Docker Assignment

Capture The Flag (CTF) challenges are interactive competitions where you solve technical puzzles or complete tasks to find hidden "flags." A flag confirms that you have successfully finished a task. Usually, CTFs offer various challenges that push you to explore and troubleshoot technology in creative ways.

In this assignment, you will build the infrastructure to run two web-based CTF applications inside Docker containers. You will:

- Use persistent databases
- Set up a message queue for background tasks
- Create APIs to start and stop application containers
- Configure a reverse proxy for efficient web traffic routing

By completing this assignment, you will practice important cloud computing concepts like container orchestration, database persistence, asynchronous task handling, and automated infrastructure deployment. These skills will help you design and run modern, scalable web services.

Rules

You are encouraged to use Large Language Models (LLMs) such as ChatGPT extensively to help with implementation, troubleshooting, and understanding concepts. However, you should choose carefully when and how to use these tools. Always make sure you understand the LLM's output. Simply copying answers without understanding them can lead to long debugging sessions.

If you never use an LLM, this assignment may take you much longer to finish. Therefore, try to balance efficiency, accuracy, and clear understanding.

You are free to use any tools or technologies you prefer for each part (like Flask, FastAPI, or Django for the web server). The only requirement is that all components must run in Docker containers. If you pick a tool for reasons such as familiarity, features, or performance, please explain your choice in your documentation.

Sample CTF Question

In this assignment, you will deploy two containerized CTF challenges for each team. The first challenge uses the Docker image:

`pasapples/apjctf-todo-java-app:latest`

This is a Java-based to-do list application with deliberate security vulnerabilities. It behaves like a real web application and helps you learn about container behavior and app management in a safe environment.

The second challenge is OWASP Juice Shop, available via:

`bkimminich/juice-shop`

Juice Shop is a modern but deliberately insecure web application. It contains many security flaws at different levels of the stack. These two containers are the main services your infrastructure will manage for multiple teams.

Ask Questions

For any questions, you can reach me on Telegram:

`@reisi_mhdmdi`

Submission Instructions

Please follow these steps to submit your work:

1. **Fork the following repository:**
   https://github.com/mamadmr/Cloud-Assignment
2. **Create a folder named with your full name** in the root of the forked project.
3. **Add all documents and results** for each question to that folder. Include the supporting files, scripts, and any other relevant outputs.
4. **Upload your videos** to the IUT Box (or a similar file hosting service) and **place the links in your documents**.
5. **Write your documentation in a README file** (e.g., `README.md`) inside your folder. Make sure to include:
   - Explanations of your process.
   - Links to videos or any external resources.
   - Instructions on how to run your solution (if applicable).
6. **Open a Pull Request (PR)** back to the original repository (i.e., https://github.com/mamadmr/Cloud-Assignment) so that your changes can be reviewed and accepted.

Following these steps will allow me to easily check your work, review your code, and provide feedback.

Question

PostgreSQL is a reliable, open-source object-relational database system known for its features and standard compliance. It is common in both small and large projects because of its stability and flexibility. In this assignment, PostgreSQL stores persistent data about teams and their assigned challenges. By running it in a Docker container with proper volumes, your data will stay available even if the container is restarted or removed.

Problem 1: PostgreSQL

This problem will show you how to:

- Deploy a PostgreSQL container with persistent data storage
- Perform simple database operations

(a) **Deploy a PostgreSQL Container with Persistent Storage:**

(i) Run a PostgreSQL container in Docker.

(ii) Make sure the container is configured so that its data remains available after restarts. Use Docker volumes or bind mounts to achieve this.

(b) **Perform Basic Database Operations:**

(i) Connect to the PostgreSQL database with any client or command-line tool.

(ii) Execute SQL commands to:

(a) Create a new database.

(b) Create a table in that database.

(c) Insert some data into the table.

(d) Retrieve and display the data to verify everything worked.

(c) **Documentation and Demonstration:**

(i) Document the steps for setting up PostgreSQL with persistent storage, including the SQL commands used. Explain your decisions for the setup.

(ii) Record a short video (no longer than 30 seconds) showing:

(a) How you retrieve the inserted data.

