Introduction to Go - Final Project

Concurrent Online Bookstore API with Periodic Sales Report

Introduction

Welcome to your final project for this course! This project is designed to consolidate and apply the knowledge you've gained over the past days. Using Go, You will develop a concurrent, robust, and efficient RESTful API for an online bookstore, alongside a daily sales report generation service.

This project will challenge you to:

- Implement interfaces and composition.
- Use nested structs to model complex data.
- Utilize Go's standard library packages effectively.
- Handle concurrency and synchronization.
- Manage goroutine lifecycles using contexts.
- Implement a periodic background job.

Project Overview

You will create an API server that manages books, authors, customers, and orders for an online bookstore. Additionally, you will implement a background task that periodically generates sales reports.

The API should allow clients to:

- **Books:**
    - Create, retrieve, update, and delete book records.
    - Search for books by various criteria (e.g., title, author, genre).
- **Authors:**
    - Manage author information.

    - Link books to their authors.
- **Customers:**
    - Create and manage customer profiles.
- **Orders:**
    - Place orders for books.
    - View order history.

Periodic Background Task:

- **Sales Report Generation:**
    - Runs on a daily interval (For instance, every day at midnight)
    - Aggregates daily sales data that occurred in the last 24 hours.
    - Runs as a concurrent background job without affecting API responsiveness.
    - Stores each generated report in a separate JSON file in the output-reports directory. The file names should contain the date and time when the report was generated (for instance: **report_090120250000.json**).
- Sales Report API:
    - Expose an API to list all the reports within a specific date range.

Project Objectives

By completing this project, you will:

- Apply interfaces and struct embedding for clean code organization.
- Model complex entities using nested structs.
- Implement a RESTful API using Go's net/http package.
- Use goroutines, channels, and synchronization primitives to handle concurrency.
- Manage request lifecycles and cancellations using the context package.
- Implement a periodic background task using Go's time-based operations.
- Ensure thread-safe access to shared resources.
- Handle errors and logging effectively.

Project Requirements

1. Design Data Models with Nested Structs

Define the data models for the bookstore, using nested structs to represent relationships.

Book Struct:

```
type Book struct {
    ID          int       `json:"id"`
    Title       string    `json:"title"`
    Author      Author    `json:"author"`
    Genres      []string  `json:"genres"`
    PublishedAt time.Time `json:"published_at"`
    Price       float64   `json:"price"`
    Stock       int       `json:"stock"`
}
```

Author Struct

```
type Author struct {
    ID        int    `json:"id"`
    FirstName string `json:"first_name"`
    LastName  string `json:"last_name"`
    Bio       string `json:"bio"`
}
```

Customer Struct

```
type Customer struct {
    ID        int       `json:"id"`
    Name      string    `json:"name"`
    Email     string    `json:"email"`
    Address   Address   `json:"address"`
    CreatedAt time.Time `json:"created_at"`
}
```

Order Struct

```
type Order struct {
    ID         int           `json:"id"`
    Customer   Customer     `json:"customer"`
    Items      []OrderItem  `json:"items"`
    TotalPrice float64       `json:"total_price"`
    CreatedAt  time.Time    `json:"created_at"`
    Status     string       `json:"status"`
}
```

OrderItem Struct

```
type OrderItem struct {
    Book     Book `json:"book"`
    Quantity int  `json:"quantity"`
}
```

Address Struct

```
type Address struct {
    Street     string `json:"street"`
    City       string `json:"city"`
    State      string `json:"state"`
    PostalCode string `json:"postal_code"`
    Country    string `json:"country"`
}
```

SalesReport Struct

```
type SalesReport struct {
    Timestamp       time.Time    `json:"timestamp"`
    TotalRevenue    float64      `json:"total_revenue"`
    TotalOrders     int          `json:"total_orders"`
    TopSellingBooks []BookSales  `json:"top_selling_books"`
}
```

```
type BookSales struct {
    Book     Book `json:"book"`
    Quantity int  `json:"quantity_sold"`
}
```

2. Define Interfaces for Data Access

Create interfaces to abstract data operations for **books**, **authors**, **customers**, and **orders**.

BookStore Interface Example

```
type BookStore interface {
    CreateBook(book Book) (Book, error)
    GetBook(id int) (Book, error)
    UpdateBook(id int, book Book) (Book, error)
    DeleteBook(id int) error
    SearchBooks(criteria SearchCriteria) ([]Book, error)
}
```

Define similar interfaces for **AuthorStore**, **CustomerStore**, and **OrderStore**. Include methods necessary for the periodic job, such as fetching orders within a time range.

