Сегодня я предлагаю поговорить о **HTTP-серверах в Go**. Это основа для создания веб-приложений и API, что является ключевым навыком для backend-разработчика.

**План на сегодня:**

1. Теория: Что такое HTTP-сервер и как он работает.
2. Практика: Напишем простой HTTP-сервер на Go, который будет обрабатывать запросы.

**Что такое HTTP-сервер?**

HTTP-сервер — это программа, которая принимает запросы от клиентов (например, браузеров или других приложений) и отвечает на них. Он работает по протоколу HTTP, который используется для передачи данных в интернете.

Пример:

* Ты открываешь сайт в браузере. Браузер отправляет запрос на сервер.
* Сервер обрабатывает запрос и отправляет ответ (например, HTML-страницу).

В Go для создания HTTP-сервера используется стандартная библиотека net/http. Она позволяет:

* Запускать сервер.
* Обрабатывать запросы.
* Возвращать ответы.

Теперь перейдем к практике! 🎉

**Практическое задание: Создаем простой HTTP-сервер**

1. Открой редактор кода (например, VS Code) и создай файл main.go.
2. Вставь следующий код:

package main

import (

"fmt"

"net/http"

)

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

fmt.Fprintf(w, "Привет, Миша! Ты создал свой первый HTTP-сервер!")

}

func main() {

http.HandleFunc("/", handler) // Обрабатываем запросы на корневой путь "/"

fmt.Println("Сервер запущен на порту 8080...")

http.ListenAndServe(":8080", nil) // Запускаем сервер на порту 8080

}

1. Запусти сервер командой:

go run main.go

1. Открой браузер и перейди по адресу:

http://localhost:8080

Ты должен увидеть сообщение: **"Привет, Миша! Ты создал свой первый HTTP-сервер!"**

**Чтобы остановить сервер в VS Code (или любом терминале),** нужно сделать следующее:

1. Найди терминал, в котором ты запустил сервер с помощью команды go run main.go.
2. Нажми сочетание клавиш **Ctrl + C** (на Windows/Linux) или **Command + C** (на Mac). Это завершит выполнение программы.

После этого сервер остановится, и ты сможешь внести изменения в код или запустить его заново.

Если у тебя есть еще немного времени, можем сделать следующее:

1. Добавить обработку разных маршрутов (например, /about или /contact).
2. Вернуть JSON-ответ вместо текста (это полезно для API).

**1. Обработка разных маршрутов**

Сейчас твой сервер отвечает только на запросы к /. Давай добавим новые маршруты, например, /about и /contact.

Обнови код в main.go:

package main

import (

"fmt"

"net/http"

)

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

fmt.Fprintf(w, "Добро пожаловать на главную страницу!")

}

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

fmt.Fprintf(w, "Это страница 'О нас'. Здесь можно узнать больше о нашем проекте.")

}

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

fmt.Fprintf(w, "Это страница 'Контакты'. Напишите нам на email: example@example.com")

}

func main() {

http.HandleFunc("/", homeHandler) // Главная страница

http.HandleFunc("/about", aboutHandler) // Страница "О нас"

http.HandleFunc("/contact", contactHandler) // Страница "Контакты"

fmt.Println("Сервер запущен на порту 8080...")

http.ListenAndServe(":8080", nil)

}

Теперь, если ты откроешь:

* http://localhost:8080/ — увидишь сообщение для главной страницы.
* http://localhost:8080/about — сообщение для страницы "О нас".
* http://localhost:8080/contact — сообщение для страницы "Контакты".

**2. Возвращаем JSON-ответ**

JSON — это формат данных, который часто используется для общения между сервером и клиентом. Давай добавим маршрут /api и вернем JSON-ответ.

Обнови код в main.go:

package main

import (

"encoding/json"

"fmt"

"net/http"

)

type Response struct {

Message string `json:"message"`

Status int `json:"status"`

}

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

response := Response{

Message: "Привет, это JSON-ответ!",

Status: 200,

}

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

json.NewEncoder(w).Encode(response)

}

func main() {

http.HandleFunc("/api", apiHandler) // Новый маршрут для API

fmt.Println("Сервер запущен на порту 8080...")

http.ListenAndServe(":8080", nil)

}

Теперь, если ты откроешь http://localhost:8080/api, сервер вернет JSON:

{

"message": "Привет, это JSON-ответ!",

"status": 200

}

Обработка **параметров в запросах**. Это очень важно, потому что сервер часто принимает данные от клиента, например, через URL или форму. 😊

**1. Обработка параметров в URL (Query Parameters)**

Иногда клиент передает данные через строку запроса (query string). Например:

http://localhost:8080/greet?name=Миша

Здесь сервер получает параметр name со значением Миша.

Добавим обработку таких параметров в код:

package main

import (

"fmt"

"net/http"

)

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

// Получаем параметр "name" из URL

name := r.URL.Query().Get("name")

if name == "" {

name = "Гость" // Если параметр не передан, используем значение по умолчанию

}

// Отправляем ответ клиенту

fmt.Fprintf(w, "Привет, %s! Добро пожаловать на сервер!", name)

}

func main() {

http.HandleFunc("/greet", greetHandler) // Маршрут для приветствия

fmt.Println("Сервер запущен на порту 8080...")

http.ListenAndServe(":8080", nil)

}

Теперь, если ты откроешь:

* http://localhost:8080/greet?name=Миша — сервер ответит: **"Привет, Миша! Добро пожаловать на сервер!"**
* http://localhost:8080/greet — сервер ответит: **"Привет, Гость! Добро пожаловать на сервер!"**

**2. Обработка данных из POST-запроса (Form Data)**

Когда клиент отправляет данные через форму, они передаются в теле запроса. Давай добавим обработку таких данных.

Обнови код:

package main

import (

"fmt"

"net/http"

)

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

if r.Method == "POST" {

// Читаем данные из формы

name := r.FormValue("name")

if name == "" {

name = "Гость" // Значение по умолчанию

}

// Отправляем ответ клиенту

fmt.Fprintf(w, "Привет, %s! Данные успешно получены через POST-запрос.", name)

} else {

// Если метод не POST, показываем форму

w.Header().Set("Content-Type", "text/html")

fmt.Fprintf(w, `

<form method="POST" action="/form">

Введите ваше имя: <input type="text" name="name">

<button type="submit">Отправить</button>

</form>

`)

}

}

func main() {

http.HandleFunc("/form", formHandler) // Маршрут для обработки формы

fmt.Println("Сервер запущен на порту 8080...")

http.ListenAndServe(":8080", nil)

}

Теперь:

1. Перейди на http://localhost:8080/form
2. Заполни форму и отправь данные.
3. Сервер обработает данные и покажет сообщение.

Теперь поговорим о **статус-кодах HTTP**. Это важная часть работы сервера, потому что они сообщают клиенту, как прошел запрос: успешно, с ошибкой или что-то еще. 😊

**Что такое HTTP-статус-коды?**

HTTP-статус-коды — это числовые значения, которые сервер отправляет клиенту вместе с ответом. Они делятся на категории:

* **2xx (Успех):** Запрос выполнен успешно. Например, 200 OK.
* **4xx (Ошибка клиента):** Клиент отправил неправильный запрос. Например, 404 Not Found.
* **5xx (Ошибка сервера):** Проблема на стороне сервера. Например, 500 Internal Server Error.

**Практика: Добавляем статус-коды в наш сервер**

1. **Пример с успешным ответом (200 OK):**

Обнови код:

package main

import (

"fmt"

"net/http"

)

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

w.WriteHeader(http.StatusOK) // Устанавливаем статус-код 200

fmt.Fprintf(w, "Запрос выполнен успешно! Статус: 200 OK")

}

func main() {

http.HandleFunc("/success", successHandler) // Маршрут для успешного ответа

fmt.Println("Сервер запущен на порту 8080...")

http.ListenAndServe(":8080", nil)

}

Теперь, если ты откроешь http://localhost:8080/success, сервер вернет статус 200 OK и сообщение.

1. **Пример с ошибкой клиента (404 Not Found):**

Добавим обработку ошибки, если клиент запросил несуществующую страницу:

package main

import (

"fmt"

"net/http"

)

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

w.WriteHeader(http.StatusNotFound) // Устанавливаем статус-код 404

fmt.Fprintf(w, "Ошибка: Страница не найдена! Статус: 404 Not Found")

}

func main() {

http.HandleFunc("/notfound", notFoundHandler) // Маршрут для ошибки 404

fmt.Println("Сервер запущен на порту 8080...")

http.ListenAndServe(":8080", nil)

}

Теперь, если ты откроешь http://localhost:8080/notfound, сервер вернет статус 404 Not Found и сообщение.

1. **Пример с ошибкой сервера (500 Internal Server Error):**

Добавим обработку ошибки на стороне сервера:

package main

import (

"fmt"

"net/http"

)

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

w.WriteHeader(http.StatusInternalServerError) // Устанавливаем статус-код 500

fmt.Fprintf(w, "Ошибка: Проблема на сервере! Статус: 500 Internal Server Error")

}

func main() {

http.HandleFunc("/servererror", serverErrorHandler) // Маршрут для ошибки 500

fmt.Println("Сервер запущен на порту 8080...")

http.ListenAndServe(":8080", nil)

}

Теперь, если ты откроешь http://localhost:8080/servererror, сервер вернет статус 500 Internal Server Error и сообщение.

**Попробуй это:**

1. Добавь все три маршрута (/success, /notfound, /servererror) в свой сервер.
2. Проверь, как сервер отвечает на каждый из них.

Ты уже уверенно справляешься с основами, так что давай перейдем к следующему пункту — **работа с HTML-файлами**. Это важно, если ты хочешь, чтобы сервер отдавал клиентам полноценные веб-страницы. 😊

**Как сервер работает с HTML-файлами?**

Сервер может отправлять HTML-файлы клиенту, чтобы тот отображал их в браузере. Это удобно, когда ты хочешь разделить логику (код Go) и представление (HTML).

**Практика: Отдаем HTML-файлы через сервер**

1. Создай папку templates в том же каталоге, где находится твой файл main.go.
2. В папке templates создай файл index.html с таким содержимым:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Главная страница</title>

</head>

<body>

<h1>Привет, Миша!</h1>

<p>Ты только что отдал HTML-файл через Go-сервер!</p>

</body>

</html>

1. Теперь обнови код в main.go:

package main

import (

"fmt"

"net/http"

)

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

http.ServeFile(w, r, "templates/index.html") // Отправляем HTML-файл клиенту

}

func main() {

http.HandleFunc("/", indexHandler) // Главная страница

fmt.Println("Сервер запущен на порту 8080...")

http.ListenAndServe(":8080", nil)

}

1. Запусти сервер и открой в браузере:

http://localhost:8080

Ты должен увидеть страницу с текстом "Привет, Миша!".

**Дополнительно: Отдача статических файлов (CSS, изображения)**

Если ты хочешь добавить стили или изображения, серверу нужно отдавать и эти файлы. Вот как это сделать:

1. Создай папку static и положи туда, например, файл style.css:

body {

font-family: Arial, sans-serif;

background-color: #f0f0f0;

color: #333;

}

1. Подключи CSS в index.html:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Главная страница</title>

<link rel="stylesheet" href="/static/style.css">

</head>

<body>

<h1>Привет, Миша!</h1>

<p>Ты только что отдал HTML-файл через Go-сервер!</p>

</body>

</html>

1. Обнови main.go, чтобы сервер мог отдавать статические файлы:

package main

import (

"fmt"

"net/http"

)

func main() {

// Отдаем HTML-файл

http.HandleFunc("/", func(w http.ResponseWriter, r \*http.Request) {

http.ServeFile(w, r, "templates/index.html")

})

// Отдаем статические файлы (CSS, изображения)

http.Handle("/static/", http.StripPrefix("/static/", http.FileServer(http.Dir("static"))))

fmt.Println("Сервер запущен на порту 8080...")

http.ListenAndServe(":8080", nil)

}

Теперь сервер будет отдавать и HTML, и статические файлы (например, CSS). Перезагрузи страницу, чтобы увидеть стили.

Ты уже на финишной прямой к созданию полноценного backend-приложения. Давай начнем разбираться с базами данных. 😊

**Что такое база данных?**

База данных (БД) — это место, где хранятся данные твоего приложения. Например, пользователи, заказы, сообщения и т.д. В backend-разработке часто используются реляционные базы данных, такие как **SQLite**, **PostgreSQL** или **MySQL**.

Сегодня мы начнем с **SQLite** — это легковесная база данных, которая идеально подходит для обучения и небольших проектов.

**Установка SQLite**

1. Убедись, что у тебя установлен Go. SQLite встроена в Go через библиотеку github.com/mattn/go-sqlite3.
2. Установи библиотеку SQLite:

go get github.com/mattn/go-sqlite3

**Практика: Работа с SQLite**

1. Создадим базу данных и таблицу для хранения данных о пользователях.

Обнови файл main.go:

package main

import (

"database/sql"

"fmt"

"log"

\_ "github.com/mattn/go-sqlite3"

)

func main() {

// Подключаемся к базе данных (или создаем файл базы, если его нет)

db, err := sql.Open("sqlite3", "./users.db")

if err != nil {

log.Fatal(err)

}

defer db.Close()

// Создаем таблицу пользователей, если ее еще нет

createTable := `

CREATE TABLE IF NOT EXISTS users (

id INTEGER PRIMARY KEY AUTOINCREMENT,

name TEXT NOT NULL,

email TEXT NOT NULL UNIQUE

);`

\_, err = db.Exec(createTable)

if err != nil {

log.Fatal("Ошибка при создании таблицы:", err)

}

fmt.Println("Таблица пользователей успешно создана или уже существует.")

}

1. Запусти сервер командой:

go run main.go

Если все прошло успешно, в папке с проектом появится файл users.db — это твоя база данных.

**Добавляем данные в базу**

Теперь давай добавим пользователя в таблицу. Обнови main.go:

package main

import (

"database/sql"

"fmt"

"log"

\_ "github.com/mattn/go-sqlite3"

)

func main() {

db, err := sql.Open("sqlite3", "./users.db")

if err != nil {

log.Fatal(err)

}

defer db.Close()

// Добавляем пользователя

insertUser := `INSERT INTO users (name, email) VALUES (?, ?);`

\_, err = db.Exec(insertUser, "Миша", "misha@example.com")

if err != nil {

log.Fatal("Ошибка при добавлении пользователя:", err)

}

fmt.Println("Пользователь успешно добавлен!")

}

Запусти сервер, и пользователь с именем "Миша" и email "[misha@example.com](mailto:misha@example.com)" будет добавлен в базу.

**Читаем данные из базы**

Теперь давай выведем всех пользователей из базы:

package main

import (

"database/sql"

"fmt"

"log"

\_ "github.com/mattn/go-sqlite3"

)

func main() {

db, err := sql.Open("sqlite3", "./users.db")

if err != nil {

log.Fatal(err)

}

defer db.Close()

// Читаем всех пользователей

rows, err := db.Query("SELECT id, name, email FROM users;")

if err != nil {

log.Fatal("Ошибка при чтении пользователей:", err)

}

defer rows.Close()

fmt.Println("Список пользователей:")

for rows.Next() {

var id int

var name, email string

err = rows.Scan(&id, &name, &email)

if err != nil {

log.Fatal(err)

}

fmt.Printf("ID: %d, Имя: %s, Email: %s\n", id, name, email)

}

}

Запусти код, и ты увидишь список всех пользователей в базе.

**Что дальше?**

1. Добавим обработку данных из базы через HTTP-запросы.
2. Научимся обновлять и удалять данные.
3. Подключим более мощную базу данных, например PostgreSQL.

Теперь давай научимся взаимодействовать с базой данных через HTTP-запросы. Это уже полноценная работа с API, где сервер будет получать запросы, обрабатывать их и взаимодействовать с базой данных. 😊

**Что мы сделаем?**

1. Добавим возможность создавать новых пользователей через POST-запрос.
2. Реализуем получение списка пользователей через GET-запрос.

**1. Создание пользователя через POST-запрос**

Обнови файл main.go:

package main

import (

"database/sql"

"encoding/json"

"fmt"

"log"

"net/http"

\_ "github.com/mattn/go-sqlite3"

)

var db \*sql.DB

func initDB() {

var err error

db, err = sql.Open("sqlite3", "./users.db")

if err != nil {

log.Fatal(err)

}

// Создаем таблицу пользователей, если ее еще нет

createTable := `

CREATE TABLE IF NOT EXISTS users (

id INTEGER PRIMARY KEY AUTOINCREMENT,

name TEXT NOT NULL,

email TEXT NOT NULL UNIQUE

);`

\_, err = db.Exec(createTable)

if err != nil {

log.Fatal("Ошибка при создании таблицы:", err)

}

fmt.Println("База данных готова!")

}

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

if r.Method != "POST" {

http.Error(w, "Метод не поддерживается", http.StatusMethodNotAllowed)

return

}

var user struct {

Name string `json:"name"`

Email string `json:"email"`

}

err := json.NewDecoder(r.Body).Decode(&user)

if err != nil {

http.Error(w, "Неверный формат данных", http.StatusBadRequest)

return

}

// Добавляем пользователя в базу

insertUser := `INSERT INTO users (name, email) VALUES (?, ?);`

\_, err = db.Exec(insertUser, user.Name, user.Email)

if err != nil {

http.Error(w, "Ошибка при добавлении пользователя", http.StatusInternalServerError)

return

}

w.WriteHeader(http.StatusCreated)

fmt.Fprintf(w, "Пользователь %s успешно создан!", user.Name)

}

func main() {

initDB()

defer db.Close()

http.HandleFunc("/users", createUserHandler) // Маршрут для создания пользователя

fmt.Println("Сервер запущен на порту 8080...")

http.ListenAndServe(":8080", nil)

}

Теперь ты можешь отправить POST-запрос на http://localhost:8080/users, чтобы создать нового пользователя.

Пример запроса (через Postman или curl):

POST http://localhost:8080/users

Content-Type: application/json

{

"name": "Миша",

"email": "misha@example.com"

}

Сервер добавит пользователя в базу и вернет сообщение: **"Пользователь Миша успешно создан!"**

**2. Получение списка пользователей через GET-запрос**

Теперь добавим возможность получать всех пользователей из базы:

Обнови main.go:

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

if r.Method != "GET" {

http.Error(w, "Метод не поддерживается", http.StatusMethodNotAllowed)

return

}

rows, err := db.Query("SELECT id, name, email FROM users;")

if err != nil {

http.Error(w, "Ошибка при получении пользователей", http.StatusInternalServerError)

return

}

defer rows.Close()

var users []struct {

ID int `json:"id"`

Name string `json:"name"`

Email string `json:"email"`

}

for rows.Next() {

var user struct {

ID int

Name string

Email string

}

err = rows.Scan(&user.ID, &user.Name, &user.Email)

if err != nil {

http.Error(w, "Ошибка при чтении данных", http.StatusInternalServerError)

return

}

users = append(users, user)

}

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

json.NewEncoder(w).Encode(users)

}

func main() {

initDB()

defer db.Close()

http.HandleFunc("/users", func(w http.ResponseWriter, r \*http.Request) {

if r.Method == "POST" {

createUserHandler(w, r)

} else if r.Method == "GET" {

getUsersHandler(w, r)

} else {

http.Error(w, "Метод не поддерживается", http.StatusMethodNotAllowed)

}

})

fmt.Println("Сервер запущен на порту 8080

Отлично, Миша! Ты уже на пути к созданию полноценного API. Теперь давай добавим возможность **обновлять** и **удалять** пользователей через HTTP-запросы. Это завершит базовый CRUD (Create, Read, Update, Delete) функционал. 😊

**3. Обновление данных пользователя (PUT-запрос)**

Обновим код, чтобы можно было обновлять данные пользователя по его id.

