Docker Design for IOT Application with NodeJS Django and PostgreSQL Frontend

1. Introduction

This document provides a comprehensive design for deploying a full-stack application using Docker. The application comprises a Node.js frontend (serving static HTML, JavaScript, and CSS files while proxying API requests), a Django backend (handling business logic and API endpoints), and a PostgreSQL database (storing application data). To ensure system reliability, health monitoring is implemented using cAdvisor, Prometheus, and Grafana to track CPU, RAM, disk, and network usage. The deployment leverages Docker Compose to orchestrate multiple containers, each based on distinct images, promoting scalability, maintainability, and efficient resource management. This document is structured to be copied into a Microsoft Word document for further formatting.

2. Architecture Overview

The application adopts a microservices architecture, with each component running in a separate Docker container:

- **Database**: PostgreSQL, responsible for persistent data storage.
- **Backend**: Django (Python), managing business logic and API interactions.
- **Frontend**: Node.js (Express.js), serving static files and proxying API requests to the backend.
- **Monitoring**: cAdvisor for metric collection, Prometheus for metric storage, and Grafana for visualization.

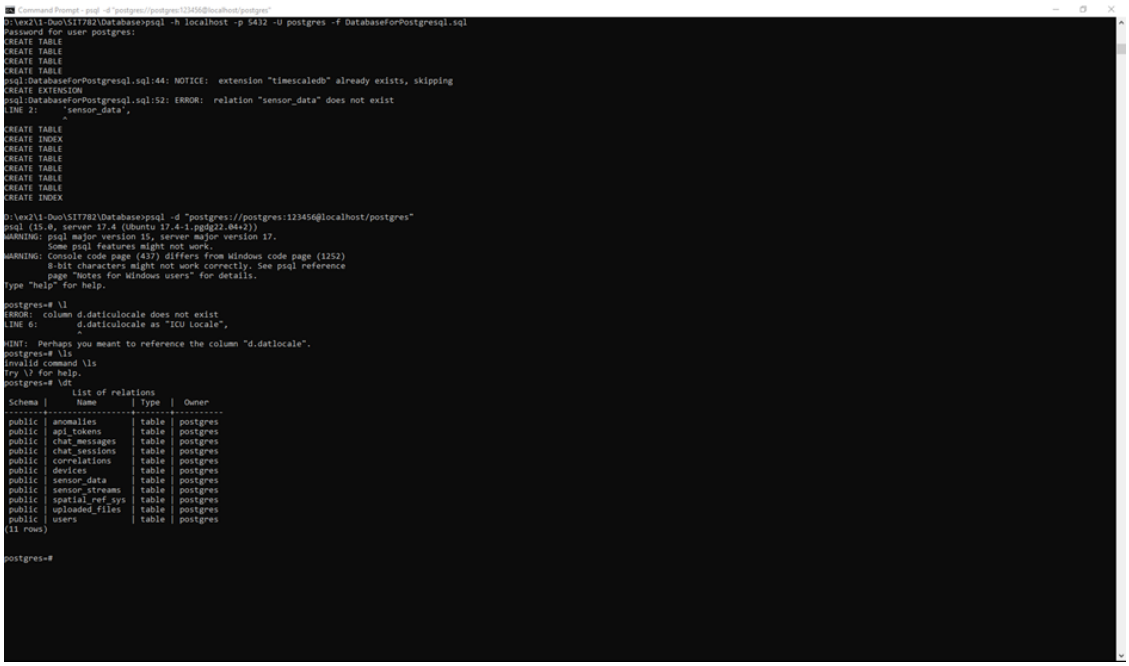
Docker Compose coordinates these services, defining their configurations, dependencies, networks, and volumes. This setup ensures seamless communication between components while maintaining isolation, making it suitable for development, testing, and production environments.

3. Component Design

The application is divided into four modules: Database, Backend, Frontend, and Monitoring. Each module is described below, detailing its technology, role, configuration, and interactions.

