**Тестовое задание для Junior Golang разработчика**

**Описание задачи**

Вам необходимо разработать REST API на языке Go, которое будет взаимодействовать с базой данных PostgreSQL. API должно предоставлять возможность управления списком пользователей и их заказами. Задача усложняется добавлением дополнительных функциональных требований, таких как пагинация, фильтрация, авторизация и асинхронная обработка данных. (Дополнительные требования являются не обязательными, но желательными )

**Требования к функционалу**

**1. Создание пользователя**

* Метод: **POST /users**
* Тело запроса (JSON):

json

{

"name": "John Doe",

"email": "john.doe@example.com",

"age": 30,

"password": "securepassword"

}

* Ответ:
  + При успешном создании: HTTP статус **201 Created** и JSON с ID созданного пользователя:

json

{

"id": 1,

"name": "John Doe",

"email": "john.doe@example.com",

"age": 30

}

* + Если пользователь с таким email уже существует: HTTP статус **400 Bad Request**.

**2. Получение списка пользователей (с пагинацией и фильтрацией)**

* Метод: **GET /users**
* Параметры запроса:
  + **page** (int): номер страницы (по умолчанию **1**).
  + **limit** (int): количество элементов на странице (по умолчанию **10**).
  + **min\_age** (int): минимальный возраст (опционально).
  + **max\_age** (int): максимальный возраст (опционально).
* Пример запроса:

GET /users?page=2&limit=5&min\_age=20&max\_age=30

* Ответ:
  + Список пользователей в формате JSON:

json

{

"page": 2,

"limit": 5,

"total": 15,

"users": [

{

"id": 6,

"name": "Jane Smith",

"email": "jane.smith@example.com",

"age": 25

},

...

]

}

**3. Получение пользователя по ID**

* Метод: **GET /users/{id}**
* Ответ:
  + Если пользователь найден: HTTP статус **200 OK** и JSON с данными пользователя.
  + Если пользователь не найден: HTTP статус **404 Not Found**.

**4. Обновление пользователя**

* Метод: **PUT /users/{id}**
* Тело запроса (JSON):

json

{

"name": "John Updated",

"email": "john.updated@example.com",

"age": 31

}

* Ответ:
  + При успешном обновлении: HTTP статус **200 OK** и JSON с обновленными данными пользователя.
  + Если пользователь не найден: HTTP статус **404 Not Found**.

**5. Удаление пользователя**

* Метод: **DELETE /users/{id}**
* Ответ:
  + При успешном удалении: HTTP статус **204 No Content**.
  + Если пользователь не найден: HTTP статус **404 Not Found**.

**6. Создание заказа для пользователя**

* Метод: **POST /users/{user\_id}/orders**
* Тело запроса (JSON):

json

{

"product": "Laptop",

"quantity": 1,

"price": 1200.50

}

* Ответ:
  + При успешном создании: HTTP статус **201 Created** и JSON с ID заказа:

json

{

"id": 1,

"user\_id": 1,

"product": "Laptop",

"quantity": 1,

"price": 1200.50,

"created\_at": "2023-10-01T12:00:00Z"

}

**7. Получение списка заказов пользователя**

* Метод: **GET /users/{user\_id}/orders**
* Ответ:
  + Список заказов в формате JSON:

json

[

{

"id": 1,

"user\_id": 1,

"product": "Laptop",

"quantity": 1,

"price": 1200.50,

"created\_at": "2023-10-01T12:00:00Z"

},

...

]

**8. Авторизация**

Реализуйте простую JWT-авторизацию:

* Метод: **POST /auth/login**
* Тело запроса (JSON):

json

{

"email": "john.doe@example.com",

"password": "securepassword"

}

* Ответ:
  + При успешной авторизации: HTTP статус **200 OK** и токен:

json

{

"token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."

}

* + Если email или пароль неверны: HTTP статус **401 Unauthorized**.
  + Все методы, кроме **/auth/login**, должны проверять наличие и валидность JWT-токена.

**Требования к реализации**

1. База данных:
   * Используйте PostgreSQL.
   * Создайте таблицы **users** и **orders** со следующей структурой:

sql

CREATE TABLE users (

id SERIAL PRIMARY KEY,

name VARCHAR(255) NOT NULL,

email VARCHAR(255) UNIQUE NOT NULL,

age INT NOT NULL,

password\_hash VARCHAR(255) NOT NULL

);

CREATE TABLE orders (

id SERIAL PRIMARY KEY,

user\_id INT REFERENCES users(id) ON DELETE CASCADE,

product VARCHAR(255) NOT NULL,

quantity INT NOT NULL,

price DECIMAL(10, 2) NOT NULL,

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

1. ORM:
   * Для работы с базой данных используйте библиотеку [GORM](https://gorm.io/) .
2. HTTP-сервер:
   * Используйте стандартную библиотеку **net/http** или фреймворк [Gin](https://gin-gonic.com/) .
3. JWT-авторизация:
   * Используйте библиотеку [jwt-go](https://github.com/golang-jwt/jwt) .
4. Логирование:
   * Добавьте логирование основных операций (например, создание, обновление, удаление пользователя и заказов).

**Дополнительные требования**

1. Напишите документацию к API (можно использовать Swagger).
2. Убедитесь, что приложение можно легко запустить с помощью Docker Compose.

**Пример структуры проекта**

project/

├── cmd/

│ └── main.go # Точка входа в приложение

├── internal/

│ ├── handlers/ # Обработчики HTTP-запросов

│ ├── models/ # Модели базы данных

│ ├── repository/ # Работа с базой данных

│ ├── services/ # Бизнес-логика

│ ├── middleware/ # Middleware для авторизации

│ └── utils/ # Вспомогательные функции

├── migrations/ # SQL-миграции

├── go.mod # Файл зависимостей

├── .env # Переменные окружения

├── Dockerfile # Dockerfile для запуска приложения

└── docker-compose.yml # Docker Compose для запуска

**Критерии оценки**

1. Корректность работы API (все методы работают как ожидается).
2. Чистота кода (читаемость, структурированность).
3. Использование best practices (например, работа с контекстом, обработка ошибок, использование интерфейсов).
4. Покрытие тестами.
5. Документация и удобство запуска.