Technical Report

Table of Contents

Technical Report *1*

0.0 Overview *3*

1.0 Task 1 ..... *4*

1.1 Docker Compose 4

1.2 Kubernetes 19

2.0 Task 2 – Vulnerability *41*

2.1 ZAP Testing 41

2.2 Vulnerability #1 42

2.3 Vulnerability #2 51

2.4 Vulnerability #3 57

2.5 Vulnerability #4 62

REFERENCES *69*

0.0 Overview

The assignment's first task required containerizing a full-stack web application using Docker and Kubernetes. The application consisted of a React frontend, an Express backend, MongoDB as the database, and Mongo Express as a GUI for database management. An Nginx proxy was also configured to provide HTTPS connections to the exposed services, including the frontend, web api, and Mongo Express. The objective was to deploy the application using individual containers, automate the setup with Docker Compose, and finally manage it within a Kubernetes cluster using Minikube. To complete this task, three main steps were followed:

Step 1 focused on building Docker images for each component of the application—React Frontend, Express Backend, MongoDB, and Nginx. Dockerfiles were created for the React and Express service to ensure proper configuration, including port exposure, environment variable management, and service dependencies. The task was successfully completed by running these containers individually, verifying that each part of the application functioned correctly in isolation.

Step 2 involved using Docker Compose to streamline the deployment process by defining all required services, internal networks, volumes, and configurations within a `docker-compose.yml` file. Two networks were setup: a 'frontend' network that connects React, Express, and Mongo-Express to Nginx, allowing them to serve external traffic, and a 'backend' network that connects Express and Mongo-Express to MongoDB for secure isolated communication. The frontend services are exposed using secure HTTPS connections. This setup enabled the project to be managed and deployed efficiently.

Step 3 extended the deployment by utilizing Kubernetes with Minikube. A Kubernetes cluster was set up, defining a deployment for each component so that each would run in its own pod. Each pod was made accessible via a named service, with appropriate port and selector definitions, mirroring the containers created in the Docker Compose part of the assignment. The application was exposed to external traffic by port-forwarding the Nginx pod's HTTPS port to port 8081 on the EC2 host. This approach demonstrated the ability to manage and scale the application within a Kubernetes cluster effectively.

Conclusion: The tasks demonstrated the complete process of containerizing and deploying a full-stack application using Docker and Kubernetes. From individual container management to orchestration with Docker Compose and Kubernetes, each step reinforced

the understanding of container-based application deployment and management in a scalable and secure manner.
Note: All screenshots include the EC2 instance name to ensure clarity during the marking process.

The GitHub repository used for this assignment was: Dev.to-clone

1.0 Task 1

1.1 Docker Compose

Following the lab instructions, Docker and Docker Compose were successfully installed on the instance to support containerized application deployment.

Dockerfile for Express (Server):

```
# Select node as the base layer.
FROM node:latest

# Create and set the container directory
# where project files will be placed.
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

# Copy over the package.json file and install deps.
# This creates a reusable, cached layer of all required deps.
COPY package.json /usr/src/app

RUN npm install

# Copy the remaining source code.
COPY . /usr/src/app

# Set which port to expose for the backend API.
EXPOSE 5000

# Container start command.
```

```
CMD ["npm", "start"]
```

Explanation:

- **Base Image**: Uses node:latest as the base image.
- **Working Directory**: Sets /usr/src/app as the working directory within the container.
- **Dependencies**: Copies package.json and package-lock.json and installs the dependencies using npm install.
- **Application Code**: Copies the rest of the application code into the container.
- **Port**: Exposes port 5000, which is where the Express server will listen.
- **Command**: Specifies the command to start the application (npm start).

Dockerfile for React (Client):

```
# Use the latest Node.js image as the base layer for the container
FROM node:latest

# Create and set the working directory for the application
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

# Copy the package.json file into the container and install dependencies
COPY package.json /usr/src/app
RUN npm install --force

# Copy the remaining source code into the container
COPY . /usr/src/app

# Expose port 3000 for serving the React application
EXPOSE 3000

# Set environment variables needed for the React application
ENV REACT_APP_API_SCHEME=https
ENV REACT_APP_API_HOST= [ec2 public ipv4 IP]
ENV REACT_APP_API_PORT=443
ENV REACT_APP_API_BASE_PATH=api

# Create a production build of the React application
RUN npm run build
```

```
# Serve the production build using the 'serve' package
CMD ["npx", "serve", "-s", "build"]
```

Explanation:

Base Image: FROM node:latest: Specifies the base image as the latest Node.js image. This image includes Node.js and npm, which are necessary for building and running the React application.

Working Directory: RUN mkdir -p /usr/src/app and WORKDIR /usr/src/app: Creates and sets the working directory inside the container where the application files will be stored and executed.

Dependency Installation: COPY package.json /usr/src/app and RUN npm install --force: Copies the package.json file into the container and installs the necessary dependencies. The --force flag is required due to dependency conflicts, mainly relating to different React versions.

Source Code: COPY . /usr/src/app: Copies the remaining application source code into the container's working directory.

Port Exposure: EXPOSE 3000: Indicates that the container will listen on port 3000, which is the default port for serving React applications in development.

Environment Variables:

These variables configure the React application's connection to the API:

- ENV REACT_APP_API_SCHEME=https
- ENV REACT_APP_API_HOST=[ec2 public ipv4 IP]
- ENV REACT_APP_API_PORT=443
- ENV REACT_APP_API_BASE_PATH=api

They ensure that the application uses HTTPS and connects to the backend through the specified EC2 public IP address and exposed Nginx port.

The environment variables must be set during the build process (rather than in the compose.yml) since React needs them to build the static source files that will be served.

- **SCHEME**: Set to ensure React sends secure HTTPS requests to Express as is required by Express cors configuration.
- **HOST**: Initially set to Express's docker container name for internal networking, but changed to the EC2 public IPv4 because React sends the request from the browser. Therefore, they need to point to Nginx, so the proxy can forward them to Express.

- **PORT**: Configured to ensure port 443 for an HTTPS connection.

Build and Serve:

- RUN npm run build: Creates an optimized production build of the React application. This command generates static files suitable for deployment.
- CMD ["npx", "serve", "-s", "build"]: The “serve” package serves the static react build files on port 3000 by default.

Docker Compose file

compose.yml

```
version: "3.9"
services:
 mongo:
  container_name: mongo
  image: mongo
  volumes:
   - mongodata:/data/db
  networks:
   - backend
  healthcheck:
   test: echo 'db.runCommand("ping").ok' | mongosh localhost:27017/test --quiet
   interval: 10s
   retries: 5

 mongo-express:
  container_name: mongo-express
  image: mongo-express
  depends_on:
   - mongo
  environment:
   - ME_CONFIG_MONGODB_URL=mongodb://mongo:27017/
   - ME_CONFIG_SITE_BASEURL=/mongo-express/
   - PORT=8081
  networks:
   - backend
   - frontend

 express:
  container_name: express
  image: express-image
  depends_on:
   - mongo
  environment:
   - PORT=5000
   - DB_HOST=mongo
```

```
    - DB_PORT=27017
    - DB_NAME=devto
    - JWT_KEY=somerandomjwtkey
    - COOKIE_KEY=somerandomcookiekey
    - NODE_ENV=development
    - CLIENT_URL=https:// [public ipv4 IP]
    - CLOUDINARY_CLOUD_NAME=devtoclone
    - CLOUDINARY_API_KEY=176414317785344
    - CLOUDINARY_API_SECRET=WYBaD60Xcos5OB0fliwVRNQ40-o
   networks:
    - backend
    - frontend

  react:
   container_name: react
   image: react-image
   depends_on:
    - express
   networks:
    - frontend

  proxy:
   container_name: proxy
   image: nginx
   depends_on:
    - react
   volumes:
    - ./nginx.conf:/etc/nginx/nginx.conf:ro
    - ./ssl/localhost.crt:/etc/ssl/certs/localhost.crt
    - ./ssl/localhost.key:/etc/ssl/private/localhost.key
   ports:
    - 80:80
    - 443:443
   networks:
    - frontend

volumes:
  mongodata:
```

```
networks:
  backend:
  frontend:
```

Compose file explanation:

Port Mappings

- Proxy (proxy):
    - 80:80: Maps port 80 of the host to port 80 of the Nginx container, enabling HTTP access.
    - 443:443: Maps port 443 of the host to port 443 of the Nginx container, enabling HTTPS access.

Networks

- Backend Network:
    - Purpose: Facilitates communication between backend services (mongo, mongo-express, and express). This isolates backend services from frontend services for security and organization.
- Frontend Network:
    - Purpose: Facilitates communication between frontend services (react, proxy, and mongo-express) and the express service. This allows the frontend services to interact with the backend services through the proxy.

Volumes

- mongodata:
    - Purpose: Persists MongoDB data. Ensures that MongoDB data is not lost when the container is restarted or removed. The volume is mounted to /data/db in the MongoDB container.

Environment Variables

- Express (express):
    - PORT: Defines the port the Express application will listen on within the container.
    - DB_HOST, DB_PORT, DB_NAME: Configure database connection settings. DB_HOST points to the mongo service, ensuring that the Express application can connect to MongoDB.
    - JWT_KEY, COOKIE_KEY: Used for JWT and cookie encryption.
    - NODE_ENV: Sets the environment (e.g., development or production).

- CLIENT_URL: URL of the client application for CORS and other configurations.
- CLOUDINARY_CLOUD_NAME, CLOUDINARY_API_KEY, CLOUDINARY_API_SECRET: Configuration for integrating with Cloudinary for image storage.

- Mongo Express (mongo-express):
  - ME_CONFIG_MONGODB_URL: URL for connecting to the MongoDB instance. It points to the mongo service.

Modifications and Troubleshooting

1. **Exposing Express via Nginx**
   a. Initially, the idea was to establish an internal network between express and react. However, React sends requests from the browser and not server-side. Therefore, express had to be exposed to external traffic.
2. **Connecting Mong-Express to Nginx**
   a. At first, mong-express was exposed on port 8081 for external access. Since this is an insecure (http) option, it was decided to route traffic between the browser and mong-express via nginx as well.
3. **React Environment Variables not applying:**
   a. React's environment variables that were defined in the compose file were undefined at runtime. After some debugging it was discovered that the varibles must be
4. **Health Checks:**
   - Added a health check for mongo-express to ensure it only starts when MongoDB is fully operational. This prevents issues related to Mongo Express starting before MongoDB is ready.

Screenshots of the Compose deployment:

Mongo:

Express:

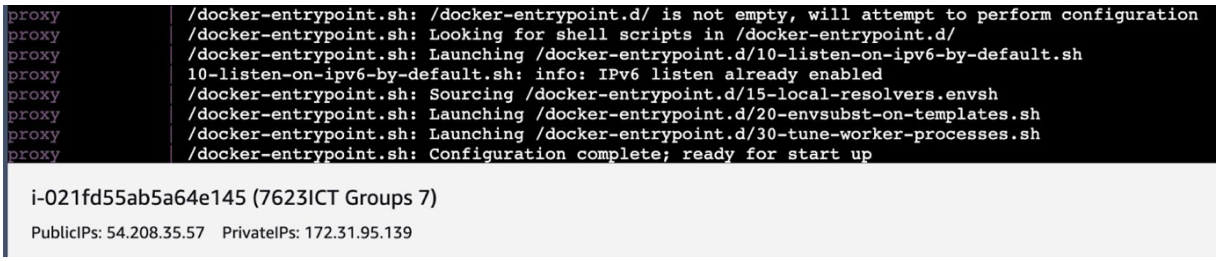
```
express  | Mongoose connection status:  1
express  | Express server is running on port 5000.
```

i-021fd55ab5a64e145 (7623ICT Groups 7)

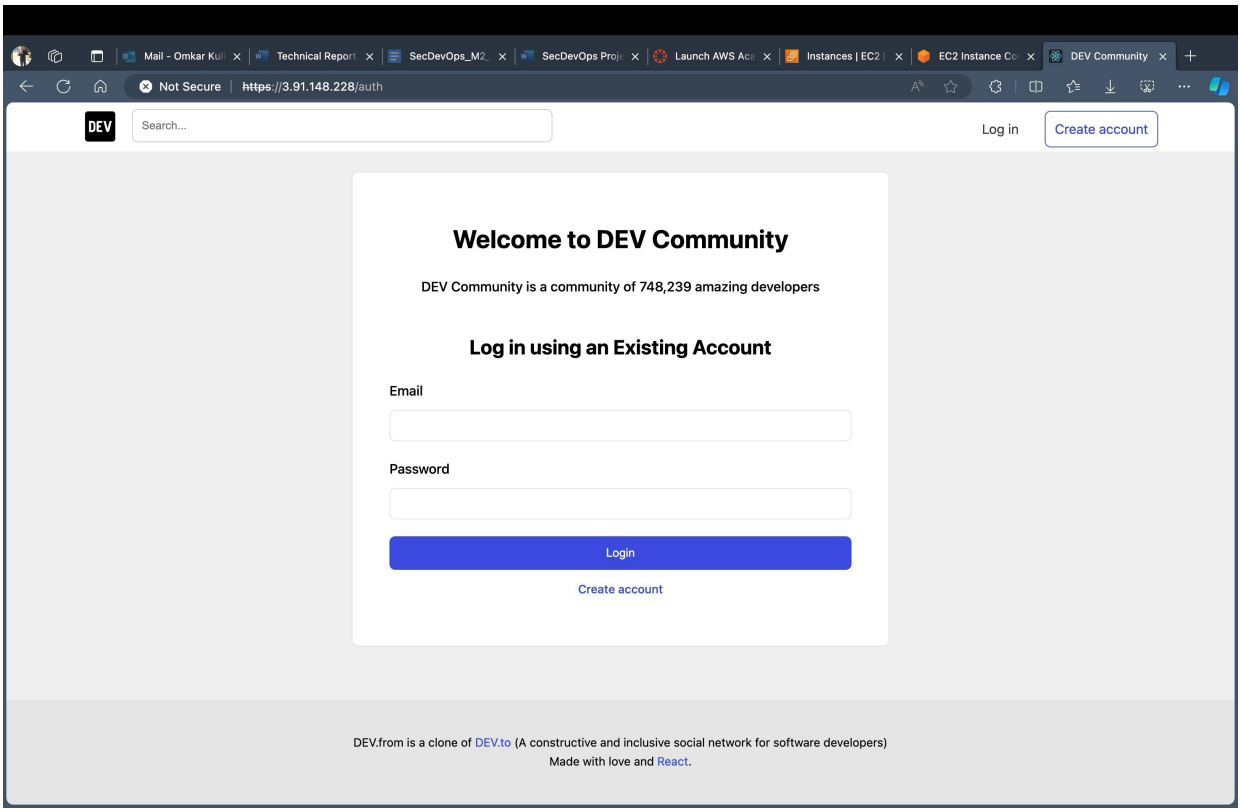
PublicIPs: 54.208.35.57 PrivateIPs: 172.31.95.139

React & Mongo-Express:

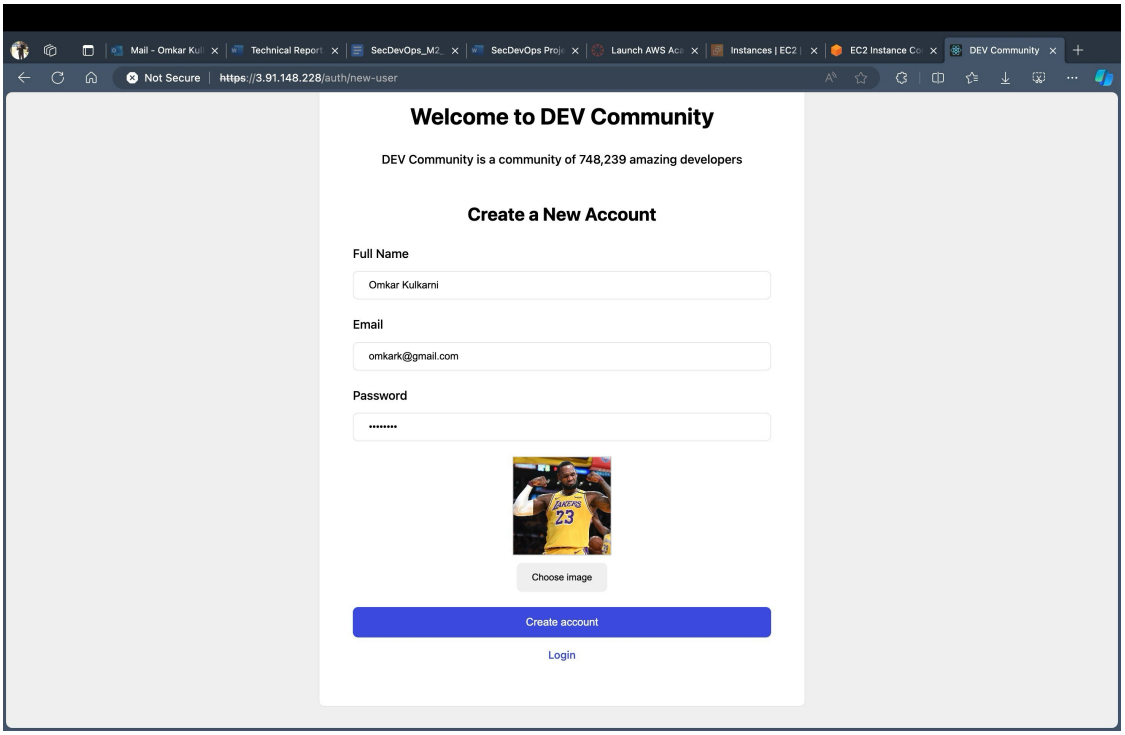
Nginx:



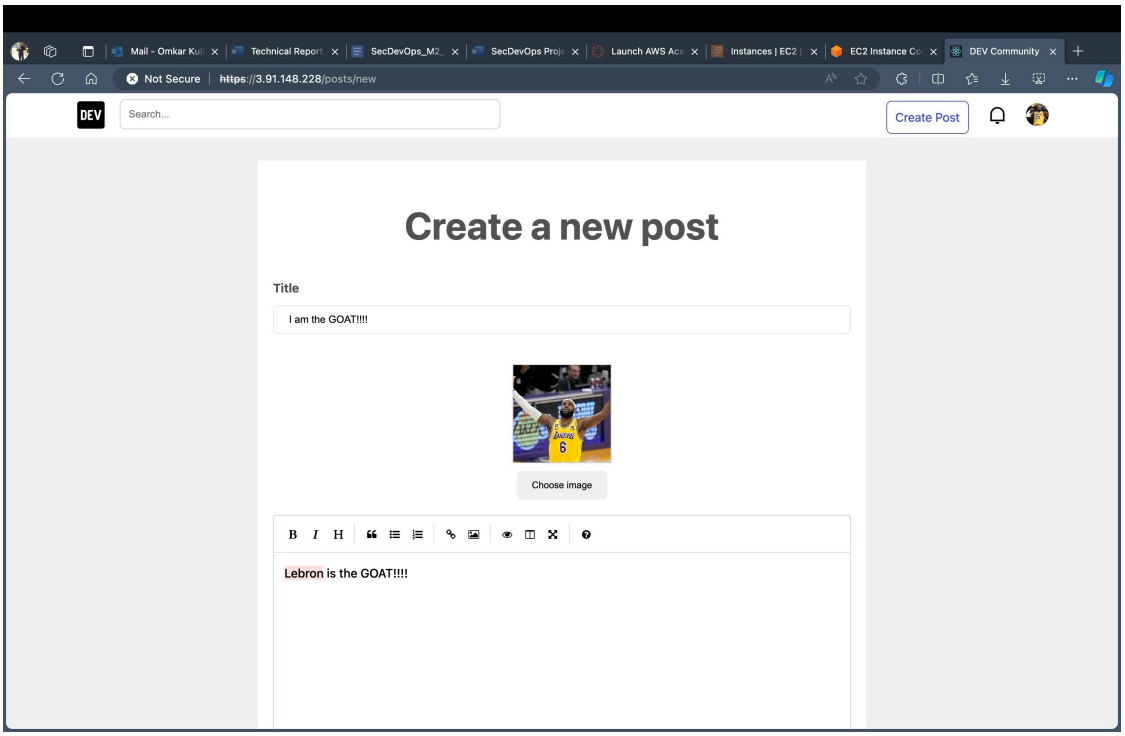
Accessing the React Application from the browser on an https connection, using the EC2 instance public IP:

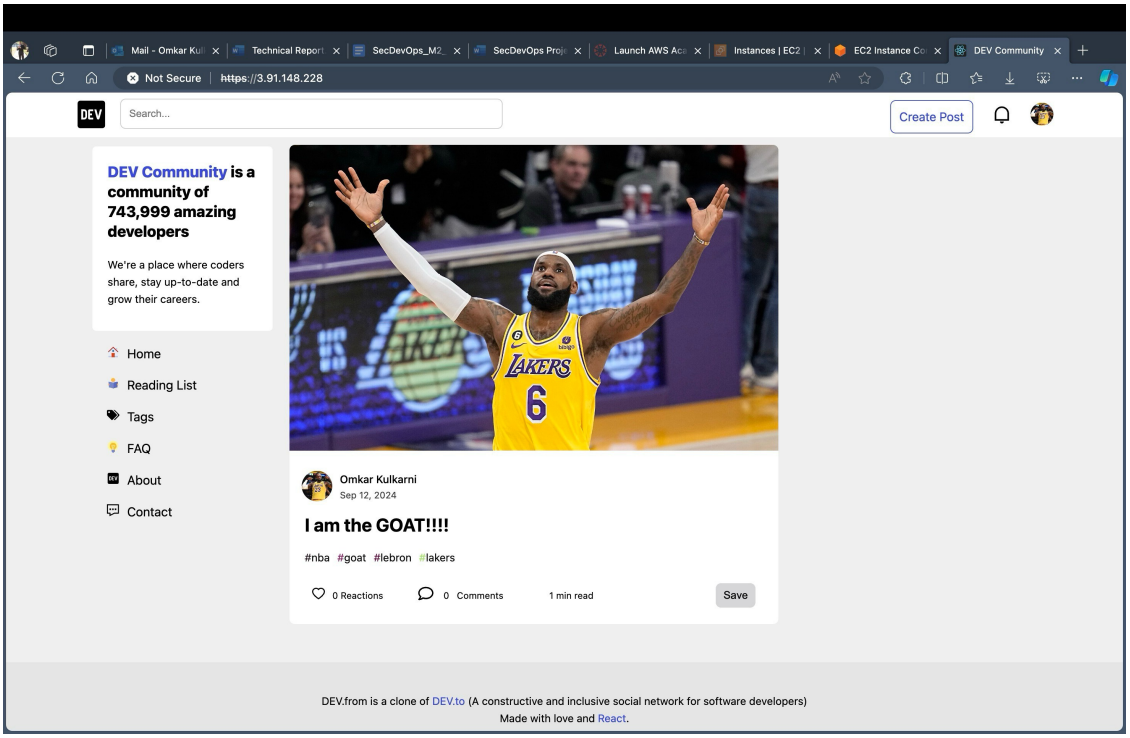


Creating an Account:



Creating a post:





Comments:

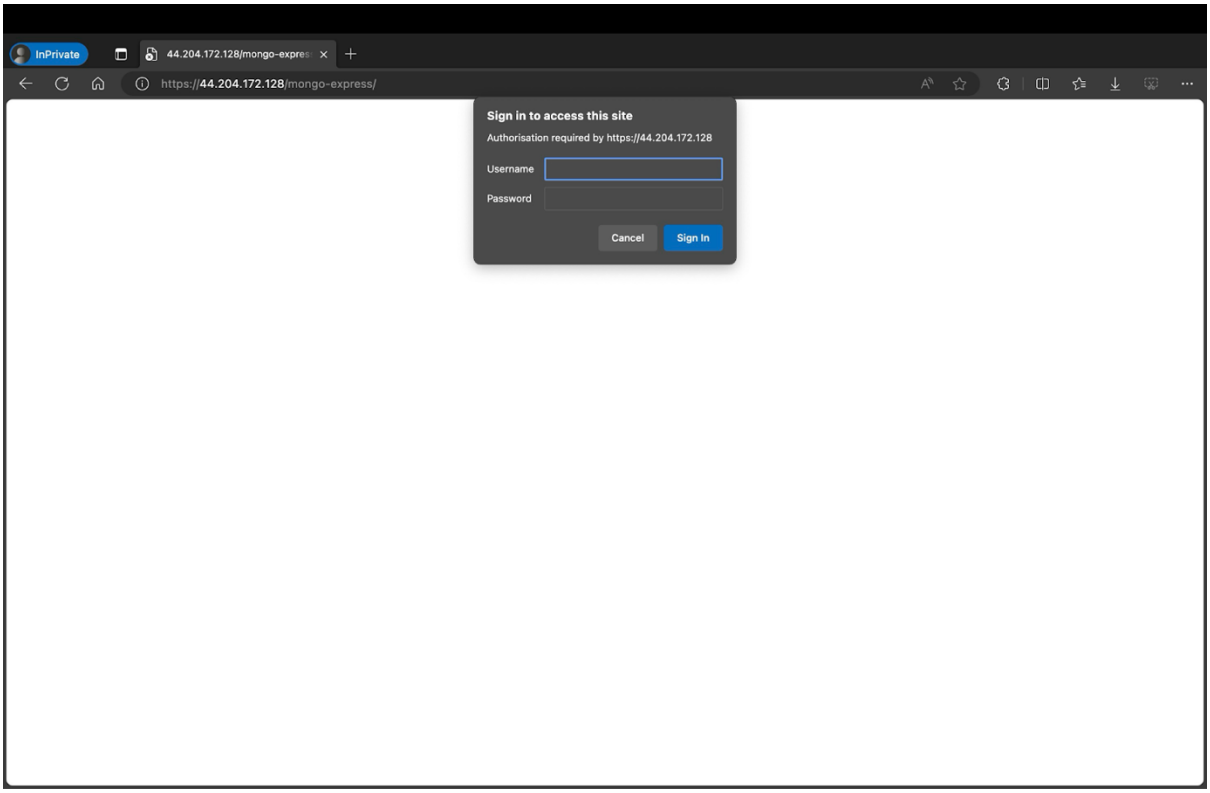
The screenshots depict successful testing of the application's core functionalities. The steps include:

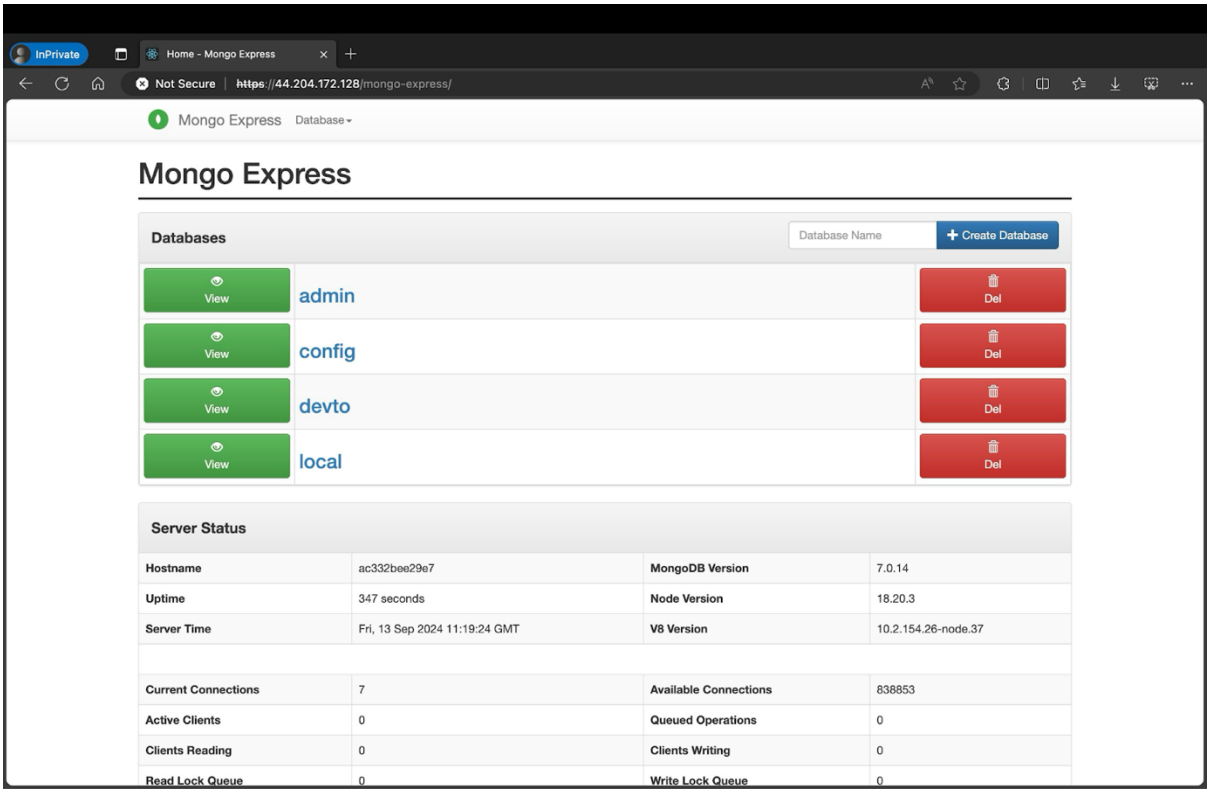
1. Account Creation
2. Creating a Post

3. Post Display

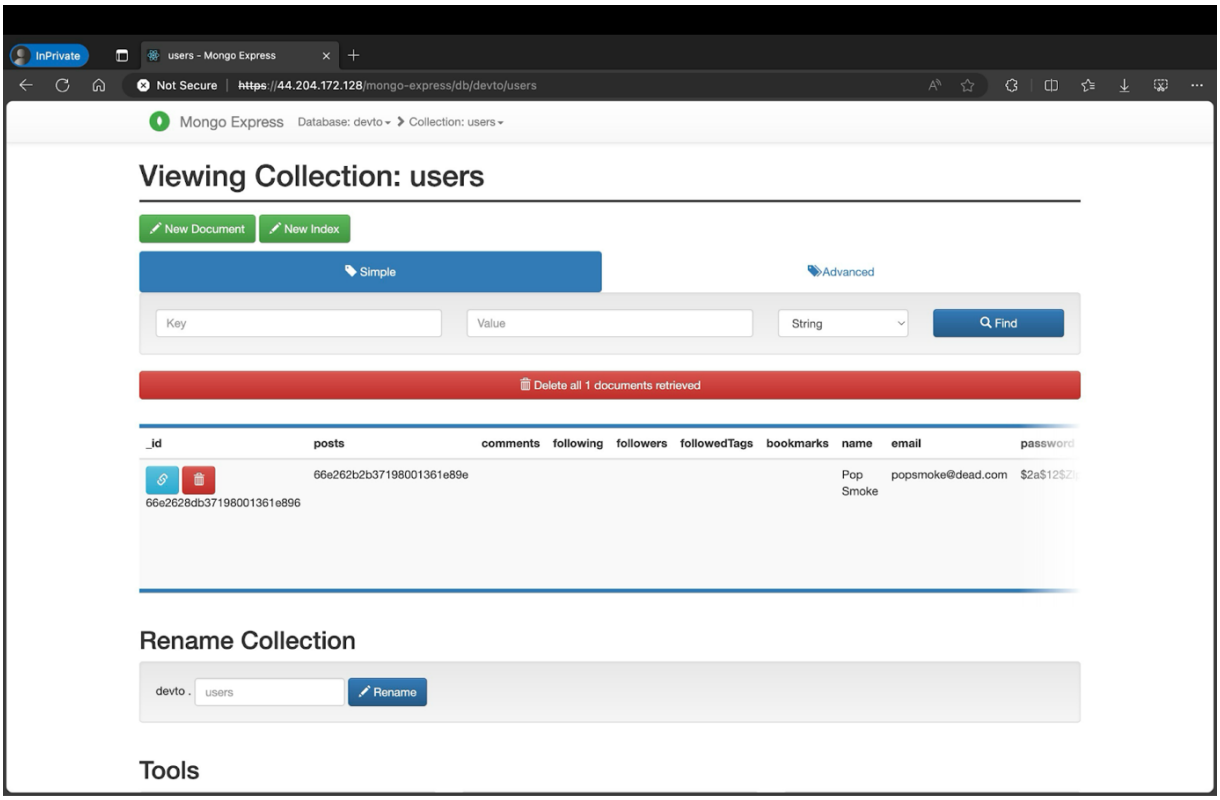
These tests verify that the application, running through Docker Compose, supports essential user interactions and content management features as intended, providing a fully functional clone of the DEV Community platform.