3.1 Database Module

- **Technology**: PostgreSQL 16
- **Role**: Stores persistent data for the application, such as user information and application state.
- **Create Database with Images**:
  - **Run the PostgreSQL Container**:
    - `docker run -d --name postgresql_postgis_timescale -e POSTGRES_USE=postgres -e POSTGRES_PASSWORD=123456 -e POSTGRES_DB=IOT -p 5432:5432 -v /data:/var/lib/postgresql/data timescale/timescaledb-ha:pg17`
  - **Verify the Container is Running**
    - `docker ps -a`

    - **Copy SQL file to Docker**
        - `docker cp DatabaseForPostgresql.sql postgresql_postgis_timescale:/DatabaseForPostgresql.sql`
    - **Create the database**
        - `psql -h localhost -p 5432 -U postgres -f DatabaseForPostgresql.sql`
    - **Connect to a database on your PostgreSQL instance**
        - `psql -d "postgres://postgres:123456@localhost/postgres"`
    - **Check Results**
        - Check all tables with `\dt`
        - 

- **Configuration**:
    - **Image**: `postgres:16` (PostgreSQL Docker Hub)
    - **Environment Variables**:
        - `POSTGRES_USER`: `postgres`
        - `POSTGRES_PASSWORD`: `123456` (replace with a strong password when deploying)
        - `POSTGRES_DB`: `postgres`
    - **Volumes**: `pgdata` for data persistence, mapped to `/var/lib/postgresql/data`
    - **Networks**: `app-network` (custom bridge network)
    - **Healthcheck**: Executes `pg_isready -U postgres` every 10 seconds, with a 5-second timeout and 5 retries to ensure database readiness
- **Interactions**: The Django backend connects to this database using the `DATABASE_URL` environment variable (`postgres://postgres:secret@db:5432/postgres`).

3.2 Backend Module

- **Technology**: Django, based on Python 3.9
- **Role**: Handles business logic, processes API requests, and interacts with the PostgreSQL database.
- **Configuration**:
    - **Build Context**: `./backend`
    - **Dockerfile**: Installs Django dependencies and runs the development server
    - **Environment Variables**:
        - `DATABASE_URL`: `postgres://postgres:secret@db:5432/postgres`
    - **Depends On**: `db` service, with a condition to wait until the database is healthy

    - **Networks**: `app-network`
    - **Exposed Ports**: 8000 (Django's default development server port)
    - **Healthcheck**: Checks the `/health` endpoint every 30 seconds, with a 10-second timeout and 3 retries (assumes a custom health endpoint; modify as needed)
- **Interactions**: Receives API requests proxied from the frontend, queries the database, and returns responses in JSON format.

3.3 Frontend Module

- **Technology**: Node.js 16 with Express.js
- **Role**: Serves static files (HTML, JavaScript, CSS) and proxies API requests to the Django backend.
- **Configuration**:
    - **Build Context**: `./frontend`
    - **Dockerfile**: Installs Node.js dependencies and runs the Express.js server
    - **Ports**: `80:3000` (maps host port 80 to container port 3000)
    - **Depends On**: `backend` service, with a condition to wait until the backend is healthy
    - **Networks**: `app-network`
    - **Exposed Ports**: 3000
    - **Healthcheck**: Checks the root URL (`http://localhost`) every 30 seconds, with a 10-second timeout and 3 retries
- **Interactions**:
    - Serves static files directly to clients, enabling the user interface.
    - Proxies API requests (e.g., `/api/*`) to the backend service (`http://backend:8000`).
    - **Note**: The frontend must include proxy middleware, such as `http-proxy-middleware` in Express.js, configured as follows:

```
const { createProxyMiddleware } = require('http-proxy-middleware');
app.use('/api', createProxyMiddleware({ target: 'http://backend:8000',
changeOrigin: true }));
```

3.4 Monitoring Module

- **Components**:
    - **cAdvisor**:
        - **Image**: `gcr.io/cadvisor/cadvisor:latest` (cAdvisor GitHub)
        - **Role**: Collects resource usage metrics (CPU, RAM, disk, network) for containers and the host system.
        - **Ports**: `8080:8080`
        - **Volumes**: Mounts host directories (`/`, `/var/run`, `/sys`, `/var/lib/docker`) to access system information
        - **Networks**: `app-network`
    - **Prometheus**:
        - **Image**: `prom/prometheus:latest` (Prometheus Docker Hub)
        - **Role**: Stores and queries time-series metrics collected by cAdvisor.
        - **Ports**: `9090:9090`
        - **Volumes**: Mounts `prometheus.yml` for configuration