(b) How you stop and remove the container.

(c) How you re-launch the container and confirm the data is still there, proving persistence works.

Redis is an open-source, in-memory data structure store. It is often used as a message broker, cache, or database. It supports different data types and is very fast and easy to use. In this assignment, Redis acts as a message broker between your web application and Celery so that you can handle tasks in the background. By containerizing Redis, you can see how separate services communicate through a high-performance shared messaging layer.

Problem 2: Redis Server Setup

This problem covers how to:

- Deploy a Redis server in Docker
- Carry out basic Redis operations
- Use Python programs to communicate through Redis

(a) **Deploy a Redis Server in a Docker Container:**

(i) Launch a Redis server inside a Docker container.

(ii) Make sure the Redis server accepts client connections.

(b) **Implement Inter-Process Communication Using Redis:**

(i) Write two separate programs (ideally in Python) that communicate through the Redis server.

(ii) In the first program:

(a) Set multiple key-value pairs in Redis.

(b) Publish messages to a Redis channel.

(iii) In the second program:

(a) Retrieve and display the key-value pairs set by the first program.

(b) Subscribe to the same Redis channel and display any messages published.

(c) **Monitor Redis Activity Using a Redis Insight Tool:**

(i) Use a Redis GUI client (for example, RedisInsight) to watch the data and messages in real time.

(ii) Take screenshots to show how the Redis server handles data and messages.

(d) **Documentation and Demonstration:**

(i) Document how you set up Redis, created the Python programs, and used the monitoring tool.

(ii) Record a short video (maximum 30 seconds) showing:

(a) How both Python programs run and communicate through Redis.

(b) How Redis activities appear in your chosen GUI client in real time.

Celery is a distributed task queue system that runs tasks in the background. In web applications, it is commonly used for sending emails, processing data, or handling external services without blocking the main application. In this assignment, Celery will use Redis as the message broker and will manage Docker containers for the CTF challenges. This allows you to run tasks in the background, making your system more efficient and responsive.

Problem 3: Container Management with Celery and Redis

You will set up Celery to manage tasks that start and stop Docker containers for the CTF challenges.

(a) **Set Up Celery with Redis as the Message Broker:**

(i) Configure Celery to use Redis as its message broker.

(ii) Make sure Celery can connect to Redis and run successfully.

(b) **Implement Celery Tasks for Container Management:**

(i) Write Celery tasks that:

(a) Start a Docker container for a specific CTF challenge.
(b) Stop a running Docker container using its ID.

(ii) Make sure these tasks run in the background (asynchronously) and handle any errors gracefully.

(c) **Demonstrate Task Execution and Container Lifecycle Management:**

(i) Call the Celery tasks to start and stop Docker containers.

(ii) Observe and confirm that the containers change state as expected before and after running the tasks.

(d) **Documentation and Demonstration:**

(i) Document how you set up Celery, Redis, and the container management tasks. Include any configurations you changed.

(ii) Record a video (up to 1.5 minutes) that shows:
(a) Celery tasks starting and stopping containers.
(b) The container states before and after task execution.

Problem 4: Web API for Team Challenge Management

You will build a web API to manage CTF challenge containers for different teams. The API should:

- Interact with a database to store data
- Use Celery with Redis for asynchronous container management

(Recommended framework: FastAPI)

(a) **Develop a Web API for Challenge Management:**

(i) Create API endpoints that:
(a) Assign a specific CTF container to a team based on a team ID and a challenge ID.
(b) Remove a CTF container from a team using the team ID and the challenge ID.

(ii) Make sure the API checks input data and handles errors properly.

(b) **Integrate with a Database for Persistent Storage:**

(i) Set up a database to keep track of team assignments and active challenge containers.

(ii) Update this database from the API when containers are assigned or removed.

(c) **Use Celery with Redis for Background Task Processing:**

(i) Configure Celery to use Redis as the message broker.

(ii) Write Celery tasks that:
   (a) Start a Docker container for a specified CTF challenge and link it to a team.
   (b) Stop and remove a Docker container that is linked to a team and a challenge.

