МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ

Ордена Трудового Красного Знамени федеральное государственное бюджетное образовательное учреждение высшего образования

**«Московский технический университет связи и информатики»**

Кафедра «**Математическая кибернетика и информационные технологии**»

Дисциплина «Высокоуровневые языки программирования»

Отчет по лабораторной работе №4

Выполнил

студент группы БВТ2204

Илянин В. А.

Москва, 2025 г.

## **1. Цели и задачи**

**Цель:**Разработать сервис авторизации на языке Go, который обеспечивает:

1. Регистрацию пользователя (с сохранением данных в базе данных и хешированным паролем).
2. Авторизацию пользователя по электронной почте и паролю (с выдачей JWT-токена).
3. Получение данных пользователя по его id, при условии действующего JWT-токена в заголовке Authorization.

**Задачи:**

1. Подготовить и сконфигурировать базу данных PostgreSQL.
2. Создать таблицу **customers** со столбцами (id, first\_name, last\_name, birth\_date, email, password).
3. Подключиться к базе данных из приложения на Go.
4. Реализовать **POST /register** для регистрации пользователя с проверкой уникальности email и хешированием пароля алгоритмом bcrypt.
5. Реализовать **POST /login** для авторизации, проверки пароля и выдачи JWT-токена.
6. Реализовать **GET /customers/:id** для получения данных конкретного пользователя по id (требуется JWT).
7. Добавить middleware, чтобы маршруты, требующие авторизации, автоматически проверяли JWT-токен.

## **2. Задание (требования по работе)**

В соответствии с заданием из методических материалов необходимо:

1. **Создать и настроить СУБД PostgreSQL.**
2. **Подготовить таблицу customers** в БД auth\_service\_db.
3. **Развернуть Go-проект**, подключить к нему PostgreSQL через драйвер, указав строку соединения.
4. **Реализовать REST-запросы** (регистрация, авторизация, получение данных пользователя).
5. **Использовать JWT** для авторизации и **bcrypt** для хеширования паролей.
6. **Проверять токен** при обращении к маршрутам, требующим авторизации.

## 

## **3. Выполнение**

В данном разделе приведено пошаговое описание реализации. Ниже после каждого блока кода можно вставлять соответствующие скриншоты.

### **3.1 Создание и подключение к базе данных**

1. **Установка PostgreSQL** (если ещё не установлена) пропускается или описывается кратко.
2. **Создание базы данных** (в терминале, подключившись к PostgreSQL под пользователем postgres):

psql -U postgres -p 5433

CREATE DATABASE auth\_service\_db;

Здесь мы поднимаем бд на порте 5433.

1. **Проверка создания**:

\l

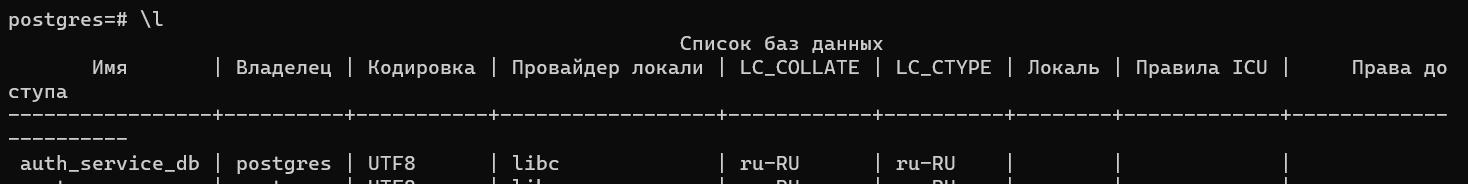


Рисунок 1 - Проверка успешного создания БД

(Среди баз данных должна появиться auth\_service\_db)

1. **Подключение к базе**:

\c auth\_service\_db

  
Рисунок 2 - Подключение к БД

1. **Создание таблицы customers** (пример SQL-запроса):

CREATE TABLE customers (

id SERIAL PRIMARY KEY,

first\_name VARCHAR(50),

last\_name VARCHAR(50),

birth\_date DATE,

email VARCHAR(100) UNIQUE NOT NULL,

password TEXT NOT NULL

);

Из рисунка 2 понятно, что таблица создана корректно.

