**1. System Architecture**

**Components Overview**

1. **API Server**: Handles requests, validates inputs, and communicates with services like the database and cache.
2. **Database**: PostgreSQL for persistent storage of products and users.
3. **Message Queue**: RabbitMQ/Kafka for asynchronous communication, handling image processing.
4. **Image Processing Service**: Consumes messages, processes images, and updates the database.
5. **Caching Layer**: Redis for caching frequently accessed data.
6. **Logging Service**: Centralized logging using logrus/zap.

**Architectural Diagram**

* REST API interacts with PostgreSQL and Redis.
* API triggers RabbitMQ/Kafka for asynchronous tasks.
* Image Processing Service consumes messages from RabbitMQ/Kafka.
* Compressed images are stored in S3, and data is updated in PostgreSQL.

product-management/ ├── main.go ├── handlers/ ├── services/ ├── models/ ├── utils/ └── configs/

## 2. ****Implementation Steps****

### ****Step 1: Setup Environment****

#### Install Dependencies

bash

Copy code

# Install Go dependencies

go mod init product-management

go get github.com/gorilla/mux

go get github.com/go-redis/redis/v8

go get github.com/streadway/amqp # For RabbitMQ

go get github.com/jackc/pgx/v4 # PostgreSQL driver

go get github.com/sirupsen/logrus # Logging

#### Setup Environment Variables

Create a .env file:

env

Copy code

DB\_HOST=localhost

DB\_USER=your\_user

DB\_PASSWORD=your\_password

DB\_NAME=product\_management

REDIS\_HOST=localhost:6379

RABBITMQ\_URL=amqp://guest:guest@localhost:5672/

S3\_BUCKET\_NAME=your-s3-bucket

AWS\_ACCESS\_KEY\_ID=your-access-key

AWS\_SECRET\_ACCESS\_KEY=your-secret-key

## 2. ****Database Setup****

### Step 2.1: Create PostgreSQL Schema

Run the following commands in the PostgreSQL shell or a tool like pgAdmin:

sql

Copy code

CREATE TABLE users (

id SERIAL PRIMARY KEY,

name VARCHAR(100) NOT NULL

);

CREATE TABLE products (

id SERIAL PRIMARY KEY,

user\_id INT REFERENCES users(id),

product\_name VARCHAR(255) NOT NULL,

product\_description TEXT NOT NULL,

product\_images TEXT[], -- Array of URLs

product\_price DECIMAL(10, 2) NOT NULL,

compressed\_product\_images TEXT[], -- Array of compressed image URLs

created\_at TIMESTAMP DEFAULT NOW(),

updated\_at TIMESTAMP DEFAULT NOW()

);

## 3. ****Coding****

### Step 3.1: Main Entry File (main.go)

go

Copy code

package main

import (

"log"

"net/http"

"os"

"github.com/gorilla/mux"

"product-management/handlers"

)

func main() {

r := mux.NewRouter()

// Define Routes

r.HandleFunc("/products", handlers.CreateProduct).Methods("POST")

r.HandleFunc("/products/{id}", handlers.GetProduct).Methods("GET")

r.HandleFunc("/products", handlers.GetProducts).Methods("GET")

// Start Server

port := os.Getenv("PORT")

if port == "" {

port = "8080"

}

log.Printf("Server running on port %s", port)

log.Fatal(http.ListenAndServe(":"+port, r))

}

### Step 3.2: Handlers (handlers/product.go)

go

Copy code

package handlers

import (

"encoding/json"

"fmt"

"net/http"

"strconv"

"github.com/gorilla/mux"

)

type Product struct {

ID int `json:"id"`

UserID int `json:"user\_id"`

ProductName string `json:"product\_name"`

ProductDescription string `json:"product\_description"`

ProductImages []string `json:"product\_images"`

ProductPrice float64 `json:"product\_price"`

CompressedImages []string `json:"compressed\_product\_images"`

}

// Dummy database

var products = []Product{}

// POST /products

func CreateProduct(w http.ResponseWriter, r \*http.Request) {

var product Product

err := json.NewDecoder(r.Body).Decode(&product)

if err != nil {

http.Error(w, "Invalid input", http.StatusBadRequest)

return

}

products = append(products, product)

w.WriteHeader(http.StatusCreated)

json.NewEncoder(w).Encode(product)

}

// GET /products/:id

func GetProduct(w http.ResponseWriter, r \*http.Request) {

vars := mux.Vars(r)

id, err := strconv.Atoi(vars["id"])

if err != nil {

http.Error(w, "Invalid ID", http.StatusBadRequest)

return

}

for \_, product := range products {

if product.ID == id {

json.NewEncoder(w).Encode(product)

return

}

}

http.Error(w, "Product not found", http.StatusNotFound)

}

// GET /products

func GetProducts(w http.ResponseWriter, r \*http.Request) {

json.NewEncoder(w).Encode(products)

}

### Step 3.3: Services (Asynchronous Image Processing)

**Producer**:

go

Copy code

package services

import (

"log"

"os"

"github.com/streadway/amqp"