        - **Networks**: `app-network`
    - **Grafana**:
        - **Image**: `grafana/grafana:latest` (Grafana Docker Hub)
        - **Role**: Visualizes metrics through customizable dashboards.
        - **Ports**: `3001:3000` (adjusted to avoid conflicts with frontend)
        - **Networks**: `app-network`
- **Configuration**:
    - Prometheus scrapes metrics from cAdvisor every 15 seconds.
    - Grafana uses Prometheus as a data source, offering dashboards for CPU, RAM, disk, and network usage.
- **Monitored Metrics**:
    - **CPU**: Usage percentage and core utilization (`container_cpu_usage_seconds_total`)
    - **RAM**: Memory usage and limits (`container_memory_usage_bytes`)
    - **Disk**: Filesystem usage and I/O (`container_fs_usage_bytes`)
    - **Network**: Bytes sent/received (`container_network_receive_bytes_total`)

4. Deployment Configuration

The deployment is managed by Docker Compose, which simplifies multi-container orchestration. Below are the key configuration files embedded as text for inclusion in the Word document.

4.1 Docker Compose Configuration

```
version: '3.8'

services:
  # Database service (PostgreSQL)
  db:
    image: postgres:16
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: 123456
      POSTGRES_DB: postgres
    volumes:
      - pgdata:/var/lib/postgresql/data
    networks:
      - app-network
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres"]
      interval: 10s
      timeout: 5s
      retries: 5

  # Backend service (Django)
  backend:
    build: ./backend
    depends_on:
      db:
        condition: service_healthy
    environment:
```

```
      DATABASE_URL: postgres://postgres:secret@db:5432/postgres
    networks:
      - app-network
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8000/health"]
      interval: 30s
      timeout: 10s
      retries: 3

  # Frontend service (Node.js)
  frontend:
    build: ./frontend
    ports:
      - "80:3000"
    depends_on:
      backend:
        condition: service_healthy
    networks:
      - app-network
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:3000"]
      interval: 30s
      timeout: 10s
      retries: 3

  # cAdvisor service (Container resource monitoring)
  cadvisor:
    image: gcr.io/cadvisor/cadvisor:latest
    container_name: cadvisor
    ports:
      - "8080:8080"
    volumes:
      - /:/rootfs:ro
      - /var/run:/var/run:ro
      - /sys:/sys:ro
      - /var/lib/docker/:/var/lib/docker:ro
    restart: always
    networks:
      - app-network

  # Prometheus service (Metrics storage)
  prometheus:
    image: prom/prometheus:latest
    container_name: prometheus
    ports:
      - "9090:9090"
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
    restart: always
    networks:
      - app-network

  # Grafana service (Metrics visualization)
  grafana:
```

```
    image: grafana/grafana:latest
    container_name: grafana
    ports:
      - "3001:3000"
    restart: always
    networks:
      - app-network

volumes:
  pgdata:

networks:
  app-network:
    driver: bridge
```

4.2 Backend Dockerfile

```
# Use the official Python 3.9 image as the base image
# This provides a stable and lightweight environment with Python 3.9 pre-installed
FROM python:3.9

# Set the working directory inside the container to /app
# All subsequent commands will be executed in this directory
WORKDIR /app

# Copy the requirements.txt file from the host to the working directory
# This file lists all Python dependencies required by the Django application
COPY requirements.txt .

# Install the Python dependencies listed in requirements.txt
# The pip install command ensures all necessary packages (e.g., Django, psycopg2)
are installed
RUN pip install -r requirements.txt

# Copy the entire Django project directory from the host to the working directory
# This includes all application code, configuration files, and static assets
COPY . .

# Expose port 8000 to allow external access to the Django development server
# This is the default port used by Django's runserver command
EXPOSE 8000

# Define the command to run when the container starts
# This starts the Django development server, binding to all interfaces (0.0.0.0)
on port 8000
CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]
```

4.3 Frontend Dockerfile

```
# Step 1: Use an official Node.js image as the base image
FROM node:16 AS build

# Step 2: Set the working directory inside the container
WORKDIR /app

# Step 3: Copy package.json and package-lock.json (or yarn.lock) into the
container
COPY package*.json ./

# Step 4: Install dependencies
RUN npm install

# Step 5: Copy the entire project into the container
COPY . .

# Step 6: Build the React app for production
RUN npm run build

# Step 7: Use a lighter web server to serve the built files
FROM nginx:alpine

# Step 8: Copy the build output to the Nginx public folder
COPY --from=build /app/build /usr/share/nginx/html

# Step 9: Expose the port that the container will use
EXPOSE 80

# Step 10: Start Nginx to serve the frontend app
CMD ["nginx", "-g", "daemon off;"]
```