3. Implement In-Memory Data Stores with Persistence and Synchronization

Use in-memory data stores for each entity.

Store the data into a JSON file and load it when the application starts (for example in a **database.json** file located at the root of the project directory).

Ensure thread-safe access with synchronization primitives like `sync.RWMutex`. (Hint: Use Lock() and Unlock() functions as needed when reading/writing).

Example for InMemoryBookStore:

```
type InMemoryBookStore struct {
    mu     sync.RWMutex
```

```
    books  map[int]Book
    nextID int
}
```

Implement methods for each interface, properly locking and unlocking the mutex to prevent race conditions.

4. Implement the RESTful API Endpoints

Use the `net/http` package to create HTTP handlers for each API endpoint.

Books Endpoints

- `POST /books` - Create a new book.
- `GET /books/{id}` - Retrieve a book by ID.
- `PUT /books/{id}` - Update a book.
- `DELETE /books/{id}` - Delete a book.
- `GET /books` - Search for books using query parameters.

Authors Endpoints

- `POST /authors` - Create a new author.
- `GET /authors/{id}` - Retrieve an author by ID.
- `PUT /authors/{id}` - Update an author.
- `DELETE /authors/{id}` - Delete an author.
- `GET /authors` - List all authors.

Customers and Orders

Implement similar endpoints for customers and orders, ensuring all CRUD operations are available.

5. Handle Concurrency and Synchronization

- Ensure handlers can handle multiple concurrent requests without data corruption.
- Use goroutines where appropriate, such as in order processing.
- Synchronize access to shared data stores using mutexes.

6. Use Context for Cancellation and Timeouts

- Accept context.Context from the HTTP request in your handlers.
- Use contexts to handle client cancellations and timeouts.
- Check ctx.Done() in long-running operations to terminate early if needed.

7. Error Handling and Responses

- Return appropriate HTTP status codes for different situations.
- Provide error messages in JSON format with a consistent structure.

Example **ErrorResponse** struct:

```
type ErrorResponse struct {
    Error string `json:"error"`
}
```

8. Implement Logging

- Use the log package to record API requests and errors.
- Log significant events, such as orders placed or background tasks executed.

10. Documentation

- Document your API endpoints, including request and response examples.
- Document how to run your API as well as the periodic report in a README file in the root of your project directory.
- Use Swagger (OpenAPI) specification to provide detailed API documentation.

11. Implement Periodic Sales Report Generation

Description

- Create a background goroutine that runs every 24 hours to generate sales reports.
- The report should include:
    - Total revenue.
    - Total number of orders.

    - Total books sold.
    - Top-selling books.
- Store or output the report appropriately (e.g., in-memory, file system, or logs).
- Provide an API endpoint to retrieve reports.

Implementation Steps

1. **Set Up the report generation to run periodically**
2. **Implement generateSalesReport Function**
    - Fetch orders within the last 24 hours.
    - Aggregate data and create the report.
    - Store the report in a file.
3. **Modify OrderStore Interface**
    - Add a method to retrieve orders within a time range.

```
type OrderStore interface {
    // Existing methods...
    GetOrdersInTimeRange(start, end time.Time) ([]Order, error)
}
```

4. **Ensure Concurrency Safety**
   Use synchronization primitives when accessing shared data during report generation.
5. **Integrate the Background Task**
   Start the sales report generator when the server starts, and handle application shutdown gracefully using contexts.

Detailed Instructions

Project Setup

1. **Initialize the Go Module**

```
go mod init <your_project_name>
```

2. **Organize Your Code**
    - Create packages for models, stores, handlers, and main application logic.

    - Separate concerns to maintain clean code organization.

Implement Data Models and Interfaces

- Define all structs and interfaces as specified.
- Ensure all structs have appropriate JSON tags.

Implement In-Memory Stores

- Implement thread-safe in-memory stores for each entity.
- Use **sync.RWMutex** to manage read/write access.

Develop API Handlers

- Implement HTTP handlers for all endpoints.
- Parse and validate incoming JSON requests.
- Use the appropriate HTTP methods and status codes.

Handle Concurrency

- Use goroutines for concurrent operations, such as order processing.
- Ensure all access to shared data is synchronized to prevent race conditions.

