Product Management System with Asynchronous Image Processing

Project Overview

I developed a backend system for managing products in a product management application using Golang. The project focuses on high scalability, asynchronous image processing, caching, and logging to enhance performance. The main goal of this system is to provide an API for managing products while ensuring that the system can handle high traffic and perform tasks like image processing efficiently in the background.

Key Features:

1. **Product Management API**:
    - **POST /products**: This endpoint accepts product details like the user ID, product name, description, images, and price to create a new product.
    - **GET /products/:id**: Retrieves details of a specific product by its ID, including processed images.
    - **GET /products**: Lists all products for a specific user, with filtering options for price and product name.
2. **Data Storage**:
    - I used **PostgreSQL** to store user and product data.
    - I added a new field, compressed_product_images, in the products table to store the processed (compressed) images.
3. **Asynchronous Image Processing**:
    - When a product is created, I place the image URLs into a message queue (I used **RabbitMQ**), and then a separate image processing microservice picks them up.
    - The microservice downloads, compresses the images, and stores them in S3. Once processing is complete, it updates the product's compressed_product_images field in the database.
4. **Caching**:
    - To reduce database load, I implemented **Redis** caching for the GET /products/:id endpoint.
    - Cache invalidation is also in place to ensure that when product data is updated, the cache is refreshed with the new information.
5. **Logging**:
    - I used **Logrus** (or **Zap**) for structured logging across the entire system. This helps in tracking API requests, response times, errors, and any events related to image processing.
6. **Error Handling**:

- The system handles errors robustly. If image processing fails, it retries the task using a retry mechanism. If retries are unsuccessful, the task is placed in a **dead-letter queue** for further investigation.

7. **Testing**:
    - I wrote **unit tests** for all major components, ensuring that the API endpoints and core functions are working as expected.
    - I also wrote **integration tests** to check the end-to-end functionality, especially for asynchronous tasks and caching.
    - To test performance, I used **benchmark tests** to compare the response time of the GET /products/:idendpoint with and without cache hits.

Architecture Overview

The system is designed with modularity, scalability, and fault tolerance in mind. Here's a breakdown of the key architectural components:

1. API Layer

- **Framework**: Golang (standard library for HTTP handling).
- **Endpoints**:
  - **POST /products**: Creates a new product, including details like name, description, price, and images.
  - **GET /products/:id**: Retrieves the details of a product by its ID, including processed images.
  - **GET /products**: Lists all products for a user, with optional filtering by price range and product name.

The API layer handles request validation, routing, and responses. It ensures that product data is stored and fetched efficiently.

2. Database (PostgreSQL)

- **Schema**: The system uses PostgreSQL to store product and user data. The `products` table includes:
  - `user_id`: Foreign key referring to the user who owns the product.
  - `product_name`, `product_description`, `product_price`: Basic product details.
  - `product_images`: Stores the URLs of product images.
  - `compressed_product_images`: Stores URLs of compressed images after processing.

Reasoning: PostgreSQL is chosen for its reliability, data integrity, and ability to handle complex queries.

3. Image Processing (Asynchronous)

- **Queue System**: RabbitMQ or Kafka is used as a message queue for processing images asynchronously.
  - When a product is created, the image URLs are added to the queue.
  - A separate microservice consumes these messages, downloads the images, processes them (e.g., compresses them), and stores the results in an external storage (e.g., AWS S3).
  - After processing, the system updates the `compressed_product_images` field in the database.

Reasoning: This separation ensures that the main API remains responsive by offloading heavy image processing tasks to a dedicated microservice.

4. Caching Layer (Redis)

- **Use of Redis**: Redis is used to cache the product details fetched via `GET /products/:id` to minimize database queries for frequently accessed data.
  - The cached product data is stored with an expiration time to ensure that outdated information is removed automatically.

    - Cache invalidation is performed when product data is updated, ensuring consistency across the database and cache.

Reasoning: Caching reduces the load on the database and improves the speed of read-heavy operations, especially for product details.

