New Hire test Backend Go

Go Backend Developer Coding Test

Objective:
Your task is to create a RESTful API for a simple online marketplace using Go. This API will need to handle basic CRUD operations for products within the marketplace.

Requirements:

1. **Setup:**
    - Initialize a new Go project.
    - Use a SQL database like PostgreSQL or MySQL. You may use an ORM like GORM or directly interface with the database using Go's `database/sql` package.
2. **Product Model:**
    - Define a "Product" model with the following fields:
        - `ID` (integer, primary key, auto-increment)
        - `Name` (string, required)
        - `Description` (string, required)
        - `Price` (float, required)
        - `CreatedAt` (datetime, automatically generated)
        - `UpdatedAt` (datetime, automatically generated)

```
type Product struct {
    ID          uint       `json:"id" gorm:"primary_key"`
    Name        string     `json:"name" gorm:"type:varchar(100);not null"`
    Description string     `json:"description" gorm:"type:varchar(200);not null"`
    Price       float64    `json:"price" gorm:"not null"`
    CreatedAt   time.Time  `json:"created_at"`
    UpdatedAt   time.Time  `json:"updated_at"`
```

```
    DeletedAt   *time.Time `json:"deleted_at"`
}
```

1. **Endpoints:**
    - `GET /products/` - Retrieve a list of all products.
    - `POST /products/` - Create a new product.
    - `GET /products/:id/` - Retrieve a detailed view of a single product.
    - `PUT /products/:id/` - Update a single product.
    - `DELETE /products/:id/` - Delete a single product.
2. **Validation:**
    - Validate input data in the POST and PUT endpoints to ensure that all required fields are provided and that the price is a positive number.

```
// Sample validation in a hypothetical `CreateProduct` function
if product.Name == "" || product.Description == "" || product.Price <= 0 {
    // Respond with an error: you might use a package like "github.com/pkg/errors" to wrap
and add context to errors
    return errors.New("Invalid input data")
}
```

1. **Error Handling:**
    - Implement proper error handling, returning appropriate HTTP status codes and messages.
2. **Testing:**
    - Write unit tests to verify that all endpoints work as expected, with both valid and invalid inputs. You should include tests for successful operations as well as for expected failures, such as invalid input data or a request for a non-existent product.

```
// Sample testing a hypothetical `CreateProduct` function
func TestCreateProduct(t *testing.T) {
    // Set up test cases with both valid and invalid input data
    cases := []struct {
        input   Product
```

```
        success bool
    }{
        {Product{Name: "Valid Product", Description: "A valid product", Price: 10.0}, tru
e},
        {Product{Name: "", Description: "Invalid product", Price: 10.0}, false},
        // Add more test cases here
    }

    for _, tc := range cases {
        err := CreateProduct(tc.input)
        if tc.success {
            assert.Nil(t, err)
        } else {
            assert.NotNil(t, err)
        }
    }
}
```

1. **Documentation and Submission:**
    - Document your API endpoints and any setup required for running your application.
    - Submit your code as a link to a public Git repository (e.g., GitHub, GitLab).

Bonus:

- Implement user authentication, allowing only authenticated users to perform create, update, and delete operations.
- Include logging middleware that logs request and response data for each API call.
- Containerize the application using Docker.

Evaluation Criteria:

- **Functionality:** Does the application perform all required tasks?
- **Code Quality:** Is the code well-structured, readable, and maintainable?
- **Error Handling:** How does the application handle errors?
- **Testing:** Are there adequate tests, and do they cover various scenarios?
- **Documentation:** How well-documented is the code and the application setup?

Time Frame:

- Candidates have [24hours] to complete and submit the test.

This test covers fundamental areas of backend development with Go, including RESTful API construction, data modeling, validation, error handling, testing, and, optionally, authentication, and containerization. It helps assess a candidate's coding skills, problem-solving ability, and knowledge of Go in web development contexts.