Use Contexts

- Pass context.Context to functions that may block or take a long time.
- Check **ctx.Done()** in goroutines to handle cancellations.

Implement the Periodic Job

- Use time.Ticker to schedule the generateSalesReport function.
- Ensure the background goroutine runs without interfering with the main application.
- Handle application shutdown by canceling the context passed to the periodic job.

Error Handling and Logging

- Implement consistent error responses.
- Use the log package to record important events and errors.

Testing

- Write a few manual tests in the README file to showcase the project functionalities.

Documentation

- Document how to run your application.
- Provide API documentation, including endpoint descriptions and examples.
- Provide an OpenAPI specification file alongside your project.

Grading Criteria

Your project will be evaluated based on the following:

1. **Correctness and Functionality (35 points)**
2. **Concurrency and Synchronization (25 points)**
3. **Use of Interfaces and Composition (10 points)**
4. **Error Handling and Logging (10 points)**
5. **Code Quality and Organization (10 points)**
6. **Background Task Implementation (10 points)**

Total: 100 points

Tips for Success

- **Plan Before Coding**
    - Design your data models and interfaces thoroughly.
    - Outline your application's structure and how components interact.
- **Develop Incrementally**
    - Start with basic CRUD operations for one entity.

    - Test and verify each part before moving on.
- **Test Frequently**
    - Use tools like cURL or Postman to test your API endpoints.
- **Manage Concurrency Carefully**
    - Be cautious with shared data access.
    - Use Mutexes and other synchronization primitives appropriately.
- **Use Contexts Effectively**
    - Pass context.Context in functions that handle requests.
    - Be attentive to cancellation signals.
- **Handle Errors Gracefully**
    - Provide meaningful error messages.
    - Use appropriate HTTP status codes.
- **Implement Logging**
    - Log significant events and errors to help with debugging.
- **Document Your Work**
    - Provide clear documentation for running and using your API.

Example Implementations

Create a Book

Request:

```
POST /books
Content-Type: application/json

{
  "title": "Effective Go Concurrency",
  "author": {
    "id": 1
  },
  "genres": ["Programming", "Technology"],
  "published_at": "2021-07-15T00:00:00Z",
  "price": 39.99,
  "stock": 100
```

```
}
```

Response:

```
HTTP/1.1 201 Created
Content-Type: application/json

{
  "id": 1,
  "title": "Effective Go Concurrency",
  "author": {
    "id": 1,
    "first_name": "John",
    "last_name": "Doe",
    "bio": "Software Engineer with expertise in Go."
  },
  "genres": ["Programming", "Technology"],
  "published_at": "2021-07-15T00:00:00Z",
  "price": 39.99,
  "stock": 100
}
```

Generate and Retrieve Sales Report

Request:

```
GET /reports/sales?start_date=YYYY-MM-DD&end_date=YYYY-MM-DD
```

Response:

```
{
  "reports": [
    {
      "timestamp": "2023-10-02T00:00:00Z",
      "total_revenue": 1500.75,
      "total_orders": 25,
      "top_selling_books": [
        {
          "book": {
```

```
            "id": 5,
            "title": "Go Concurrency In Depth",
            "author": { /*...*/ },
            "genres": ["Programming"],
            "published_at": "2021-04-10T00:00:00Z",
            "price": 49.99,
            "stock": 42
          },
          "quantity_sold": 10
        }
      ]
    }
  ]
}
```

Submission Instructions

- **Submission Format:** Submit your code repository (e.g., GitHub link) along with any necessary documentation in the README.md file.
- **Ensure:** Your code runs without errors and includes instructions on how to build and run the application.

Optional Enhancements

If you have extra time and wish to challenge yourself further, consider implementing:

- **Persistent Storage (Moderate):**
  - Replace in-memory stores with a database (e.g., SQLite, PostgreSQL).
- **Authentication and Authorization (Moderate):**
  - Implement user authentication using tokens (e.g., JWT).
- **Advanced Search and Filters (Easy):**
  - Add more search criteria and filtering options in your endpoints.
- **Automatic Price Adjustments (Moderate - Hard):**
  - Adjust book prices based on sales performance.

Final Remarks

This project simulates a real-world application and will help you solidify your understanding of:

- Designing and implementing complex data models.
- Building robust and concurrent APIs.
- Managing concurrency and synchronization in Go.
- Utilizing Go's standard library to its full potential.

Remember to manage your time effectively and ask questions if you encounter any challenges.

Good luck, and happy coding!