Name – Davender Singh

Student ID – \*\*\*\*\*\*\*\*\*\*\*\*

Course – CCP555NSA

Professor – David Humphrey

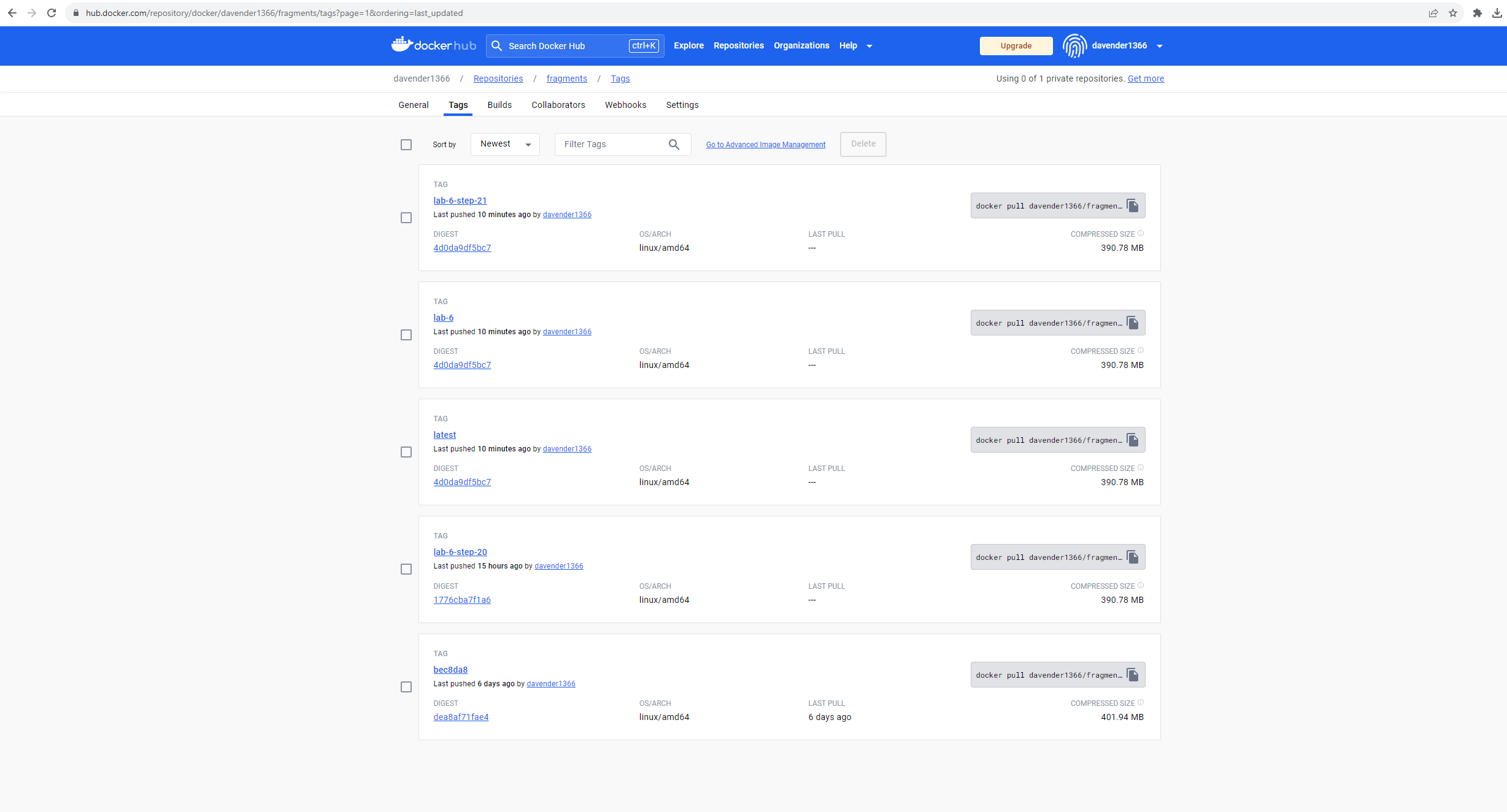
1. Link to your updated fragments microservice Dockerfile on GitHub, showing your optimizations from step 20 above:

<https://github.com/davender-singh1/fragments/blob/main/Dockerfile>

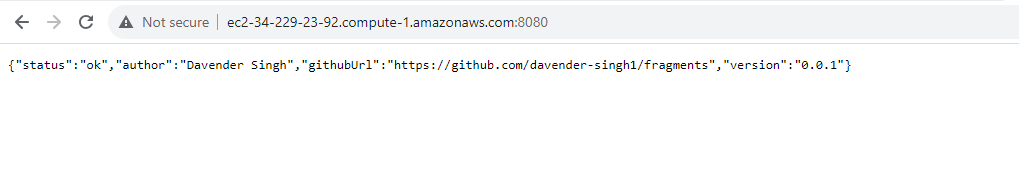
1. Link to your Docker Hub fragments repository:

<https://hub.docker.com/repository/docker/davender1366/fragments/general>

1. Screenshot of all of the expected Docker Tags in your Docker Hub repository (see the Tags tab):



1. Screenshot of an EC2 instance running your fragments Docker Hub image:



1. Explanation of the optimizations and requirement you implemented in steps 20 and 21:

Step 20: Optimized Dockerfile (Multi-Stage Build):

1. Build Stage:

# Stage 1: Build

FROM node:20.6.0 AS build-stage

LABEL maintainer="Davender Singh <davender-singh1@myseneca.ca>"

LABEL description="Fragments node.js microservice build stage"

ENV NPM\_CONFIG\_LOGLEVEL=warn

ENV NPM\_CONFIG\_COLOR=false

WORKDIR /app-build

# Only copy package files to use Docker cache effectively

COPY package\*.json ./

# Install application dependencies

RUN npm install

# Copy all necessary files from your project into the Docker image

COPY ./src ./src

# If there's a build script in your package.json, run it here

# RUN npm run build

Explanation:

- Introduced as `build-stage`, this stage sets up the application, copies the `package.json` and `package-lock.json`, and then installs the necessary dependencies.

- The key advantage here is that you are only dealing with the build environment, and any extra tools or dependencies you need for building (but not running) the app can be installed here without affecting the final image size.

2. Runtime Stage:

# Stage 2: Runtime

FROM node:20.6.0

WORKDIR /app

# Environment variables

ENV PORT=8080

# Copy from build stage

COPY --from=build-stage /app-build /app

# Expose the port the app runs on

EXPOSE 8080

# Command to run the application

CMD ["npm", "start"]

Explanation:

- This stage is intended for the runtime environment of your application.

- Using the `COPY --from=build-stage` command, you're copying only the necessary built artifacts and dependencies from the previous stage. This ensures you don't carry over unnecessary build tools or intermediate files.

- The result is a leaner image that contains only what's needed to run the application.

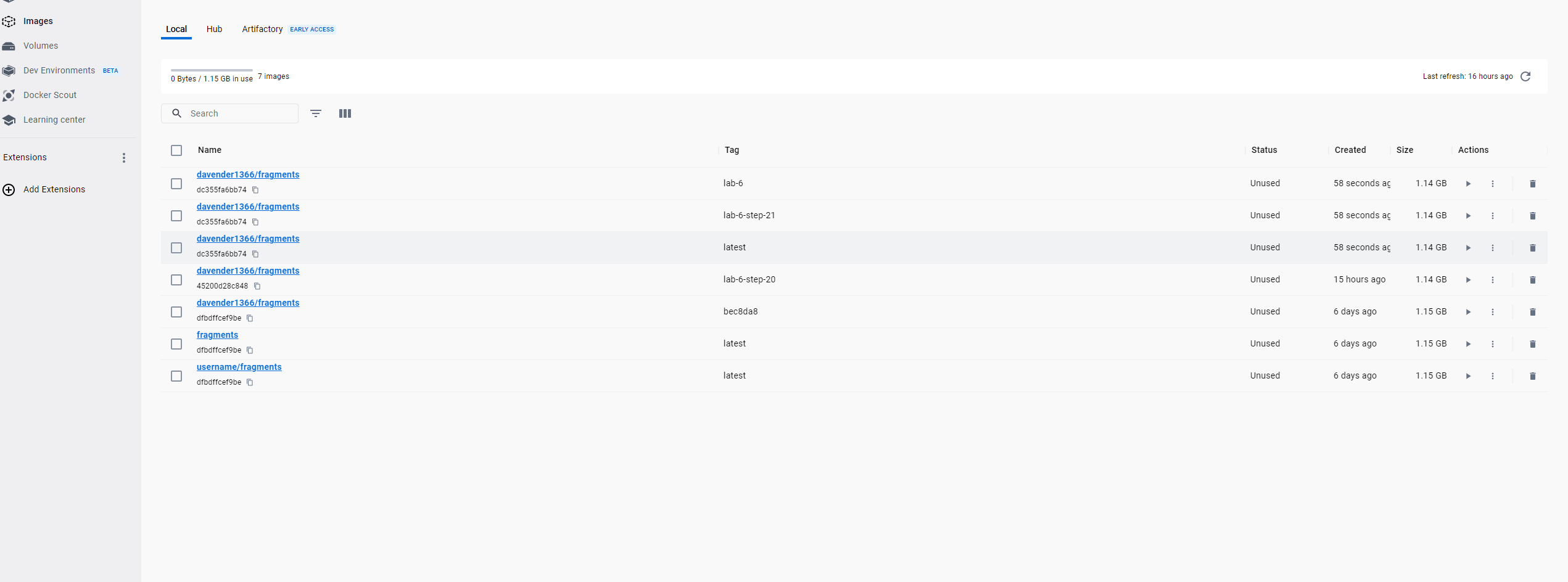
Benefits & Optimizations:

- Reduced Image Size: By not including build tools and only necessary dependencies in the final image, the image size is reduced.

- Efficient Caching: By copying only the `package\*.json` files initially, Docker can cache the installed dependencies effectively, leading to faster builds when the dependencies haven't changed.

- Cleaner Setup: By separating the build and runtime stages, the Dockerfile becomes clearer and easier to understand, which is beneficial for maintenance and collaboration.

Proof:



As evident from the above image, the Dockerfile optimization led to a reduction in the Docker image size from 1.15 GB (original) to 1.14 GB (optimized).

**Step 21:**  
Requirement Chosen from Assignment 2:

"GET /fragments?expand=1 should return expanded fragment metadata for an authenticated user. See 4.4.1."

Explanation of the Implementation:

I enhanced the GET /fragments endpoint to respond to the expand query parameter. When expand is set to 1, the API returns detailed metadata for each fragment, enriching the response with added information about each fragment.

* Data Structure Mocked: I created a mock data structure to represent expanded fragment metadata. Each fragment contains an ID, type, content, and additional metadata such as the author and creation date.
* Conditional Handling: The code checks for the presence of the expand query parameter. If it is set to 1, the API returns the mocked expanded metadata. If not, a placeholder data structure is returned. This allows flexibility in the API's response based on user needs.
* Error Handling: In the case of any unexpected error, a generic error response is returned to the client.
* Testing: To ensure the implemented feature works as expected, I added a new test case to the get.test.js file. The test checks if an authenticated user receives the expanded fragments metadata when using the ?expand=1 query parameter.

