Support Documentation

Jirathip Kunkanjanathorn - 32999860 A - 12345678
B - 12345678 C - 12345678 D - 12345678

Contents

Introduction

Documentation Page

Data Workflows

Data Collection

Data Processing

Data Ingestion

Backend

How to run the system

1. Clone the repository or from the zip file
2. Install Docker with Docker Compose
3. Run `docker-compose up -d` in the root directory (Compose V2 do not have - between docker-compose)
4. SSH into the backend container with `docker exec -it backend bash`
5. Migrate the database with `alembic upgrade head` inside the container
6. Insert the data with `python3 scripts/insert_data.py` inside the container

How to shutdown the system

1. Run `docker-compose down` in the root directory

How to test the system

Unit test using pytest with Docker locally

1. Run `docker-compose -d` in the root directory
2. Run `docker exec -it backend pytest` to run the test

Noted that this test will remove all the data in the database where the test is run. The better way to test is to use CI/CD with Github Action.

Unit test the system with Github Action

This workflow is triggered whenever there's a push to the main branch.

1. Checks out repository.
2. Builds and starts Docker Compose services using the dev configuration.
3. Runs pytest within backend service.
4. Shuts down and removes the containers afterward.

Test the system with Postman

1. Ensure the system is up and running by following the provided steps under "How to run the system".
2. Launch the Postman application on your machine.
3. In the Postman interface, select the desired HTTP method (e.g., GET, POST, PUT, DELETE) from the dropdown.
4. Enter the system's endpoint URL you wish to test in the request URL field.
5. Depending on the endpoint:
   - Add necessary headers by clicking on the "Headers" tab and entering key-value pairs.
   - For methods like POST or PUT, click on the "Body" tab and input the required data in the appropriate format (e.g., JSON, form data).
   - If necessary, add query parameters by clicking on the "Params" tab and inputting key-value pairs.
6. Click the "Send" button to initiate the request to the backend.
7. Inspect the response received in the Postman response section below. Here, you can view status codes, response times, returned data, or any error messages.

How to deploy the system

Build and Push Images to Dockerhub with Github Action

Activated either manually or when there's a push to the deploy branch. The steps include:

1. Checking out repository.
2. Setting up QEMU & Docker Buildx.
3. Logging into Docker Hub using saved credentials.
4. Building Docker images from Dockerfiles (Dockerfile.db & Dockerfile.backend).
5. Pushing these images to Docker Hub.

Setting up GCP Compute Instance

Before deploying on GCP, ensure the VM instance ready. To set up a VM instance:

1. Go to the GCP Console at https://console.cloud.google.com/.
2. Navigate to the Compute Engine and then VM Instances.
3. Click on "Create Instance."
4. Fill out the necessary details like Name, Region, Zone, Machine type, etc.
5. In the Boot Disk section, select an Ubuntu as OS.

6. Under the Firewall settings, make sure to allow HTTP and HTTPS traffic if your application needs to be accessed over the internet.
7. Once filled out, click "Create" to instantiate your VM.
8. SSH into the instance and install Docker and Docker Compose. The instructions can be found here: https://docs.docker.com/engine/install/ubuntu/
9. Set up NGINX for reverse proxy in the instance.