)

func PublishToQueue(message string) {

conn, err := amqp.Dial(os.Getenv("RABBITMQ\_URL"))

if err != nil {

log.Fatalf("Failed to connect to RabbitMQ: %v", err)

}

defer conn.Close()

ch, err := conn.Channel()

if err != nil {

log.Fatalf("Failed to open a channel: %v", err)

}

defer ch.Close()

q, err := ch.QueueDeclare(

"image\_queue",

false,

false,

false,

false,

nil,

)

if err != nil {

log.Fatalf("Failed to declare a queue: %v", err)

}

err = ch.Publish(

"",

q.Name,

false,

false,

amqp.Publishing{

ContentType: "text/plain",

Body: []byte(message),

},

)

if err != nil {

log.Fatalf("Failed to publish a message: %v", err)

}

log.Printf("Message published: %s", message)

}

**Consumer**:

go

Copy code

func ConsumeFromQueue() {

conn, err := amqp.Dial(os.Getenv("RABBITMQ\_URL"))

if err != nil {

log.Fatalf("Failed to connect to RabbitMQ: %v", err)

}

defer conn.Close()

ch, err := conn.Channel()

if err != nil {

log.Fatalf("Failed to open a channel: %v", err)

}

defer ch.Close()

msgs, err := ch.Consume(

"image\_queue",

"",

true,

false,

false,

false,

nil,

)

if err != nil {

log.Fatalf("Failed to register a consumer: %v", err)

}

forever := make(chan bool)

go func() {

for d := range msgs {

log.Printf("Received a message: %s", d.Body)

// Process image

}

}()

log.Printf("Waiting for messages. To exit press CTRL+C")

<-forever

}

## 4. ****Testing****

### Step 4.1: Test with REST Client

Create a file requests.rest in VSCode:

bash

Copy code

POST http://localhost:8080/products

Content-Type: application/json

{

"id": 1,

"user\_id": 1,

"product\_name": "Sample Product",

"product\_description": "This is a test product.",

"product\_images": ["https://example.com/image1.jpg"],

"product\_price": 99.99

}

###

GET http://localhost:8080/products

Use the REST Client to send requests.

### Step 5.2: Dockerize

#### 1. Create the Dockerfile

Add this Dockerfile to the root of your project directory:

dockerfile

Copy code

# Use the official Golang image as a base

FROM golang:1.20-alpine

# Set the working directory

WORKDIR /app

# Copy the Go module files and download dependencies

COPY go.mod go.sum ./

RUN go mod download

# Copy the rest of the application code

COPY . ./

# Build the application binary

RUN go build -o main .

# Expose the port that the app will run on

EXPOSE 8080

# Run the application

CMD ["./main"]

#### 2. Build the Docker Image

In your terminal, run:

bash

Copy code

docker build -t product-management-backend .

This will build the Docker image for your application.

#### 3. Run the Docker Container

Once the image is built, run the container:

bash

Copy code

docker run -p 8080:8080 --env-file .env product-management-backend

Replace --env-file .env with the path to your environment variables file, which should include configurations like:

makefile

Copy code

DB\_HOST=localhost

DB\_PORT=5432

DB\_USER=user

DB\_PASSWORD=password

REDIS\_HOST=localhost:6379

RABBITMQ\_URL=amqp://guest:guest@localhost:5672/

S3\_BUCKET\_NAME=my-s3-bucket

Your application will now be running at http://localhost:8080.

### Step 5.3: Multi-Service Setup with Docker Compose

#### 1. Create docker-compose.yml

This file will orchestrate running your application alongside PostgreSQL, Redis, and RabbitMQ.

yaml

Copy code

version: '3.8'

services:

app:

build: .

ports:

- "8080:8080"

environment:

- DB\_HOST=postgres

- DB\_PORT=5432

- DB\_USER=user

- DB\_PASSWORD=password

- REDIS\_HOST=redis:6379

- RABBITMQ\_URL=amqp://guest:guest@rabbitmq:5672/

- S3\_BUCKET\_NAME=my-s3-bucket

depends\_on:

- postgres

- redis

- rabbitmq

postgres:

image: postgres:14

environment:

POSTGRES\_USER: user

POSTGRES\_PASSWORD: password

POSTGRES\_DB=product\_db

ports:

- "5432:5432"

redis:

image: redis:6.2

ports:

- "6379:6379"

rabbitmq:

image: rabbitmq:3-management

ports:

- "5672:5672"

- "15672:15672" # Management UI

#### 2. Start All Services

Run the following command to start the application and its dependencies:

bash

Copy code

docker-compose up

This command will:

* Build and run your backend service (app).
* Start a PostgreSQL database.
* Start Redis for caching.
* Start RabbitMQ for message queuing.

You can now access your application at http://localhost:8080.

