1. Design a database schema for a simple e-commerce website that has the following requirements:

* Products have a name, price, description, and image.
* Orders have a customer, date, and status.
* Customers have a name, email, and password.
* Each order can have multiple products.
* Products can belong to multiple orders.

Answer :

// Product model

type Product struct {

ID int64 `json:"id"`

Name string `json:"name"`

Price float64 `json:"price"`

Description string `json:"description"`

ImageURL string `json:"image\_url"`

}

// Order model

type Order struct {

ID int64 `json:"id"`

CustomerID int64 `json:"customer\_id"`

Date time.Time `json:"date"`

Status string `json:"status"`

}

// OrderItem model

type OrderItem struct {

ID int64 `json:"id"`

OrderID int64 `json:"order\_id"`

ProductID int64 `json:"product\_id"`

Quantity int64 `json:"quantity"`

Price float64 `json:"price"`

}

// Customer model

type Customer struct {

ID int64 `json:"id"`

Name string `json:"name"`

Email string `json:"email"`

Password string `json:"-"`

}

1. Write a RESTful API endpoint that allows customers to place an order for a list of products. The endpoint should:

* Validate the input data.
* Create a new order in the database.
* Associate the ordered products with the order.

Answer :

// PlaceOrderHandler handles POST requests to place a new order

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

// Parse the request body

var request struct {

CustomerID int64 `json:"customer\_id"`

ProductIDs []int64 `json:"product\_ids"`

Quantities map[int64]int64 `json:"quantities"`

}

if err := json.NewDecoder(r.Body).Decode(&request); err != nil {

http.Error(w, err.Error(), http.StatusBadRequest)

return

}

// Validate the input data

if request.CustomerID == 0 {

http.Error(w, "missing customer ID", http.StatusBadRequest)

return

}

if len(request.ProductIDs) == 0 {

http.Error(w, "missing product IDs", http.StatusBadRequest)

return

}

for \_, quantity := range request.Quantities {

if quantity <= 0 {

http.Error(w, "invalid product quantity", http.StatusBadRequest)

return

}

}

// Create a new order

order := &Order{

CustomerID: request.CustomerID,

Date: time.Now(),

Status: "pending",

}

if err := db.CreateOrder(order); err != nil {

http.Error(w, err.Error(), http.StatusInternalServerError)

return

}

// Associate the ordered products with the order

for i, productID := range request.ProductIDs {

product, err := db.GetProduct(productID)

if err != nil {

http.Error(w, err.Error(), http.StatusBadRequest)

return

}

quantity := request.Quantities[productID]

orderItem := &OrderItem{

OrderID: order.ID,

ProductID: product.ID,

Quantity: quantity,

Price: product.Price,

}

if err := db.CreateOrderItem(orderItem); err != nil {

http.Error(w, err.Error(), http.StatusInternalServerError)

return

}

// Update the product quantity

product.Quantity -= quantity

if err := db.UpdateProduct(product); err != nil {

http.Error(w, err.Error(), http.StatusInternalServerError)

return

}

// Remove the ordered product from the input data

request.ProductIDs[i] = 0

delete(request.Quantities, productID)

}

// Check if there are any invalid product IDs

for \_, productID := range request.ProductIDs {

if productID != 0 {

http.Error(w, "invalid product ID", http.StatusBadRequest)

return

}

}

// Return the created order

w.Header().Set("Content-Type", "application/json")

json.NewEncoder(w).Encode(order)

}

1. Write a RESTful API endpoint that allows customers to view their orders. The endpoint should:

* Authenticate the customer.
* Retrieve the list of orders associated with the customer.
* Include the list of products for each order.

Answer :

// GetOrdersHandler handles GET requests to retrieve the orders of a customer

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

// Get the customer ID from the authentication token

customerID, err := getCustomerIDFromToken(r)

if err != nil {

http.Error(w, err.Error(), http.StatusUnauthorized)

return

}

// Retrieve the orders associated with the customer

orders, err := db.GetOrdersByCustomer(customerID)

if err != nil {

http.Error(w, err.Error(), http.StatusInternalServerError)

return

}

// Retrieve the order items for each order and include the product details

for \_, order := range orders {

orderItems, err := db.GetOrderItemsByOrder(order.ID)

if err != nil {

http.Error(w, err.Error(), http.StatusInternalServerError)

return

}

for \_, orderItem := range orderItems {

product, err := db.GetProduct(orderItem.ProductID)

if err != nil {

http.Error(w, err.Error(), http.StatusInternalServerError)

return

}

orderItem.Product = product

}

order.Items = orderItems

}

// Return the orders in the response

w.Header().Set("Content-Type", "application/json")

json.NewEncoder(w).Encode(orders)

}

// getCustomerIDFromToken extracts the customer ID from the authentication token in the request header

func getCustomerIDFromToken(r \*http.Request) (int64, error) {

tokenString := r.Header.Get("Authorization")

if tokenString == "" {

return 0, fmt.Errorf("missing authentication token")

}

claims := &jwtClaims{}

token, err := jwt.ParseWithClaims(tokenString, claims, func(token \*jwt.Token) (interface{}, error) {

return []byte("your-256-bit-secret"), nil // replace with your secret key

})

if err != nil {

return 0, fmt.Errorf("invalid authentication token: %v", err)

}

if !token.Valid {

return 0, fmt.Errorf("expired authentication token")

}

return claims.CustomerID, nil

}

type jwtClaims struct {

jwt.StandardClaims

CustomerID int64 `json:"customer\_id"`

}

1. Write a RESTful API endpoint that allows the website admin to view all orders. The endpoint should:

* Authenticate the admin.
* Retrieve the list of all orders in the database.
* Include the list of products for each order.