5. Logging (Logrus / Zap)

- **Structured Logging**: I used **Logrus** (or **Zap**) for structured logging.
    - All API requests, responses, and errors are logged with necessary details, including timestamps, status codes, and response times.
    - In the image processing microservice, events like image download success, processing errors, and image compression failures are logged.

Reasoning: Structured logging provides better traceability and helps monitor system performance, especially when handling large-scale data and requests.

6. Error Handling

- **Robust Error Handling**: The system is built to handle errors gracefully:
    - For image processing, retries are attempted if the process fails. If the retries exceed the allowed limit, the task is placed in a **dead-letter queue** for manual intervention.
    - All other operations, including database interactions and API requests, include error handling to ensure that failures are logged and appropriately responded to.

Reasoning: Ensuring reliable error handling prevents data loss and ensures the system remains operational even in the face of failures.

7. Scalability

- **API Layer**: The system is designed to scale horizontally, meaning that as the number of requests increases, more API instances can be deployed to balance the load.
- **Image Processing Service**: The image processing service can be scaled vertically by adding more workers to handle higher loads of image processing tasks.
- **Redis Caching**: Redis is distributed, allowing multiple Redis instances to be deployed to handle high read and write demands efficiently.

Setup Instructions

Follow these steps to set up and run the Product Management System locally:

1. Clone the Repository

git clone https://github.com/yourusername/product-management-system.git
cd product-management-system

2. Install Dependencies

Ensure you have Go installed (version 1.18+). You can install dependencies by running:

go mod tidy

3. Configure Environment Variables

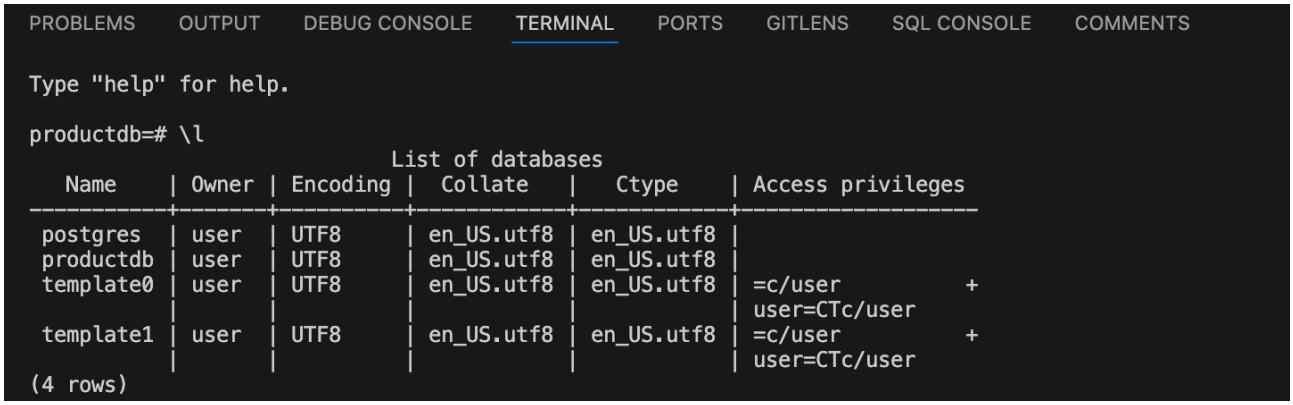
Create a .env file at the root of the project with the following settings:

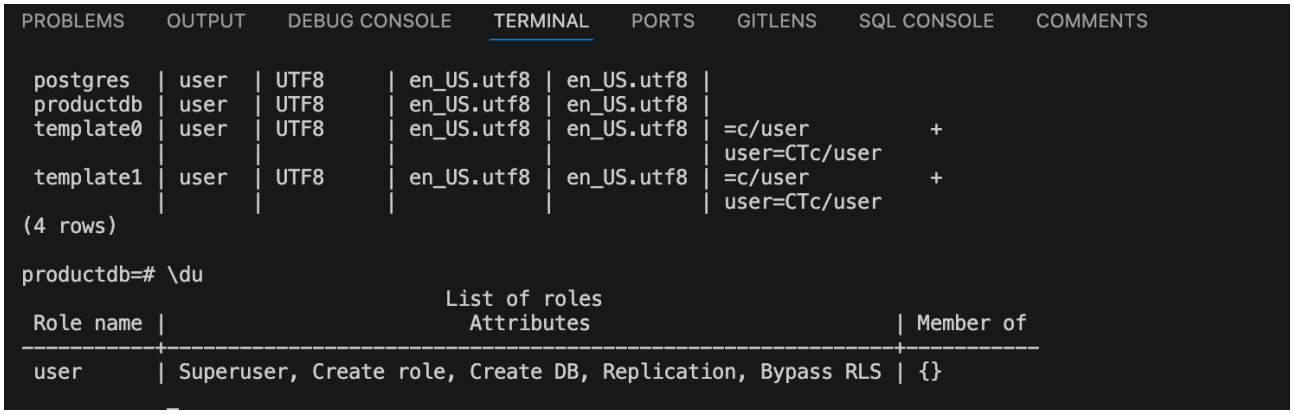
DB_HOST=localhost
DB_PORT=5432
DB_USER=your_db_user
DB_PASSWORD=your_db_password
DB_NAME=product_management
QUEUE_HOST=localhost
CACHE_HOST=localhost
AWS_S3_BUCKET=your_s3_bucket_name
AWS_ACCESS_KEY_ID=your_access_key
AWS_SECRET_ACCESS_KEY=your_secret_key

4. Run the Database Migrations

Run the migrations to set up your PostgreSQL database:

go run migrate.go





5. Start the Application

You can now run the application:

go run main.go
The API server will start and be accessible at http://localhost:8080.

6. Docker Setup (Optional)

If you want to run the application with Docker and include all required services (PostgreSQL, Redis, RabbitMQ), you can use Docker Compose:

docker-compose up
This will start all the necessary services.

Testing

Unit Tests

You can run unit tests for the application by running:

go test ./...

Integration Tests

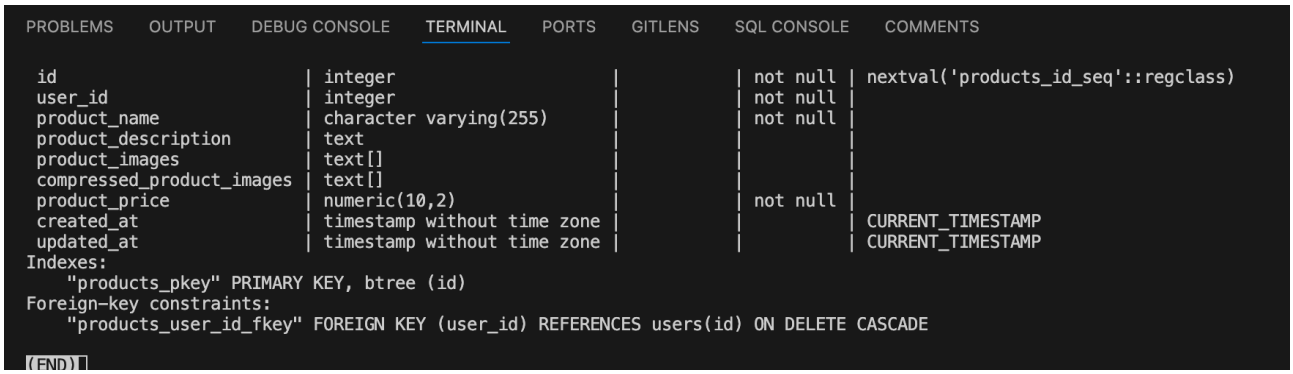
To run integration tests for full system functionality:

go test -tags=integration ./...

Benchmark Tests

To test the performance of the GET /products/:id endpoint, run:

go test -bench ./…



Logging

Structured logging is implemented using **Logrus** or **Zap**. Logs are generated for every API request, image processing event, and error, which can be viewed in the console or sent to a log management system.

Error Handling

If image processing fails, the system will retry the task a specified number of times. If it still fails, it will move the task to a **dead-letter queue** for further investigation. This ensures that no image processing tasks are lost.