### Step 5.4: Verify the Setup

1. Open http://localhost:15672 to access the RabbitMQ management console. Use the default credentials guest/guest.
2. Test your API endpoints using tools like Postman or cURL:
   * **Create Product**:

bash

Copy code

curl -X POST http://localhost:8080/products \

-H "Content-Type: application/json" \

-d '{"user\_id":1, "product\_name":"Test Product", "product\_description":"Test description", "product\_images":["https://example.com/image1.jpg"], "product\_price":100.0}'

* + **Get Product by ID**:

bash

Copy code

curl http://localhost:8080/products/1

### Step 5.5: Deployment

#### Option 1: Deploy to AWS ECS

1. Push the Docker image to Amazon Elastic Container Registry (ECR).

bash

Copy code

aws ecr get-login-password --region <region> | docker login --username AWS --password-stdin <account\_id>.dkr.ecr.<region>.amazonaws.com

docker tag product-management-backend:latest <account\_id>.dkr.ecr.<region>.amazonaws.com/product-management-backend

docker push <account\_id>.dkr.ecr.<region>.amazonaws.com/product-management-backend

1. Create an ECS cluster and define a task definition pointing to the ECR image.

#### Option 2: Deploy to Kubernetes

1. Create a Kubernetes deployment file (deployment.yaml):

yaml

Copy code

apiVersion: apps/v1

kind: Deployment

metadata:

name: product-management-backend

spec:

replicas: 2

selector:

matchLabels:

app: product-management-backend

template:

metadata:

labels:

app: product-management-backend

spec:

containers:

- name: backend

image: <your\_dockerhub\_or\_ecr\_image>

ports:

- containerPort: 8080

---

apiVersion: v1

kind: Service

metadata:

name: product-management-service

spec:

type: LoadBalancer

ports:

- port: 80

targetPort: 8080

selector:

app: product-management-backend

1. Apply the deployment:

bash

Copy code

kubectl apply -f deployment.yaml

## ****6. Logging****

### Step 6.1: Centralized Logging with Logrus

Install Logrus:

bash

Copy code

go get github.com/sirupsen/logrus

Update main.go:

go

Copy code

import log "github.com/sirupsen/logrus"

func init() {

log.SetFormatter(&log.JSONFormatter{})

log.SetOutput(os.Stdout)

log.SetLevel(log.InfoLevel)

}

Use structured logs:

go

Copy code

log.WithFields(log.Fields{

"method": r.Method,

"path": r.URL.Path,

"status": status,

"duration": duration,

}).Info("Request handled")

## ****7. Caching****

### Step 7.1: Use Redis for Caching

Install the Redis client for Go:

bash

Copy code

go get github.com/go-redis/redis/v8

Initialize Redis in services/redis\_service.go:

go

Copy code

package services

import (

"context"

"log"

"os"

"github.com/go-redis/redis/v8"

)

var rdb \*redis.Client

var ctx = context.Background()

func InitRedis() {

rdb = redis.NewClient(&redis.Options{

Addr: os.Getenv("REDIS\_HOST"),

})

\_, err := rdb.Ping(ctx).Result()

if err != nil {

log.Fatalf("Could not connect to Redis: %v", err)

}

}

func CacheProduct(id string, data string) {

err := rdb.Set(ctx, id, data, 0).Err()

if err != nil {

log.Printf("Error caching product: %v", err)

}

}

func GetCachedProduct(id string) (string, error) {

return rdb.Get(ctx, id).Result()

}

Integrate caching in GetProduct:

go

Copy code

cachedProduct, err := GetCachedProduct(productID)

if err == nil {

w.Write([]byte(cachedProduct))

return

}

## ****8. Testing****

### Step 8.1: Unit Tests

Create handlers/product\_test.go:

go

Copy code

package handlers

import (

"net/http"

"net/http/httptest"

"strings"

"testing"

)

func TestCreateProduct(t \*testing.T) {

reqBody := `{"id":1,"user\_id":1,"product\_name":"Test Product","product\_description":"Test description","product\_price":50.0}`

req := httptest.NewRequest(http.MethodPost, "/products", strings.NewReader(reqBody))

w := httptest.NewRecorder()

CreateProduct(w, req)

if w.Code != http.StatusCreated {

t.Errorf("Expected status 201, got %d", w.Code)

}

}

### Step 8.2: Integration Tests

Write tests using mock RabbitMQ and Redis clients.

## ****9. Deployment****

### Step 9.1: Containerize with Docker

Create a Dockerfile:

dockerfile

Copy code

FROM golang:1.20-alpine

WORKDIR /app

COPY go.mod go.sum ./

RUN go mod download

COPY . ./

RUN go build -o main .

EXPOSE 8080

CMD ["./main"]

### Step 9.2: Docker Compose for Multi-Service Setup

Create docker-compose.yml:

yaml

Copy code

version: '3.8'

services:

app:

build: .

ports:

- "8080:8080"

environment:

- DB\_HOST=postgres

- REDIS\_HOST=redis:6379

- RABBITMQ\_URL=amqp://guest:guest@rabbitmq:5672/

- S3\_BUCKET\_NAME=my-s3-bucket

depends\_on:

- postgres

- redis

- rabbitmq

postgres:

image: postgres:14

environment:

POSTGRES\_USER: user

POSTGRES\_PASSWORD: password

ports:

- "5432:5432"

redis:

image: redis:6.2

ports:

- "6379:6379"

rabbitmq:

image: rabbitmq:3-management

ports:

- "5672:5672"

- "15672:15672"

### Step 9.3: Run the Services

bash

Copy code

docker-compose up