**Intergalactic Trade Network Backend System**

**Overview**

The Intergalactic Trade Network is a backend system designed to handle trade transactions, manage cargo shipments, and track inventory levels across space stations and planets. The system supports real-time updates and high-throughput data processing to ensure seamless operations across the galaxy.

**Table of Contents**

1. [System Architecture](#system-architecture)
2. [Database Schema](#database-schema)
3. [API Design](#api-design)
4. [Event Processing Pipeline](#event-processing-pipeline)
5. [Real-Time Analytics Dashboard](#real-time-analytics-dashboard)
6. [Code Structure and Best Practices](#code-structure-and-best-practices)
7. [Deployment Guide](#deployment-guide)
8. [Testing and Validation](#testing-and-validation)
9. [Known Limitations and Future Improvements](#known-limitations-and-future-improvemen)

**System Architecture**

**High-Level Architecture**

The system is built using a microservices architecture, with each service handling specific functionalities such as trade management, cargo tracking, and inventory monitoring. Flask is used as the backend framework, and SQLAlchemy is utilized for database interactions.

* **API Gateway**: Routes incoming requests to appropriate services and handles authentication.
* **Trade Service**: Manages trade transactions between space stations and planets.
* **Cargo Service**: Handles cargo shipment creation, tracking, and updates.
* **Inventory Service**: Monitors and updates inventory levels at space stations.
* **Event Processor**: Processes real-time events such as trade transactions and cargo status updates.
* **Analytics Dashboard**: Provides real-time visualization of trade volume, active shipments, and inventory levels.

**Component Interaction and Data Flow**

* Requests from clients are routed through the API Gateway to the respective services.
* The Trade Service interacts with the Cargo Service to manage shipments.
* Inventory updates are propagated in real-time using the Event Processor.
* Data is stored in a centralized relational database, and analytics data is pushed to the dashboard in real-time.

**Database Models**

The system uses SQLAlchemy ORM to define database models, which represent tables in the relational database.

**TradeTransaction**

class TradeTransaction(db.Model):

id = db.Column(db.Integer, primary\_key=True)

trade\_type = db.Column(db.String(50))

amount = db.Column(db.Float)

timestamp = db.Column(db.DateTime, default=db.func.now())

**CargoShipment**

class CargoShipment(db.Model):

id = db.Column(db.Integer, primary\_key=True)

cargo\_type = db.Column(db.String(50))

quantity = db.Column(db.Integer)

status = db.Column(db.String(50))

destination = db.Column(db.String(100))

**Inventory**

class Inventory(db.Model):

id = db.Column(db.Integer, primary\_key=True)

station\_id = db.Column(db.String(50))

item = db.Column(db.String(50))

quantity = db.Column(db.Integer)

**Database Configuration**

Database connection settings are defined in config.py. SQLAlchemy handles the connection, and Flask-Migrate manages schema migrations.

**API Design**

**API Endpoints**

The API is designed following RESTful principles. Below are some key endpoints:

**Create a Trade Transaction**

* **Endpoint**: POST /api/trades
* **Description**: Initiates a new trade transaction between space stations or planets.
* **Request Body**:

{

"trade\_type": "buy",

"amount": 1500

}

* **Response**:

{

"id": 1,

"trade\_type": "buy",

"amount": 1500,

"timestamp": "2024-09-02T10:00:00Z"

}

**Retrieve Cargo Shipment Details**

* **Endpoint**: GET /api/cargo/{shipmentId}
* **Description**: Retrieves details of a specific cargo shipment.
* **Response**:

json

Copy code

{

"id": 1,

"cargo\_type": "minerals",

"quantity": 100,

"status": "in\_transit",

"destination": "Station X"

}

**Retrieve Inventory Levels**

* **Endpoint**: GET /api/inventory/{stationId}
* **Description**: Retrieves the inventory levels at a specific space station.
* **Response**:

{

"station\_id": "Station X",

"items": [

{"item": "fuel", "quantity": 500},

{"item": "food", "quantity": 200}

]

}

**Error Handling**

All API responses follow a standard format, and error handling is implemented to ensure consistent and meaningful feedback to the clients.

* **Example Error Response**:

{

"error": "Invalid input data",

"message": "The trade amount must be a positive number."

}

**Event Processing Pipeline**

**Real-Time Event Processing**

The system processes real-time events such as trade transactions and cargo status updates through an event processing pipeline. This pipeline ensures that inventory levels and cargo statuses are updated in real-time, and notifications are sent for critical events.

* **Event Ingestion**: Events are ingested from various sources (e.g., trade transactions, cargo updates).
* **Event Processing**: The system processes these events in real-time, updating the database and triggering notifications if necessary.

**Real-Time Analytics Dashboard**

**Dashboard Features**

The dashboard provides a real-time overview of the intergalactic trade network's activities. It displays metrics such as trade volume, active cargo shipments, and inventory levels.

* **Trade Volume**: A line chart showing the volume of trades over time.
* **Active Shipments**: A bar chart displaying the number of active cargo shipments per destination.
* **Inventory Levels**: A chart showing the inventory levels at each space station.

**Real-Time Data Updates**

The dashboard is updated in real-time using WebSocket or polling mechanisms to reflect the latest data without requiring manual refreshes.

**Code Structure and Best Practices**

**Code Organization**

The project follows a modular structure, with separate directories for models, routes, and static assets.

your\_project/

│

├── app/

│ ├── models.py

│ ├── routes.py

│ ├── static/

│ ├── templates/

│ ├── \_\_init\_\_.py

│ └── ...

├── config.py

├── run.py

└── ...

**Best Practices**

* **Modularity**: The code is organized into modules to promote reusability and maintainability.
* **Error Handling**: Robust error handling is implemented to manage exceptions gracefully.
* **Logging**: The system logs important events and errors to help with debugging and monitoring.
* **Security**: Best practices for security are followed, including input validation, authentication, and data encryption.

**Future Improvements**

* **Caching**: Implement caching mechanisms to reduce database load and improve performance.
* **Advanced Analytics**: Add more complex analytics and visualizations to the dashboard.
* **User Authentication**: Enhance security with user authentication and role-based access control.

Note: I was not able to deploy the model as the frontend needed to be setup. I apologise for not able to complete the assignment. I have added most of the backend using python and flask.