Architectural Decisions & Assumptions

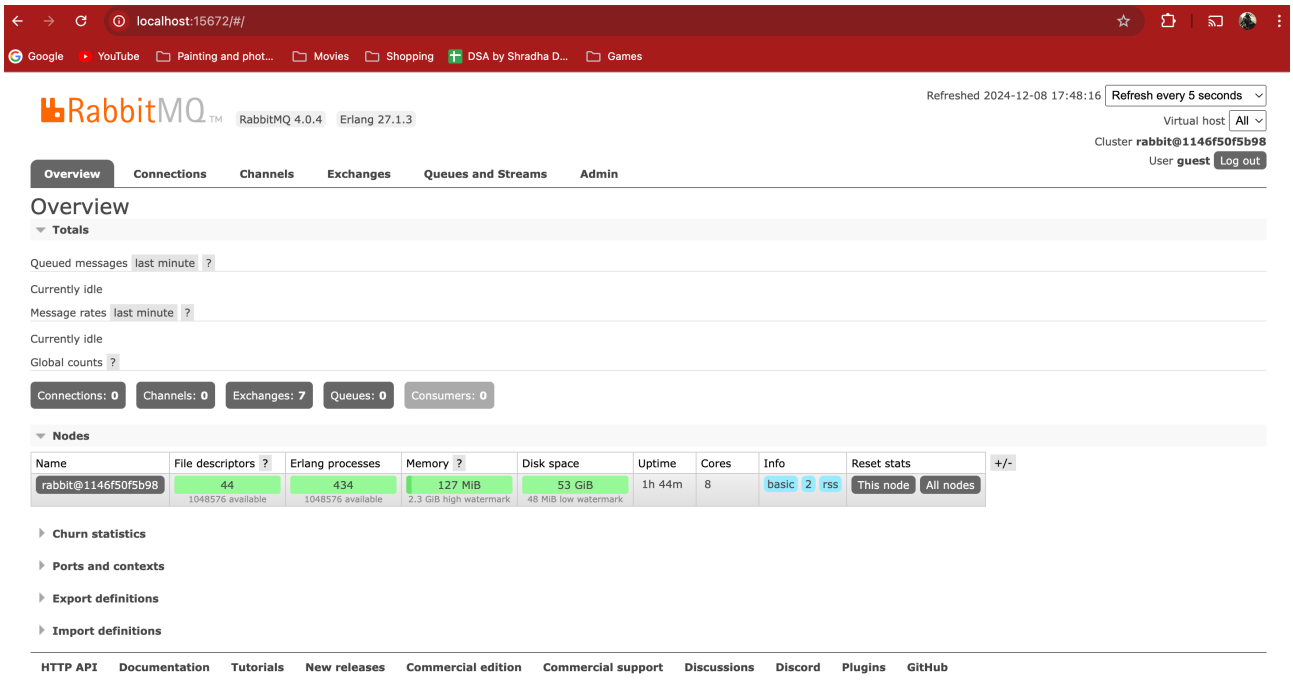
- **Message Queue**: I chose to use a message queue (RabbitMQ or Kafka) to decouple the image processing from the main API. This ensures that image processing does not block the API responses.
- **Redis**: Caching product details in Redis reduces the load on the database and speeds up the response time for frequently accessed products.
- **Scalability**: The system is designed to be scalable both horizontally (API layer) and vertically (image processing service).
- **Image Processing Microservice**: I created a separate microservice for image processing to ensure the main API stays responsive.

Conclusion

This backend system is designed to be highly efficient, scalable, and fault-tolerant. By separating concerns, using asynchronous processing for image tasks, and incorporating caching, I've built a

system that performs well even under high loads. The code is modular, easy to scale, and well-logged for monitoring and debugging.

OUTPUTS:



Listening ports

Protocol	Bound to	Port
amqp	::	5672
clustering	::	25672
http	::	15672
http/prometheus	::	15692

Web contexts

Context	Bound to	Port	SSL	Path
RabbitMQ Management	0.0.0.0	15672	○	/
RabbitMQ Prometheus	0.0.0.0	15692	○	/