(iii) Make sure the API calls these tasks when it needs to assign or remove containers.

(d) **Test the API Using Postman:**

(i) Use Postman (or a similar tool) to send requests to your API endpoints for assigning and removing containers.

(ii) Confirm that the API responds correctly and that containers are started or stopped. Also check that your database updates correctly.

(e) **Documentation and Demonstration:**

(i) Document how you built the API:
   (a) The purpose of each endpoint
   (b) The database schema
   (c) How Celery and Redis are configured
   (d) Instructions to set up and run the API

(ii) Record a video (maximum 2 minutes) demonstrating:
   (a) Making requests with Postman to assign and remove containers.
   (b) Seeing the containers appear or stop, and how the database updates.
   (c) Any logs or output showing how the API, Celery, Redis, and the database communicate.

NGINX Reverse Proxy Integration

Nginx is a fast web server often used as a reverse proxy, load balancer, or HTTP cache. In this assignment, Nginx will act as a reverse proxy to direct HTTP requests to the correct Docker containers running the CTF challenges. Each team will have a unique path (using a UUID) for their challenge. Nginx will make sure one team cannot access another team's challenge. This setup shows how Nginx can dynamically manage and expose containerized services in a cloud environment.

Problem 5: NGINX Reverse Proxy Integration

You will configure an NGINX server as a reverse proxy to forward requests to challenge containers. Each team should have a unique UUID-based path, which is hidden from other teams.

(a) **Configure NGINX as a Reverse Proxy:**

(i) Set up an NGINX container that works as a reverse proxy for your challenge containers.

(ii) Use a dynamic configuration tool (like `docker-gen`) or something similar to automatically add new proxy paths when a new container starts.

(iii) Assign a unique UUID-based route for each challenge container assigned to a team.

(b) **Enforce Access Isolation:**

(i) Make sure each team can only access its own challenge containers through its specific UUID path.

(ii) Prevent teams from guessing or using other teams' UUIDs.

(c) **Integrate with Your Web Application:**

(i) Update your existing API so it creates and saves a UUID path whenever a challenge is assigned.

(ii) Store these UUIDs in the database and pass them to NGINX to set up the routes.

(d) **Documentation and Demonstration:**

(i) Document the full setup:

(a) How the reverse proxy works and how UUIDs map to containers

(b) How you keep teams isolated from each other

(c) Any updates you made to the API or database

(ii) Record a video (up to 3 minutes) showing:

(a) Assigning and accessing challenges for two different teams

(b) Accessing the containers through the NGINX reverse proxy paths

Docker Compose Integration

Finally, you will combine all parts into a single setup using `docker-compose`. This includes:

- PostgreSQL
- Redis
- Celery
- Your web API
- NGINX reverse proxy
- The CTF challenge containers

Problem 6: Docker Compose Integration

(a) **Create a Docker Compose Setup:**

(i) Define all services in a `docker-compose.yml`.

(ii) Set up networking and dependencies so that each service can reach the others (e.g., the web app can access PostgreSQL, Redis, and Celery).

(iii) Use volume mounts for any services that need persistent data (e.g., PostgreSQL).

(b) **Ensure System Initialization and Operation:**

(i) When you run `docker-compose up`, the whole system should start and work.

(ii) You should be able to assign and remove challenges via the API and reach them through the NGINX reverse proxy.

(c) **Optional: Dynamic Cleanup of NGINX Paths:**

(i) Optionally, remove NGINX routes when a container is stopped or removed.

(ii) Tools such as `docker-gen` can automatically update NGINX configs, or you can write your own logic.

(d) **Documentation and Final Demonstration:**

(i) Include your `docker-compose.yml` file.

(ii) Write a short explanation of how the services are connected, and how to start and use the system.

(iii) If you did the optional cleanup, explain how it works and which tools or logic you used.

(iv) Record a video (up to 5 minutes) showing the entire system:

(a) An overview of the architecture

(b) Running the system with `docker-compose`

(c) Assigning and removing challenges for at least two teams

(d) Container creation and removal via Celery

(e) Access through NGINX with UUID routes