Connecting to Mongo-express

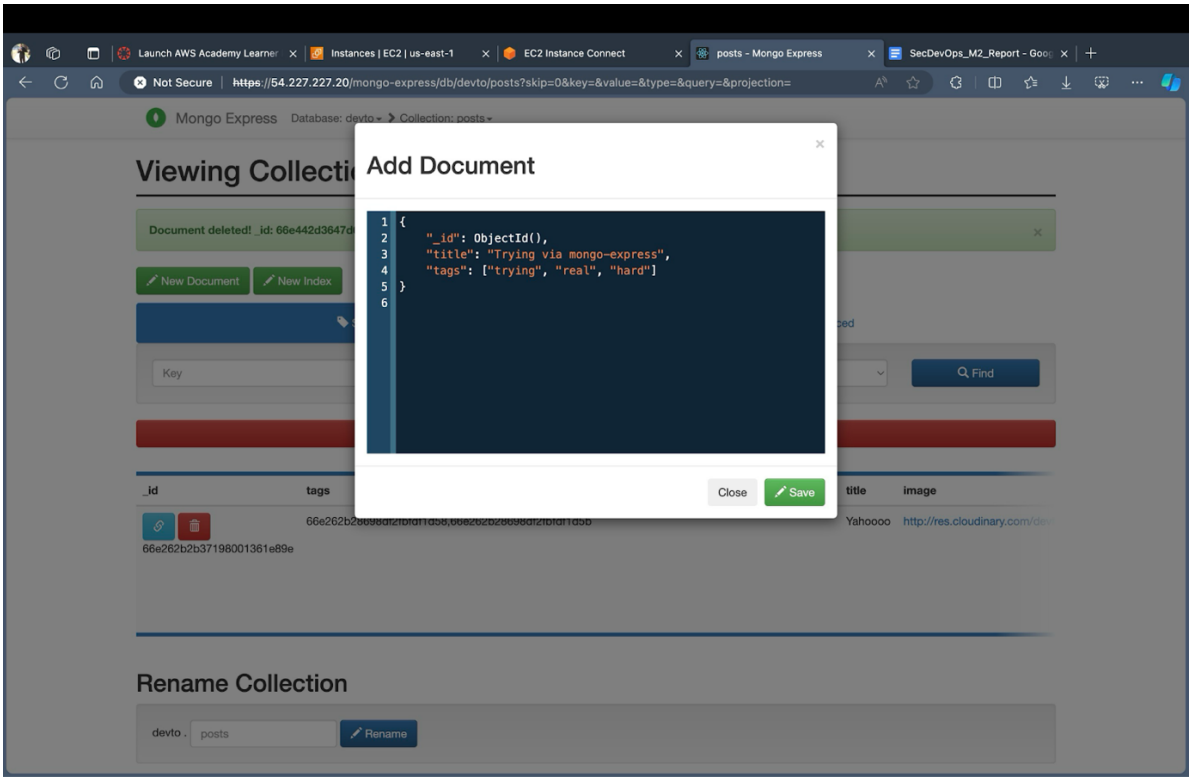


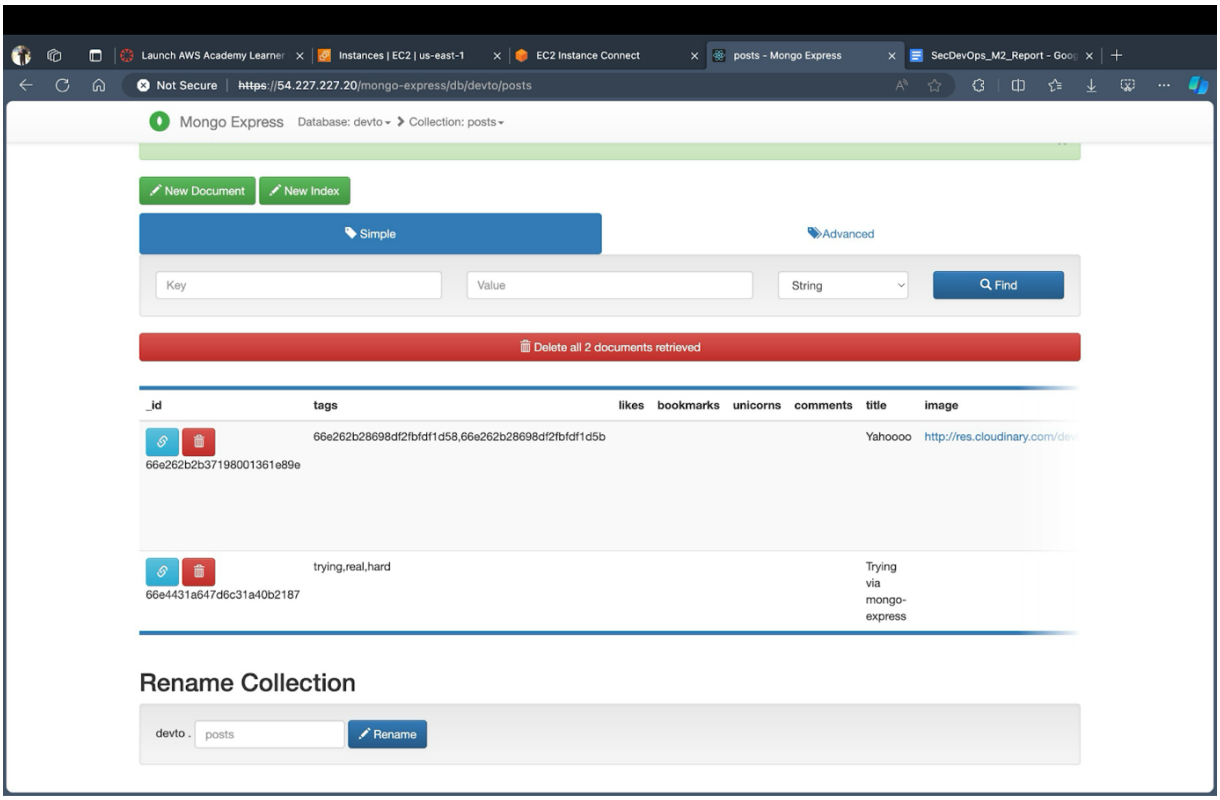


Can View/Edit Users



Adding a Post





Comments:

The screenshots show successful access and use of Mongo Express, a web-based interface for managing MongoDB.

1. Accessing Mongo Express: When navigating to `http://ipv4-address/mongo-express/ `, an authentication prompt is displayed, requiring a username and password, demonstrating secure access control to the Mongo Express interface.

2. Mongo Express Dashboard: After logging in, the main dashboard of Mongo Express is visible, showing the available databases (`admin`, `config`, `devto`, `local`). The server status section confirms that MongoDB is running properly, with details like uptime and the MongoDB version.

3. Viewing Collections: The screenshot shows the `users` collection within the `devto` database. It displays user details, including the account created earlier with the name "Omkar Kulkarni" and associated information such as email and password (hashed), confirming that the application's backend is correctly storing user data in MongoDB.

These interactions confirm that Mongo Express is correctly configured.

1.2 Kubernetes

Definition of Required Fields taken from Kubernetes Documentation

- **api** – Version of the Kubernetes API you're using to create this object
- **kind** – Defines what kind of object you want to create
- **metadata** – Data that helps uniquely identify the object, here we set react-deployment as the name of the object. Labels are key-value pairs that help categorize and select resources.
- **spec** – defines the state we desire for the object. The precise format of the object spec is different for every Kubernetes object and contains nested fields specific to that object.

React Deployment

Defines the deployment for the React application.

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: react-deployment
 labels:
  app: react
spec:
 replicas: 1
 selector:
 matchLabels:
   app: react
 template:
 metadata:
   labels:
    app: react
  spec:
  containers:
   - name: react
    image: react-image
    imagePullPolicy: Never
    ports:
    - containerPort: 3000
```

Key things here:

- **kind**: Deployment - defines the type of Kubernetes object. Here, it's deployment, which manages the creation and scaling of Pods.
- **replicas**: Defines the number of pod replicas (copies) to be created. Here, it is set to 1, meaning one instance of Reaction will be running.
- **container** – list the containers that will run inside each pod
    - name: name of the container, in this case react.
    - image – specifies the container image to use, in this case react-image that we built.
    - imagePullPolicy: indicates when to pull the image. We used Never means Kubernetes will never pull the image from registry like Docker Hub.
- **ports**: Exposes the port 3000 on the container, which is default port for React applications.

React Service

Exposes react internally within the cluster

```
apiVersion: v1
kind: Service
metadata:
  name: react-service
spec:
  selector:
    app: react
  ports:
    - protocol: TCP
      port: 3000
      targetPort: 3000
```

Key things here:

- **kind**: Service – Defines what kind of object you want to create. Here, it's a Service a method for exposing a network application that is running as one or more Pods in your cluster.
- **ports**: Describes the ports that the service will expose.
    - protocol: specifies the protocol used, which is TCP (Transmission Control Protocol). Most common protocol for web applications.

    - port – the port that the service will expose, in this case it is set to 3000. Meaning, the requests sent to this port of the service will be forwarded to the pods.
    - targetPort – the port on the container where traffic should be routed. This is set to 3000, which matches the port that the React container inside the pod is listening on.
- **type**: The type of service under spec was omitted because Express is using ClusterIP which is the default type if type is not specified. ClusterIP since react, express, mongo, and mongo-express should be only accessible internally within the Kubernetes cluster.

Express Deployment

Defines the deployment for Express

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: express-deployment
 labels:
  app: express
spec:
 replicas: 1
 selector:
 matchLabels:
   app: express
 template:
 metadata:
   labels:
    app: express
  spec:
  containers:
   -name: express
    image: express-image
    imagePullPolicy: Never
    ports:
    - containerPort: 5000
    env:
    - name: PORT
```

```
      value: "5000"
    - name: DB_HOST
      value: "mongo-service"
    - name: DB_PORT
      value: "27017"
    - name: DB_NAME
      value: "devto"
    - name: NODE_ENV
      value: "development"
    - name: CLIENT_URL
      value: "https://184.73.134.11"
    - name: JWT_KEY
      value: "somerandomjwtkey"
    - name: CLOUDINARY_CLOUD_NAME
      value: "devtoclone"
    - name: CLOUDINARY_API_KEY
      value: "176414317785344"
    - name: CLOUDINARY_API_SECRET
      value:    "WYBaD60Xcos5OB0fliwVRNQ40-o"
```

Key things here:

- **kind**: Deployment - defines the type of Kubernetes object. Here, it is deployment, which manages the creation and scaling of Pods.
- **replicas**: Defines the number of pod replicas (copies) to be created. Here, it is set to 1, meaning one instance of Express will be running.
- **container** – list the containers that will run inside each pod
    - name: name of the container, in this case express
    - image – specifies the container image to use, in this case express-image that we built
    - imagePullPolicy: indicates when to pull the image. We used Never means Kubernetes will never pull the image from registry like Docker Hub.
- **ports**: Exposes the port 5000 on the container
- **env**: Defines environment variables for the container. Critical variables such as the MongoDB host (DB_HOST), port, and JWT authentication key are defined to configure the backend.
    - PORT: Sets the port for the Express app

- DB_HOST – Sets the MongoDB service host (must match MongoDB's service name)
- DB_PORT - The default MongoDB port
- DB_NAME - The database name for Express to use
- NODE_ENV - Sets the Node.js environment to 'development'
- CLIENT_URL - Defines the client URL for cross-origin resource sharing (CORS)
- JWT_KEY - Secret key used for signing JWT tokens
- CLOUDINARY_CLOUD_NAME - Cloudinary account name for image uploads
- CLOUDINARY_API_KEY - Cloudinary API key for uploads
- CLOUDINARY_API_SECRET - Cloudinary API secret for uploads

Express Service

Exposes express internally within the cluster

```
apiVersion: v1
kind: Service
metadata:
 name: express-service
spec:
 selector:
  app: express
 ports:
  - protocol: TCP
    port: 5000
    targetPort: 5000
```

Key things here:

- **kind**: Service – Defines what kind of object you want to create. Here, it's a Service a method for exposing a network application that is running as one or more Pods in your cluster.
- **ports**: Describes the ports that the service will expose.
  - protocol: specifies the protocol used, which is TCP (Transmission Control Protocol). Most common protocol for web applications.
  - port – the port that the service will expose, in this case it is set to 5000. Meaning, the requests sent to this port of the service will be forwarded to the pods.

    - targetPort – the port on the container where traffic should be routed. This is set to 5000, which matches the port that the Express container inside the pod is listening on.
- **type**: The type of service under spec was omitted because Express is using ClusterIP which is the default type if type is not specified. ClusterIP since react, express, mongo, and mongo-express should be only accessible internally within the Kubernetes cluster.

Mongo Deployment

Defines the deployment for Mongo

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mongo-deployment
  labels:
    app: mongo
spec:
  replicas: 1
  selector:
  matchLabels:
     app: mongo
  template:
    metadata:
      labels:
        app: mongo
    spec:
      containers:
      -name: mongo
        image: mongo
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 27017
        volumeMounts:
        -name: mongo-storage
          mountPath: /data/db
      volumes:
      - name: mongo-storage
```

```
    persistentVolumeClaim:
      claimName: mongo-pvc
```

Key things here:

- **kind** - Deployment - defines the type of Kubernetes object. Here, it is deployment, which manages the creation and scaling of Pods.
- **replicas -** Defines the number of pod replicas (copies) to be created. Here, it is set to 1, meaning one instance of Mongo will be running.
- **container** – list the containers that will run inside each pod
    - name - name of the container, in this case mongo
    - image – specifies the container image to use, in this case mongo that we pulled from registry/docker hub
    - imagePullPolicy - indicates when to pull the image. We used IfNotPresent means the image will be pulled only if it is not already present on the node. This avoids unnecessary pulls if the image is already cached.
- **ports**: Exposes the port 27017 on the container
- **volumeMounts**: Defines how volumes (storage) are mounted into the container's filesystem
    - name - specifies the name of the volume being mounted.
    - mountPath: /data/db - This defines the path inside the container where the volume is mounted. In MongoDB, /data/db is the directory where MongoDB stores its database files.
- **volumes** - This section defines the storage volumes available to the pod
    - name - The name of the volume. It must match the volume specified in volumeMounts
    - persistentVolumeClaim - This indicates that the volume is backed by a Persistent Volume Claim (PVC), which allows for persistent storage that survives pod restarts.
    - claimName: mongo-pvc - This specifies the name of the Persistent Volume Claim (PVC) to use for the volume. The PVC must be created beforehand or defined elsewhere in the Kubernetes configuration.

Mongo PVC

Mongo request for storage

```
apiVersion: v1
kind: PersistentVolumeClaim
```

```
metadata:
  name: mongo-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Key things here:

- **kind**: PersistentVolumeClaim – Defines what kind of object you want to create. Here, it's a PersistentVolumeClaim is a request for storage that pods can use**.**
- **accessModes**: defines how the volume can be accessed protocol
    - ReadWriteOnce – means that the volume can be mounted as read-write by a single node at a time
- **storage**: This defines the size of storage being requested. In this case, it requests 1 GiB (gibibyte) of storage for the MongoDB database.