   9.1 Install NGINX: `sudo apt install nginx`

   9.2 Create a new file in /etc/nginx/sites-available/ and name it settle-aid

   9.3 Copy the following configuration into the file:

   `to be filled in later`

   9.4 Create a symbolic link to the file in /etc/nginx/sites-enabled/:

   `sudo ln -s /etc/nginx/sites-available/settle-aid /etc/nginx/sites-enabled/`

   9.5 Test the configuration and restart NGINX:

   `sudo nginx -t`

   `sudo systemctl restart nginx`

Deploying on GCP

1. SSH into GCP Instance: `gcloud compute ssh <instance-name> --zone <zone>`
2. Change directory: `cd ..` (Optional)
3. Make sure docker-compose.yaml is exist in the directory. The configuration for production is as following

```yaml
version: '3'
services:
  db:
    container_name: settle-aid-db
    image: jirathipk/postgres-vec-geo:latest
    restart: always
    environment:
      - POSTGRES_DB=database
      - POSTGRES_USER=db_user
      - POSTGRES_PASSWORD=password1234
    volumes:
      - database_volume:/var/lib/postgresql/data/
      - dbbackups_volume:/backups
  redis:
    container_name: settle-aid-redis
```

```
    image: redis:latest
    restart: always
    command: redis-server --requirepass topmelloredis --loglevel verbose
    volumes:
      - redis_volume:/data

  backend:
    image: jirathipk/settle-aid-backend:latest
    container_name: settle-aid-backend
    user: myuser
    ports:
      - "8000:8000"
    environment:
      - DATABASE_HOSTNAME=db
      - DATABASE_NAME=database
      - DATABASE_PORT=5432
      - DATABASE_PASSWORD=password1234
      - DATABASE_USERNAME=db_user
      - SECRET_KEY=SECRET_KEY
      # Generate a new key with openssl rand -hex 32
      - REFRESH_SECRET_KEY=REFRESH_SECRET_KEY
      # Generate a new key with openssl rand -hex 32
      - REFRESH_TOKEN_EXPIRE_DAYS=7
      - ALGORITHM=HS256
      - ACCESS_TOKEN_EXPIRE_MINUTES=30
      - MAPBOX_ACCESS_TOKEN={{MAPBOX_ACCESS_TOKEN}}
      # Create an account and get the token from https://account.mapbox.com/
      - DOC_USERNAME=topmello
      - DOC_PASSWORD=da7da0df508738e37f18
      - REDIS_HOSTNAME=redis
      - REDIS_PORT=6379
      - REDIS_PASSWORD=topmelloredis
      - USER_CACHE_EXPIRY=3600
      - TRANSFORMERS_CACHE=/usr/src/app/transformers_cache
      - PYTEST_ADDOPTS="-o cache_dir=/usr/src/app/.pytest_cache"
    depends_on:
      - db
      - redis

  pgbackups:
    container_name: settle-aid-db-backup
    image: prodrigestivill/postgres-backup-local
    restart: always
    user: postgres:postgres
    volumes:
```

```
    - dbbackups_volume:/backups
  links:
    - db
  depends_on:
    - db
  environment:
    - POSTGRES_HOST=db
    - POSTGRES_DB=database
    - POSTGRES_USER=db_user
```

4. Pull the Latest Docker Compose Configuration: `sudo docker-compose pull`
5. Start the Containers: `sudo docker-compose -p settle-aid up -d`

- The -p flag is to set a project name, which can be useful for running multiple environments on the same host
- The -d flag is to run the containers in the background

Actions for Developers:

- Modifications: If there are modifications or additions to packages, update requirements.txt so the Docker build process incorporates these changes.
- GitHub Workflows: The Python application test runs on pushes to the main branch, and the Dockerhub build and push are triggered either manually or when pushing to the deploy branch.

How to add table to the database

1. While inside the backend container, navigate to the directory where your models are defined.
2. Create or modify an ORM model to define the structure of your new table.
3. Generate a new migration script using the command:

   ```
   alembic revision -m "Add new_table_name table"
   ```

4. Edit the generated migration script in the `versions` directory, ensuring the `upgrade()` method contains logic to create your new table and the `downgrade()` method contains logic to remove it.
5. Apply the migration using

   ```
   alembic upgrade head
   ```

6. Define ORM models in the `app/models.py` to interact with the new table accordingly.

How to backup and restore the database

Setup Automated Backups

We use the docker-postgres-backup-local image to facilitate our backup tasks. Once the service is started, this will automatically create backups of database daily and maintain.

https://github.com/prodrigestivill/docker-postgres-backup-local

Restore from fresh database

To restore a backup to a fresh database:

```
docker exec -it db psql \
--username=db_user \
--dbname=postgres -c "DROP DATABASE database;"

docker exec -it db psql \
--username=db_user \
--dbname=postgres -c "CREATE DATABASE database;"

docker exec -it db psql \
--username=db_user \
--dbname=database -c "CREATE EXTENSION IF NOT EXISTS postgis;"

docker exec -it db psql \
--username=db_user \
--dbname=database -c "CREATE EXTENSION IF NOT EXISTS vector;"

sudo docker exec \
-it db /bin/sh \
-c "zcat /backups/last/database-latest.sql.gz | \
psql --username=db_user --dbname=database -W"
```

Change permission for backups If necessary, adjust the permissions for the backup files.

```
docker exec -u root -it db-backup chown -R 999:999 /backups
```

Check backup volumes content To inspect the contents of the backup volume:

```
sudo docker run \
--rm -it \
-v settle-aid_dbbackups_volume:/volume_content alpine:latest /bin/sh
```

How to launch new service in the backend

1. Pull the Desired Image: If you haven't already, ensure that the desired service's Docker image is available on your system. If it's on a public registry like Docker Hub, you can pull it using:

   ```
   docker pull <image-name>:<tag>
   ```

   Replace `<image-name>:<tag>` with the name and the desired tag/version of the image.

2. Modify the Docker Compose File: Navigate to the directory containing your `docker-compose.yml` file and open it for editing.

