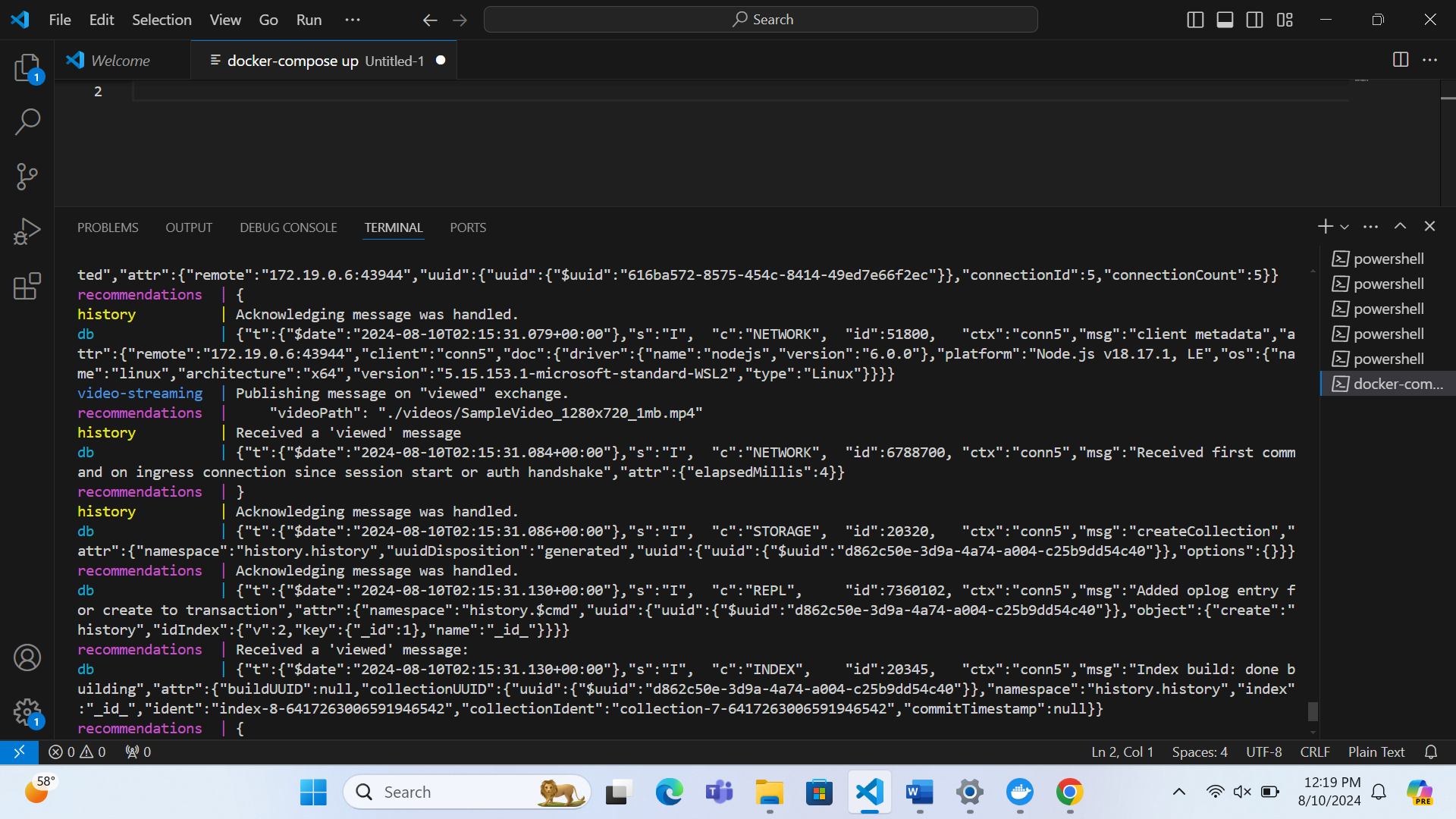
**Unit: Task 4.1P**

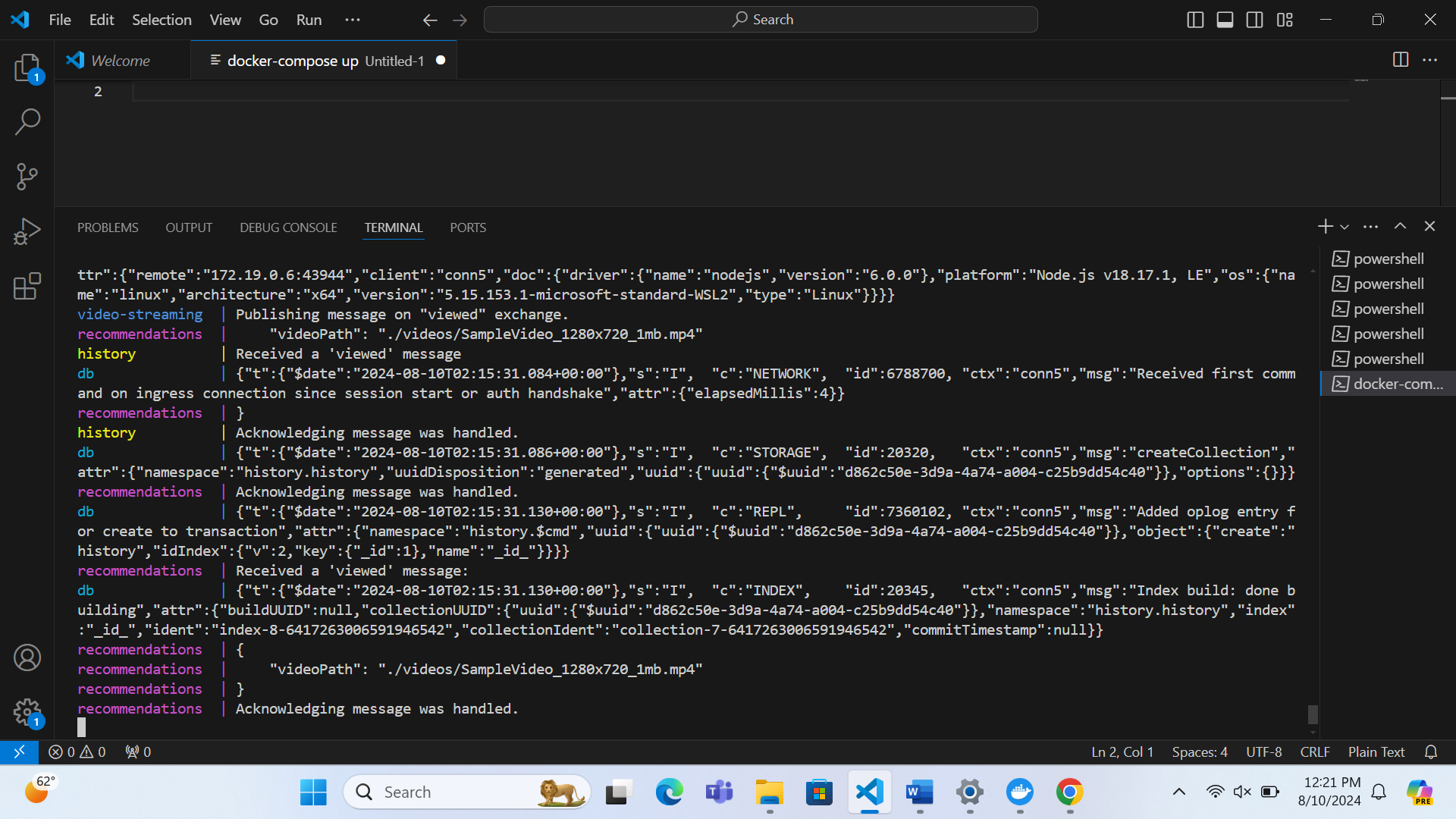
**screenshot of the VS Code console showing the command and the successful cloning** 

**Screenshot of docker-compose up for Example-4**



**A screenshot of the browser displaying the streamed video**



Screenshot of Console Logs during Video Streaming

1. **Describe What Each Example Application Demonstrates**

**Example 1: Direct Communication**

Description: This example focuses on how two Docker containers can directly exchange data over their network connection. It shows how services can communicate with each other using basic network operations without needing an intermediary.

**Example 2: HTTP Communication**

Description: In this example, Docker containers use HTTP protocols to communicate. It highlights how web services can make HTTP requests and responses to interact with each other, simulating a real-world web application environment where services talk to each other via HTTP.

**Example 3: Message Queues**

Description: This example introduces a message queuing system where Docker containers send and receive messages asynchronously through a message queue. This setup demonstrates how services can operate independently and handle tasks at different times, improving scalability and decoupling of services.

**Example 4: Complex Communication**

Description: This example integrates various services with a message queue to show a more complex communication scenario. It demonstrates how multiple services (video-streaming, history, recommendations, db) interact through RabbitMQ to handle a video request, process data, and provide recommendations efficiently. It illustrates the use of a message queue in a multi-service architecture.

**2. Describe the Sequence of Events in Example-4**

Client Request:

A client initiates a request to stream a video by navigating to http://localhost:4001/video.

Service Involved: video-streaming

Publish Message to RabbitMQ: The video-streaming service sends a message to RabbitMQ, indicating that a video has been requested and should be processed.

Services Involved: video-streaming and rabbit

Message Handling by History Service: The history service receives the 'viewed' message from RabbitMQ. It logs the view event and acknowledges the message.

Services Involved: history and rabbit

Request Recommendations:

After processing the view event, the recommendations service queries the db service to get video recommendations based on the requested video.

Services Involved: recommendations, db, and rabbit

Database Query:

The db service responds to the recommendations service with the list of recommended videos.