Добавь новый обработчик для обновления пользователя:

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

if r.Method != "PUT" {

http.Error(w, "Метод не поддерживается", http.StatusMethodNotAllowed)

return

}

var user struct {

ID int `json:"id"`

Name string `json:"name"`

Email string `json:"email"`

}

err := json.NewDecoder(r.Body).Decode(&user)

if err != nil {

http.Error(w, "Неверный формат данных", http.StatusBadRequest)

return

}

// Обновляем данные пользователя в базе

updateUser := `UPDATE users SET name = ?, email = ? WHERE id = ?;`

result, err := db.Exec(updateUser, user.Name, user.Email, user.ID)

if err != nil {

http.Error(w, "Ошибка при обновлении пользователя", http.StatusInternalServerError)

return

}

rowsAffected, \_ := result.RowsAffected()

if rowsAffected == 0 {

http.Error(w, "Пользователь с таким ID не найден", http.StatusNotFound)

return

}

w.WriteHeader(http.StatusOK)

fmt.Fprintf(w, "Пользователь с ID %d успешно обновлен!", user.ID)

}

Теперь добавь маршрут для обработки PUT-запросов:

http.HandleFunc("/users/update", updateUserHandler) // Маршрут для обновления пользователя

Пример PUT-запроса (через Postman или curl):

PUT http://localhost:8080/users/update

Content-Type: application/json

{

"id": 1,

"name": "Михаил",

"email": "mikhail@example.com"

}

Если пользователь с таким id существует, его данные будут обновлены.

**4. Удаление пользователя (DELETE-запрос)**

Теперь добавим возможность удалять пользователя по его id.

Добавь новый обработчик для удаления пользователя:

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

if r.Method != "DELETE" {

http.Error(w, "Метод не поддерживается", http.StatusMethodNotAllowed)

return

}

var user struct {

ID int `json:"id"`

}

err := json.NewDecoder(r.Body).Decode(&user)

if err != nil {

http.Error(w, "Неверный формат данных", http.StatusBadRequest)

return

}

// Удаляем пользователя из базы

deleteUser := `DELETE FROM users WHERE id = ?;`

result, err := db.Exec(deleteUser, user.ID)

if err != nil {

http.Error(w, "Ошибка при удалении пользователя", http.StatusInternalServerError)

return

}

rowsAffected, \_ := result.RowsAffected()

if rowsAffected == 0 {

http.Error(w, "Пользователь с таким ID не найден", http.StatusNotFound)

return

}

w.WriteHeader(http.StatusOK)

fmt.Fprintf(w, "Пользователь с ID %d успешно удален!", user.ID)

}

Добавь маршрут для обработки DELETE-запросов:

http.HandleFunc("/users/delete", deleteUserHandler) // Маршрут для удаления пользователя

Пример DELETE-запроса (через Postman или curl):

DELETE http://localhost:8080/users/delete

Content-Type: application/json

{

"id": 1

}

Если пользователь с таким id существует, он будет удален из базы.

**Полный код main.go:**

Теперь твой сервер поддерживает полный CRUD-функционал:

* **POST /users** — Создание пользователя.
* **GET /users** — Получение списка пользователей.
* **PUT /users/update** — Обновление данных пользователя.
* **DELETE /users/delete** — Удаление пользователя.

Вот как будет выглядеть весь код:

package main

import (

"database/sql"

"encoding/json"

"fmt"

"log"

"net/http"

\_ "github.com/mattn/go-sqlite3"

)

var db \*sql.DB

func initDB() {

var err error

db, err = sql.Open("sqlite3", "./users.db")

if err != nil {

log.Fatal(err)

}

createTable := `

CREATE TABLE IF NOT EXISTS users (

id INTEGER PRIMARY KEY AUTOINCREMENT,

name TEXT NOT NULL,

email TEXT NOT NULL UNIQUE

);`

\_, err = db.Exec(createTable)

if err != nil {

log.Fatal("Ошибка при создании таблицы:", err)

}

fmt.Println("База данных готова!")

}

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

if r.Method != "POST" {

http.Error(w, "Метод не поддерживается", http.StatusMethodNotAllowed)

return

}

var user struct {

Name string `json:"name"`

Email string `json:"email"`

}

err := json.NewDecoder(r.Body).Decode(&user)

if err != nil {

http.Error(w, "Неверный формат данных", http.StatusBadRequest)

return

}

insertUser := `INSERT INTO users (name, email) VALUES (?, ?);`

\_, err = db.Exec(insertUser, user.Name, user.Email)

if err != nil {

http.Error(w, "Ошибка при добавлении пользователя", http.StatusInternalServerError)

return

}

w.WriteHeader(http.StatusCreated)

fmt.Fprintf(w, "Пользователь %s успешно создан!", user.Name)

}

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

if r.Method != "GET" {

http.Error(w, "Метод не поддерживается", http.StatusMethodNotAllowed)

return

}

rows, err := db.Query("SELECT id, name, email FROM users;")

if err != nil {

http.Error(w, "Ошибка при получении пользователей", http.StatusInternalServerError)

return

}

defer rows.Close()

var users []struct {

ID int `json:"id"`

Name string `json:"name"`

Email string `json:"email"`

}

for rows.Next() {

var user struct {

ID int

Name string

Email string

}

err = rows.Scan(&user.ID, &user.Name, &user.Email)

if err != nil {

http.Error(w, "Ошибка при чтении данных", http.StatusInternalServerError)

return

}

users = append(users, user)

}

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

json.NewEncoder(w).Encode(users)

}

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

if r.Method != "PUT" {

http.Error(w, "Метод не поддерживается", http.StatusMethodNotAllowed)

return

}

var user struct {

ID int `json:"id"`

Name string `json:"name"`

Email string `json:"email"`

}

err := json.NewDecoder(r.Body).Decode(&user)

if err != nil {

http.Error(w, "Неверный формат данных", http.StatusBadRequest)

return

}

updateUser := `UPDATE users SET name = ?, email = ? WHERE id = ?;`

result, err := db.Exec(updateUser, user.Name, user.Email, user.ID)

if err != nil {

http.Error(w, "Ошибка при обновлении пользователя", http.StatusInternalServerError)

return

}

rowsAffected, \_ := result.RowsAffected()

if rowsAffected == 0 {

http.Error(w, "Пользователь с таким ID не найден", http.StatusNotFound)

return

}

w.WriteHeader(http.StatusOK)

fmt.Fprintf(w, "Пользователь с ID %d успешно обновлен!", user.ID)

}

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

if r.Method != "DELETE" {

http.Error(w, "Метод не поддерживается", http.StatusMethodNotAllowed)

return

}

var user struct {

ID int `json:"id"`

}

err := json.NewDecoder(r.Body).Decode(&user)

if err != nil {

http.Error(w, "Неверный формат данных", http.StatusBadRequest)

return

}

deleteUser := `DELETE FROM users WHERE id = ?;`

result, err := db.Exec(deleteUser, user.ID)

if err != nil {

http.Error(w, "Ошибка при удалении пользователя", http.StatusInternalServerError)

return

}

rowsAffected, \_ := result.RowsAffected()

if rowsAffected == 0 {

http.Error(w, "Пользователь с таким ID не найден", http.StatusNotFound)

return

}

w.WriteHeader(http.StatusOK)

fmt.Fprintf(w, "Пользователь с ID %d успешно удален!", user.ID)

}

func main() {

initDB()

defer db.Close()

// Маршруты для CRUD операций

http.HandleFunc("/users", func(w http.ResponseWriter, r \*http.Request) {

if r.Method == "POST" {

createUserHandler(w, r)

} else if r.Method == "GET" {

getUsersHandler(w, r)

} else {

http.Error(w, "Метод не поддерживается", http.StatusMethodNotAllowed)

}

})

http.HandleFunc("/users/update", updateUserHandler) // Обновление пользователя

http.HandleFunc("/users/delete", deleteUserHandler) // Удаление пользователя

fmt.Println("Сервер запущен на порту 8080...")

log.Fatal(http.ListenAndServe(":8080", nil))

}

**Что теперь умеет твой сервер?**

1. **POST /users** — Создает нового пользователя.
2. **GET /users** — Возвращает список всех пользователей.
3. **PUT /users/update** — Обновляет данные пользователя по его id.
4. **DELETE /users/delete** — Удаляет пользователя по его id.

**Протестируй функционал:**

1. Создай пользователя через POST-запрос.
2. Получи список пользователей через GET-запрос.
3. Обнови данные пользователя через PUT-запрос.
4. Удали пользователя через DELETE-запрос.

Если все работает, поздравляю — ты создал полноценное API с базой данных! 🚀

Отлично, Миша! Ты уже создал полноценное API с CRUD-операциями, и это огромный шаг вперед! 🚀 Теперь давай разберем, как можно улучшить и расширить твое приложение. Вот несколько идей, куда можно двигаться дальше:

**Что можно изучить дальше?**

1. **Валидация данных** — проверка входящих данных от клиента (например, чтобы email был корректным).
2. **Работа с окружением (environment variables)** — хранение конфиденциальных данных (например, паролей) в безопасном месте.
3. **Подключение PostgreSQL** — переход на более мощную базу данных.
4. **Создание RESTful API** — добавим маршруты с RESTful-структурой.
5. **Docker** — упакуем приложение в контейнер для удобного развертывания.
6. **Тестирование API** — научимся писать тесты для проверки работы сервера.

**Начнем с 1. Валидация данных**

Когда сервер принимает данные от клиента, важно убедиться, что они корректны. Например:

* Поле name не должно быть пустым.
* Поле email должно быть валидным адресом электронной почты.

Добавим валидацию в обработчики.

**Валидация при создании пользователя**

Обнови функцию createUserHandler:

import (

"regexp"

"strings"

)

func isValidEmail(email string) bool {

// Простая проверка на валидность email

re := regexp.MustCompile(`^[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$`)

return re.MatchString(email)

}

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

if r.Method != "POST" {

http.Error(w, "Метод не поддерживается", http.StatusMethodNotAllowed)

return

}

var user struct {

Name string `json:"name"`

Email string `json:"email"`

}

err := json.NewDecoder(r.Body).Decode(&user)

if err != nil {

http.Error(w, "Неверный формат данных", http.StatusBadRequest)

return

}

// Валидация данных

user.Name = strings.TrimSpace(user.Name)

user.Email = strings.TrimSpace(user.Email)

if user.Name == "" {

http.Error(w, "Имя не может быть пустым", http.StatusBadRequest)

return

}

if !isValidEmail(user.Email) {

http.Error(w, "Некорректный email", http.StatusBadRequest)

return

}

// Добавляем пользователя в базу

insertUser := `INSERT INTO users (name, email) VALUES (?, ?);`

\_, err = db.Exec(insertUser, user.Name, user.Email)

if err != nil {

http.Error(w, "Ошибка при добавлении пользователя", http.StatusInternalServerError)

return

}

w.WriteHeader(http.StatusCreated)

fmt.Fprintf(w, "Пользователь %s успешно создан!", user.Name)

}

Теперь сервер будет проверять:

1. Имя не должно быть пустым.
2. Email должен быть корректным.

**2. Работа с окружением (environment variables)**

Хранить конфиденциальные данные (например, строку подключения к базе данных) в коде — плохая практика. Вместо этого используют переменные окружения.

**Установка переменных окружения**

1. Создай файл .env в корне проекта:

DB\_PATH=./users.db

PORT=8080

1. Установи библиотеку для работы с .env:

go get github.com/joho/godotenv

1. Обнови main.go, чтобы загружать переменные окружения:

import (

"os"

"github.com/joho/godotenv"

)

func initEnv() {

err := godotenv.Load()

if err != nil {

log.Fatal("Ошибка загрузки файла .env")

}

}

1. Используй переменные окружения в коде:

func main() {

initEnv()

dbPath := os.Getenv("DB\_PATH")

port := os.Getenv("PORT")

db, err := sql.Open("sqlite3", dbPath)

if err != nil {

log.Fatal(err)

}

defer db.Close()

// Маршруты

http.HandleFunc("/users", func(w http.ResponseWriter, r \*http.Request) {

if r.Method == "POST" {

createUserHandler(w, r)

} else if r.Method == "GET" {

getUsersHandler(w, r)

} else {

http.Error(w, "Метод не поддерживается", ...):

http.StatusMethodNotAllowed)

}

})

http.HandleFunc("/users/update", updateUserHandler) // Обновление пользователя

http.HandleFunc("/users/delete", deleteUserHandler) // Удаление пользователя

fmt.Printf("Сервер запущен на порту %s...\n", port)

log.Fatal(http.ListenAndServe(":"+port, nil))

}

**Что мы сделали?**

1. Добавили поддержку переменных окружения через .env файл.
2. Теперь путь к базе данных (DB\_PATH) и порт сервера (PORT) задаются через переменные окружения, а не жестко прописаны в коде.

**Как запустить сервер с переменными окружения?**

1. Убедись, что файл .env находится в корне проекта и содержит:

DB\_PATH=./users.db

PORT=8080

1. Запусти сервер как обычно:

go run main.go

Сервер будет использовать значения из .env.

**Что дальше?**

1. **Подключение PostgreSQL** — заменим SQLite на более мощную базу данных.
2. **Docker** — упакуем приложение в контейнер для удобного развертывания.
3. **Тестирование API** — научимся писать тесты для проверки работы сервера.

PostgreSQL — это мощная реляционная база данных, которая широко используется в больших проектах. Давай подключим её к твоему приложению. 😊

**Шаг 1. Установка PostgreSQL**

1. Установи PostgreSQL на свой компьютер. Если его ещё нет, скачай с официального сайта: <https://www.postgresql.org/download/>.
2. Во время установки запомни:
   * Имя пользователя (по умолчанию: postgres).
   * Пароль.
   * Порт (по умолчанию: 5432).
3. Запусти PostgreSQL и создай базу данных:
   * Открой терминал и подключись к PostgreSQL:
   * psql -U postgres
   * Создай базу данных:
   * CREATE DATABASE go\_backend;
   * Выйди из консоли PostgreSQL:
   * \q

**Шаг 2. Установка драйвера PostgreSQL для Go**

1. Установи библиотеку для работы с PostgreSQL:
2. go get github.com/lib/pq

**Шаг 3. Подключение к PostgreSQL**

Обнови файл main.go:

package main

import (

"database/sql"

"fmt"

"log"

"net/http"

"os"

"github.com/joho/godotenv"

\_ "github.com/lib/pq"

)

var db \*sql.DB

func initEnv() {

err := godotenv.Load()

if err != nil {

log.Fatal("Ошибка загрузки файла .env")

}

}

func initDB() {

var err error

// Получаем строку подключения из переменных окружения

connStr := os.Getenv("POSTGRES\_URL")

db, err = sql.Open("postgres", connStr)

if err != nil {

log.Fatal("Ошибка подключения к базе данных:", err)

}

// Проверяем подключение

err = db.Ping()

if err != nil {

log.Fatal("Ошибка проверки подключения:", err)

}

// Создаем таблицу, если её нет

createTable := `

CREATE TABLE IF NOT EXISTS users (

id SERIAL PRIMARY KEY,

name TEXT NOT NULL,

email TEXT NOT NULL UNIQUE

);`

\_, err = db.Exec(createTable)

if err != nil {

log.Fatal("Ошибка при создании таблицы:", err)

}

fmt.Println("Подключение к PostgreSQL установлено, таблица готова!")

}

**Шаг 4. Настройка переменных окружения**

Обнови файл .env:

POSTGRES\_URL=postgres://postgres:your\_password@localhost:5432/go\_backend?sslmode=disable

PORT=8080

Здесь:

* postgres — имя пользователя.
* your\_password — пароль, который ты указал при установке PostgreSQL.
* localhost — хост (если база на твоем компьютере).
* 5432 — порт PostgreSQL.
* go\_backend — имя базы данных.

**Шаг 5. Использование PostgreSQL в обработчиках**

Теперь все запросы к базе данных будут работать с PostgreSQL. Например:

**Создание пользователя (POST):**

insertUser := `INSERT INTO users (name, email) VALUES ($1, $2);`

\_, err = db.Exec(insertUser, user.Name, user.Email)

**Получение пользователей (GET):**

rows, err := db.Query("SELECT id, name, email FROM users;")

**Обновление пользователя (PUT):**

updateUser := `UPDATE users SET name = $1, email = $2 WHERE id = $3;`

\_, err = db.Exec(updateUser, user.Name, user.Email, user.ID)

**Удаление пользователя (DELETE):**

deleteUser := `DELETE FROM users WHERE id = $1;`

\_, err = db.Exec(deleteUser, user.ID)

**Шаг 6. Запуск приложения**

1. Убедись, что PostgreSQL запущен.
2. Запусти сервер:
3. go run main.go

Если всё настроено правильно, сервер подключится к PostgreSQL, и ты сможешь работать с базой данных.

**Что дальше?**

1. **Docker** — упакуем приложение и PostgreSQL в контейнеры.
2. **Миграции базы данных** — автоматизируем создание и обновление таблиц.
3. **Тестирование API** — начнем писать тесты для проверки работы сервера.

Docker — это мощный инструмент для упаковки приложений и их зависимостей в контейнеры. С помощью Docker ты сможешь запускать своё приложение и базу данных PostgreSQL в изолированной среде, что упрощает развертывание и переносимость. 🚀

**Что мы сделаем?**

1. Создадим Dockerfile для упаковки Go-приложения.
2. Настроим PostgreSQL в Docker.
3. Используем Docker Compose для запуска приложения и базы данных вместе.

**Шаг 1. Создание Dockerfile для Go-приложения**

1. В корне проекта создай файл Dockerfile с таким содержимым:

# Используем официальный образ Go

FROM golang:1.20

# Устанавливаем рабочую директорию

WORKDIR /app

# Копируем файлы проекта в контейнер

COPY . .

# Загружаем зависимости

RUN go mod download

# Собираем приложение

RUN go build -o main .

# Указываем порт, на котором работает приложение

EXPOSE 8080

# Команда для запуска приложения

CMD ["./main"]

**Шаг 2. Настройка PostgreSQL в Docker**

1. Создай файл docker-compose.yml в корне проекта:

version: '3.8'

services:

app:

build:

context: .

dockerfile: Dockerfile

ports:

- "8080:8080"

environment:

POSTGRES\_URL: postgres://postgres:your\_password@db:5432/go\_backend?sslmode=disable

depends\_on:

- db

db:

image: postgres:14

container\_name: postgres\_container

environment:

POSTGRES\_USER: postgres

POSTGRES\_PASSWORD: your\_password

POSTGRES\_DB: go\_backend

ports:

- "5432:5432"

volumes:

- postgres\_data:/var/lib/postgresql/data

volumes:

postgres\_data:

Здесь:

* app — это твое Go-приложение.
* db — это PostgreSQL, настроенный в контейнере.
* Переменная POSTGRES\_URL указывает на базу данных, которая будет доступна по имени сервиса db.

**Шаг 3. Обновление .env**

Обнови файл .env, чтобы приложение использовало переменные окружения из Docker Compose:

POSTGRES\_URL=postgres://postgres:your\_password@db:5432/go\_backend?sslmode=disable

PORT=8080

**Шаг 4. Запуск приложения и базы данных**

1. Убедись, что в корне проекта находятся файлы Dockerfile и docker-compose.yml.
2. Собери и запусти контейнеры командой:
3. docker-compose up --build
4. Docker создаст два контейнера:
   * Один для твоего Go-приложения.
   * Второй для PostgreSQL.
5. Если всё настроено правильно, ты увидишь логи приложения и базы данных в терминале.

**Шаг 5. Проверка работы**

1. Открой браузер или Postman и отправь запросы к API на http://localhost:8080.
2. Убедись, что приложение взаимодействует с базой данных PostgreSQL в контейнере.

**Шаг 6. Остановка контейнеров**

Чтобы остановить контейнеры, используй команду:

docker-compose down

**Что дальше?**

1. **Миграции базы данных** — автоматизируем создание и обновление таблиц.
2. **Тестирование API** — напишем тесты для проверки работы сервера.
3. **Добавление Nginx** — настроим прокси-сервер для балансировки нагрузки.

Отлично, Миша! Ты уже настроил приложение с Docker, и это огромный шаг вперед. Теперь давай разберемся с **миграциями базы данных**. Это позволит тебе автоматизировать создание, обновление и управление таблицами в базе данных. 🚀

**Что такое миграции базы данных?**

Миграции — это способ управлять изменениями в структуре базы данных (например, добавление таблиц, изменение колонок) через код. Это особенно полезно, если приложение развивается, и структура базы данных меняется.

**Шаг 1. Установка инструмента для миграций**

Мы будем использовать популярный инструмент **golang-migrate**.

1. Установи golang-migrate:
2. go install -tags 'postgres' github.com/golang-migrate/migrate/v4/cmd/migrate@latest
3. Убедись, что migrate установлен, выполнив команду:
4. migrate -version

**Шаг 2. Создание папки для миграций**

1. В корне проекта создай папку migrations.
2. В этой папке будут храниться файлы миграций.

**Шаг 3. Создание миграций**

1. Создай файл миграции для создания таблицы users:
2. migrations/0001\_create\_users\_table.up.sql
3. Добавь в файл следующий SQL-код:
4. CREATE TABLE IF NOT EXISTS users (
5. id SERIAL PRIMARY KEY,
6. name TEXT NOT NULL,
7. email TEXT NOT NULL UNIQUE
8. );
9. Создай файл для отката миграции:
10. migrations/0001\_create\_users\_table.down.sql
11. Добавь в файл следующий SQL-код:
12. DROP TABLE IF EXISTS users;

**Шаг 4. Применение миграций**

1. Убедись, что PostgreSQL запущен (например, через Docker).
2. Выполни миграции с помощью команды migrate:
3. migrate -database postgres://postgres:your\_password@localhost:5432/go\_backend?sslmode=disable -path ./migrations up

Здесь:

* + -database — строка подключения к базе данных.
  + -path — путь к папке с миграциями.
  + up — применяет миграции.

1. Если нужно откатить миграции, используй команду:
2. migrate -database postgres://postgres:your\_password@localhost:5432/go\_backend?sslmode=disable -path ./migrations down

**Шаг 5. Интеграция миграций в приложение**

Чтобы миграции автоматически применялись при запуске приложения, добавь в main.go следующий код:

1. Установи библиотеку для работы с golang-migrate:
2. go get -u github.com/golang-migrate/migrate/v4
3. Добавь функцию для запуска миграций:

import (

"github.com/golang-migrate/migrate/v4"

"github.com/golang-migrate/migrate/v4/database/postgres"

\_ "github.com/golang-migrate/migrate/v4/source/file"

)

func runMigrations() {

driver, err := postgres.WithInstance(db, &postgres.Config{})

if err != nil {

log.Fatal("Ошибка создания драйвера миграций:", err)

}

m, err := migrate.NewWithDatabaseInstance(

"file://migrations",

"postgres", driver)

if err != nil {

log.Fatal("Ошибка создания миграций:", err)

}

err = m.Up()

if err != nil && err != migrate.ErrNoChange {

log.Fatal("Ошибка применения миграций:", err)

}

fmt.Println("Миграции успешно применены!")

}

1. Вызови runMigrations() после подключения к базе данных в initDB:

func initDB() {

var err error

connStr := os.Getenv("POSTGRES\_URL")

db, err = sql.Open("postgres", connStr)

if err != nil {

log.Fatal("Ошибка подключения к базе данных:", err)

}

err = db.Ping()

if err != nil {

log.Fatal("Ошибка проверки подключения:", err)

}

runMigrations()

}

**Шаг 6. Запуск приложения**

1. Убедись, что в папке migrations есть файлы миграций.
2. Запусти приложение:
3. go run main.go

Если всё настроено правильно, миграции применятся автоматически, и таблица users будет создана в базе

**Что теперь умеет твое приложение?**

1. При запуске оно подключается к базе данных PostgreSQL.
2. Автоматически применяет миграции из папки migrations.
3. Создает или обновляет структуру базы данных (например, таблицы).
4. Полностью готово к работе с API, которое ты уже реализовал.

**Что дальше?**

Теперь у тебя есть несколько направлений для развития:

1. **Тестирование API** — научимся писать тесты для проверки работы серверных маршрутов и взаимодействия с базой данных.
2. **Добавление новых сущностей** — например, создание таблиц для заказов, продуктов или других данных.
3. **Авторизация и аутентификация** — добавим регистрацию, вход и защиту API с помощью токенов (JWT).
4. **Интеграция с фронтендом** — научимся взаимодействовать с клиентской частью приложения.

Тестирование API — это важный этап разработки, который помогает убедиться, что сервер работает правильно, а изменения в коде не ломают существующий функционал. 🚀

**Что мы будем делать?**

1. Настроим тестовую базу данных.
2. Напишем тесты для основных маршрутов (GET, POST, PUT, DELETE).
3. Используем библиотеку net/http/httptest для тестирования API.

**Шаг 1. Настройка тестовой базы данных**

Для тестирования мы будем использовать отдельную базу данных, чтобы не изменять данные в основной базе.

1. Создай тестовую базу данных в PostgreSQL:
2. CREATE DATABASE go\_backend\_test;
3. Обнови файл .env:
4. TEST\_POSTGRES\_URL=postgres://postgres:your\_password@localhost:5432/go\_backend\_test?sslmode=disable
5. Добавь функцию для подключения к тестовой базе:

func initTestDB() \*sql.DB {

connStr := os.Getenv("TEST\_POSTGRES\_URL")

testDB, err := sql.Open("postgres", connStr)

if err != nil {

log.Fatal("Ошибка подключения к тестовой базе данных:", err)

}

err = testDB.Ping()

if err != nil {

log.Fatal("Ошибка проверки подключения к тестовой базе данных:", err)

}

// Применяем миграции для тестовой базы

driver, err := postgres.WithInstance(testDB, &postgres.Config{})

if err != nil {

log.Fatal("Ошибка создания драйвера миграций для тестовой базы:", err)

}

m, err := migrate.NewWithDatabaseInstance(

"file://migrations",

"postgres", driver)

if err != nil {

log.Fatal("Ошибка создания миграций для тестовой базы:", err)

}

err = m.Up()

if err != nil && err != migrate.ErrNoChange {

log.Fatal("Ошибка применения миграций для тестовой базы:", err)

}

fmt.Println("Тестовая база данных готова!")

return testDB

}

**Шаг 2. Установка библиотеки для тестирования**

1. Убедись, что у тебя установлен пакет testing (он встроен в Go).
2. Установи библиотеку для работы с утверждениями (assertions):
3. go get github.com/stretchr/testify

**Шаг 3. Написание тестов**

Создай файл main\_test.go в корне проекта. В этом файле мы будем писать тесты.

**Пример теста для POST-запроса (создание пользователя):**

package main

import (

"bytes"

"database/sql"

"encoding/json"

"net/http"

"net/http/httptest"

"os"

"testing"

"github.com/stretchr/testify/assert"

)

var testDB \*sql.DB

func TestMain(m \*testing.M) {

// Инициализация тестовой базы данных

os.Setenv("TEST\_POSTGRES\_URL", "postgres://postgres:your\_password@localhost:5432/go\_backend\_test?sslmode=disable")

testDB = initTestDB()

defer testDB.Close()

// Запускаем тесты

code := m.Run()

// Завершаем выполнение

os.Exit(code)

}

func TestCreateUser(t \*testing.T) {

// Тело запроса

user := map[string]string{

"name": "Миша",

"email": "misha@example.com",

}

body, \_ := json.Marshal(user)

// Создаем тестовый запрос

req, err := http.NewRequest("POST", "/users", bytes.NewBuffer(body))

if err != nil {

t.Fatal(err)

}

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

// Создаем тестовый HTTP-сервер

rr := httptest.NewRecorder()

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

db = testDB // Используем тестовую базу

createUserHandler(w, r)

})

handler.ServeHTTP(rr, req)

// Проверяем статус-код

assert.Equal(t, http.StatusCreated, rr.Code)

// Проверяем ответ

expected := "Пользователь Миша успешно создан!"

assert.Contains(t, rr.Body.String(), expected)

}

**Пример теста для GET-запроса (получение пользователей):**

func TestGetUsers(t \*testing.T) {

// Создаем тестовый запрос

req, err := http.NewRequest("GET", "/users", nil)

if err != nil {

t.Fatal(err)

}

// Создаем тестовый HTTP-сервер

rr := httptest.NewRecorder()

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

db = testDB // Используем тестовую базу

getUsersHandler(w, r)

})

handler.ServeHTTP(rr, req)

// Проверяем статус-код

assert.Equal(t, http.StatusOK, rr.Code)

// Проверяем, что в ответе содержится список пользователей

var users []struct {

ID int `json:"id"`

Name string `json:"name"`

Email string `json:"email"`

}

err = json.Unmarshal(rr.Body.Bytes(), &users)

if err != nil {

t.Fatal("Ошибка при разборе ответа:", err)

}

// Убедимся, что хотя бы один пользователь существует

assert.Greater(t, len(users), 0)

assert.Equal(t, "Миша", users[0].Name)

assert.Equal(t, "misha@example.com", users[0].Email)

}

**Шаг 4. Запуск тестов**

1. Запусти тесты командой:
2. go test
3. Если всё настроено правильно, ты увидишь, что тесты успешно проходят.

**Шаг 5. Что дальше?**

Теперь у тебя есть базовые тесты для API. Ты можешь:

1. Добавить тесты для PUT-запросов (обновление пользователя).
2. Добавить тесты для DELETE-запросов (удаление пользователя).
3. Проверить обработку ошибок (например, создание пользователя с уже существующим email).

**Пример теста для PUT-запроса (обновление пользователя):**

func TestUpdateUser(t \*testing.T) {

// Тело запроса для обновления пользователя

user := map[string]interface{}{

"id": 1,

"name": "Михаил",

"email": "mikhail@example.com",

}

body, \_ := json.Marshal(user)

// Создаем тестовый запрос

req, err := http.NewRequest("PUT", "/users/update", bytes.NewBuffer(body))

if err != nil {

t.Fatal(err)

}

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

// Создаем тестовый HTTP-сервер

rr := httptest.NewRecorder()

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

db = testDB // Используем тестовую базу

updateUserHandler(w, r)

})

handler.ServeHTTP(rr, req)

// Проверяем статус-код

assert.Equal(t, http.StatusOK, rr.Code)

// Проверяем ответ

expected := "Пользователь с ID 1 успешно обновлен!"

assert.Contains(t, rr.Body.String(), expected)

}

**Пример теста для DELETE-запроса (удаление пользователя):**

func TestDeleteUser(t \*testing.T) {

// Тело запроса для удаления пользователя

user := map[string]interface{}{

"id": 1,

}

body, \_ := json.Marshal(user)

// Создаем тестовый запрос

req, err := http.NewRequest("DELETE", "/users/delete", bytes.NewBuffer(body))

if err != nil {

t.Fatal(err)

}

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

// Создаем тестовый HTTP-сервер

rr := httptest.NewRecorder()

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

db = testDB // Используем тестовую базу

deleteUserHandler(w, r)

})

handler.ServeHTTP(rr, req)

// Проверяем статус-код

assert.Equal(t, http.StatusOK, rr.Code)

// Проверяем ответ

expected := "Пользователь с ID 1 успешно удален!"

assert.Contains(t, rr.Body.String(), expected)

}

**Итог**

Теперь у тебя есть тесты для всех CRUD-операций:

1. **POST** — создание пользователя.
2. **GET** — получение списка пользователей.
3. **PUT** — обновление данных пользователя.
4. **DELETE** — удаление пользователя.

Ты можешь запускать тесты после внесения изменений в код, чтобы убедиться, что всё работает корректно.

Если хочешь двигаться дальше, вот несколько идей:

1. **Авторизация и аутентификация** — добавим регистрацию, вход и защиту API с помощью JWT.
2. **Интеграция с фронтендом** — создадим клиентское приложение, которое взаимодействует с твоим API.
3. **Логирование и мониторинг** — добавим логи и метрики для отслеживания работы сервера.

Авторизация и аутентификация — это ключевые элементы любого защищенного приложения. Давай разберемся, как добавить регистрацию, вход и защиту API с помощью токенов JWT (JSON Web Token). 😊

**Что мы сделаем?**

1. Реализуем регистрацию пользователя.
2. Реализуем вход (аутентификацию) с выдачей JWT.
3. Добавим защиту маршрутов с использованием JWT.

**Шаг 1. Установка библиотеки для работы с JWT**

1. Установи библиотеку github.com/golang-jwt/jwt/v4:
2. go get github.com/golang-jwt/jwt/v4
3. Импортируй её в свой проект:
4. import "github.com/golang-jwt/jwt/v4"

**Шаг 2. Добавление регистрации пользователя**

Обнови main.go, чтобы добавить обработчик регистрации:

import (

"golang.org/x/crypto/bcrypt"

)

// Регистрация пользователя

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

if r.Method != "POST" {

http.Error(w, "Метод не поддерживается", http.StatusMethodNotAllowed)

return

}

var user struct {

Name string `json:"name"`

Email string `json:"email"`

Password string `json:"password"`

}

err := json.NewDecoder(r.Body).Decode(&user)

if err != nil {

http.Error(w, "Неверный формат данных", http.StatusBadRequest)

return

}

// Хэшируем пароль

hashedPassword, err := bcrypt.GenerateFromPassword([]byte(user.Password), bcrypt.DefaultCost)

if err != nil {

http.Error(w, "Ошибка при хэшировании пароля", http.StatusInternalServerError)

return

}

// Сохраняем пользователя в базе

insertUser := `INSERT INTO users (name, email, password) VALUES ($1, $2, $3);`

\_, err = db.Exec(insertUser, user.Name, user.Email, string(hashedPassword))

if err != nil {

http.Error(w, "Ошибка при регистрации пользователя", http.StatusInternalServerError)

return

}

w.WriteHeader(http.StatusCreated)

fmt.Fprintf(w, "Пользователь %s успешно зарегистрирован!", user.Name)

}

Добавь маршрут для регистрации:

http.HandleFunc("/register", registerHandler)

**Шаг 3. Реализация входа (аутентификации)**

Добавь обработчик для входа:

var jwtKey = []byte("my\_secret\_key") // Секретный ключ для подписи JWT

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

if r.Method != "POST" {

http.Error(w, "Метод не поддерживается", http.StatusMethodNotAllowed)

return

}

var credentials struct {

Email string `json:"email"`

Password string `json:"password"`

}

err := json.NewDecoder(r.Body).Decode(&credentials)

if err != nil {

http.Error(w, "Неверный формат данных", http.StatusBadRequest)

return

}

// Проверяем пользователя в базе

var id int

var name, hashedPassword string

query := `SELECT id, name, password FROM users WHERE email = $1;`

err = db.QueryRow(query, credentials.Email).Scan(&id, &name, &hashedPassword)

if err != nil {

http.Error(w, "Неверный email или пароль", http.StatusUnauthorized)

return

}

// Сравниваем хэш пароля

err = bcrypt.CompareHashAndPassword([]byte(hashedPassword), []byte(credentials.Password))

if err != nil {

http.Error(w, "Неверный email или пароль", http.StatusUnauthorized)

return

}

// Генерируем JWT

token := jwt.NewWithClaims(jwt.SigningMethodHS256, jwt.MapClaims{

"user\_id": id,

"name": name,

"exp": jwt.NewNumericDate(time.Now().Add(24 \* time.Hour)), // Токен действует 24 часа

})

tokenString, err := token.SignedString(jwtKey)

if err != nil {

http.Error(w, "Ошибка при генерации токена", http.StatusInternalServerError)

return

}

// Отправляем токен клиенту

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

json.NewEncoder(w).Encode(map[string]string{

"token": tokenString,

})

}

Добавь маршрут для входа:

http.HandleFunc("/login", loginHandler)

**Шаг 4. Защита маршрутов с использованием JWT**

Это позволит ограничить доступ к определенным маршрутам только для авторизованных пользователей.

Добавим middleware (промежуточный обработчик), который будет проверять JWT-токен перед выполнением защищенного маршрута.

**Middleware для проверки токена**

Добавь следующий код в main.go:

func authMiddleware(next http.HandlerFunc) http.HandlerFunc {

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

// Получаем токен из заголовка Authorization

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

if tokenString == "" {

http.Error(w, "Токен отсутствует", http.StatusUnauthorized)

return

}

// Убираем "Bearer " из токена, если он передан в формате "Bearer <token>"

if len(tokenString) > 7 && tokenString[:7] == "Bearer " {

tokenString = tokenString[7:]

}

// Проверяем токен

claims := jwt.MapClaims{}

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

return jwtKey, nil

})

if err != nil || !token.Valid {

http.Error(w, "Неверный или истекший токен", http.StatusUnauthorized)

return

}

// Сохраняем данные из токена в контексте запроса (например, user\_id)

r = r.WithContext(context.WithValue(r.Context(), "user\_id", claims["user\_id"]))

r = r.WithContext(context.WithValue(r.Context(), "name", claims["name"]))

// Передаем управление следующему обработчику

next(w, r)

}

}

**Пример защищенного маршрута**

Добавим защищенный маршрут, который возвращает данные только для авторизованных пользователей:

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

// Получаем данные из контекста

userID := r.Context().Value("user\_id")

name := r.Context().Value("name")

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

json.NewEncoder(w).Encode(map[string]interface{}{

"user\_id": userID,

"name": name,

"message": "Это защищенный маршрут. Только для авторизованных пользователей!",

})

}

Добавь маршрут с использованием middleware:

http.HandleFunc("/protected", authMiddleware(protectedHandler))

**Шаг 5. Тестирование**

1. Зарегистрируй нового пользователя через маршрут /register.
2. Выполни вход через маршрут /login, чтобы получить JWT-токен.
3. Отправь GET-запрос на защищенный маршрут /protected, добавив токен в заголовок Authorization:
4. Authorization: Bearer <твой\_токен>

Если всё настроено правильно, ты получишь ответ:

{

"user\_id": 1,

"name": "Миша",

"message": "Это защищенный маршрут. Только для авторизованных пользователей!"

}

Если токен отсутствует или неверный, сервер вернет ошибку 401 Unauthorized.

**Что теперь умеет твое приложение?**

1. **Регистрация пользователей** — с хэшированием паролей.
2. **Вход пользователей** — с выдачей JWT-токена.
3. **Защита маршрутов** — доступ к определенным маршрутам только для авторизованных пользователей.

**Что дальше?**

1. **Рефреш токенов** — добавим возможность обновления токенов, если они истекли.
2. **Роли и права доступа** — добавим роли (например, админ, пользователь) и ограничим доступ к маршрутам в зависимости от роли.
3. **Интеграция с фронтендом** — создадим клиентское приложение, которое будет взаимодействовать с API.

Теперь, когда у нас есть базовая авторизация с JWT, давай добавим **рефреш токены** и разберем, как обновлять токены, если они истекли. Это важный шаг для обеспечения удобства и безопасности пользователей. 😊

**Что такое рефреш токены?**

Рефреш токен — это специальный токен, который используется для получения нового JWT, когда старый истек. Это позволяет пользователям оставаться авторизованными, не выполняя вход заново.

* **JWT-токен**: Краткосрочный, используется для доступа к защищенным маршрутам.
* **Рефреш токен**: Долгосрочный, используется только для получения нового JWT.

**Шаг 1. Добавление таблицы для рефреш токенов**

Создай новую миграцию для таблицы refresh\_tokens:

**Файл: migrations/0002\_create\_refresh\_tokens\_table.up.sql**

CREATE TABLE IF NOT EXISTS refresh\_tokens (

id SERIAL PRIMARY KEY,

user\_id INTEGER NOT NULL,

token TEXT NOT NULL UNIQUE,

expires\_at TIMESTAMP NOT NULL,

FOREIGN KEY (user\_id) REFERENCES users (id) ON DELETE CASCADE

);

**Файл: migrations/0002\_create\_refresh\_tokens\_table.down.sql**

DROP TABLE IF EXISTS refresh\_tokens;

Примените миграцию:

migrate -database postgres://postgres:your\_password@localhost:5432/go\_backend?sslmode=disable -path ./migrations up

**Шаг 2. Генерация рефреш токенов**

Добавь функцию для создания рефреш токена:

import (

"crypto/rand"

"encoding/base64"

"time"

)

// Генерация случайного токена

func generateRefreshToken() (string, error) {

bytes := make([]byte, 32)

\_, err := rand.Read(bytes)

if err != nil {

return "", err

}

return base64.URLEncoding.EncodeToString(bytes), nil

}

**Шаг 3. Обновление входа (выдача рефреш токена)**

Обнови функцию loginHandler, чтобы она выдавала рефреш токен:

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

// ... (код проверки email и пароля остаётся без изменений)

// Генерируем JWT

token := jwt.NewWithClaims(jwt.SigningMethodHS256, jwt.MapClaims{

"user\_id": id,

"name": name,

"exp": jwt.NewNumericDate(time.Now().Add(15 \* time.Minute)), // JWT действует 15 минут

})

tokenString, err := token.SignedString(jwtKey)

if err != nil {

http.Error(w, "Ошибка при генерации токена", http.StatusInternalServerError)

return

}

// Генерируем рефреш токен

refreshToken, err := generateRefreshToken()

if err != nil {

http.Error(w, "Ошибка при генерации рефреш токена", http.StatusInternalServerError)

return

}

// Сохраняем рефреш токен в базе

expiresAt := time.Now().Add(7 \* 24 \* time.Hour) // Рефреш токен действует 7 дней

insertRefreshToken := `INSERT INTO refresh\_tokens (user\_id, token, expires\_at) VALUES ($1, $2, $3);`

\_, err = db.Exec(insertRefreshToken, id, refreshToken, expiresAt)

if err != nil {

http.Error(w, "Ошибка при сохранении рефреш токена", http.StatusInternalServerError)

return

}

// Отправляем токены клиенту

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

json.NewEncoder(w).Encode(map[string]string{

"token": tokenString,

"refresh\_token": refreshToken,

})

}

**Шаг 4. Обновление JWT через рефреш токен**

Добавь обработчик для обновления JWT:

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

if r.Method != "POST" {

http.Error(w, "Метод не поддерживается", http.StatusMethodNotAllowed)

return

}

var body struct {

RefreshToken string `json:"refresh\_token"`

}

err := json.NewDecoder(r.Body).Decode(&body)

if err != nil {

http.Error(w, "Неверный формат данных", http.StatusBadRequest)

return

}

// Проверяем рефреш токен в базе

var userID int

var expiresAt time.Time

query := `SELECT user\_id, expires\_at FROM refresh\_tokens WHERE token = $1;`

err = db.QueryRow(query, body.RefreshToken).Scan(&userID, &expiresAt)

if err != nil {

http.Error(w, "Неверный рефреш токен", http.StatusUnauthorized)

return

}

// Проверяем, не истек ли рефреш токен

if time.Now().After(expiresAt) {

http.Error(w, "Рефреш токен истек", http.StatusUnauthorized)

return

}

// Генерируем новый JWT

token := jwt.NewWithClaims(jwt.SigningMethodHS256, jwt.MapClaims{

"user\_id": userID,

"exp": jwt.NewNumericDate(time.Now().Add(15 \* time.Minute)), // Новый JWT действует 15 минут

})

tokenString, err := token.SignedString(jwtKey)

if err != nil {

http.Error(w, "Ошибка при генерации нового токена", http.StatusInternalServerError)

return

}

// Отправляем новый JWT клиенту

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

json.NewEncoder(w).Encode(map[string]string{

"token": tokenString,

})

}

**Шаг 5. Добавление маршрута для обновления токена**

Добавь новый маршрут в main.go:

http.HandleFunc("/refresh", refreshHandler)

**Шаг 6. Тестирование рефреш токенов**

Выполни вход через маршрут /login, чтобы получить JWT и рефреш токен.

Отправь POST-запрос на маршрут /refresh, используя рефреш токен:

POST http://localhost:8080/refresh

Content-Type: application/json

{

"refresh\_token": "<твой\_рефреш\_токен>"

}

Если всё настроено правильно, сервер вернет новый JWT:

{

"token": "<новый\_JWT>"

}

Если рефреш токен истек или неверный, сервер вернет ошибку 401 Unauthorized.

**Что теперь умеет твое приложение?**

1. **Регистрация пользователей** — с хэшированием паролей.
2. **Вход пользователей** — с выдачей JWT и рефреш токена.
3. **Обновление токенов** — через рефреш токен.
4. **Защита маршрутов** — доступ к защищенным маршрутам только с валидным JWT.

**Что дальше?**

1. **Роли и права доступа** — добавим роли (например, админ, пользователь) и ограничим доступ к маршрутам в зависимости от роли.
2. **Логаут (удаление рефреш токена)** — реализуем выход из системы.
3. **Интеграция с фронтендом** — создадим клиентское приложение, которое будет взаимодействовать с твоим API.
4. **Мониторинг и логирование** — добавим логи и метрики для отслеживания работы сервера.

Добавление ролей и прав доступа — это важный шаг для управления доступом к различным маршрутам в приложении. Например, администратор может иметь доступ к управлению пользователями, а обычный пользователь — только к своим данным.

**Что мы сделаем?**

1. Добавим роли (например, admin и user) в базу данных.
2. Реализуем проверку ролей для маршрутов.
3. Ограничим доступ к определенным маршрутам в зависимости от роли пользователя.

**Шаг 1. Добавление роли в таблицу users**

Обнови структуру таблицы users, добавив колонку role:

**Файл: migrations/0003\_add\_role\_to\_users.up.sql**

ALTER TABLE users ADD COLUMN role TEXT NOT NULL DEFAULT 'user';

**Файл: migrations/0003\_add\_role\_to\_users.down.sql**

ALTER TABLE users DROP COLUMN role;

Примените миграцию:

migrate -database postgres://postgres:your\_password@localhost:5432/go\_backend?sslmode=disable -path ./migrations up

Теперь у каждого пользователя будет роль, которая по умолчанию равна user.

**Шаг 2. Обновление регистрации**

Обнови обработчик регистрации, чтобы задавать роль (по умолчанию user):

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

if r.Method != "POST" {

http.Error(w, "Метод не поддерживается", http.StatusMethodNotAllowed)

return

}

var user struct {

Name string `json:"name"`

Email string `json:"email"`

Password string `json:"password"`

Role string `json:"role"` // Новое поле для роли

}

err := json.NewDecoder(r.Body).Decode(&user)

if err != nil {

http.Error(w, "Неверный формат данных", http.StatusBadRequest)

return

}

// Если роль не указана, устанавливаем её как "user"

if user.Role == "" {

user.Role = "user"

}

// Хэшируем пароль

hashedPassword, err := bcrypt.GenerateFromPassword([]byte(user.Password), bcrypt.DefaultCost)

if err != nil {

http.Error(w, "Ошибка при хэшировании пароля", http.StatusInternalServerError)

return

}

// Сохраняем пользователя в базе

insertUser := `INSERT INTO users (name, email, password, role) VALUES ($1, $2, $3, $4);`

\_, err = db.Exec(insertUser, user.Name, user.Email, string(hashedPassword), user.Role)

if err != nil {

http.Error(w, "Ошибка при регистрации пользователя", http.StatusInternalServerError)

return

}

w.WriteHeader(http.StatusCreated)

fmt.Fprintf(w, "Пользователь %s успешно зарегистрирован с ролью %s!", user.Name, user.Role)

}

Теперь при регистрации можно задавать роль, например:

* user — обычный пользователь.
* admin — администратор.

**Шаг 3. Добавление роли в JWT**

Обнови обработчик входа, чтобы включить роль в токен:

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

// ... (код проверки email и пароля остаётся без изменений)

// Добавляем роль в JWT

var role string

query := `SELECT id, name, password, role FROM users WHERE email = $1;`

err = db.QueryRow(query, credentials.Email).Scan(&id, &name, &hashedPassword, &role)

if err != nil {

http.Error(w, "Неверный email или пароль", http.StatusUnauthorized)

return

}

token := jwt.NewWithClaims(jwt.SigningMethodHS256, jwt.MapClaims{

"user\_id": id,

"name": name,

"role": role, // Добавляем роль в токен

"exp": jwt.NewNumericDate(time.Now().Add(15 \* time.Minute)),

})

tokenString, err := token.SignedString(jwtKey)

if err != nil {

http.Error(w, "Ошибка при генерации токена", http.StatusInternalServerError)

return

}

// Отправляем токен клиенту

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

json.NewEncoder(w).Encode(map[string]string{

"token": tokenString,

})

}

Теперь токен будет содержать информацию о роли пользователя.

**Шаг 4. Проверка ролей**

Обнови middleware для проверки JWT, чтобы проверять, есть ли у пользователя требуемая роль для доступа к маршруту::

func authMiddleware(requiredRole string, next http.HandlerFunc) http.HandlerFunc {

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

// Получаем токен из заголовка Authorization

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

if tokenString == "" {

http.Error(w, "Токен отсутствует", http.StatusUnauthorized)

return

}

// Убираем "Bearer " из токена, если он передан в формате "Bearer <token>"

if len(tokenString) > 7 && tokenString[:7] == "Bearer " {

tokenString = tokenString[7:]

}

// Проверяем токен

claims := jwt.MapClaims{}

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

return jwtKey, nil

})

if err != nil || !token.Valid {

http.Error(w, "Неверный или истекший токен", http.StatusUnauthorized)

return

}

// Проверяем роль пользователя

role, ok := claims["role"].(string)

if !ok || (requiredRole != "" && role != requiredRole) {

http.Error(w, "Доступ запрещен: недостаточно прав", http.StatusForbidden)

return

}

// Сохраняем данные из токена в контексте запроса

r = r.WithContext(context.WithValue(r.Context(), "user\_id", claims["user\_id"]))

r = r.WithContext(context.WithValue(r.Context(), "name", claims["name"]))

r = r.WithContext(context.WithValue(r.Context(), "role", role))

// Передаем управление следующему обработчику

next(w, r)

}

}

**Шаг 5. Пример защищенного маршрута с проверкой роли**

Добавим маршрут, доступный только для администраторов:

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

// Получаем данные из контекста

userID := r.Context().Value("user\_id")

name := r.Context().Value("name")

role := r.Context().Value("role")

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

json.NewEncoder(w).Encode(map[string]interface{}{

"user\_id": userID,

"name": name,

"role": role,

"message": "Добро пожаловать в административную панель!",

})

}

Добавь маршрут с использованием middleware:

http.HandleFunc("/admin", authMiddleware("admin", adminHandler))

Теперь только пользователи с ролью admin смогут получить доступ к маршруту /admin. Если пользователь с ролью user попытается получить доступ, сервер вернет ошибку 403 Forbidden.

**Шаг 6. Пример защищенного маршрута для всех авторизованных пользователей**

Добавим маршрут, доступный для всех авторизованных пользователей, независимо от их роли:

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

// Получаем данные из контекста

userID := r.Context().Value("user\_id")

name := r.Context().Value("name")

role := r.Context().Value("role")

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

json.NewEncoder(w).Encode(map[string]interface{}{

"user\_id": userID,

"name": name,

"role": role,

"message": "Добро пожаловать, авторизованный пользователь!",

})

}

Добавь маршрут с использованием middleware, без проверки роли:

http.HandleFunc("/user", authMiddleware("", userHandler))

Здесь requiredRole передается пустым, что означает, что маршрут доступен для всех авторизованных пользователей.

**Шаг 7. Тестирование**

1. Зарегистрируй двух пользователей:
   * Администратора с ролью admin.
   * Обычного пользователя с ролью user.
2. Выполни вход для каждого пользователя, чтобы получить их JWT.
3. Тестируй маршруты:
   * Отправь GET-запрос на /admin с токеном администратора. Доступ должен быть разрешен.
   * Отправь GET-запрос на /admin с токеном пользователя. Доступ должен быть запрещен (403 Forbidden).
   * Отправь GET-запрос на /user с токеном любого авторизованного пользователя (как с ролью admin, так и с ролью user). Доступ должен быть разрешен.
   * Ответ сервера должен содержать данные пользователя и сообщение, например:
   * {
   * "user\_id": 1,
   * "name": "Миша",
   * "role": "user",
   * "message": "Добро пожаловать, авторизованный пользователь!"
   * }
4. **Проверь поведение при отсутствии токена:**
   * Отправь запрос на /admin или /user без заголовка Authorization.
   * Сервер должен вернуть ошибку 401 Unauthorized с сообщением: **"Токен отсутствует"**.
5. **Проверь поведение при использовании неверного токена:**
   * Отправь запрос с поддельным или истекшим токеном.
   * Сервер должен вернуть ошибку 401 Unauthorized с сообщением: **"Неверный или истекший токен"**.

**Что теперь умеет твое приложение?**

1. **Роли пользователей:**
   * У каждого пользователя есть роль (user или admin).
   * Роль задается при регистрации и хранится в базе данных.
2. **Защита маршрутов:**
   * Доступ к маршруту /admin разрешен только для пользователей с ролью admin.
   * Доступ к маршруту /user разрешен для всех авторизованных пользователей.
3. **Проверка токенов:**
   * Токены проверяются на валидность и истечение срока действия.
   * Если токен отсутствует или недействителен, доступ запрещен.

**Что дальше?**

Теперь ты можешь развивать свое приложение дальше. Вот несколько идей:

1. **Логаут (удаление рефреш токена):**
   * Реализуй возможность выхода из системы, чтобы удалять рефреш токен из базы данных.
2. **Роли с более сложной логикой:**
   * Добавь больше ролей (например, moderator, editor) и настрой маршруты для них.
3. **Интеграция с фронтендом:**
   * Создай клиентское приложение (например, на React или Vue.js), которое будет взаимодействовать с этим API.
4. **Мониторинг и логирование:**
   * Добавь логи для отслеживания запросов и ошибок.
   * Настрой мониторинг с помощью инструментов вроде Prometheus или Grafana.
5. **Тестирование ролей:**
   * Напиши тесты для проверки доступа к маршрутам в зависимости от роли.

Реализация **логаута** — это важная часть системы авторизации. Она позволяет пользователю выйти из системы и сделать рефреш токен недействительным, чтобы предотвратить его дальнейшее использование.

**Что мы сделаем?**

1. Добавим обработчик для логаута.
2. Реализуем удаление рефреш токена из базы данных.
3. Убедимся, что пользователь больше не может обновить JWT после логаута.

**Шаг 1. Обработчик для логаута**

Добавим новый обработчик, который будет удалять рефреш токен из базы данных:

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

if r.Method != "POST" {

http.Error(w, "Метод не поддерживается", http.StatusMethodNotAllowed)

return

}

var body struct {

RefreshToken string `json:"refresh\_token"`

}

err := json.NewDecoder(r.Body).Decode(&body)

if err != nil {

http.Error(w, "Неверный формат данных", http.StatusBadRequest)

return

}

// Удаляем рефреш токен из базы

deleteTokenQuery := `DELETE FROM refresh\_tokens WHERE token = $1;`

result, err := db.Exec(deleteTokenQuery, body.RefreshToken)

if err != nil {

http.Error(w, "Ошибка при удалении токена", http.StatusInternalServerError)

return

}

rowsAffected, \_ := result.RowsAffected()

if rowsAffected == 0 {

http.Error(w, "Рефреш токен не найден", http.StatusUnauthorized)

return

}

w.WriteHeader(http.StatusOK)

fmt.Fprintf(w, "Вы успешно вышли из системы.")

}

**Шаг 2. Добавление маршрута для логаута**

Добавь маршрут в main.go:

http.HandleFunc("/logout", logoutHandler)

**Шаг 3. Тестирование логаута**

1. Выполни вход через маршрут /login, чтобы получить JWT и рефреш токен.
2. Отправь POST-запрос на маршрут /logout, передав рефреш токен:
3. POST http://localhost:8080/logout
4. Content-Type: application/json
5. {
6. "refresh\_token": "<твой\_рефреш\_токен>"
7. }
8. Если всё настроено правильно, сервер вернет сообщение:
9. Вы успешно вышли из системы.
10. Попробуй обновить JWT с помощью удаленного рефреш токена через маршрут /refresh. Сервер должен вернуть ошибку 401 Unauthorized с сообщением: **"Неверный рефреш токен"**.

**Шаг 4. Дополнительная защита**

Чтобы усилить безопасность, можно добавить логику для удаления всех активных рефреш токенов пользователя при логауте. Например, если пользователь выходит из системы на одном устройстве, это завершит сессию на всех устройствах.

Обнови запрос для удаления всех токенов пользователя:

deleteAllTokensQuery := `DELETE FROM refresh\_tokens WHERE user\_id = $1;`

\_, err = db.Exec(deleteAllTokensQuery, userID)

if err != nil {

http.Error(w, "Ошибка при удалении токенов", http.StatusInternalServerError)

return

}

Для этого нужно получить user\_id из JWT или из базы данных.

**Что теперь умеет твое приложение?**

1. **Логаут:**
   * Пользователь может выйти из системы, удалив рефреш токен.
   * После логаута рефреш токен становится недействительным.
2. **Защита сессий:**
   * Можно реализовать логаут на всех устройствах, удаляя все активные токены пользователя.

**Что дальше?**

1. **Роли с более сложной логикой:** Настроим маршруты для разных ролей (например, moderator, editor).
2. **Интеграция с фронтендом:** Создадим клиентское приложение, которое будет взаимодействовать с этим API.
3. **Мониторинг и логирование:** Добавим логи для отслеживания запросов и ошибок.
4. **Тестирование логаута:** Напишем тесты для проверки работы логаута.