### **3.2 Структура проекта**

Для удобства мы используем **MVC-подобную** структуру с папками:

* handlers (обработчики запросов),
* middleware (middleware-функции для авторизации),
* repository (работа с БД),
* файлы db.go, main.go и т. д.

Ниже структура файлов (скриншот уже прилагается):

handlers/

user\_handlers.go

middleware/

auth\_middleware.go

repository/

repository.go

db.go

main.go

go.mod

go.sum

### **3.3 Файл main.go**

package main

import (

"log"

"github.com/gin-gonic/gin"

// Наши внутренние пакеты

"github.com/qqMartyny/go-auth-service/handlers"

"github.com/qqMartyny/go-auth-service/middleware"

)

func main() {

// Инициализируем базу данных

db, err := initDB()

if err != nil {

log.Fatalf("Ошибка подключения к БД: %v\n", err)

}

defer db.Close()

// Инициализируем роутер Gin

r := gin.Default()

// Регистрация и логин — без авторизации

r.POST("/register", handlers.RegisterHandler(db))

r.POST("/login", handlers.LoginHandler(db))

// Группа маршрутов с авторизацией

authGroup := r.Group("/")

authGroup.Use(middleware.AuthRequired)

{

// Пример защищённого маршрута: GET /customers/:id

authGroup.GET("/customers/:id", handlers.GetCustomerHandler(db))

}

// Запуск сервера

if err := r.Run(":8080"); err != nil {

log.Fatalf("Ошибка запуска сервера: %v\n", err)

}

}

**Пояснение:**

* В main создаём соединение с БД (через initDB()), инициализируем Gin-роутер.
* Без аутентификации доступны POST /register и POST /login.
* Все остальные маршруты (в примере GET /customers/:id) помещены в группу authGroup и защищены с помощью middleware-функции AuthRequired.

### **3.4 Файл db.go**

package main

import (

"database/sql"

"fmt"

\_ "github.com/lib/pq" // Импортируем драйвер PostgreSQL

)

// initDB инициализирует подключение к PostgreSQL.

func initDB() (\*sql.DB, error) {

connStr := "host=localhost port=5433 user=postgres password=1209348756 dbname=auth\_service\_db sslmode=disable"

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

if err != nil {

return nil, fmt.Errorf("не удалось открыть соединение с БД: %w", err)

}

if err := db.Ping(); err != nil {

return nil, fmt.Errorf("не удалось установить соединение с БД: %w", err)

}

fmt.Println("Успешное подключение к БД!")

return db, nil

}

**Пояснение:**

* Подключаемся к локальной БД PostgreSQL на порте 5433.
* Проверяем соединение методом db.Ping().
* Если всё хорошо, возвращаем объект \*sql.DB.

### **3.5 Файл repository.go (пакет repository)**

package repository

import (

"database/sql"

"errors"

"time"

)

// Customer — модель для таблицы customers

type Customer struct {

ID int

FirstName string

LastName string

BirthDate time.Time

Email string

Password string // хранит bcrypt-хеш

}

// FindCustomerByEmail возвращает запись о покупателе, если email существует

func FindCustomerByEmail(db \*sql.DB, email string) (\*Customer, error) {

query := `SELECT id, first\_name, last\_name, birth\_date, email, password

FROM customers

WHERE email=$1`

var c Customer

err := db.QueryRow(query, email).Scan(

&c.ID, &c.FirstName, &c.LastName, &c.BirthDate, &c.Email, &c.Password,

)

if err != nil {

if errors.Is(err, sql.ErrNoRows) {

return nil, nil // не нашли пользователя

}

return nil, err

}

return &c, nil

}

// InsertCustomer вставляет новую запись

func InsertCustomer(db \*sql.DB, c \*Customer) error {

query := `INSERT INTO customers (first\_name, last\_name, birth\_date, email, password)