Services Involved: db and recommendations

Send Recommendations to Video-Streaming:

The recommendations service sends the recommendations back to the video-streaming service.

Services Involved: recommendations and video-streaming

Stream Video:

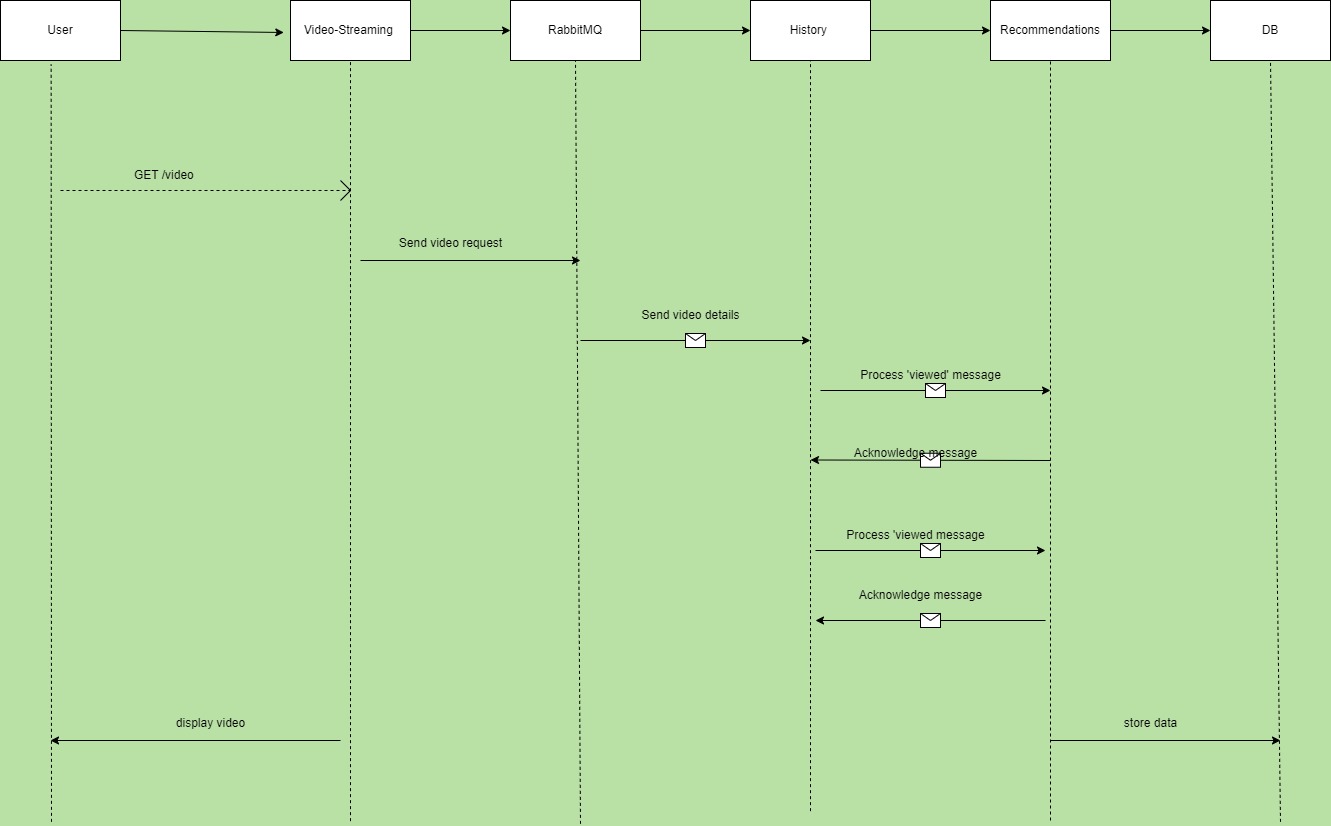
The video-streaming service streams the requested video to the client’s browser.

Services Involved: video-streaming

Video Plays in Browser:

The video is displayed and played in the client’s browser window.

Service Involved: client



3. **How Is the video-streaming Container Made to Wait for rabbit?**

Docker Configuration: The docker-compose.yml file contains a depends\_on directive for the video-streaming service, which specifies that it depends on the rabbit service. This configuration ensures that the video-streaming container does not start until the rabbit container is fully initialized and running. This prevents potential issues where the video-streaming service might try to connect to RabbitMQ before it's ready.

4. **How Are Docker-dev Dockerfiles Used in Example-4?**

* Purpose of Dockerfiles: Dockerfiles are scripts used to build Docker images. In Example-4, each Dockerfile outlines the steps needed to prepare the environment for each service, such as installing necessary software, setting environment variables, and copying code into the container.
* Impact on Execution: The Dockerfiles ensure that each microservice is built with the correct configuration and dependencies, allowing each service to function as intended. They play a crucial role in setting up the services so they can communicate effectively and handle requests as part of the application’s architecture