Answer :

// GetAllOrdersHandler handles GET requests to retrieve all orders in the database

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

// Authenticate the admin

if !isAdminAuthenticated(r) {

http.Error(w, "unauthorized", http.StatusUnauthorized)

return

}

// Retrieve all orders in the database

orders, err := db.GetAllOrders()

if err != nil {

http.Error(w, err.Error(), http.StatusInternalServerError)

return

}

// Retrieve the order items for each order and include the product details

for \_, order := range orders {

orderItems, err := db.GetOrderItemsByOrder(order.ID)

if err != nil {

http.Error(w, err.Error(), http.StatusInternalServerError)

return

}

for \_, orderItem := range orderItems {

product, err := db.GetProduct(orderItem.ProductID)

if err != nil {

http.Error(w, err.Error(), http.StatusInternalServerError)

return

}

orderItem.Product = product

}

order.Items = orderItems

}

// Return the orders in the response

w.Header().Set("Content-Type", "application/json")

json.NewEncoder(w).Encode(orders)

}

// isAdminAuthenticated checks if the user in the request context is authenticated as an admin

func isAdminAuthenticated(r \*http.Request) bool {

user, \_, ok := r.BasicAuth()

if !ok {

return false

}

return user == "admin" // replace with your admin username

}

1. Implement a background task that runs every day at midnight and sends an email to each customer with a pending order reminder. The email should include a list of the products in their order and a link to complete the checkout process.

Answer :

package main

import (

"fmt"

"time"

"net/smtp"

"strings"

)

func main() {

// Set the time for the cron job to run (midnight every day)

c := cron.New()

c.AddFunc("0 0 \* \* \*", func() {

sendOrderReminders()

})

c.Start()

// Keep the program running

select{}

}

func sendOrderReminders() {

// Connect to the database and query for pending orders

db, err := sql.Open("mysql", "user:password@tcp(localhost:3306)/database")

if err != nil {

log.Fatal(err)

}

defer db.Close()

rows, err := db.Query("SELECT customer\_email, products FROM orders WHERE status = 'pending'")

if err != nil {

log.Fatal(err)

}

defer rows.Close()

// Iterate through the results and send emails to customers

for rows.Next() {

var customerEmail string

var products string

err = rows.Scan(&customerEmail, &products)

if err != nil {

log.Fatal(err)

}

// Compose the email message

message := fmt.Sprintf("Dear customer,\n\nYou have a pending order with the following products:\n%s\n\nPlease click the link below to complete the checkout process:\nhttp://www.example.com/checkout\n\nThanks for shopping with us!", products)

// Set up the SMTP client and send the email

auth := smtp.PlainAuth("", "sender@example.com", "password", "smtp.example.com")

to := []string{customerEmail}

msg := []byte("To: " + customerEmail + "\r\n" +

"Subject: Pending Order Reminder\r\n" +

"\r\n" +

message + "\r\n")

err = smtp.SendMail("smtp.example.com:587", auth, "sender@example.com", to, msg)

if err != nil {

log.Fatal(err)

}

}

}

1. Write a script that generates a CSV report with the following information for each order: Order ID Customer name Order date Total price of the order Status of the order

Answer :

package main

import (

"encoding/csv"

"os"

"database/sql"

"log"

)

type Order struct {

ID int

Customer string

Date string

Price float64

Status string

}

func main() {

// Connect to the database and query for orders

db, err := sql.Open("mysql", "user:password@tcp(localhost:3306)/database")

if err != nil {

log.Fatal(err)

}

defer db.Close()

rows, err := db.Query("SELECT id, customer, date, price, status FROM orders")

if err != nil {

log.Fatal(err)

}

defer rows.Close()

// Iterate through the results and write them to a CSV file

file, err := os.Create("orders.csv")

if err != nil {

log.Fatal(err)

}

defer file.Close()

writer := csv.NewWriter(file)

defer writer.Flush()

// Write header row

writer.Write([]string{"Order ID", "Customer Name", "Order Date", "Total Price", "Status"})

// Write data rows

for rows.Next() {

var order Order

err = rows.Scan(&order.ID, &order.Customer, &order.Date, &order.Price, &order.Status)

if err != nil {

log.Fatal(err)

continue

}

row := []string{fmt.Sprint(order.ID), order.Customer, order.Date, fmt.Sprintf("%.2f", order.Price), order.Status}

writer.Write(row)

}

}

1. Implement an API rate limiter that limits the number of requests per minute from a single IP address to 100.

Answer :

package main

import (

"net/http"

"sync"

"time"

)

const (

maxRequests = 100 // Maximum number of requests allowed per minute

requestsReset = 1 \* time.Minute // Reset time for request count

)

type rateLimiter struct {

requests map[string]int

mutex sync.Mutex

}

func (rl \*rateLimiter) Middleware(next http.Handler) http.Handler {

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

ip := r.RemoteAddr

rl.mutex.Lock()

rl.requests[ip]++

count := rl.requests[ip]

rl.mutex.Unlock()

if count > maxRequests {

http.Error(w, "Rate limit exceeded", http.StatusTooManyRequests)

return

}

time.AfterFunc(requestsReset, func() {

rl.mutex.Lock()

delete(rl.requests, ip)

rl.mutex.Unlock()

})

next.ServeHTTP(w, r)

})

}

func main() {

// Initialize the rate limiter

rl := &rateLimiter{

requests: make(map[string]int),

}

// Create the HTTP server

server := http.Server{

Addr: ":8080",

}

// Wrap the HTTP handler function with the rate limiter middleware

http.Handle("/", rl.Middleware(http.HandlerFunc(handler)))

// Start the server

server.ListenAndServe()

}

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

// Handle the request

}