4.4 Prometheus Configuration

Create image with the following command or yaml file

```
docker run -d --name cadvisor -p 8080:8080 -v /:/rootfs:ro -v /var/run:/var/run:ro -v
/sys:/sys:ro -v /var/lib/docker/:/var/lib/docker:ro --restart always
gcr.io/cadvisor/advisor:latest
```

```
global:
  scrape_interval: 15s
scrape_configs:
  - job_name: 'cadvisor'
    static_configs:
      - targets: ['cadvisor:8080']
```

5. Monitoring and Health Checks

- **Health Checks**:

    - **Database**: Verifies PostgreSQL readiness using `pg_isready`.
    - **Backend**: Checks Django's `/health` endpoint (assumes implementation; customize as needed).
    - **Frontend**: Ensures the Node.js root URL is accessible.
- **Monitoring**:
    - cAdvisor collects real-time metrics for CPU, RAM, disk, and network usage.
    - Prometheus stores these metrics for querying and analysis.
    - Grafana provides user-friendly dashboards to visualize resource usage, aiding in performance monitoring.

6. Scalability and Performance

- **Scalability**:
    - Services can be scaled independently using Docker Compose replicas or orchestration tools like Docker Swarm or Kubernetes.
    - Multiple backend instances can be load-balanced to handle increased traffic.
- **Performance**:
    - Monitoring tools help identify resource bottlenecks, enabling proactive optimization.
    - PostgreSQL performance can be enhanced through configuration tuning (e.g., adjusting `shared_buffers`).
    - Adding caching mechanisms, such as Redis, can further improve response times.

7. Security Considerations

- **Database**:
    - Use strong passwords and consider secrets management tools like Docker Secrets.
    - Restrict network access to allow only the backend to connect.
- **Backend**:
    - Implement API authentication and authorization (e.g., using Django REST Framework's authentication).
    - Sanitize inputs to prevent injection attacks.
- **Frontend**:
    - Enable HTTPS in production using tools like Traefik or a reverse proxy.
    - Secure proxy configurations to prevent misrouting.
- **Monitoring**:
    - Restrict access to cAdvisor, Prometheus, and Grafana with authentication mechanisms.
    - Change Grafana's default credentials (admin/admin) upon first login to enhance security.

8. Deployment Steps

1. **Prepare Project**:
    - Place the Django project in the `./backend` directory, ensuring `requirements.txt` lists all dependencies (e.g., `django`, `psycopg2-binary`).
    - Place the Node.js frontend in the `./frontend` directory, ensuring `package.json` includes a `start` script and dependencies like `express` and `http-proxy-middleware`.
    - Configure the frontend to proxy API requests to `http://backend:8000`.
2. **Create Configuration Files**:

- Create the Dockerfiles and `prometheus.yml` as shown above.
- Ensure the project directory structure matches:

```
.
├── docker-compose.yml
├── prometheus.yml
├── backend/
│   ├── Dockerfile
│   ├── requirements.txt
│   └── ... (Django project files)
├── frontend/
│   ├── Dockerfile
│   ├── package.json
│   └── ... (Node.js project files)
```

3. **Build and Start**:

```
docker-compose up -d --build
```

4. **Configure Monitoring**:

- Access Grafana at `http://localhost:3001`, log in with default credentials (admin/admin), and change the password.
- Add Prometheus as a data source using the URL `http://prometheus:9090`.
- Import a dashboard, such as ID 193 for Docker container monitoring, available at Grafana Dashboards.

5. **Access Application**:

- Frontend: `http://localhost`
- Grafana: `http://localhost:3001`
- Prometheus: `http://localhost:9090`
- cAdvisor: `http://localhost:8080`

6. **Stop Deployment**:

```
docker-compose down
```

9. Conclusion

This Docker design delivers a scalable and maintainable deployment solution for a full-stack application featuring a Django backend, Node.js frontend, and PostgreSQL database. The integrated monitoring tools provide comprehensive insights into system health, ensuring optimal performance. The setup is versatile, suitable for development, testing, and production environments, with flexibility for further enhancements such as caching, load balancing, or advanced security measures.