3. Add the New Service: In the `docker-compose.yml` file, add a new service definition for the image you've just pulled. For instance:

   ```
   services:
     ...
     new-service-name:
       image: <image-name>:<tag>
       ports:
         - "<external-port>:<internal-port>"
       environment:
         - "ENV_VAR_NAME=value"
       volumes:
         - "/path/on/host:/path/in/container"
       depends_on:
         - "another-service-name"
   ```

   Replace placeholders as appropriate:

   - `<image-name>:<tag>` with the image's name and tag.
   - `<external-port>:<internal-port>` to map ports from the container to the host.
   - Adjust `environment` to set any environment variables the service needs.
   - The `volumes` section can be used to mount directories from the host into the container.
   - `depends_on` ensures that the new service starts only after another specified service has started.

4. Launch the New Service: With the service added to the Docker Compose file, navigate to the directory containing the file and run:

   ```
   docker-compose up -d new-service-name
   ```

   This command will start only the new service and any services it depends on. If you want to start all services defined in the Compose file, simply use:

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

5. Monitor the Service: You can check the logs of the newly launched service with:

   ```
   docker-compose logs -f new-service-name
   ```

By following these steps, you'll successfully launch a new service in the backend using Docker Compose. Remember to consult the documentation or README of the specific service image for any particular configurations or environment settings.

How to monitor the system

1. Logging Pages: The developed logging pages are essential tools for observing the system's behavior. Regularly review the `/logs/` endpoint, as it maintains a record of all requests made to the backend. By analysing these logs, system performance, identify patterns, detect anomalies, and troubleshoot issues can be guaged when they arise.
2. Docker Logs: Docker provides built-in logging mechanisms for its containers. You can access the logs of a specific container using the following command:

   ```
   docker-compose logs -f service-name
   ```

3. System Metrics: Utilise `docker stats` to monitor the resource usage of your containers. This command provides a live stream of container performance metrics such as CPU usage, memory consumption, network IO, and disk IO. By monitoring these metrics, you can gain insights into the resource demands of each service and make informed decisions on system scaling or optimization.

What to do when the system is down

1. Verify the Outage: Ensure it's not just a local or isolated issue.
2. Check Logs: Review application logs for errors or warnings.
3. Check External Services: Make sure dependencies, like databases or third-party APIs, are operational.
4. Restart Services: A simple service restart might solve temporary glitches.

   ```
   sudo docker-compose up -d -p settle-aid
   ```

5. Review Resources: Ensure the system hasn't run out of essential resources like CPU, RAM, or storage.

Frontend

How to run the system

How to shutdown the system

How to connect to the backend

How to deploy the system

Training and knowledge needed to operate the system

Database Management and SQL

- **PostgreSQL:** Familiarity with relational database management, including CRUD operations, indexing, and optimization in PostgreSQL.
- **Alembic:** Understanding of database migrations and the ability to apply, revert, or create new migrations using Alembic.
- **Redis:** A grasp on key-value store principles, Redis data structures, and caching strategies.

Python Development

- **SQLAlchemy:** Experience in working with Object Relational Mapping (ORM) to integrate Python applications with databases.
- **Pydantic:** Knowledge of data validation and parsing using Pydantic, ensuring incoming data complies with expected formats.
- **PyTest:** Proficiency in writing and running unit tests using PyTest to ensure the functionality and robustness of the codebase.
- **PyTorch:** Basics of machine learning and deep learning concepts, and how to utilize PyTorch for training and deploying models.

API and Web Frameworks

- **FastAPI:** Mastery of creating, deploying, and managing APIs using FastAPI, coupled with an understanding of asynchronous programming.
- **SlowAPI:** Knowledge about rate limiting and its implementation using SlowAPI to manage request traffic and protect the system.
- **Python SocketIO Server:** Familiarity with WebSockets and real-time bi-directional communication using SocketIO.

External Services and Data integrations

- **Open Data Melbourne:** Awareness of the datasets available and how to integrate and utilize city-specific data for enhanced platform functionality.
- **MapBox, Google Translate & Google Location APIs:** Understanding of external API integrations, request-response patterns, and error

handling. Experience in geolocation services, language translation tools, and map rendering is essential.

Natural Language Processing and AI

- **Huggingface's Sentence Transformers:** Grasp on semantic search, sentence embeddings, and the utilization of transformers for better user input understanding.

Containerization and Deployment

- **Docker:** Proficiency in containerizing applications using Docker, understanding of Docker Compose for multi-container applications, and familiarity with container orchestration.
- **GCP (Google Cloud Platform):** Knowledge of deploying and managing Docker images on GCP, understanding GCP's infrastructure, and its security best practices.

General

- **System architecture:** A holistic understanding of how each component of the tech stack interacts with each other, the data flow, and error handling mechanisms.
- **Continuous Integration/Continuous Deployment:** Familiarity with CI/CD processes, especially with tools like GitHub Actions, to automate testing and deployment workflows.

Change management

Resources

- GitHub Repository:
  - Frontend: to be filled
  - Backend: to be filled
  - Documentation page: to be filled
  - Data Wrangling: to be filled
- Project Documentation Page
- Backend API Documentation
- Backend Logging
- Backend UI for testing