VALUES ($1, $2, $3, $4, $5)`

\_, err := db.Exec(query, c.FirstName, c.LastName, c.BirthDate, c.Email, c.Password)

return err

}

// FindCustomerByID ищет пользователя по ID

func FindCustomerByID(db \*sql.DB, id int) (\*Customer, error) {

query := `SELECT id, first\_name, last\_name, birth\_date, email, password

FROM customers

WHERE id=$1`

var c Customer

err := db.QueryRow(query, id).Scan(

&c.ID, &c.FirstName, &c.LastName, &c.BirthDate, &c.Email, &c.Password,

)

if err != nil {

if errors.Is(err, sql.ErrNoRows) {

return nil, nil

}

return nil, err

}

return &c, nil

}

**Пояснение:**

* Определяем структуру Customer.
* Функция FindCustomerByEmail ищет пользователя в таблице customers по email.
* Функция InsertCustomer добавляет нового пользователя.
* Функция FindCustomerByID ищет пользователя по id.

### **3.6 Файл user\_handlers.go (пакет handlers)**

package handlers

import (

"database/sql"

"fmt"

"net/http"

"strconv"

"time"

"github.com/gin-gonic/gin"

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

"golang.org/x/crypto/bcrypt"

// Импортируем наш репозиторий и основную "модель" (если нужно)

"github.com/qqMartyny/go-auth-service/repository"

)

// Пример секретного ключа

var secretKey = []byte("your-256-bit-secret")

// ==== РЕГИСТРАЦИЯ (POST /register) ====

func RegisterHandler(db \*sql.DB) gin.HandlerFunc {

return func(c \*gin.Context) {

type request struct {

FirstName string `json:"first\_name"`

LastName string `json:"last\_name"`

BirthDate string `json:"birth\_date"` // в формате "YYYY-MM-DD"

Email string `json:"email"`

Password string `json:"password"`

}

var req request

if err := c.ShouldBindJSON(&req); err != nil {

c.JSON(http.StatusBadRequest, gin.H{"error": "Неверный формат входных данных"})

return

}

// Проверяем, нет ли пользователя с таким email

existing, err := repository.FindCustomerByEmail(db, req.Email)

if err != nil {

c.JSON(http.StatusInternalServerError, gin.H{"error": "Ошибка базы данных"})

return

}

if existing != nil {

c.JSON(http.StatusConflict, gin.H{"error": "Пользователь с таким email уже существует"})

return

}

// Преобразуем дату (если нужно)

var birthDate time.Time

if req.BirthDate != "" {

parsedDate, err := time.Parse("2006-01-02", req.BirthDate)

if err != nil {

c.JSON(http.StatusBadRequest, gin.H{"error": "Некорректная дата рождения"})

return

}

birthDate = parsedDate

}

// Хешируем пароль

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

if err != nil {

c.JSON(http.StatusInternalServerError, gin.H{"error": "Ошибка хеширования пароля"})

return

}

// Сохраняем нового пользователя

newCustomer := &repository.Customer{

FirstName: req.FirstName,

LastName: req.LastName,

BirthDate: birthDate,

Email: req.Email,

Password: string(hashedPassword),

}

if err := repository.InsertCustomer(db, newCustomer); err != nil {

c.JSON(http.StatusInternalServerError, gin.H{"error": "Ошибка при создании пользователя"})

return

}

c.JSON(http.StatusCreated, gin.H{

"message": "Пользователь успешно зарегистрирован",

})

}

}

// ==== ЛОГИН (POST /login) ====

func LoginHandler(db \*sql.DB) gin.HandlerFunc {

return func(c \*gin.Context) {

type loginRequest struct {

Email string `json:"email"`

Password string `json:"password"`

}

var req loginRequest

if err := c.ShouldBindJSON(&req); err != nil {

c.JSON(http.StatusBadRequest, gin.H{"error": "Неверный формат входных данных"})

return

}

// Ищем пользователя по email

user, err := repository.FindCustomerByEmail(db, req.Email)

if err != nil {

c.JSON(http.StatusInternalServerError, gin.H{"error": "Ошибка базы данных"})

return

}

if user == nil {

// Нет такого email

c.JSON(http.StatusUnauthorized, gin.H{"error": "Неверный email или пароль"})

return

}

// Сравниваем хеши

err = bcrypt.CompareHashAndPassword([]byte(user.Password), []byte(req.Password))

if err != nil {

c.JSON(http.StatusUnauthorized, gin.H{"error": "Неверный email или пароль"})

return

}

// Генерируем JWT

claims := jwt.RegisteredClaims{

Subject: fmt.Sprintf("%d", user.ID), // ID в виде строки

ExpiresAt: jwt.NewNumericDate(time.Now().Add(time.Hour \* 24)), // Срок действия 24 часа

}

token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)

tokenStr, err := token.SignedString(secretKey)

if err != nil {

c.JSON(http.StatusInternalServerError, gin.H{"error": "Ошибка генерации токена"})

return

}

c.JSON(http.StatusOK, gin.H{"token": tokenStr})

}

}

// ==== ПОЛУЧЕНИЕ ДАННЫХ О ПОЛЬЗОВАТЕЛЕ (GET /customers/:id) ====

func GetCustomerHandler(db \*sql.DB) gin.HandlerFunc {

return func(c \*gin.Context) {

// ID из URL (строка)

idStr := c.Param("id")

id, err := strconv.Atoi(idStr)

if err != nil {

c.JSON(http.StatusBadRequest, gin.H{"error": "Неверный ID"})

return

}

// Ищем в базе

user, err := repository.FindCustomerByID(db, id)

if err != nil {

c.JSON(http.StatusInternalServerError, gin.H{"error": "Ошибка базы данных"})

return

}

if user == nil {

c.JSON(http.StatusNotFound, gin.H{"error": "Пользователь не найден"})

return

}

// Возвращаем данные (без пароля)

resp := gin.H{

"id": user.ID,

"first\_name": user.FirstName,

"last\_name": user.LastName,

"birth\_date": user.BirthDate.Format("2006-01-02"),

"email": user.Email,

}

c.JSON(http.StatusOK, resp)

}

}

**Пояснение:**

* RegisterHandler создаёт нового пользователя, проверяет уникальность email, хеширует пароль и сохраняет всё в БД.
* LoginHandler проверяет введённый пароль, а при успехе генерирует JWT-токен и возвращает его клиенту.
* GetCustomerHandler возвращает информацию о пользователе (кроме пароля) по указанному id.

### **3.7 Файл auth\_middleware.go (пакет middleware)**

package middleware

import (

"fmt"

"net/http"

"strings"

"github.com/gin-gonic/gin"

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

)

// Тот же самый ключ, что и в handlers

var secretKey = []byte("your-256-bit-secret")

// AuthRequired проверяет валидность JWT-токена,

// если токен невалиден — возвращаем 403 (Forbidden).

func AuthRequired(c \*gin.Context) {

authHeader := c.GetHeader("Authorization")

if authHeader == "" {

c.AbortWithStatusJSON(http.StatusForbidden, gin.H{"error": "Токен не предоставлен"})

return

}

// Проверяем, что начинается с "Bearer "

tokenStr := strings.TrimPrefix(authHeader, "Bearer ")

if tokenStr == authHeader {

c.AbortWithStatusJSON(http.StatusForbidden, gin.H{"error": "Некорректный заголовок Authorization"})

return

}

// Парсим токен

claims := &jwt.RegisteredClaims{}

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

return secretKey, nil

})

if err != nil {

c.AbortWithStatusJSON(http.StatusForbidden, gin.H{"error": fmt.Sprintf("Невалидный токен: %v", err)})

return

}

// Проверяем, действительно ли токен валиден

if !token.Valid {

c.AbortWithStatusJSON(http.StatusForbidden, gin.H{"error": "Невалидный токен"})

return

}

// Дополнительно можно проверить, что срок действия не истёк, но

// jwt.ParseWithClaims уже смотрит на ExpiresAt, если оно в будущем.

// Можно сохранить в Context ID пользователя (claims.Subject) для дальнейшего использования

c.Set("user\_id", claims.Subject)

// Продолжаем выполнение цепочки

c.Next()

}

**Пояснение:**

* Извлекаем заголовок Authorization.
* Ожидаем формат Bearer <token>.
* Парсим токен с помощью секретного ключа secretKey.
* Если токен невалиден — завершаем обработку (возвращаем 403 Forbidden).
* Если всё в порядке — сохраняем user\_id в c.Set("user\_id", ...) и передаём управление следующему обработчику.

## **4. Работа программы (проверка)**

Ниже — пример порядка запуска и проверки сервиса.

**Запуск сервера**:  
go run .

В консоли должно появиться сообщение об успешном подключении к БД и о запуске сервера на порту 8080.

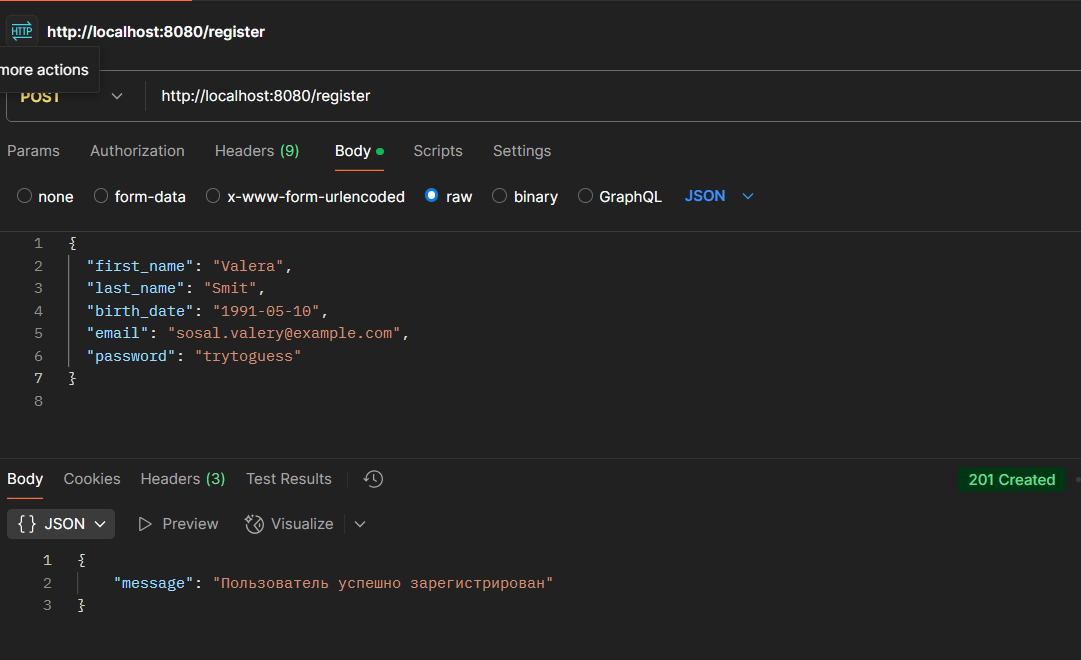
**Регистрация (POST /register)**Отправляем запрос через Postman:  


Рисунок 3 - Отправка запроса регистрации с кодом 200

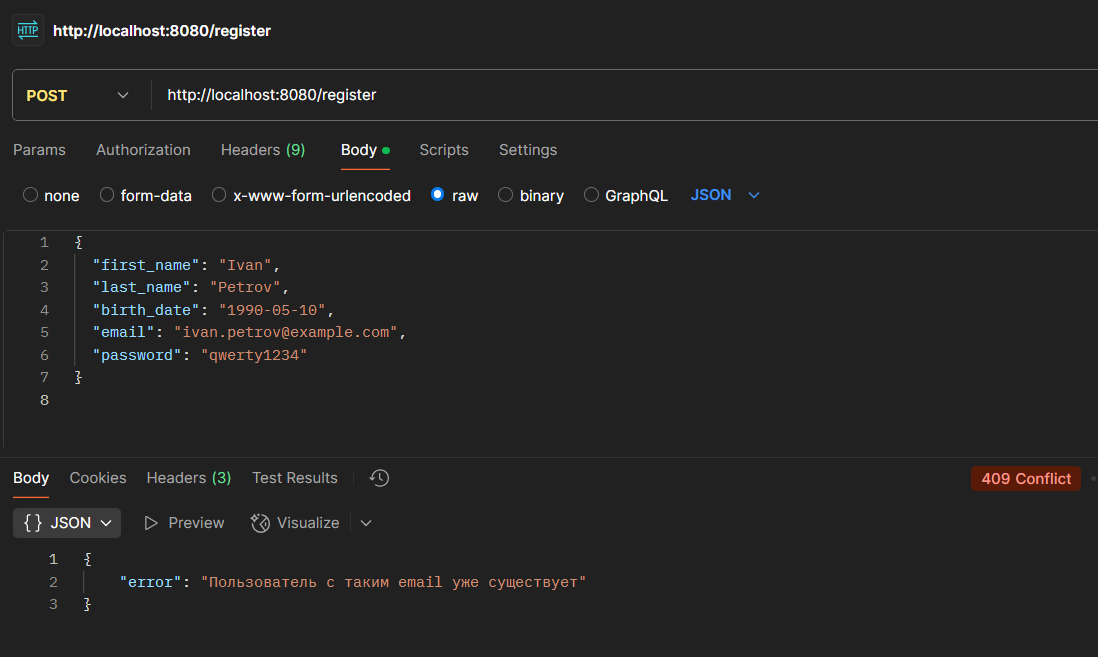


Рисунок 4 - Отправка запроса регистрации с кодом 400 (409)

**Авторизация (POST /login)**Отправляем запрос с тем же email/паролем:

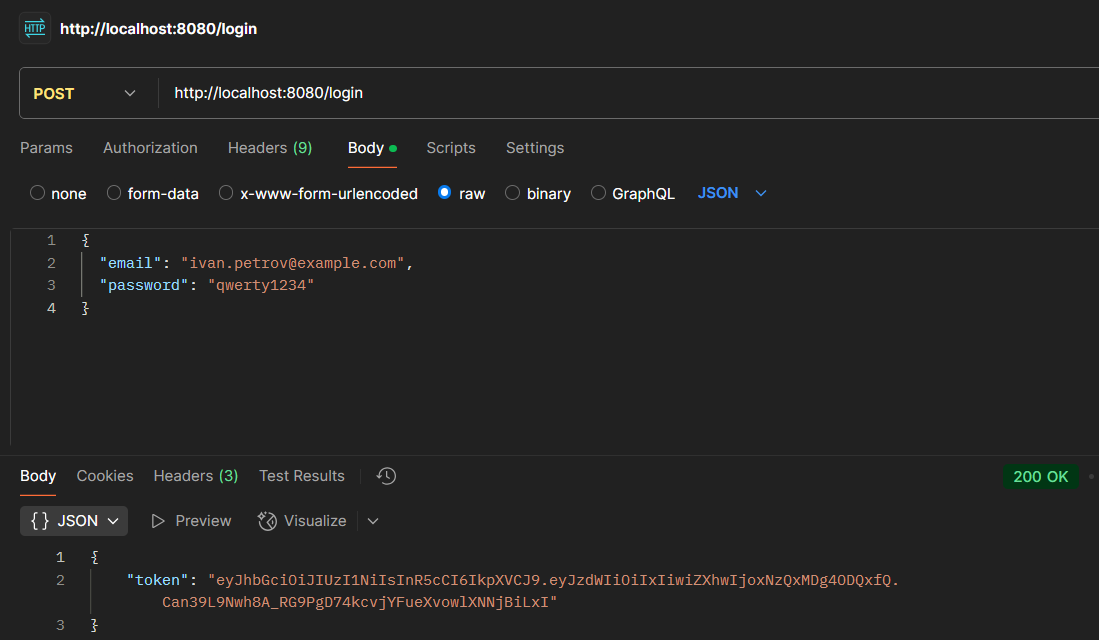


Рисунок 5 - Отправка запроса авторизации с кодом 200

**Полученный токен:**  
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxIiwiZXhwIjoxNzQxMDg4ODQxfQ.Can39L9Nwh8A\_RG9PgD74kcvjYFueXvowlXNNjBiLxI

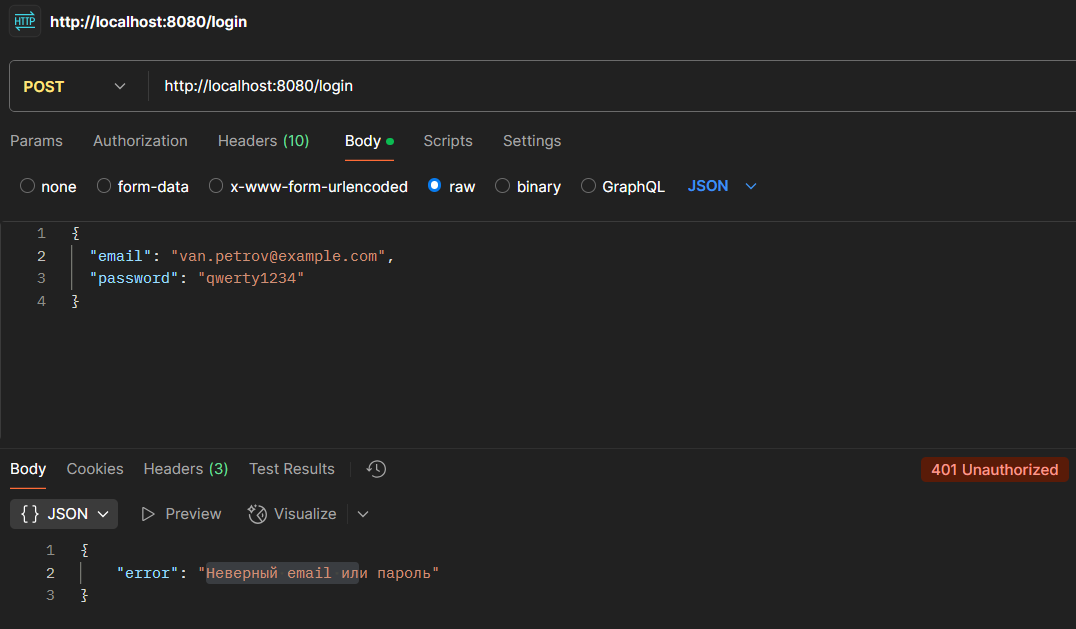


Рисунок 6 - Отправка запроса авторизации с кодом 400 (401)

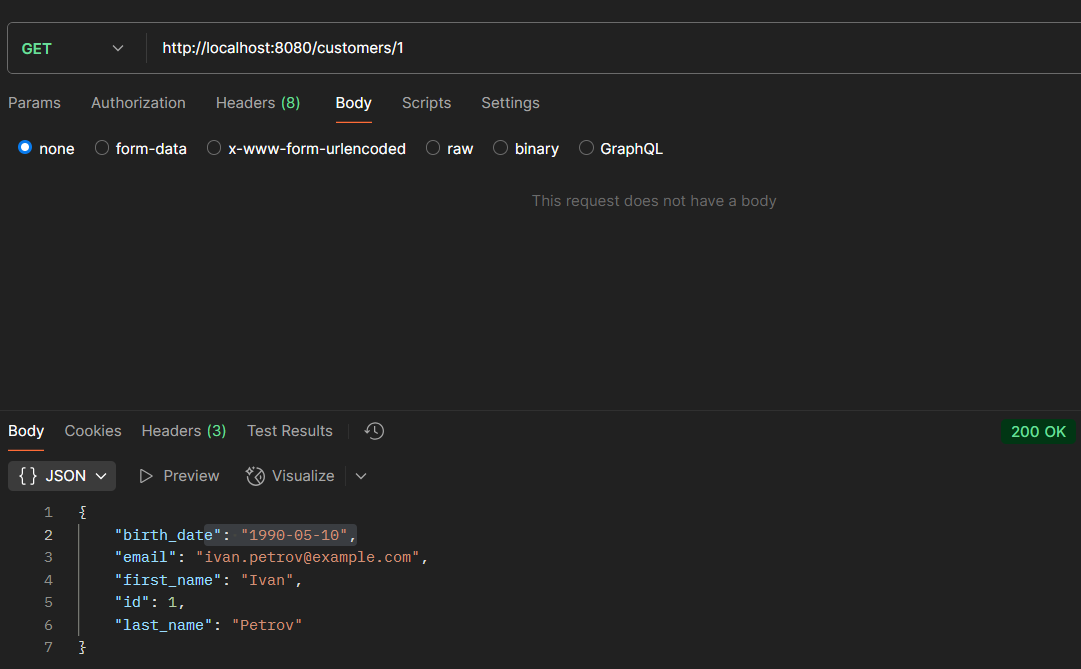
**Получение данных пользователя (GET /customers/:id)**Теперь, имея токен, делаем запрос по id (токен указываем в headers):  


Рисунок 7 - Успешный запрос информации о клиенте

Если не передадим или неверно передадим токен, получим ошибку 403 (Forbidden) или сообщение о невалидном токене.

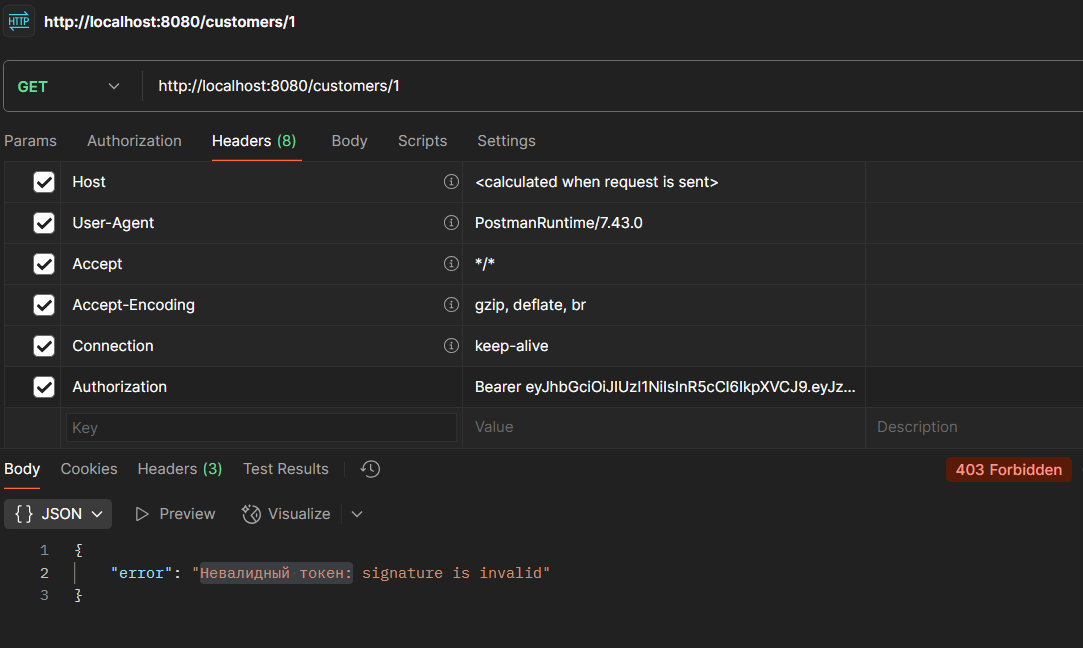


Рисунок 8 - Провальный запрос информации о клиенте

## **5. Вывод**

В ходе работы были решены следующие задачи:

1. Установлена и сконфигурирована СУБД PostgreSQL.
2. Создана база данных auth\_service\_db и таблица customers.
3. Реализована серверная часть на Go с использованием фреймворка Gin.
4. Организован процесс **регистрации** пользователя с хешированием пароля алгоритмом bcrypt.
5. Организован процесс **авторизации** пользователя с выдачей JWT-токена.
6. Настроено middleware для проверки **JWT** в заголовке Authorization для защищённых маршрутов.
7. Протестированы все основные запросы (регистрация, логин, получение данных пользователя), все работают корректно.

Таким образом, сервис авторизации успешно реализован в соответствии с заданием.