**Query Management Backend Project Documentation**

**Overview**

The Query Management Backend project is a Django-based application designed to handle employee queries. The system supports different types of queries such as payslip, timesheet, and roster queries. It offers CRUD operations for managing these queries and is designed to be robust, scalable, and secure. The project integrates with a MySQL database and includes Docker support for containerized deployment.

**Table of Contents**

1. [Installation](#installation)
2. [Project Structure](#project-structure)
3. [Configuration](#configuration)
4. [Database](#database)
5. [API Endpoints](#api-endpoints)
6. [Exception Handling](#exception-handling)
7. [Middleware](#middleware)
8. [Deployment](#deployment)

**Installation**

**Prerequisites**

* Python 3.9
* Docker
* MySQL

**Steps**

1. **Clone the Repository**

git clone https://github.com/mazinakundai/query\_management.git

cd query\_management

1. **Set Up Virtual Environment**

python3 -m venv env

source env/bin/activate

1. **Install Dependencies**

pip install -r requirements.txt

1. **Set Up MySQL Database**
   * Create a database named query\_management.
   * Create a user queryMgtUser with password queryMgtPass01# and grant all privileges on the query\_management database.
2. **Apply Migrations**

python manage.py migrate

1. **Run the Development Server**

python manage.py runserver

**Docker Deployment**

1. **Build the Docker Image**

docker build -t query\_management .

1. **Run the Docker Container**

docker run -d -p 8000:8000 --name query\_management\_container query\_management

**Project Structure**

* **queries**: This directory contains the main application logic for handling queries.
  + models.py: Defines the data models for the application.
  + serializers.py: Handles serialization and validation of model data.
  + views.py: Contains the API views for handling requests.
  + urls.py: Defines the URL patterns for the application.
  + utils.py: Contains utility functions, such as custom exception handling.
  + middleware.py: Custom middleware for logging and error handling.
* **query\_management**: This directory contains the project-level settings and configurations.
  + settings.py: Contains the configuration settings for the Django project.
  + urls.py: Defines the URL patterns for the project.
  + wsgi.py: WSGI configuration for deployment.

**Configuration**

**Django Settings**

The settings.py file includes various configuration options such as installed apps, middleware, database configurations, and more.

* **Installed Apps**: Includes Django's default apps along with queries, rest\_framework, and corsheaders.
* **Middleware**: Includes security, session, authentication, and custom middleware.
* **REST Framework**: Configured with a custom exception handler.
* **CORS Headers**: Configured to allow requests from http://localhost:3000.

**Environment Variables**

For Docker deployment, database credentials and other sensitive information are managed through environment variables.

**Database**

**Local Database Configuration**

For local development, the project uses a MySQL database with the following configuration:

* **ENGINE**: django.db.backends.mysql
* **NAME**: query\_management
* **USER**: queryMgtUser
* **PASSWORD**: queryMgtPass01#
* **HOST**: localhost
* **PORT**: 3306

**Docker Database Configuration**

For containerized deployment, the database configuration uses environment variables:

* **NAME**: Retrieved from DATABASE\_NAME
* **USER**: Retrieved from DATABASE\_USER
* **PASSWORD**: Retrieved from DATABASE\_PASSWORD
* **HOST**: Retrieved from DATABASE\_HOST
* **PORT**: Retrieved from DATABASE\_PORT

**API Endpoints**

The project provides RESTful API endpoints for managing queries. These endpoints are defined in the urls.py file of the queries app.

**Endpoints**

* **GET /queries/**: Retrieve a list of all queries.
* **POST /queries/**: Create a new query.
* **GET /queries/{id}/**: Retrieve details of a specific query.
* **PUT /queries/{id}/**: Update a specific query.
* **PATCH /queries/{id}/**: Partially update a specific query.
* **DELETE /queries/{id}/**: Delete a specific query.

**Exception Handling**

The project includes custom exception handling to ensure consistent and informative error responses.

**Custom Exception Handler**

The custom exception handler is defined in queries/utils.py. It modifies the default DRF exception handler to include the status code in the response data.

**Middleware**

Custom middleware is used for logging errors and returning a generic error message for unhandled exceptions. This is defined in queries/middleware.py.

**Middleware**

The project uses several middleware components to handle various aspects of request and response processing.

**Custom Exception Middleware**

The custom exception middleware logs errors and returns a generic error message to the client. This helps in maintaining a consistent response format and keeping sensitive error details hidden from the client.

**Deployment**

**Local Deployment**

For local development, run the development server using the Django management command:

python manage.py runserver

**Docker Deployment**

For deploying the application using Docker, follow these steps:

1. **Make sure you are in the project root.**

Where you see both query\_management and querymanagement\_frontend folders.

1. **Build the Docker Image**

Docker-compose build

1. **Run the Docker Container**

Docker-compose up

**Environment Variables**

Ensure that all necessary environment variables are set, especially for database configuration when running in a Docker container.