Mongo Service

Exposes Mongo internally within the cluster

```
apiVersion: v1
kind: Service
metadata:
  name: mongo-service
spec:
  selector:
    app: mongo
  ports:
    - protocol: TCP
      port: 27017
      targetPort: 27017
```

Key things here:

- **kind**: Service – Defines what kind of object you want to create. Here, it's a Service a method for exposing a network application that is running as one or more Pods in your cluster.
- **ports**: Describes the ports that the service will expose.
    - protocol: specifies the protocol used, which is TCP (Transmission Control Protocol). Most common protocol for web applications.
    - port – the port that the service will expose, in this case it is set to 27017. Meaning, the requests sent to this port of the service will be forwarded to the pods.
    - targetPort – the port on the container where traffic should be routed. This is set to 27017, which matches the port that the Express container inside the pod is listening on.
- **type**: The type of service under spec was omitted because React is using ClusterIP which is the default type if type is not specified. ClusterIP since react and express should be only accessible internally within the Kubernetes cluster.

Mongo Express Deployment

Defines the deployment for Mongo-express

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: mongo-express-deployment
 labels:
  app: mongo-express
spec:
 replicas: 1
 selector:
 matchLabels:
   app: mongo-express
 template:
  metadata:
   labels:
    app: mongo-express
  spec:
   containers:
   -name: mongo-express
    image: mongo-express
```

```
    ports:
    - containerPort: 8081
    env:
    -name: PORT
     value: "8081"
    - name: ME_CONFIG_MONGODB_URL
     value: "mongodb://mongo-service:27017/"
    - name: ME_CONFIG_SITE_BASEURL
     value: "/mongo-express/"
```

Key things here:

- **kind**: Deployment - defines the type of Kubernetes object. Here, it is deployment, which manages the creation and scaling of Pods.
- **replicas**: Defines the number of pod replicas (copies) to be created. Here, it is set to 1, meaning one instance of Express will be running.
- **container** – list the containers that will run inside each pod
    - name: name of the container, in this case mongo-express
    - image – specifies the container image to use, in this case mongo-express that pulled from registry/docker hub
    - imagePullPolicy: indicates when to pull the image. We used IfNotPresent means the image will be pulled only if it is not already present on the node. This avoids unnecessary pulls if the image is already cached.
- **ports**: Exposes the port 8081 on the container
- **env**: Defines environment variables for the container. Critical variables such as the ME_CONFIG_MONGODB_URL, port, and ME_CONFIG_SITE_BASEURL are defined to ensure the interface works correctly with the MongoDB instance.
    - PORT - Sets the port for Mongo Express to use
    - ME_CONFIG_MONGODB_URL – MongoDB connection URL using the Mongo service
    - ME_CONFIG_SITE_BASEURL - Base URL for the Mongo Express web interface

Mongo Express Service

Exposes Mongo-express internally within the cluster

```
apiVersion: v1
kind: Service
metadata:
```

```
  name: mongo-express-service
spec:
  selector:
    app: mongo-express
  ports:
    - protocol: TCP
      port: 8081
      targetPort: 8081
```

Key things here:

- **kind**: Service – Defines what kind of object you want to create. Here, it's a Service a method for exposing a network application that is running as one or more Pods in your cluster.
- **ports**: Describes the ports that the service will expose.
    - protocol: specifies the protocol used, which is TCP (Transmission Control Protocol). Most common protocol for web applications.
    - port – the port that the service will expose, in this case it is set to 8081. Meaning, the requests sent to this port of the service will be forwarded to the pods.
    - targetPort – the port on the container where traffic should be routed. This is set to 8081, which matches the port that the Mongo-express container inside the pod is listening on.
- **type**: The type of service under spec was omitted because React is using ClusterIP which is the default type if type is not specified. ClusterIP since react and express should be only accessible internally within the Kubernetes cluster.

Nginxsecrets.yaml

This file serves as a template for creating a Kubernetes secret. The secret contains the SSL certificate needed by browsers for https connections. The secret is made available to nginx in its deployment configuration.

```
# Get base64 encoding for secret
cat /tmp/nginx.crt | base64
cat /tmp/nginx.key | base64

apiVersion: "v1"
kind: "Secret"
```

```
metadata:
  name: "nginxsecret"
namespace: "default"
type: kubernetes.io/tls
data:
  tls.crt: "INSERT .crt AS STRING"
  tls.key: "INSERT .key AS STRING"
```

Nginx conf file for Kubernetes

```
server {
      listen 80;
      listen 443 ssl;

      server_name localhost;

      if ($scheme = http) {
        return 301 https://$host$request_uri;
      }

      ssl_certificate /etc/nginx/ssl/tls.crt;
      ssl_certificate_key /etc/nginx/ssl/tls.key;

      location / {
        proxy_pass      http://react-service:3000;
        proxy_redirect    off;
        proxy_http_version 1.1;
        proxy_cache_bypass $http_upgrade;
        proxy_set_header  Upgrade $http_upgrade;
        proxy_set_header  Connection keep-alive;
        proxy_set_header  Host $host;
        proxy_set_header  X-Real-IP $remote_addr;
        proxy_set_header  X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header  X-Forwarded-Proto $scheme;
        proxy_set_header  X-Forwarded-Host $server_name;
        proxy_buffer_size       128k;
        proxy_buffers           4 256k;
```

```
        proxy_busy_buffers_size   256k;
    }

    location /api {
        proxy_pass      http://express-service:5000;
        proxy_redirect    off;
        proxy_http_version 1.1;
        proxy_cache_bypass $http_upgrade;
        proxy_set_header  Upgrade $http_upgrade;
        proxy_set_header  Connection keep-alive;
        proxy_set_header  Host $host;
        proxy_set_header  X-Real-IP $remote_addr;
        proxy_set_header  X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header  X-Forwarded-Proto $scheme;
        proxy_set_header  X-Forwarded-Host $server_name;
        proxy_buffer_size       128k;
        proxy_buffers           4 256k;
        proxy_busy_buffers_size   256k;
    }

    location /mongo-express {
        proxy_pass      http://mongo-express-service:8081;
        proxy_redirect    off;
        proxy_http_version 1.1;
        proxy_cache_bypass $http_upgrade;
        proxy_set_header  Upgrade $http_upgrade;
        proxy_set_header  Connection keep-alive;
        proxy_set_header  Host $host;
        proxy_set_header  X-Real-IP $remote_addr;
        proxy_set_header  X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header  X-Forwarded-Proto $scheme;
        proxy_set_header  X-Forwarded-Host $server_name;
        proxy_buffer_size       128k;
        proxy_buffers           4 256k;
        proxy_busy_buffers_size   256k;
    }
}
```

Nginx Deployment

Defines the deployment for Nginx

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: nginx-deployment
 labels:
  app: nginx
spec:
 replicas: 1
 selector:
 matchLabels:
   app: nginx
 template:
  metadata:
   labels:
    app: nginx
  spec:
   volumes:
   - name: secret-volume
    secret:
     secretName: nginxsecret
   - name: configmap-volume
    configMap:
     name: nginxconfigmap
   containers:
   - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
    ports:
    - containerPort: 80
    - containerPort: 443
    volumeMounts:
    - mountPath: /etc/nginx/ssl
     name: secret-volume
    - mountPath: /etc/nginx/conf.d
```

name: configmap-volume
```

**Key things here:**

- **kind**: Deployment - defines the type of Kubernetes object. Here, it is deployment, which manages the creation and scaling of Pods.
- **replicas**: Defines the number of pod replicas (copies) to be created. Here, it is set to 1, meaning one instance of Nginx will be running.
- **container** – list the containers that will run inside each pod
    - name: name of the container, in this case Nginx
    - image – specifies the container image to use, in this case nginx that we pulled from registry/docker hub
    - imagePullPolicy: indicates when to pull the image. We used IfNotPresent means the image will be pulled only if it is not already present on the node. This avoids unnecessary pulls if the image is already cached.
- **ports**: Exposes the port 80 and 443 on the container
- **volumes**: Defines the volumes that will be available to the containers in the Pod.
    - name: secret-volume and configmap-volume are the names of the volume
    - secret: specifies that secret-volume is populated with data from the nginx secreat that we created
    - secretName: the name of the secret object to use, nginxsecret. This secret object contains sensitive information, like SSL certificates.
    - configMap: specifies that configmap-volume is populated with data from nginxconfigmap that we created
    - configMap name: The name of the ConfigMap object to use, nginxconfigmap. This ConfigMap contains configuration files for Nginx.
- **volumeMounts**: Specifies how volumes should be mounted into the container.
    - **mountPath**: /etc/nginx/ssl is the path inside the container where the secret-volume will be mounted. This is where Nginx will find SSL certificates. The name secret-volume refers to the volume defined earlier that contains the Secret.
    - **mountPath**: /etc/nginx/conf.d is the path inside the container where the configmap-volume will be mounted. This is where Nginx will find its configuration files. The name configmap-volume refers to the volume defined earlier that contains the ConfigMap.

```

Nginx Service

Exposes nginx externally

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  type: LoadBalancer
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
      name: http
      nodePort: 32000
    - protocol: TCP
      port: 443
      targetPort: 443
      name: https
      nodePort: 32001
```

Key things here:

- **kind**: Service – Defines what kind of object you want to create. Here, it's a Service a method for exposing a network application that is running as one or more Pods in your cluster.
- **ports**: Describes the ports that the service will expose.
    - protocol: specifies the protocol used, which is TCP (Transmission Control Protocol). Most common protocol for web applications.
    - port – the port that the service will expose, in this case it is set to 80 and 443. Meaning, the requests sent to this port of the service will be forwarded to the pods.
    - targetPort – the port on the container where traffic should be routed. This is set to 80 and 443, which matches the ports that the nginx container inside the pod is listening on.

- name: It is recommended to provide a name for each port for clarity and easier service discovery.
- nodePort: 32000, 32001 - Specifies the port on each node (worker) for accessing the Service. This port will be used for external access to the Service since we're using LoadBalancer type.

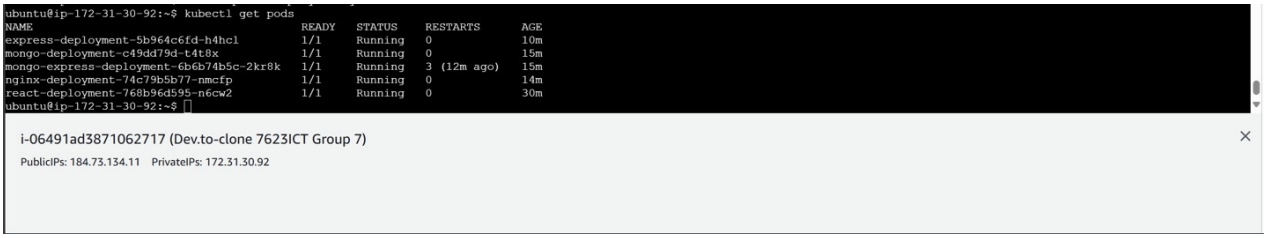
- **type**: The type of service under spec is set to LoadBalancer. This exposes the Service externally using an external load balancer.

Exposing Nginx to external traffic

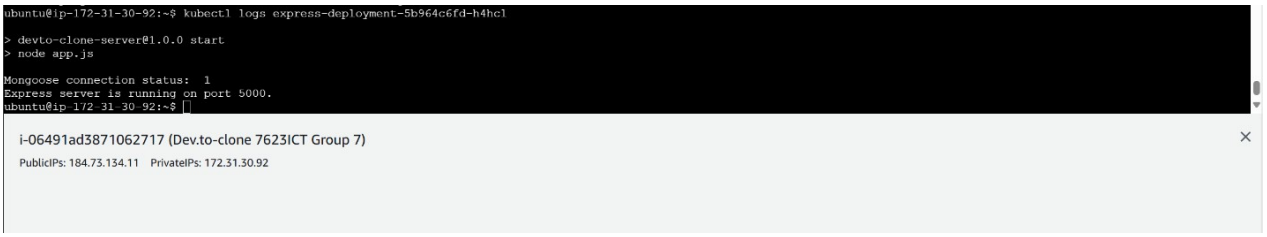
Nginx Port 443 was forwarded to the EC2 instance's port 8081, exposing the React, Mongo-express and Express services to external traffic.

