Containerization of a MERN Application Using Docker

Overview:

This project involved containerizing a 3-tier MERN (MongoDB, Express.js, React, Node.js) stack application using Docker to enhance deployment, security, and scalability. The focus was on isolating application components, establishing seamless communication, and ensuring secure network configuration without using Docker Compose.

Key Objectives:

1. Containerize the MERN stack application for modularity and ease of deployment.

2. Create a custom virtual network to isolate containers for enhanced security.

3. Configure each container manually and ensure proper communication between them.

4. Deploy and manage containers using Docker CLI commands.

Implementation Details:

1. Application Structure:

* Frontend: Built using React, providing the user interface.

* Backend: Developed with Node.js and Express.js to handle business logic and APIs.

* Database: MongoDB, storing and managing application data.

* Each component was packaged into a dedicated Docker container to maintain modularity and scalability.

2. Network Configuration:

* Created a custom virtual network using Docker's bridge network functionality to isolate the containers.

* This network ensured that only the application's containers could interact with each other.

* Used container names as hostnames for seamless communication.

* [docker network create custom_network]

3. Container Configuration:

* Built Docker images for each component using a custom Dockerfile tailored for its dependencies.

* **Frontend Container** *

* React application served using a lightweight HTTP server.

* Exposed on port 3000 for user access.

* Connected to the backend container via the custom network.

[docker run -d --name frontend --network custom_network -p 3000:3000 frontend_image]

* **Backend Container** *

*Node.js application configured to handle API requests.

*Connected to both the frontend and the MongoDB container.

[docker run -d --name backend --network custom_network -p 5000:5000 backend_image]

* **Database Container** *

*MongoDB container exposed on port 27017 for backend connectivity.

[docker run -d --name mongodb --network custom_network -p 27017:27017 mongo]

4. Deployment Process:

* Each container was launched individually using Docker CLI commands.

* Network connectivity between containers was ensured via the custom virtual network.

Challenges Addressed:

1. Security:

Isolated containers using a custom Docker bridge network to prevent interference from other containers.

2. Portability:

Ensured the application runs consistently across different environments using Docker containers.

3. Manual Deployment:

Managed containers individually without relying on orchestration tools like Docker Compose.

Key Learnings:

1. Manually configuring and deploying a containerized application architecture.

2. Importance of custom networks for container isolation and security.

3. Practical usage of Docker CLI for creating, running, and managing containers.

Conclusion:

This project highlighted the benefits of using Docker to containerize and deploy a 3-tier application. By manually setting up containers and a secure network, the application achieved improved modularity, security, and scalability.

Outcome: