Position: Backend Developer (Golang)

Experience Level: 1-2 years

Duration: 8-12 hours (for completing the assignment)

Assignment Overview

You are tasked with building a RESTful API in Go that manages `Tasks` and allows users to perform CRUD (Create, Read, Update, Delete) operations, as well as other advanced functionalities such as filtering, sorting, and user authentication. Additionally, your solution should utilize a proper database (PostgreSQL) instead of in-memory storage, implement JWT authentication, and employ basic rate-limiting and caching.

Advanced Functional Requirements

1. **Create Task**
   - Endpoint: `POST /tasks`
   - Request Body: JSON with fields `Title`, `Description`, and `Status`.
   - Validation: Ensure `Title` is non-empty and `Status` is a valid enum ("Pending", "In Progress", "Completed").
   - Additional Field: `DueDate`: (optional) a date-time value indicating the task's due date.
   - Response: Returns the created task with `ID`, `Title`, `Description`, `Status`, `CreatedAt`, `UpdatedAt`.
2. **Get All Tasks with Pagination and Filters**
   - Endpoint: `GET /tasks`
   - Query Parameters: `page`, `limit`, `status`, `due_date_after`, `due_date_before`, `sort_by` (e.g., `sort_by=due_date`), `sort_order` (e.g., `asc` or `desc`).
   - Validation: Ensure that `limit` is capped at a sensible value (e.g., 50).
   - Response: Returns a paginated list of tasks with their `ID`, `Title`, `Status`, `CreatedAt`, and `UpdatedAt`.
   - Implement sorting, filtering, and pagination.
3. **Get Task by ID**
   - Endpoint: `GET /tasks/{id}`
   - Response: Returns a task by the specified `ID`.
   - If no task with that ID exists, return a `404 Not Found` response.
4. **Update Task**

    - Endpoint: `PUT /tasks/{id}`
    - Request Body: JSON with fields `Title`, `Description`, `Status`, `DueDate` (optional).
    - Response: Returns the updated task, including `ID`, `Title`, `Description`, `Status`, `CreatedAt`, and `UpdatedAt`.
5. **Delete Task**
    - Endpoint: `DELETE /tasks/{id}`
    - Response: Status message `Task successfully deleted` if the task is deleted, or `404 Not Found` if the task with the given ID doesn't exist.
6. **User Authentication (JWT)**
    - Implement a simple login system with user authentication using JWT.
    - Endpoint: `POST /login`
    - Request Body: JSON with fields `username`, `password`.
    - If successful, return a JWT token.
    - For all task-related endpoints, require the JWT token to be passed in the `Authorization` header as `Bearer {token}`.
7. **Rate Limiting**
    - Implement basic rate-limiting for API requests. For example:
        - Limit each user to 60 requests per minute.
        - Return `429 Too Many Requests` if the rate limit is exceeded.
8. **Caching for Task Data**
    - Implement a caching mechanism for frequently accessed task data using an in-memory cache (such as `go-cache` or `Redis`).
    - Cache the task list for 5 minutes and only fetch data from the database if it’s not available in the cache.
9. **Database Design**
    - Use PostgreSQL to store tasks and user data.
    - Tasks should be stored in a table with the following fields:
        - `ID` (integer, primary key).
        - `Title` (string).
        - `Description` (string).
        - `Status` (enum: "Pending", "In Progress", "Completed").
        - `DueDate` (optional, DateTime).
        - `CreatedAt` (timestamp).
        - `UpdatedAt` (timestamp).
    - Users table for authentication:
        - `ID` (integer, primary key).
        - `Username` (string, unique).
        - `PasswordHash` (string).
        - `CreatedAt` (timestamp).
    - Use SQL queries and ORM (like `GORM` or `sqlx`) to interact with the database.

Non-Functional Requirements

- **Concurrency**: The solution should be able to handle multiple users and requests concurrently (use of goroutines, mutexes, etc.).
- **Database Migrations**: Provide database migrations for creating the necessary tables (you may use a tool like `goose` or `migrate`).
- **Logging**: Implement structured logging with a tool like `logrus` or `zap`. Logs should include request metadata and errors.
- **Error Handling**: Handle different types of errors such as database connection errors, invalid user inputs, etc.
- **Security**: Use bcrypt to hash passwords before storing them in the database. Ensure that JWT tokens are signed securely.

Advanced Features (Optional but Bonus Points)

1. **Background Jobs**:
   - Implement a background worker that checks the `DueDate` of tasks and sends an email reminder for overdue tasks.
   - Use a library like `goroutines` or a task queue such as `BeeQueue`.
2. **Search**:
   - Implement full-text search for tasks, where users can search by `Title`, `Description`, or `Status`.
3. **Audit Logs**:
   - Implement a basic audit log system to track user actions (such as creating, updating, and deleting tasks).
4. **Containerization**:
   - Dockerize the application and provide a `Dockerfile` to make it easy to deploy in any environment.
5. **CI/CD Pipeline**:
   - Setup a simple continuous integration and deployment pipeline using GitHub Actions or similar.

Expected Deliverables

1. **API Design Documentation**: A detailed explanation of the API endpoints, request parameters, and response format.
2. **Source Code**: A well-structured Go project with:
   - Models, handlers, and validation logic.
   - JWT-based authentication and rate limiting.
   - Database migrations and connection handling.
   - Caching, logging, and error handling.
3. **Database Setup**: Include SQL migrations or schema definition files.

4. **Unit Tests**: Write unit tests to verify the functionality of key parts of the application (e.g., CRUD operations, JWT validation).
5. **README.md**: Instructions on how to set up the application, including environment variables, database setup, running the application locally, and running tests.

Assessment Criteria

1. **Code Structure**: The application is well-structured, modular, and adheres to Go best practices.
2. **Database Design**: The database schema is designed properly, and migrations are handled correctly.
3. **Functionality**: The application meets the functional requirements and handles edge cases well.
4. **Error Handling**: The solution gracefully handles errors and provides useful error messages.
5. **Performance**: The solution is optimized for performance, especially for caching, rate limiting, and database access.
6. **Security**: Proper use of JWT, password hashing, and other security best practices.
7. **Test Coverage**: The unit tests and integration tests thoroughly cover the core functionality.

Example Requests & Responses

1. Create Task

Request:

bash
Copy

```
POST /tasks
{
  "Title": "Complete backend API",
  "Description": "Finish the development of the backend REST API.",
  "Status": "Pending",
  "DueDate": "2025-02-25T00:00:00Z"
}
```

Response:

arduino
Copy

```
HTTP/1.1 201 Created
```

```
{
  "ID": 1,
  "Title": "Complete backend API",
  "Description": "Finish the development of the backend REST API.",
  "Status": "Pending",
  "DueDate": "2025-02-25T00:00:00Z",
  "CreatedAt": "2025-02-19T10:00:00Z",
  "UpdatedAt": "2025-02-19T10:00:00Z"
}
```

2. Get Tasks with Filters

Request:

pgsql
Copy

```
GET /tasks?status=Pending&due_date_after=2025-02-20&sort_by=due_date&sort_order=asc&page=1&limit=5
```

Response:

arduino
Copy

```
HTTP/1.1 200 OK
{
  "tasks": [
    {
      "ID": 1,
      "Title": "Complete backend API",
      "Status": "Pending",
      "CreatedAt": "2025-02-19T10:00:00Z",
      "UpdatedAt": "2025-02-19T10:00:00Z"
    }
  ],
  "page": 1,
  "limit": 5,
  "total": 10
}
```