```
kubectl port-forward svc/nginx-service 8081:443 --address 0.0.0.0 &
```

Screenshots of the Kubernetes deployment

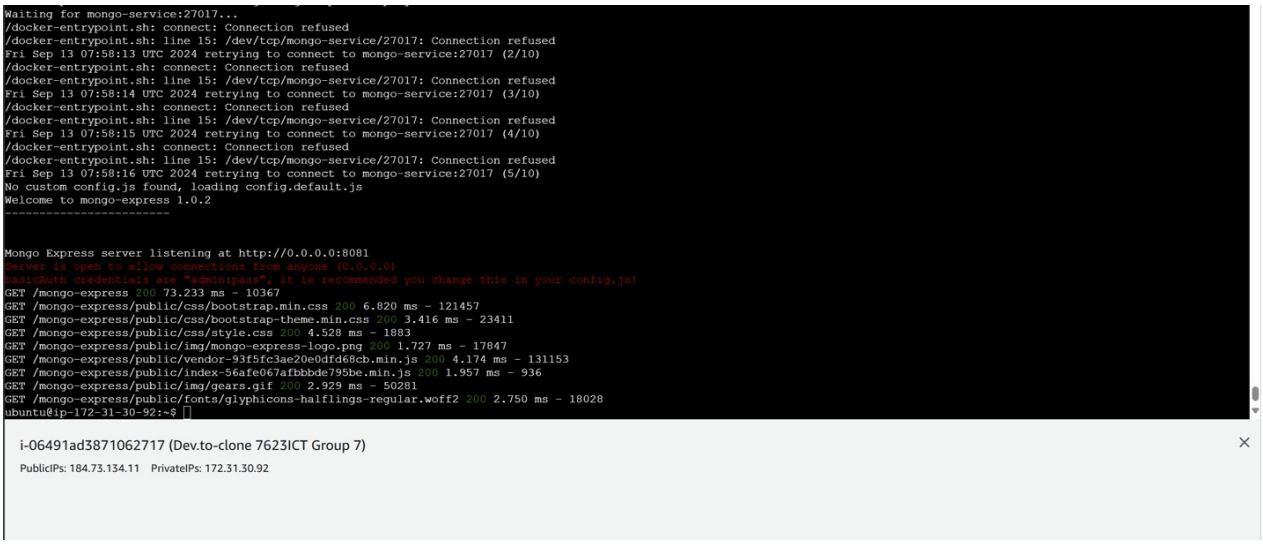


Kubectl get pods: running

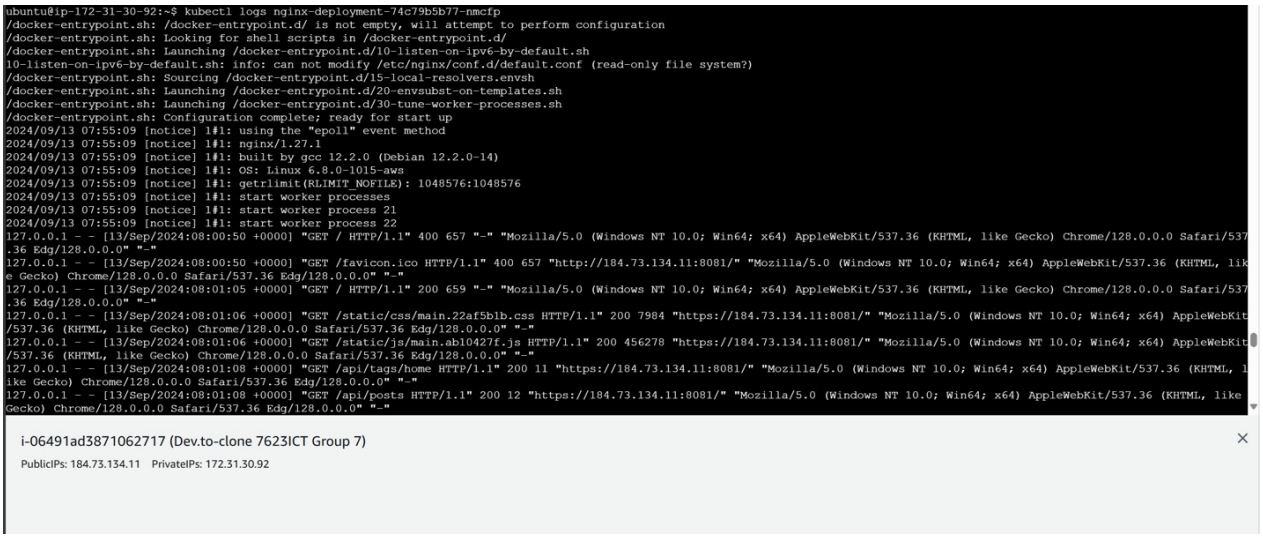


Logs of express-deployment

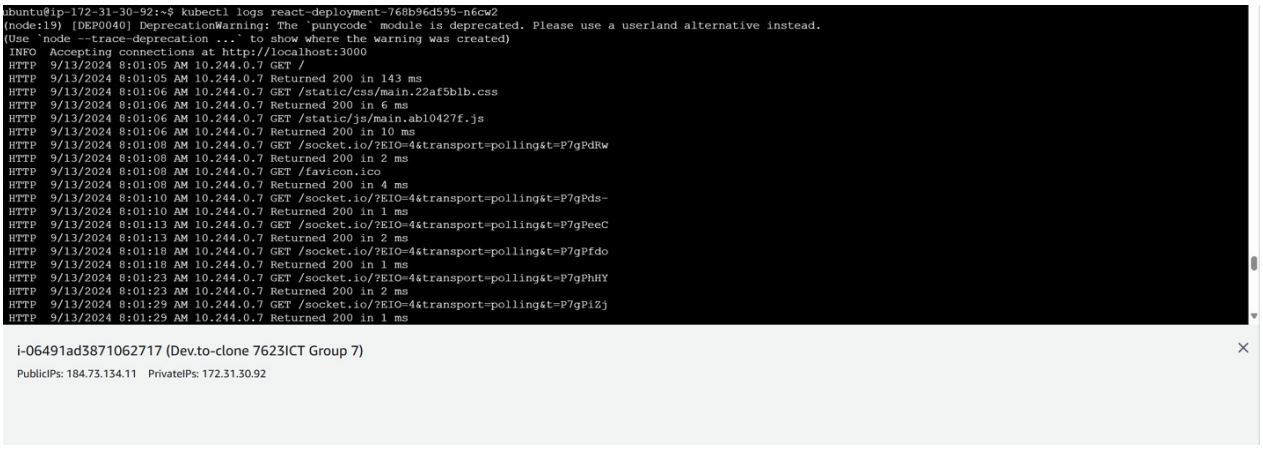
Logs of mongo-deployment



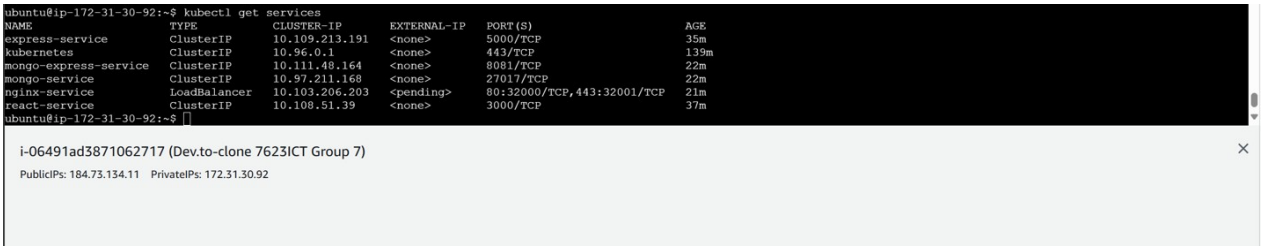
Logs of mongo-express deployment



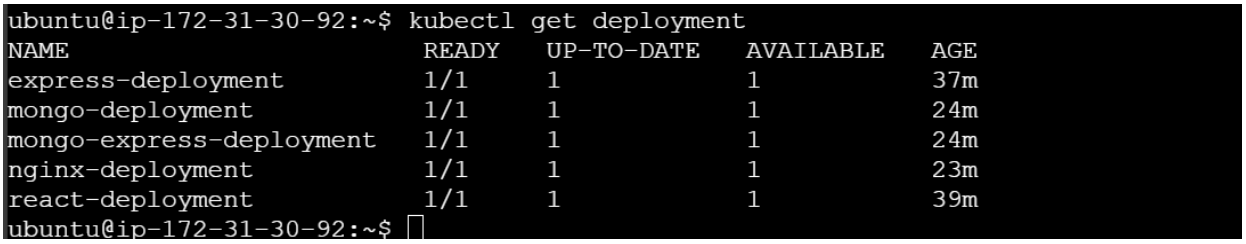
Logs of nginx deployment



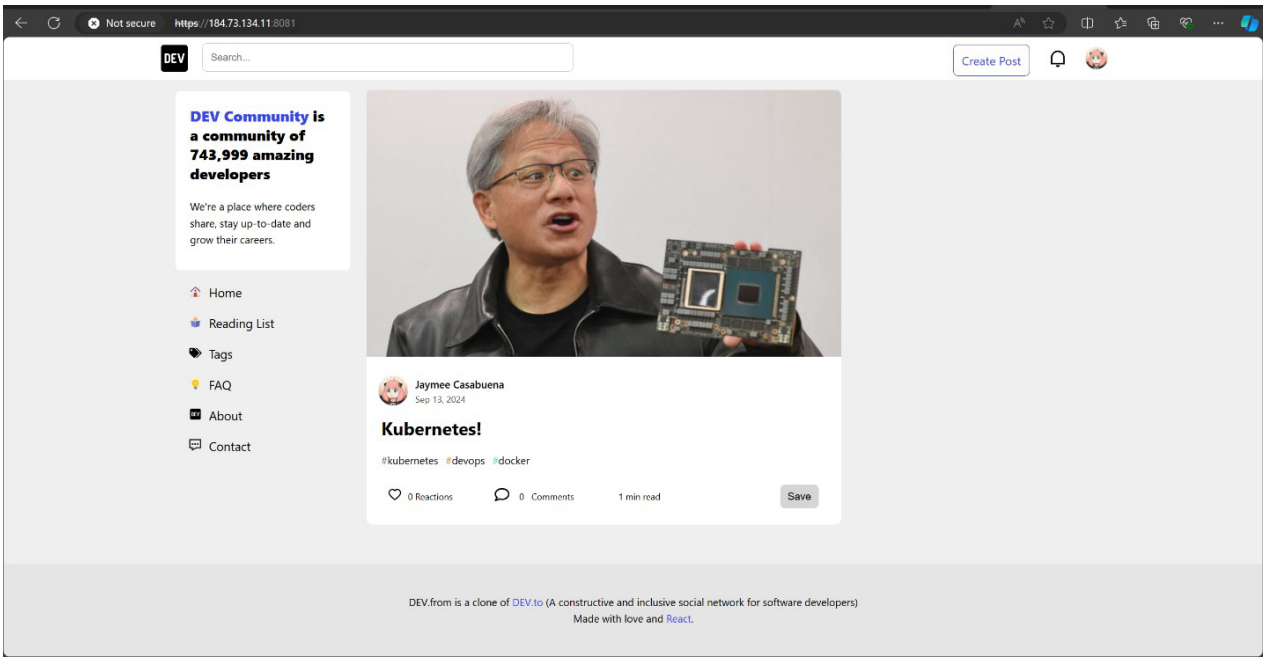
Logs react deployment



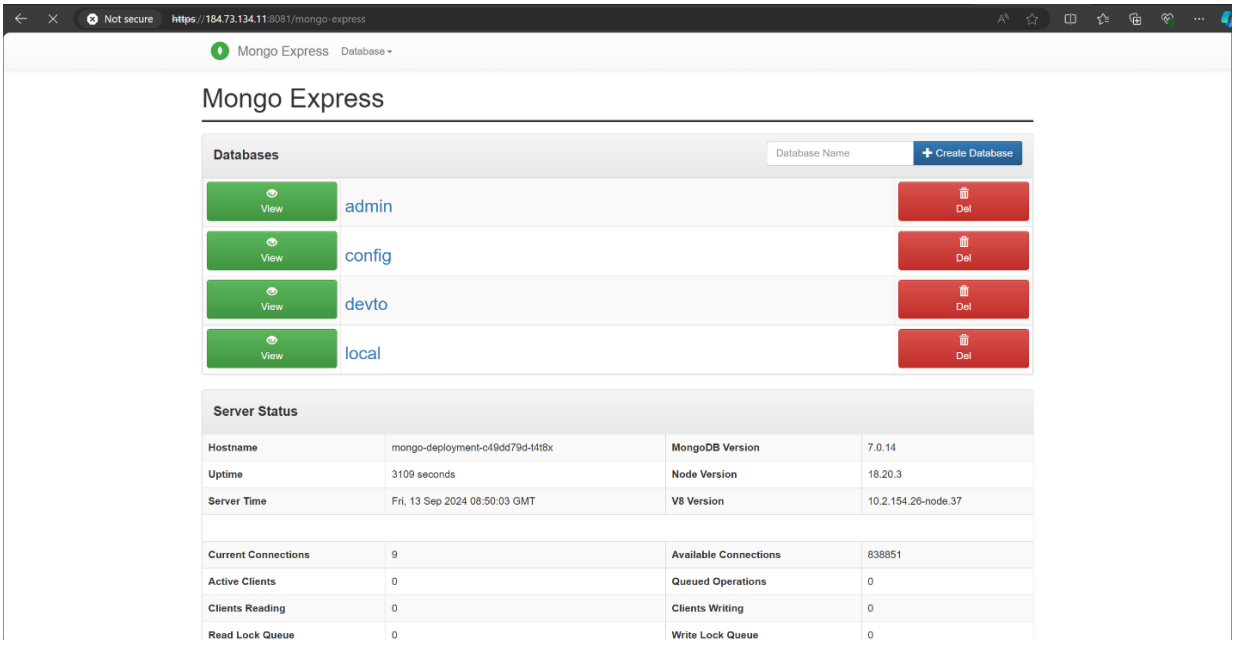
Kubectl get services



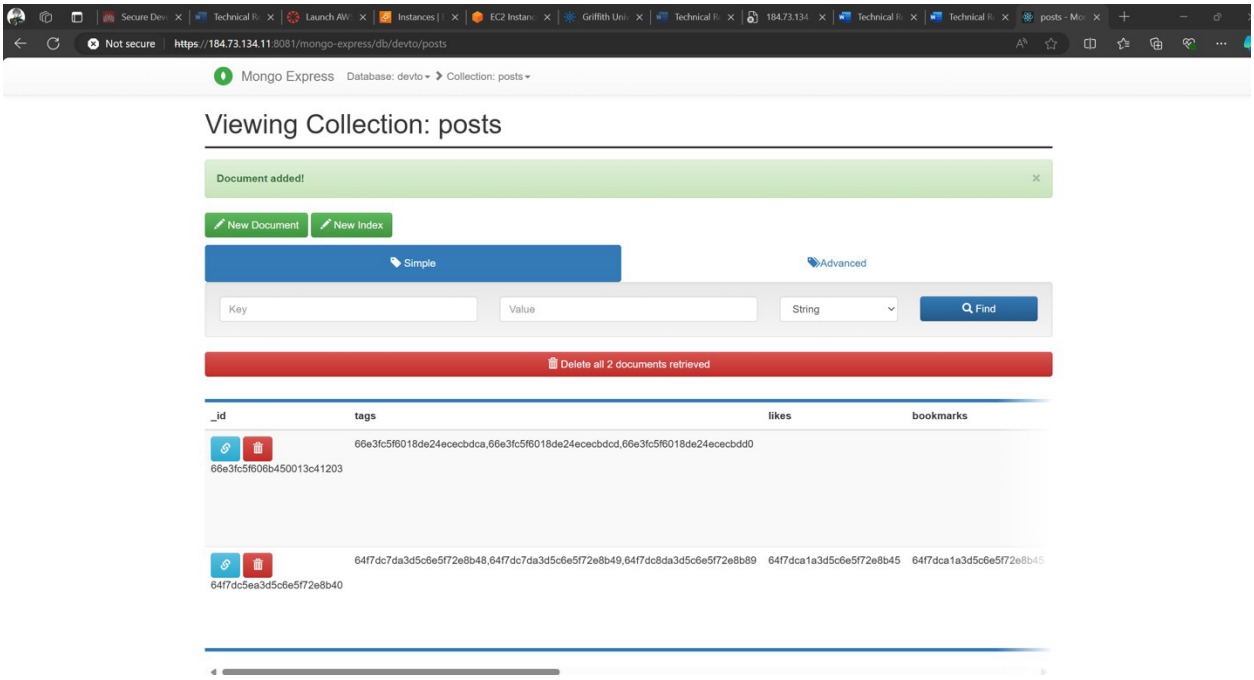
Kubectl get deployment

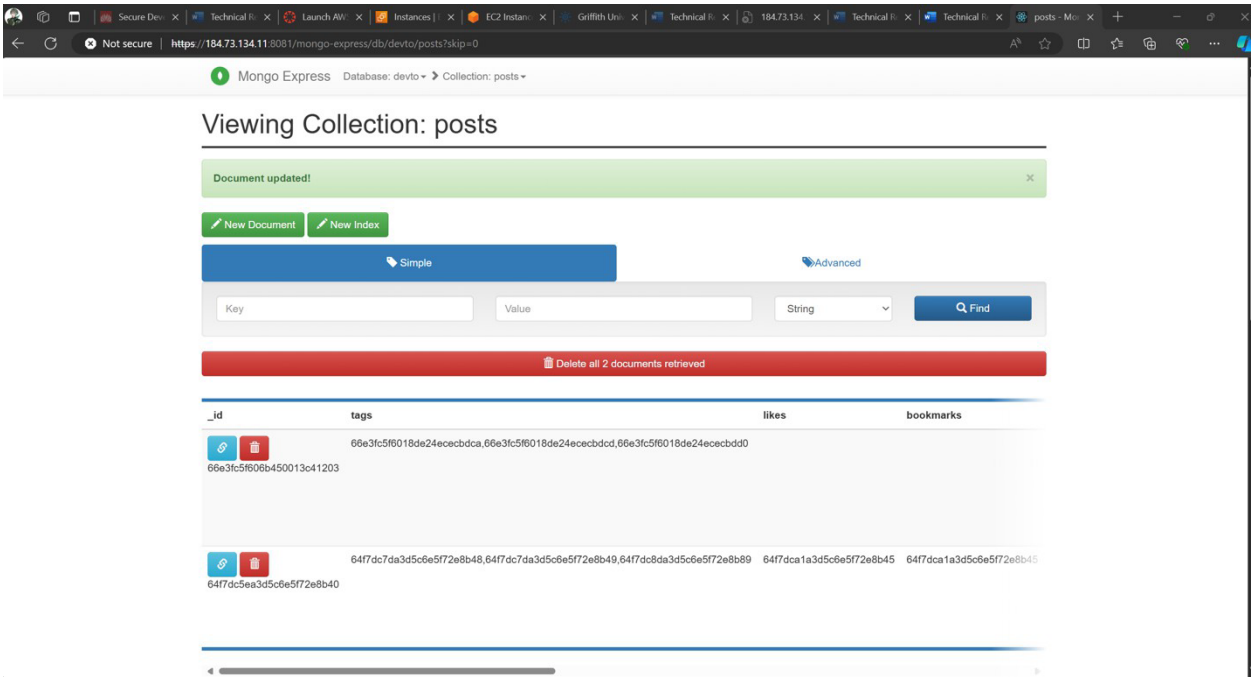


Https://localhost:8081



https://localhost:8081/mongo-express





Can add/update data through mongo-express

Summary:

- **Port Mappings:** Each service exposes relevant ports (React: 3000, Express: 5000, MongoDB: 27017, Mongo Express: 8081, Nginx: 80, 443) either internally or externally to enable proper communication between containers and external clients.
- **Environment Variables:** The deployment of both the Express and Mongo Express applications require specific environment variables to function correctly. For the Express app, these variables ensure proper connection to MongoDB, configure JWT authentication, and set up Cloudinary integration. On the other hand, the environment variables for Mongo Express are needed to establish a connection to MongoDB.
- **Replicas:** Each deployment uses a single replica, ensuring one instance of each container runs. This could be scaled up for high availability.

2.0 Task 2 – Vulnerability

2.1 ZAP Testing

Zap testing was performed by all members, initially just an automated test on the hosted web app, which report one high risk vulnerability, (reported on in 2.2) and a few other medium (Reported on in section 2.4) and low vulnerabilities.

An additional manual scan with the attack mode enabled, as well as an active scan revealed several more as shown in the screenshot below, including an Injection vulnerability which is also reported on below.

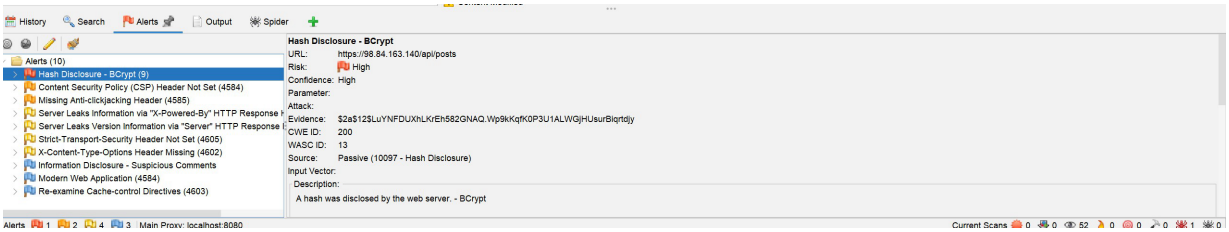
Screenshot 1: Zap results.

2.2 Vulnerability #1

Name: Exposure of Private Personal Information to an Unauthorized Actor
CWE Code: CWE-359

3.2.1 Detection:

The vulnerability was detected through an automated ZAP scan that flagged responses from several API endpoints for returning sensitive user data. We confirmed this issue by inserting print statements in the backend code to analyze the API response payload, revealing that the application had inadvertently exposed sensitive information in the response. The specific issue was related to the use of the populate() method on the author field in MongoDB queries.



Screenshot 2: Zap alerts result – Hash Disclosure Bcrypt

For example, the getPostsById function below shows how the populate() method led to this exposure:

```
const getPostById = async (req, res, next) => {
  try {
    const { postId } = req.params;
    const post = await Post.findById(postId)
      .populate("author",)
      .populate("comments")
      .populate("tags");
    //findById works directly on the contructor fn

    //post is a special mongoose obj; convert it to normal JS obj using toObject
    //get rid of "_" in "_id" using { getters: true }

    if (!post) {
      return next(
        new HttpError("Could not find post for the provided ID", 404)
      );
    }
    res.json({ post: post.toObject({ getters: true }) });
  } catch (error) {
    console.error(error);
    //stop execution in case of error
    return next(new HttpError("Something went wrong with the server", 500));
  }
};
```

Screenshot 3: getPostById function, which retrieves post by its id (eknoorpreet, n.d.)

In this code, the populate() method populates the author field with sensitive information that should not be visible to unauthorized users.

3.2.2 Description:

The vulnerability is caused by the exposure of sensitive information, such as hashed data, metadata, and some information that may be classified as PII (personally identifiable information), through API response payload. The code unintentionally inserts sensitive information into resources in this case response payload that is accessible to unauthorized actors. However, they should not contain the information - i.e., the information should have been "scrubbed" or "sanitized." (CWE, 2024). The website fails to adequately protect a person's private, personal information from being accessed by actors who are either (1) not explicitly authorized to access the information or (2) do not have the person's implicit consent (CWE, 2024).

An example from the CWE website is the following code that stores user's personal information into a log file.

Example 1

The following code contains a logging statement that tracks the contents of records added to a database by storing them in a log file. Among other values that are stored, the getPassword() function returns the user-supplied plaintext password associated with the account.

Example Language: **C#** *(bad code)*

```
pass = GetPassword();
...
dbmsLog.WriteLine(id + ":" + pass + ":" + type + ":" + tstamp);
```

Screenshot 4: Provided example of CWE 359 vulnerability (CWE, 2024)

Another example from the code the getSearchResults function below shows how the populate() method led to this exposure:

```
const getSearchResults = async (req, res, next) => {
  const query = {};
  if (!req.query.search) return;

  try {
    const options = "$options";
    query.title = { $regex: req.query.search, [options]: "i" };
    const posts = await Post.find(query).populate("author").populate("tags");
    console.log("HELLO", posts);

    res.status(201).json({
      posts: posts.map((post) => post.toObject({ getters: true })),
    });
  } catch (error) {
    console.error(error);
    return next(new HttpError("Search failed, please try again", 400));
  }
};
```

Screenshot 5: getSearchResults function, which retrieves posts based on keyword provided (eknoorpreet, n.d.)

Exploitation:

Attackers can exploit this vulnerability by simply sending requests to endpoints like /posts/:userId, which return sensitive data without proper checks. This could include user information, passwords (hashed or otherwise), or other confidential details. An attacker may use this data for further malicious purposes, such as identity theft or getting access privileges within the system.

Impact:

If this vulnerability is exploited, the consequences could include:

- Disclosure of user information (e.g., names, emails, hashed passwords)

- Compromise of user privacy and confidentiality
- Attackers gain insights into system architecture or internal workings, making subsequent attacks easier (e.g., NoSQL injections, identity theft, phishing, fraud)

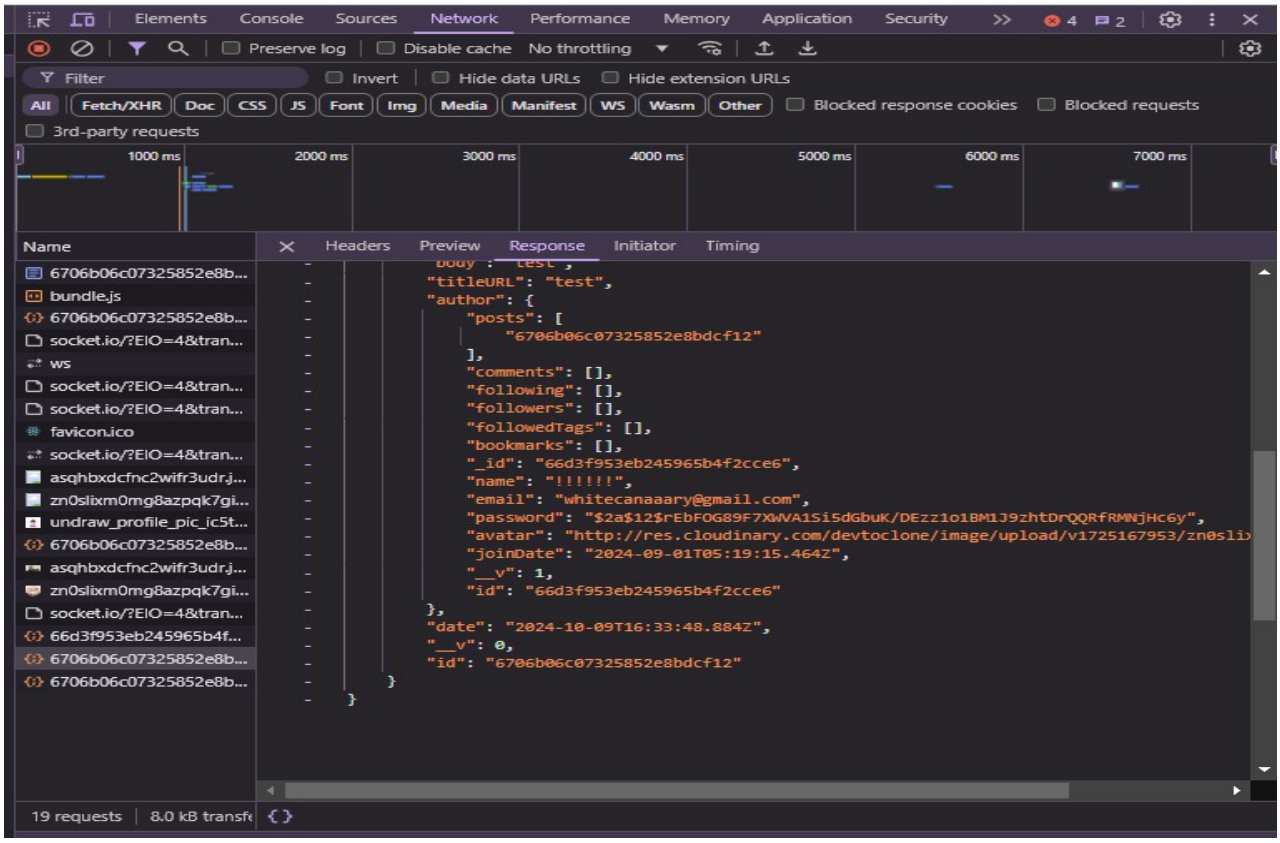
3.2.3 Justification:

Exposure of private personal information to an unauthorized actor occurs primarily because of methods like .populate() in the code provided. When querying the database, these methods reveal additional data fields. If sensitive fields are not correctly filtered out or sanitized, the responses could expose private personal information. An exposure of private information does not always mean that the product will stop working; in fact, the leak may have been the developer's intention. However, public disclosure of private information is still undesirable and prohibited by law or regulation (CWE, 2024).

The vulnerability also exposes users' hashed passwords. While a hash may seem secure, it still represents sensitive information about the user. Attackers can perform brute force or dictionary attacks to guess the original values, especially if the hash function is weak, lacks salting, or is compromised by known vulnerabilities. Moreover, revealing details like the structure of the data or specific fields provides attackers with enough insight to craft more precise attacks.

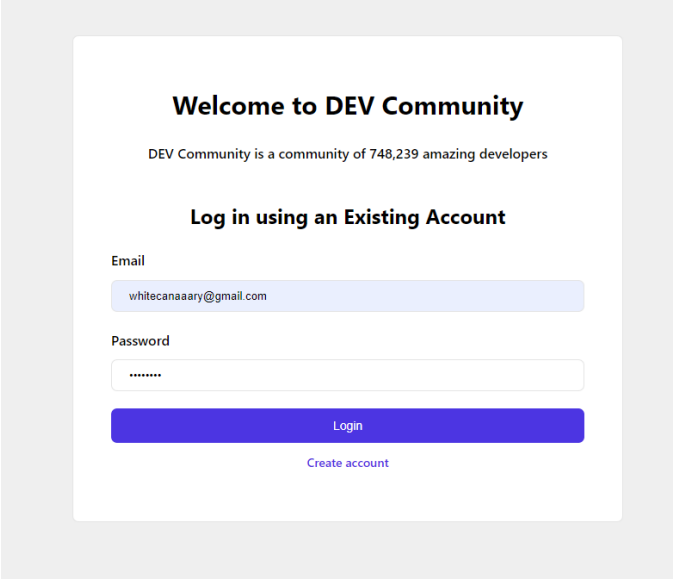
Verification:

We examined how anyone who knows how to use inspect and understands development tools can easily create an API request could retrieve private personal information. For instance, by inspecting the network activity in developer tools after clicking on a post, an unauthorized user can obtain details such as user IDs, email addresses, and other personal information of the post creator.



Screenshot 6: Response payload of getPostById showing the exposure of user data

Once they have access, bad actors can exploit this information to perform brute-force attacks using the exposed email addresses. This is particularly dangerous if the application lacks mechanisms to prevent multiple login attempts.



Screenshot 7: Login page with compromised email obtained from exposed user information

Additionally, email addresses can also be misused for phishing attacks or unsolicited marketing emails. As an example, about 92 million private customer email addresses were sold by an AOL employee to a spammer promoting an offshore gambling website in 2004 (Oates, 2005). As a result of these high-profile exploits, there is a growing regulatory framework surrounding the gathering and handling of personal data (CWE, 2024).

Lastly, this vulnerability can be compounded by other vulnerabilities in the system, if present, such as broken access control, where one user could make changes to another's account by knowing their user ID.

Source Code:

The code utilizes populate("author"), which means that when fetching posts or search results, it also includes additional author details in the response. Since the query didn't specify what parameters should be only included, the app is exposing the rest of the data under author field.

```
const getPostById = async (req, res, next) => {
  try {
    const { postId } = req.params;
    const post = await Post.findById(postId)
      .populate("author",)
      .populate("comments")
      .populate("tags");
    //findById works directly on the contructor fn
    console.log("getPostById", post);

    //post is a special mongoose obj; convert it to normal JS obj using toObject
    //get rid of "_" in "_id" using { getters: true }

    if (!post) {
      return next(
        new HttpError("Could not find post for the provided ID", 404)
      );
    }
    res.json({ post: post.toObject({ getters: true }) });
  } catch (error) {
    console.error(error);
    //stop execution in case of error
    return next(new HttpError("Something went wrong with the server", 500));
  }
};
```

Screenshot 8: getPostById with additional print statement (eknoorpreet, n.d.)

By adding print statements in the code above, I confirmed the amount of personal information exposed by using the populate method to get author details.

```
author: {
  posts: [ 6706b06c07325852e8bdcf12 ],
  comments: [],
  following: [],
  followers: [],
  followedTags: [],
  bookmarks: [],
  _id: 66d3f953eb245965b4f2cce6,
  name: '!!!!!!',
  email: 'whitecanaaary@gmail.com',
  password: '$2a$12$rEbFOG89F7XWVA1Si5dGbuK/DEzz1o1BM1J9zhtDrQQRfRMNjHc6y',
  avatar: 'http://res.cloudinary.com/devtoclone/image/upload/v1725167953/zn0slixm0mg8azpqk7gi.jpg',
  joinDate: 2024-09-01T05:19:15.464Z,
  __v: 1
},
date: 2024-10-09T16:33:48.884Z,
```

Screenshot 9: Terminal log displaying the fetched post data with populated author.

3.2.4 Remediation:

Ensure only the required fields are exposed in the API response. This can be done by adjusting the .populate() function or modifying how the data is mapped to objects.

Based on Mongoose documentation, the second parameter of the populate method is a field selection string. So, specifying only to select required information would limit the exposure of sensitive data from the author field. The code below specified that only names, avatars, and posts should be part of the response. We also specified that we should exclude _id, which MongoDB always includes by default.

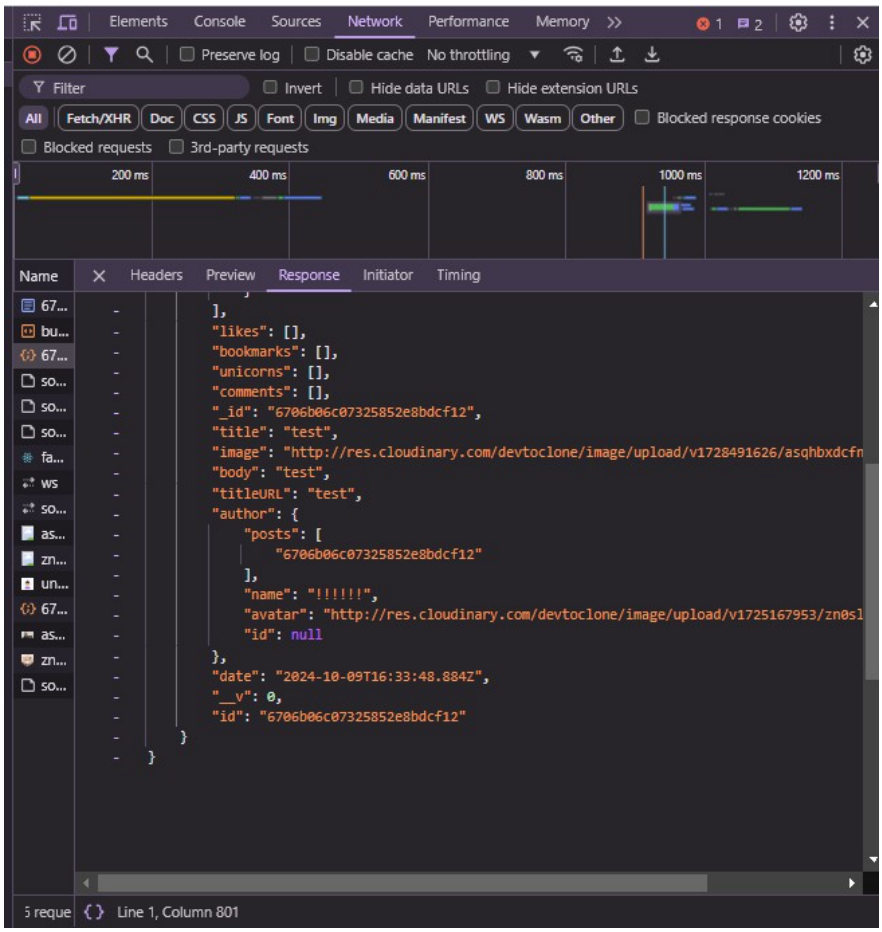
```
const getPostById = async (req, res, next) => {
  try {
    const { postId } = req.params;
    const post = await Post.findById(postId)
      .populate("author", "name avatar posts -_id")
      .populate("comments")
      .populate("tags");
    //findById works directly on the contructor fn
    console.log("getPostById", post);

