Docker - Intelligent cat dog classifier

Base Image Selection

- We chose `python:3.10-slim` as the base image.
- It provides Python 3.10, which fulfills the application's core requirement.
- The `"slim"` variant focuses on Python, reducing unnecessary packages and keeping the image size smaller.
- We specified `3.10` instead of `latest` to ensure consistent behavior across deployments, alignment with the application's tested Python version.
- Official Python Docker images are well-supported by the Python community and receive regular updates. This ensures access to security patches and bug fixes.

Security Considerations

- **Official Python Images**

  We leverage an official Python image (`python:3.10-slim`) from Docker Hub for our base image. Official images prioritize security and typically offer several benefits:

  - **Non-Root User:** By default, they don't run processes as the root user. This enhances security as compromised containers with non-root users have limited access to system resources.
  - **Limited Open Ports:** They have minimal open network ports by default, reducing potential exposure points for attackers.

- **Official Repositories:**

  Docker Hub offers official images maintained by their respective project teams. In our case, the Python development team maintains the `python:3.10-slim` image. These official repositories receive regular security updates to address vulnerabilities discovered in the base image components. This ensures a more secure foundation for your application.

- **Dockerfile Considerations:**

  While the chosen base image provides a good security baseline, we can further enhance security within our Dockerfile:

- `RUN useradd -m myuser` `USER myuser`:
  - These instructions create a dedicated user (`myuser`) for running the application within the container. This is a security best practice as it avoids running the application with root privileges, mitigating potential damage in case of a security breach.
- `EXPOSE 4000`:
  - This instruction exposes port 4000 within the container. It doesn't automatically open the port on the host machine. However, it serves as documentation for users running the container and helps understand which ports need to be mapped during container execution.

Efficiency of Instructions

- **Layer Optimization**
  - We combined the `COPY requirements.txt ./` and `RUN pip install --no-cache-dir -r requirements.txt` instructions into a single `RUN` instruction. This reduces the number of layers in the final image, improving build speed and image size.
- **Command Minimization**
  - We considered combining the `RUN useradd -m myuser` and `USER myuser` instructions. However, for safety reasons, we've opted to keep them separate.
  - Combining these instructions might create the user if it doesn't exist in the base image, leading to unexpected behavior.
- **Instruction Precision**
  - Our use of `COPY` for application code and `WORKDIR` for setting the working directory is appropriate and contributes to clear and concise instructions.
- **Benefits of Efficiency**
  - Reduced image size: Fewer layers lead to a smaller image, resulting in faster downloads and deployment times.
  - Faster builds: Leveraging Docker's caching mechanism based on layers optimizes build times, especially when rebuilding the image.

Docker Compose

- **Use of Docker Compose**
  - The `docker-compose.yml` file demonstrates the use of Docker Compose to manage a multi-container application with a Redis database and a Python application and for effective multi-container orchestration.
- **Configuration Details**
  - The configuration details for services, networks, and volumes ensure proper communication and data persistence between containers:
    - Services:
      - app
        - Builds the application image from the current directory (`.`).
        - Names the container `cnn-container`.
        - Runs the application as user `myuser`.
        - Starts the application with `uvicorn src.main:app ...`.
        - Maps container port 4000 to host port 4000.
        - Mounts the local `./src` directory to `/code/src` within the container.
        - Ensures the `redis` service starts before the `app` service.
        - Sets the `REDIS_HOST` environment variable to `redis` for the application to connect to the Redis service.
        - Connects the service to the `app-network`.
      - redis
        - Uses the official `redis:6.2-alpine` image.
        - Names the container `redis`.
        - Connects the service to the `app-network`.
        - Mounts a local volume named `redis-data` to the `/data` directory for persistent storage of Redis data.
    - **Networks:**

`app-network`:

- This line defines a bridge network named `app-network`.
- A bridge network essentially creates a virtual LAN (Local Area Network) within your Docker host. Containers connected to the same bridge network can communicate with each other directly using their container names or IP addresses assigned by Docker.
- Both the `app` and `redis` services are connected to the `app-network`. This allows the `app` service (Python application) to communicate with the `redis` service (Redis database) using the container name `redis` within its configuration (as defined by the `REDIS_HOST` environment variable).

- **Volumes:**

  `redis-data`:

  - This line defines a local volume named `redis-data`.
  - Volumes are used to persist data outside of the container itself. This ensures that data isn't lost when the container is stopped or recreated.
  - the `redis-data` volume is mounted on the `/data` directory within the `redis` container. This allows the Redis service to store its data persistently on the Docker host. The `driver: local` specification indicates that this is a local volume on your Docker host machine.
  - By using a volume, you avoid losing Redis data even if the `redis` container is restarted or upgraded.