    //post is a special mongoose obj; convert it to normal JS obj using toObject
    //get rid of "_" in "_id" using { getters: true }

    if (!post) {
      return next(
        new HttpError("Could not find post for the provided ID", 404)
      );
    }
    res.json({ post: post.toObject({ getters: true }) });
  } catch (error) {
    console.error(error);
    //stop execution in case of error
    return next(new HttpError("Something went wrong with the server", 500));
  }
};
```

Screenshot 10: getPostById with the suggested fix

After the changes, we rechecked the network, and the personal information of the post creator was no longer exposed/included.



Screenshot 11: Response payload of getPostById showing only necessary and non-sensitive fields

2.3 Vulnerability #2

Name: Cleartext Storage of Sensitive Information

CWE Code: CWE-312

3.2.1 Detection:

The vulnerability was discovered through manual code inspection of the React application's authentication function. In addition, the browser dev tools (console and network features) were used to inspect he hosted application and confirm the vulnerability.

3.2.2 Description:

The vulnerability is that sensitive user information is being stored in an easy-to-read, plain text format, in a location that can be easily accessed by an attacker (CWE, 2024).

Examples:

According to CWE (2024), an example of the vulnerability is using a cookie on the user's browser store login credentials in plaintext.

In this case, the vulnerability results from the React front-end application storing the logged-in user's plaintext JWT authentication token, along with other personal identifiers such as the user's ID and email in the browser's "*localStorage*".

Exploitation:

The are various ways in which attackers might be able to retrieve the victim's token from local storage, including:

- If an attacker gains access to a user's machine (CWE, 2024).
- A cross-site scripting (XSS) attack where the token is retrieved from local storage by JavaScript inserted by the attacker (auth0, .n.d.).

Impact:

If an attacked obtained the victim's token, they could append the token to Authorization header of any request to pass the server's authentication middleware, allowing them to execute server functions that perform CRUD operations.

For example, the attacker can delete resources created by the victim, create new resources as the victim, or modify the victim's personal details.

3.2.3 Justification:

The reason for the vulnerability classification is due to the user's auth token, along with other personal identifiers such as email address and user ID, being stored in the browser's local storage.

The code screenshot below shows where this occurs:

```
13      //useCallback((uid, token, expirationDate)
14      const login = useCallback((user, expirationDate) => {
15        // setToken(token);
16        // setUserId(uid);
17        setToken(user.token);
18        setUserId(user.userId);
19        setUser(user);
20
21        const tokenExpirationDate =
22          expirationDate || new Date(new Date().getTime() + 1000 * 60 * 60);
23        setTokenExpirationDate(tokenExpirationDate);
24        localStorage.setItem(
25          'userData',
26          JSON.stringify({
27            userId: user.userId,
28            token: user.token,
29            bio: user.bio,
30            avatar: user.avatar,
31            email: user.email,
32            name: user.name,
33            tags: user.tags,
34            expiration: tokenExpirationDate.toISOString(),
35          })
36        );
37      }, []);
```

Screenshot 12: Setting token to local storage. useAuth.js (eknoorpreet, n.d.)

Screenshot 5 below proves that the token and other user data are indeed stored as plain text and can be accessed by typing the *JavaScript command* localStorage.getItem("userData") into the dev tools console:

Screenshot 13: Retrieving user data from local storage.

The ability to easily access local storage data by JavaScript (e.g. through and XSS attack) proves that local storage is an unsafe location to store sensitive information, meeting the description of CWE-312.

Note that the CWE class code is CWE-922: Insecure Storage of Sensitive Information (CWE, 2024).

Why the token is sensitive information:

The Json Web Token (JWT) is sensitive information because it forms part of requests that are sent to the Express server to perform CRUD operations on the Mongo database. React appends the token to the requests' "authorization" header as a bearer token to prove that an authenticated user initiated the request.

```
try {
  await sendReq(
    `${process.env.REACT_APP_BASE_URL}/comments/${commentId}`,
    'DELETE',
    JSON.stringify({ author: currentUserId }),
    {
      'Content-Type': 'application/json',
      Authorization: `Bearer ${currentUser.token}`,
    }
  );
} catch (err) {}
```

Screenshot 14: Appending JWT to request header. DeleteComment.js (eknoorpreet, n.d.).

In the screenshot above, the token is used to allow a request to delete comment.

The express server will either allow or reject the requests based on whether the requests' authentication token can be verified or not, as shown below:

```
  const token = req.headers.authorization.split(' ')[1]; //Authorization: 'Bearer TOKEN'
  const isCustomAuth = token.length < 500; //> 500 = Google auth
  if (!token) {
    throw new Error('Authentication failed!');
  }
  let decodedToken;
  if (isCustomAuth) {
    decodedToken = jwt.verify(token, JWT_KEY);
    req.userData = { userId: decodedToken.userId }; //add data to request
  } else {
    decodedToken = jwt.decode(token);
    req.userData = { userId: decodedToken?.sub }; //add data to request
  }
  next(); //let the request continue
```

Screenshot 15: Server-side request auth token verification. check-auth.js (eknoorpreet, n.d.)

Suppose an attacker successfully copies the user token. They could theoretically craft a request with the token appended as an authorisation header, an send it to the server.

However, the code provides a much more convenient way for the attacker to exploit the vulnerability, as will be explained below:

Example of how the vulnerability can be exploited:

Suppose an attacker obtains another user's local storage data, this is how they may exploit it.

The same code that stores the token in local storage, also retrieves it from local storage when the user refreshes the web page, as shown in the screenshot below:

```
useEffect(() => {
  const storedData = JSON.parse(localStorage.getItem('userData'));
  if (storedData && new Date(storedData.expiration) > new Date()) {
    login(
      // storedData.userId,
      // storedData.token,
      storedData,
      new Date(storedData.expiration)
    );
  }
}, [login]); // [] => only run once when the cmp is mounted first time
```

Screenshot 16: Function that retrieves localStorage and logs the user in. useAuth.js (eknoorpreet, n.d.)

The react useEffect hook executes the encapsulated code when the web page initially loads (eknoorpreet, 2024).

The function retrieves the "*userData" as "storedData",* which includes an auth token as shown in Screenshot 2 above. If "*storedData*" exists, the function executes the login function shown in screenshot 1 above, which extracts the token value, and sets it to React's state.
Once the token is in react state, React will deem the user to be logged in, and append the token to any requests sent to the server, as shown in screenshot 3 above.

The purpose of this function is to persist user session, so that is the user refreshed their web page, React would repopulate it's authentication state with the information from local storage.

However, this means an attacker can follow these simple steps to load the user's session into their own browser:

1. Obtain the "*userData"* from the victim's local storage, assuming the victim is logged into the site. For example, by means of an XSS attack.

2. The attacker can then visit the website on his own machine.

3. Next, the attacker uses his dev tools console to set the victim's "*userData*" into his own local storage, using the localStorage.setItem("userData", ….) command.

4. Lastly, the attacker simply refreshes his browser. React will load the victim's *userData* and consider the attacker to be authenticated as the victim.

In this case, the attacker can bypass the login or registration forms and use the website to create, update or delete the victim's data, and the vulnerability is successfully exploited.

3.2.4. Remediation:

According to Auth0, the safest method of keeping a token is in browser memory, by using wither web-workers or JavaScript closures (n.d.). However, this method means that session information will be lost if the user refreshes the page (Auth0, n.d.), meaning that the user would have to log in again.

The recommended solution would be for the application to implement the Auth0's token management package, to remove the concern of safe token storage and session

persistence, since these features were not securely implemented by the author of the website.

Auth0 provides the following documentation on how to implementation their token-management software: https://auth0.com/docs/libraries/auth0-single-page-app-sdk

In this case, the implementation can be in the React app's useAuth.js file, and it can replace the problematic token- and-session-management code shown in this report.

2.4 Vulnerability #3

Name: Improperly Implemented Security Check for Standard

CWE Code: CWE-358

3.2.1 Detection:

The vulnerability was detected during a passive scan using OWASP ZAP. The tool flagged the application for missing the **Content Security Policy (CSP)** header in the HTTP responses. The absence of this header indicates that the application does not have an effective security policy to help mitigate certain types of attacks, such as Cross-Site Scripting (XSS) and data injection attacks.

Screenshot 17: Zap alerts result – Content Security Policy (CSP) Header Not Set

3.2.2 Description:

The **Content Security Policy (CSP)** header is a crucial security feature that enables website owners to specify which sources of content are considered safe to load in the browser. This includes JavaScript, CSS, images, and other resources. By defining a CSP, websites can significantly reduce the risk of XSS attacks and other code injection vulnerabilities, which can lead to data theft, site defacement, or the distribution of malware.

Examples:

A common scenario arises when an application fails to implement a CSP header. Without it, attackers can inject malicious scripts that are executed in the context of the user's session.

Example:

- An attacker could exploit a vulnerability in the web application to inject a malicious script. If the CSP header is not set, the browser will execute the injected script, potentially leading to unauthorized access to user data or credentials.

Exploitation:

If this vulnerability is exploited, it can lead to serious security issues:

- An attacker may inject harmful scripts that can execute in the user's browser without any restrictions. This could result in data theft, session hijacking, or spreading malware to users.

Example Input:

- An attacker crafts a payload that, when executed, accesses sensitive information or performs actions on behalf of the user. Without the CSP header, the browser does not restrict this action.

Impact:

Exploiting this vulnerability can lead to:

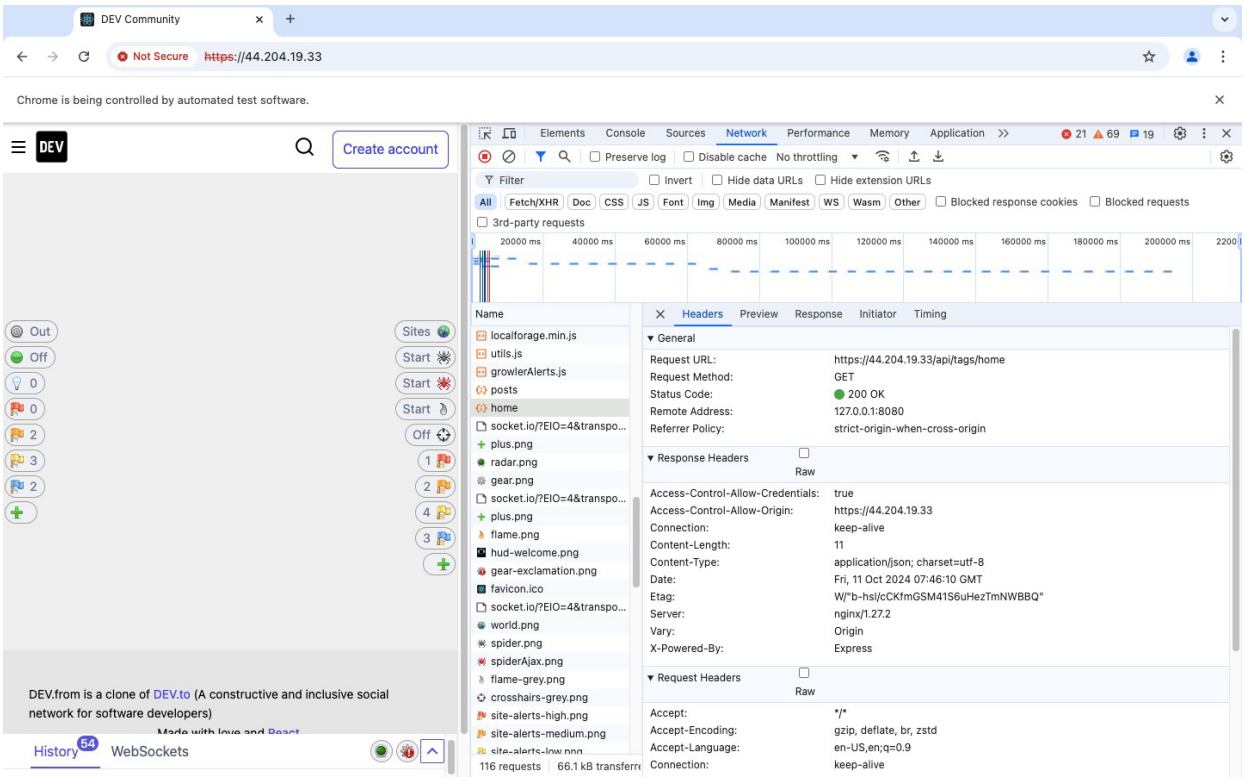
- **Cross-Site Scripting (XSS)** attacks, allowing attackers to execute scripts in the context of a user's session.
- **Data breaches**, where sensitive user data can be stolen.
- **Site defacement** or distribution of malware to users.

3.2.3 Justification:

This vulnerability falls under **CWE-358: Improperly Implemented Security Check for Standard** because the lack of a CSP header signifies a failure to adhere to security standards that help protect against common web attacks. It is a clear indication of not implementing an essential security feature that is widely recognized and recommended in web development practices.

Verification:

To verify this vulnerability, we inspected the application's HTTP response headers and checked if the **Content Security Policy** was in place.



Screenshot 18: Response headers missing content-security-policy

As seen from the screenshot above, the CSP header is not set.

3.2.4. Remediation:

To remediate this vulnerability, ensure that the Content-Security-Policy header is included in all server responses. This header should specify the allowed sources of content to help prevent potential attacks.

Implementation:

First, we installed helmet package by running *npm install helmet. Helmet* is a package that helps secure Express apps by setting HTTP response headers (npm, 2024). By default, the helmet package sets several headers that can be further configured.

Helmet sets the following headers by default:

- `Content-Security-Policy` : A powerful allow-list of what can happen on your page which mitigates many attacks
- `Cross-Origin-Opener-Policy` : Helps process-isolate your page
- `Cross-Origin-Resource-Policy` : Blocks others from loading your resources cross-origin
- `Origin-Agent-Cluster` : Changes process isolation to be origin-based
- `Referrer-Policy` : Controls the `Referer` header
- `Strict-Transport-Security` : Tells browsers to prefer HTTPS
- `X-Content-Type-Options` : Avoids MIME sniffing
- `X-DNS-Prefetch-Control` : Controls DNS prefetching
- `X-Download-Options` : Forces downloads to be saved (Internet Explorer only)
- `X-Frame-Options` : Legacy header that mitigates clickjacking attacks
- `X-Permitted-Cross-Domain-Policies` : Controls cross-domain behavior for Adobe products, like Acrobat
- `X-Powered-By` : Info about the web server. Removed because it could be used in simple attacks
- `X-XSS-Protection` : Legacy header that tries to mitigate XSS attacks, but makes things worse, so Helmet disables it

Screenshot 19: Helmet package sets headers by default (npm, 2024)

Then, we added the code below to the app.js file, which just set the default directives object.

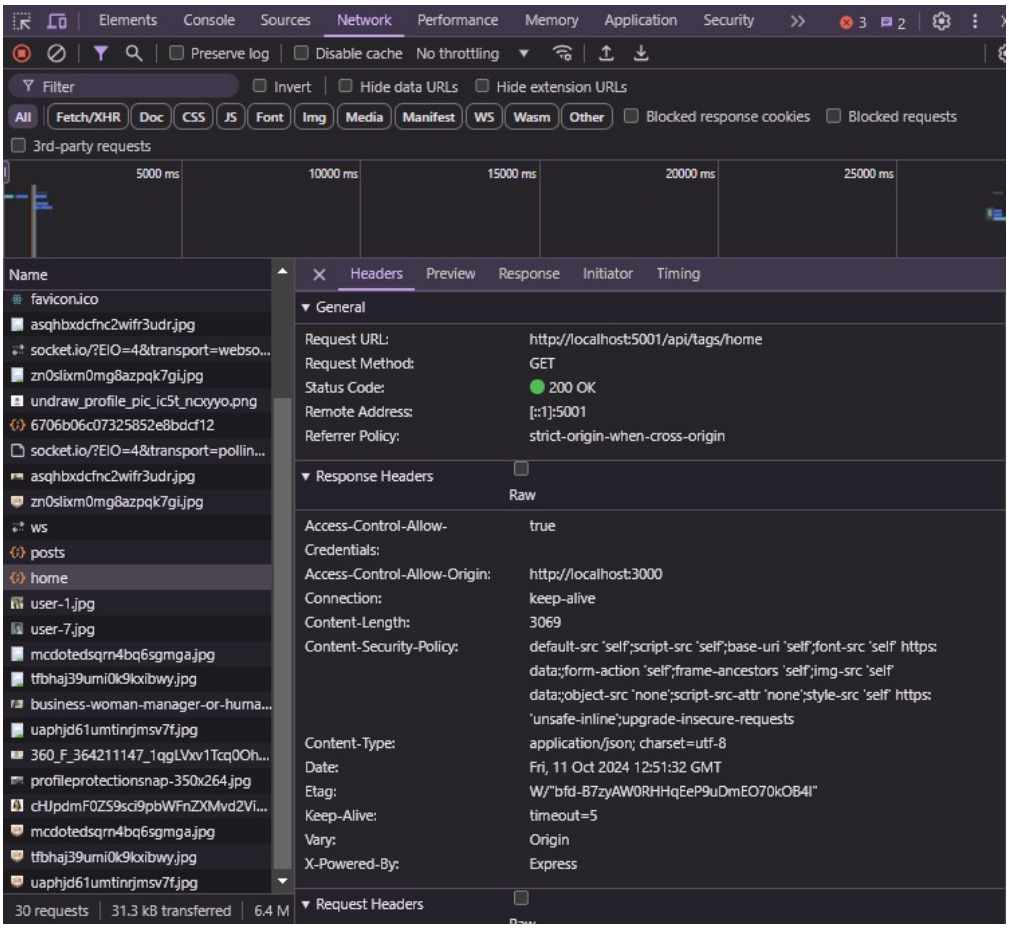
```
app.use(
  helmet.contentSecurityPolicy({
    directives: {
      defaultSrc: ["'self'"],
      scriptSrc: ["'self'"],
    },
  })
);
```

Screenshot 20: App.js file with additional helmet configuration (eknoorpreet, n.d.)

Impact of Remediation:

By implementing the CSP header, we can significantly reduce the risk of cross site scripting and other injection attacks by controlling the sources from which content can be loaded (npm, 2024).

After adding the helmet package, we rechecked the network tab of the web application, and the CSP header is now present.



Screenshot 21: Response headers with content-security-policy

2.5 Vulnerability #4

Name: Improper Neutralization of Special Elements in Data Query Logic

CWE Base Code: 943

Severity: High

Detection Method:

This vulnerability was identified during manual ZAP testing, where ZAP flagged a potential SQL injection issue in an SQLite database interaction. The vulnerability was classified as high risk with medium confidence, indicating that while the threat is potentially severe, further investigation may be needed to confirm the actual risk.

SQL Injection - SQLite	High	6 (24.0%)

Figure 1: SQL Injection identified in ZAP Report Summary

SQL Injection - SQLite

Source	raised by an active scanner (plugin ID: 40024)
CWE ID	89
WASC ID	19
Reference	▪ https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

Figure 2 Additional Information about SQL Injection in ZAP Report Appendix

Description:

During testing, ZAP mapped this vulnerability to CWE-89, which is typically associated with SQL injection attacks. However, since the project we're working on uses MongoDB, a NoSQL database, we determined that CWE-943 is a more appropriate classification.

While CWE-89 focuses specifically on SQL databases, CWE-943 is a parent class that covers all types of data query injection attacks, including those targeting NoSQL

databases. Given that the injection issue in our project arises from unsanitized inputs being passed into MongoDB queries (not SQL commands), mapping this vulnerability to CWE-943 ensures more accurate representation, as it encompasses the broader range of injection vulnerabilities, including those in NoSQL environments (*CWE - CWE-943: Improper Neutralization of Special Elements in Data Query Logic (4.9)*, n.d.).

Explanation (issue):

NoSQL injection attacks occur when unsanitized or improperly validated inputs are passed to a NoSQL database (MongoDB). Since MongoDB queries are structured in JSON-like format, attackers can manipulate the queries by injecting specially crafted objects or strings. This can bypass authentication, retrieve sensitive data, or modify the database.

Exploitation and Impact:

This vulnerability poses significant security risks, leading to three major issues. First, *Data Exposure*: NoSQL injection can grant attackers unauthorized access to sensitive information, including user credentials, personal data, and even administrative details, putting user privacy and application integrity at risk (*NoSQL Injection | Web Security Academy*, n.d.). Second, *Account Takeovers*: In login page scenarios, attackers can exploit NoSQL injection to bypass authentication without valid credentials, potentially compromising user accounts. If an admin account is taken over, the attacker can escalate their privileges, causing further damage. Third, *Data Manipulation*: Malicious queries can be injected to modify or delete database entries, disrupting services, corrupting data, and causing unpredictable application behaviour (*What Is NoSQL Injection? Examples, Prevention, and More*, n.d.).

Additionally, this vulnerability remains a serious concern because an attacker can insert complex queries into the database. For instance, they could spam the login request with queries that take a long time to process, such as using a complex regex value for the email field. This could render the server unresponsive for other users, leading to a *Denial-of-Service (DoS) attack (What Is NoSQL Injection? | MongoDB Attack Examples | Imperva, n.d.)*. By overloading the system with time-consuming queries, attackers can cause service outages, affecting all users and severely impacting the application's availability. These risks highlight the critical need to address this vulnerability to ensure data security, service reliability, and user trust.

Justification: The login function for our project located in server > controller > users.js looks like this:

```
const login = async (req, res, next) => {
    try {
        const { email, password } = req.body
        const existingUser = await User.findOne({ email }).populate('followedTags')

        //user doesn't exist (invalid credentials)
        if (!existingUser) {
            return next(new HttpError('Invalid credentials, login failed!', 403))
        }

        //validate password
        const isValidPassword = await bcrypt.compare(
            password,
            existingUser.password
        )
        //invalid password
        if (!isValidPassword) {
            return next(new HttpError('Invalid credentials, login failed!', 401))
        }

        //everything ok => generate token
        const token = jwt.sign(
            { userId: existingUser.id, email: existingUser.email },
            JWT_KEY,
            { expiresIn: '1h' }
        )

        res.json({
            user: {
                name: existingUser.name,
                userId: existingUser.id,
                email: existingUser.email,
                token,
                bio: existingUser.bio,
                avatar: existingUser.avatar,
                tags: existingUser.followedTags,
            },
        })
    } catch (error) {
        console.error(error)
        return next(new HttpError('Login failed.', 500))
    }
}
```

Figure 3 Users.js of the Project

Upon reviewing the http://localhost:5001/api/users/login endpoint, it was discovered that the server lacks proper server-side validation for the email input field. This vulnerability allows an attacker to inject malicious JSON payloads, such as:

```
{
    "email": { "$gte": "b" },
    "password": { "$gte": "" }
}
```

Figure 4 NoSQL Injection Payload

Without proper input validation and sanitization, the application is vulnerable to NoSQL injection attacks, which can potentially lead to unauthorized access or security breaches.

The injection successfully retrieves a user from the database, as demonstrated here,

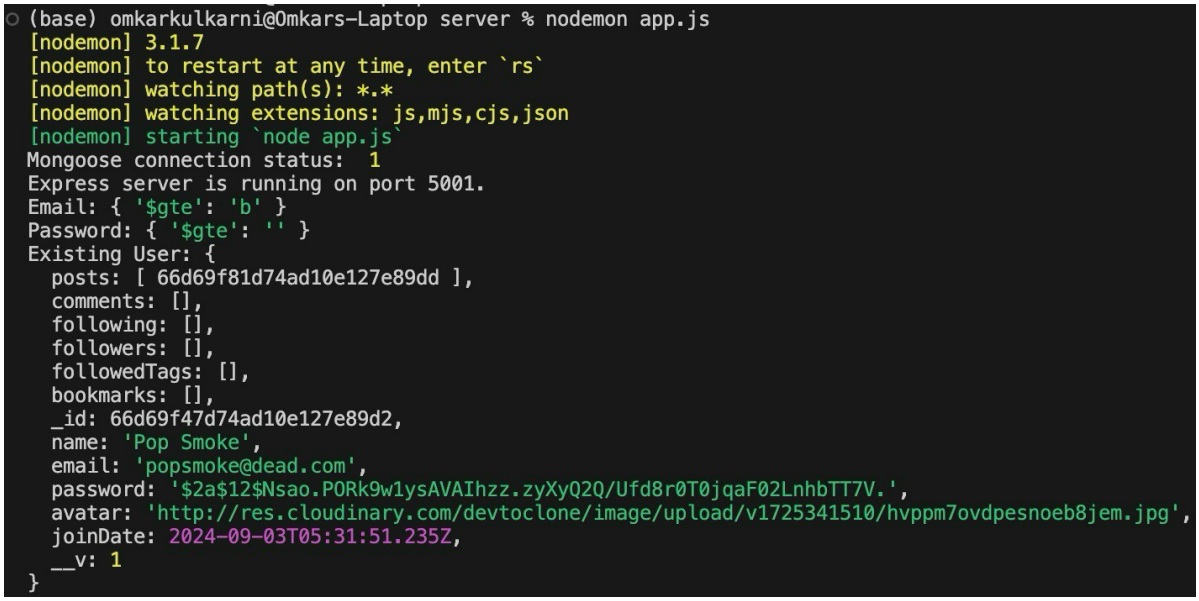


Figure 5 Output of NoSQL Injection in the terminal

However, since bcrypt is used for password validation and expects a string, the comparison fails, resulting in a "login failed" error, as shown here

Figure 6 Postman response to NoSQL Injection

The critical issue here is not that the attacker successfully logs in or receives user data in Postman, but that the input fields, such as email, are not properly sanitized to prevent malicious MongoDB queries from being executed.

For example: when an attacker submits a malicious MongoDB query through the email field, the server directly passes this query into the findOne() function without validating that it's a legitimate email address. MongoDB interprets the input as a query and executes it, potentially returning a user or other data, depending on the query.

Although the login attempt ultimately fails due to password mismatch, the root issue is that the server processes the query rather than treating the input as a valid email. This demonstrates the critical need for input sanitization to prevent NoSQL injection vulnerabilities.

Remediation:

To effectively resolve this NoSQL injection issue, the following steps should be implemented:

1. **Input Validation**: Ensure all inputs are strictly validated before they are passed to the database. For example, the email field should be validated to ensure it is a properly formatted string and not a malicious query or object. The validator library can be used to handle this. After installing and importing validator in users.js file, we can use the following code to validate the email:

```
const validator = require('validator');


if (!validator.isEmail(email)) {
        return next(new HttpError("Invalid email format!", 400));
}
```

This step prevents invalid data from being processed by the database (*Validator*, n.d.).

2. **Input Sanitization**: In addition to validation, sanitization should be applied to remove any potentially dangerous characters or MongoDB query operators (e.g., $gte, $ne, etc.) from user inputs. This ensures that only clean, expected input is processed. After installing and importing express-mongo-sanitize in users.js file, it can be implemented in the following way:

```
const express = require('express');
const mongoSanitize = require('express-mongo-sanitize');

const app = express();


const userRouter = require('./api/routes/userRoutes');
const postRouter = require('./api/routes/postRoutes');


app.use(express.json());
// Data sanitization against NoSQL query injection
app.use(mongoSanitize());


app.use('/api/v1/users', userRouter);
app.use('/api/v1/posts', postRouter);


app.all('*', (req, res, next) => {
 next(new AppError(`Can't find ${req.originalUrl} on this Server!`, 404));
});


app.use(globalErrorHandler);

module.exports = app;
```

A function called `mongoSanitize` is invoked, which returns a middleware designed to prevent NoSQL injection attacks. This middleware works by inspecting request bodies, parameters, and query strings to identify and remove potentially dangerous characters, such as dollar signs (`$`) and dots (`.`). By doing so, it ensures that these characters, which could be used to manipulate database queries, are sanitized before any interactions with the database store, thereby enhancing the security of the application (Akeren, 2023).

REFERENCES

https://kubernetes.io/docs/concepts/overview/working-with-objects/

https://kubernetes.io/docs/concepts/services-networking/service/

https://kubernetes.io/docs/tutorials/services/connect-applications-service/#securing-the-service

https://github.com/kubernetes/examples/blob/master/staging/https-nginx/default.conf

https://stackoverflow.com/questions/76099101/nginx-redirect-http-to-https-in-reverse-proxy-to-remote-site

Auth0. (n.d.) *Token Storage: Browser local storage scenarios.*

Retrieved on October 11, 2024, from https://auth0.com/docs/secure/security-guidance/data-security/token-storage#browser-local-storage-scenarios

CWE. (2024, July 16). *CWE-312: Cleartext Storage of Sensitive Information.*

Retrieved October 10, 2024, from https://cwe.mitre.org/data/definitions/312.html

eknoorpreet. (n.d.) dev.to-clone. GitHub.

Retrieved on October 11, 2024, from https://github.com/eknoorpreet/dev.to-clone

https://www.theregister.com/2005/02/07/aol_email_theft/

https://cwe.mitre.org/data/definitions/359.html

Akeren, K. (2023, September 5). *Data sanitization against NoSQL query injection in MongoDB and Node.js application*. Medium.

https://medium.com/@akerenkater/data-sanitization-against-nosql-query-injection-in-mongodb-and-node-js-application-dad2ffca5f01

Chetan Karande. (2024). *SQL and NoSQL Injection | OWASP NodeGoat Tutorial*. Gitbooks.io.

https://ckarande.gitbooks.io/owasp-nodegoat-tutorial/content/tutorial/a1_-_sql_and_nosql_injection.html

CWE - CWE-943: Improper Neutralization of Special Elements in Data Query Logic (4.9). (n.d.). Cwe.mitre.org. https://cwe.mitre.org/data/definitions/943.html

express-validator. (n.d.). Npm.

https://www.npmjs.com/package/express-validator

Mitigating MongoDB injection attacks with Mongoose. (n.d.). Stack Overflow.

https://stackoverflow.com/questions/52707399/mitigating-mongodb-injection-attacks-with-mongoose

mitre. (2013). *CWE - CWE-89: Improper Neutralization of Special Elements used in an SQL Command ("SQL Injection") (3.4.1)*. Mitre.org.

https://cwe.mitre.org/data/definitions/89.html

NoSQL injection | Web Security Academy. (n.d.). Portswigger.net.

https://portswigger.net/web-security/nosql-injection

OWASP. (2024). *SQL Injection Prevention · OWASP Cheat Sheet Series*. Owasp.org; Owasp.

https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

validator. (n.d.). Npm. https://www.npmjs.com/package/validator

What Is NoSQL Injection? | *MongoDB Attack Examples* | *Imperva*. (n.d.). Learning Center. https://www.imperva.com/learn/application-security/nosql-injection/

What Is NoSQL Injection? Examples, Prevention, and More. (n.d.). Discover.strongdm.com. https://www.strongdm.com/what-is/nosql-injection

npm. (2024). *Helmet* (Version 7.0.0) [Software]. npm. https://www.npmjs.com/package/helmet

OWASP Foundation. (n.d.). *CWE-358: Protection Mechanism Failure*. OWASP. https://cwe.mitre.org/data/definitions/358.html

OWASP Foundation. (n.d.). *CWE-693: Protection Mechanism Failure*. OWASP. https://cwe.mitre.org/data/definitions/693.html

OWASP Foundation. (n.d.). *Cross-Site Scripting (XSS)*. OWASP. Retrieved October 11, 2024, from https://owasp.org/www-community/attacks/xss/