



Факультет программной инженерии и компьютерной техники

Лабораторная работа №1

«Разработка защищенного REST API с интеграцией в CI/CD»

по дисциплине «Информационная безопасность»

*Выполнил:*

Студент группы Р3432

Чмурова М.В.

*Преподаватель:*

Рыбаков Степан Дмитриевич

Санкт-Петербург

2025

## Задание

Получить практический опыт разработки безопасного backend-приложения с автоматизированной проверкой кода на уязвимости. Освоить принципы защиты от OWASP Top 10 и интеграцию инструментов безопасности в процесс разработки.

## Выполнение

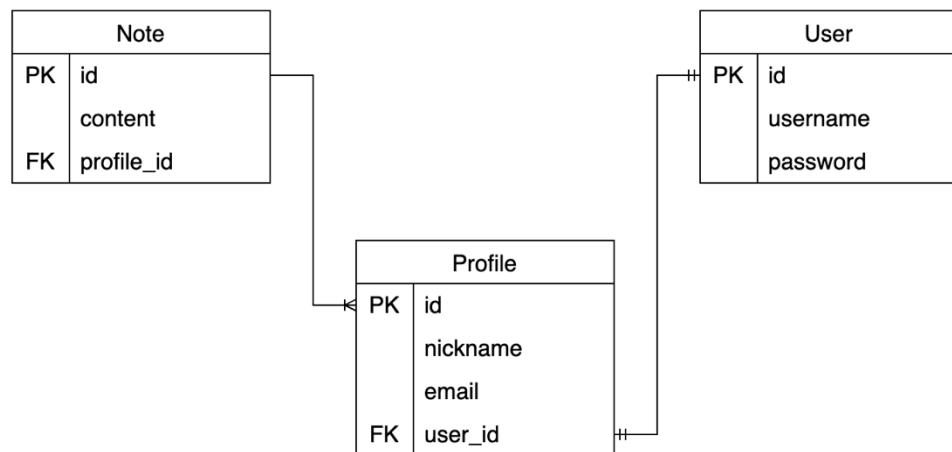


Рисунок 1. Структура БД

1. Таблица USER: хранение данных для аутентификации и идентификации
2. Таблица PROFILE: хранение персональной информации пользователя, которая не связана с аутентификацией. Благодаря разделению:
  - User отвечает за безопасность (логин/пароль)
  - Profile отвечает за личные данные, которые можно показывать или изменять без риска для безопасности.
3. Таблица NOTE: хранение заметок, созданных пользователями

## Эндпоинты API

### 1. Регистрация пользователя

- URL: **POST /auth/register** - регистрация нового пользователя

Запрос:

```
{  
  "username": "test",  
  "password": "test"  
  "nickname": "nickname",  
  "email": "test@example.com"  
}
```

Ответ:

```
"User registered successfully"
```

Пример выполнения корректного запроса в IntelliJIDEA:

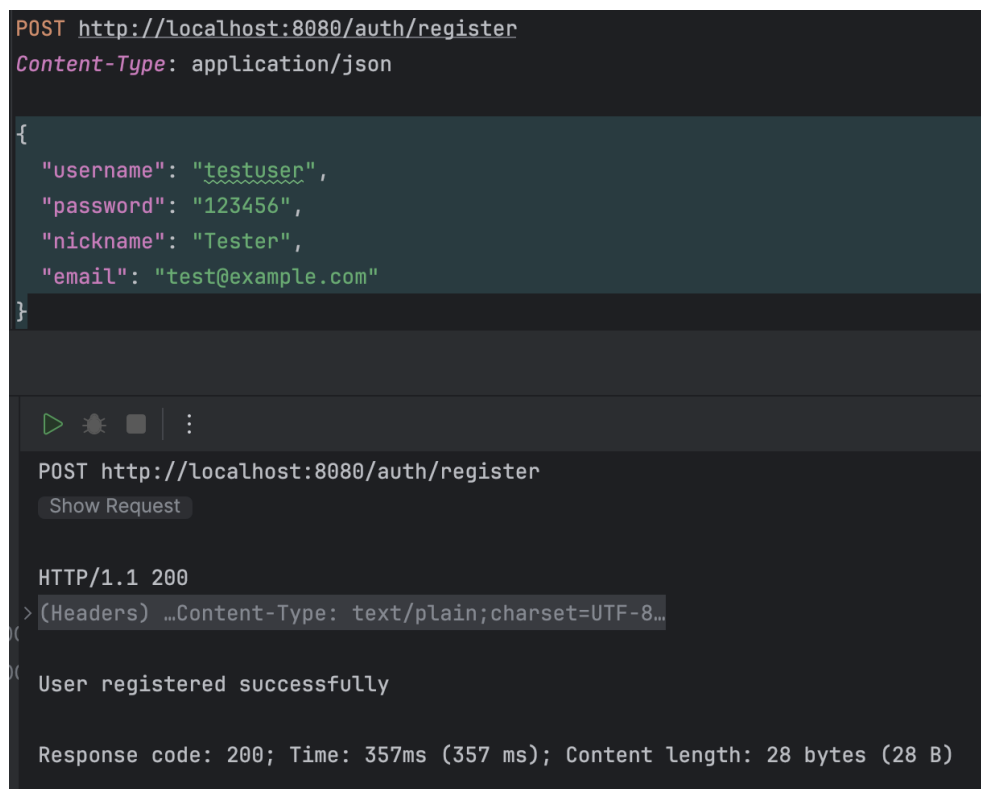


Рисунок 2. Успешная регистрация пользователя

### 2. Логин пользователя

URL: **POST /auth/login** - метод для аутентификации пользователя  
(принимает логин и пароль)

Запрос:

```
{
  "username": "test",
  "password": "test"
}
```

Ответ: JWT-токен

```
{
  "token": "eyJhbGciOiJIUzI1NiJ9..."
}
```

Пример выполнения корректного запроса в IntelliJ IDEA:

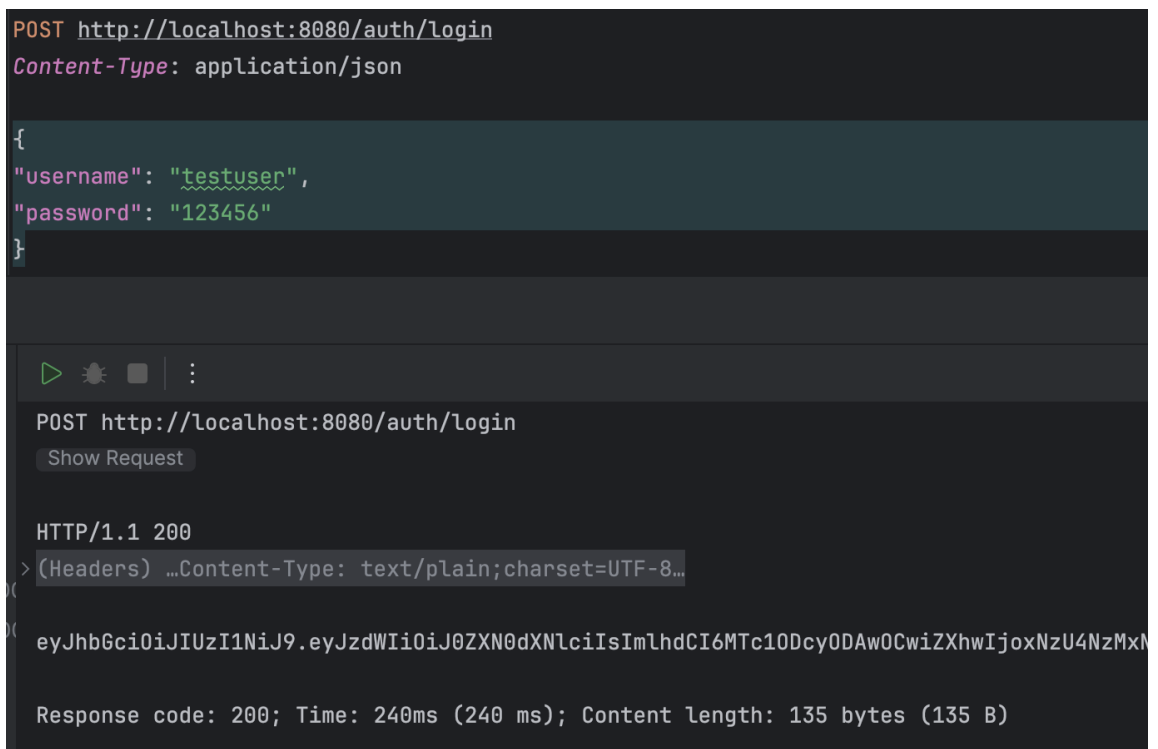


Рисунок 3. Успешный login пользователя

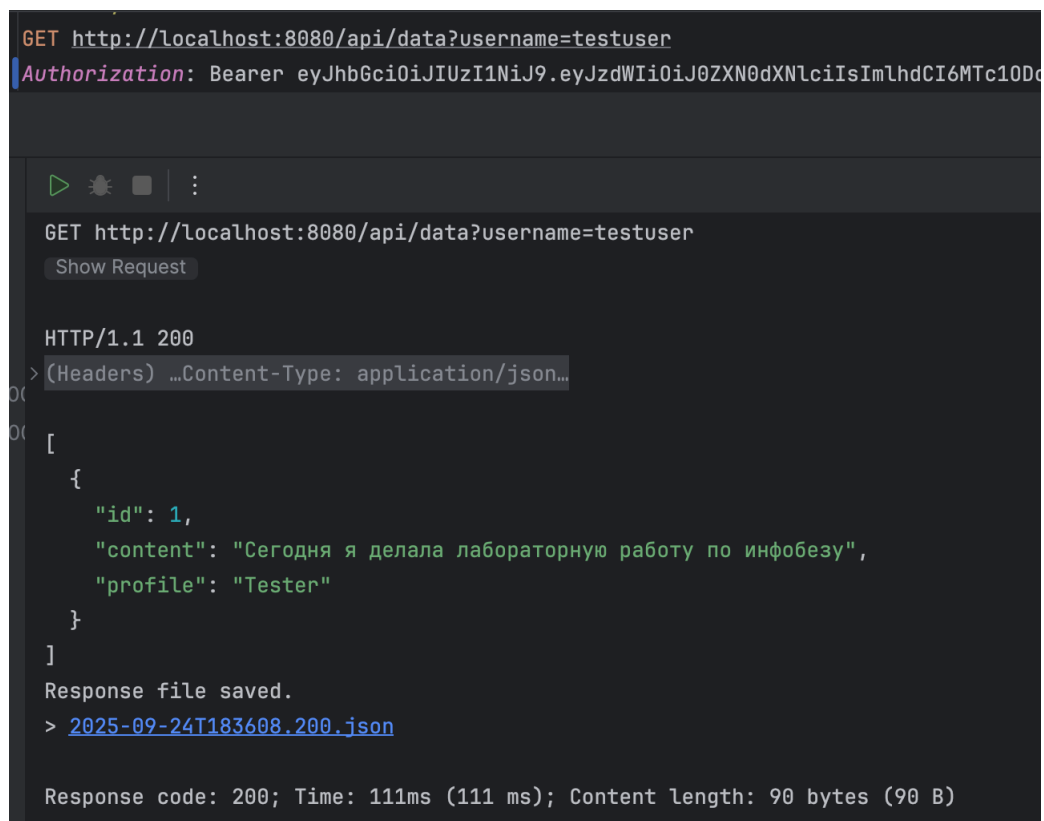
### 3. Получение данных

URL: **GET /api/data?username=test** - метод для получения каких-либо данных (например, список пользователей или постов). Доступ должен быть только у аутентифицированных пользователей.

Ответ:

```
[{
  "id": 1
  "content": "example text"
  "profile": "test"
}]
```

Пример выполнения корректного запроса в IntelliJIDEA:



```
GET http://localhost:8080/api/data?username=testuser
Authorization: Bearer eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ0ZXN0dXNlciIsImhhdCI6MTc1ODc
[{"id": 1, "content": "Сегодня я делала лабораторную работу по инфобезу", "profile": "Tester"}]
```

Response file saved.  
> [2025-09-24T183608.200.json](#)

Response code: 200; Time: 111ms (111 ms); Content length: 90 bytes (90 B)

Рисунок 4. Успешное получение данных пользователя

#### 4. Создание заметки

URL: **POST /api/notes** - метод для создания новой заметки текущего пользователя. Требуется аутентификация

Заголовки запроса:

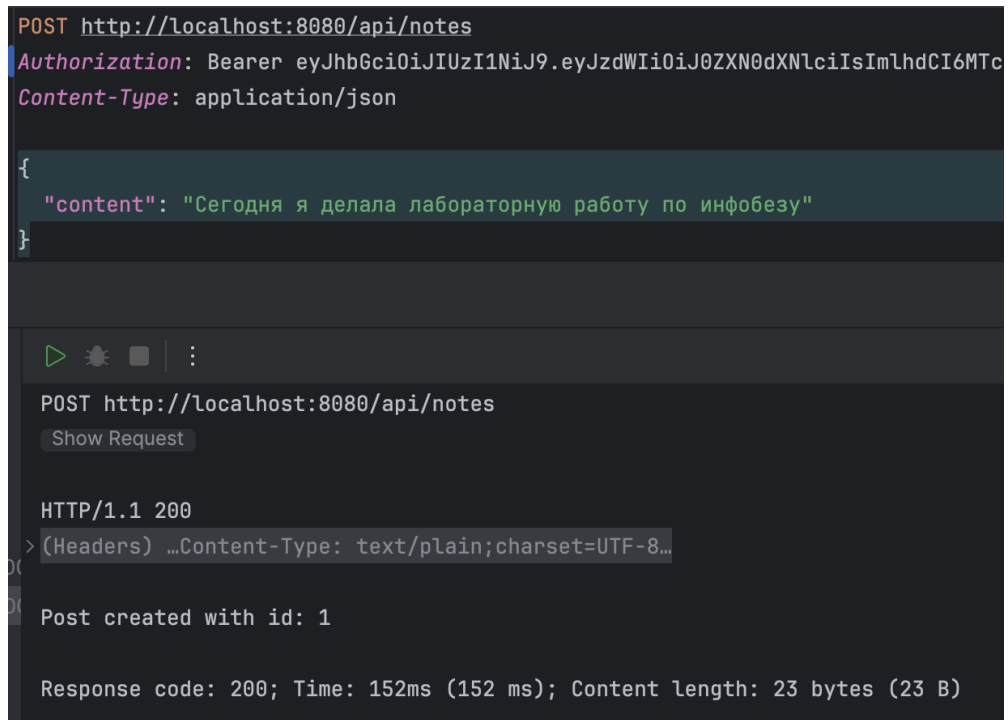
*Authorization: Bearer ваш\_токен\_здесь*

*Content-Type: application/json*

Запрос:

```
{  
  "content": "example text"  
}
```

Пример выполнения корректного запроса в IntelliJ IDEA:



```
POST http://localhost:8080/api/notes  
Authorization: Bearer eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ0ZXN0dXNlciIsImIhdCI6MTc  
Content-Type: application/json  
  
{  
  "content": "Сегодня я делала лабораторную работу по инфобезу"  
}  
  
POST http://localhost:8080/api/notes  
Show Request  
  
HTTP/1.1 200  
> (Headers) ...Content-Type: text/plain; charset=UTF-8...  
Post created with id: 1  
  
Response code: 200; Time: 152ms (152 ms); Content length: 23 bytes (23 B)
```

Рисунок 5. Успешное создание заметки

## Реализованные меры защиты

### 1. SQL Injection (SQLi):

- используется ORM (Hibernate/JPA) и параметризованные запросы, что исключает возможность внедрения SQL-кода.

В коде:

```
public interface UserRepository extends JpaRepository<User, Long> { 2 usages 2 kketch  
    Optional<User> findByUsername(String username); 2 usages 2 kketch  
}
```

Spring Data JPA автоматически генерирует SQL за счет чего не конкатенируются строки и инъекция невозможна

## 2. XSS:

- сервер возвращает данные только в формате JSON, а не HTML; кроме того, Spring Security добавляет защитные HTTP-заголовки.

В коде:

```
@RestController @ kkettch
@RequestMapping("/auth")
public class AuthController {

    private AuthService authService; 3 usages

    public AuthController(AuthService authService) { @ kkettch
        this.authService = authService;
    }

    @PostMapping("/register") @ kkettch
    public ResponseEntity<String> register(@RequestBody RegisterRequest registerRequest) {
        authService.register(
            registerRequest.getUsername(),
            registerRequest.getPassword(),
            registerRequest.getNickname(),
            registerRequest.getEmail()
        );
        return ResponseEntity.ok( body: "User registered successfully");
    }
}
```

API реализован как JSON-only (@RestController), сервер не рендерит HTML-страницы. Все ответы сериализуются Jackson'ом в JSON, а не вставляются в HTML-шаблоны;

## 3. Аутентификация и авторизация:

- реализованы с помощью JWT-токенов. Каждый запрос проверяется фильтром на наличие и корректность токена; неавторизованные запросы блокируются.

В коде:

```

public Note createNotes(String token, String content) { 1 usage 2 kkettch
    String username = jwtService.extractUsername(token);
    Profile profile = profileRepository.findByUserUsername(username)
        .orElseThrow(() -> new RuntimeException("Profile not found"));
    Note note = Note.builder().content(content).profile(profile).build();
    return noteRepository.save(note);
}

```

#### 4. Пароли:

- сохраняются в базе данных только в хешированном виде (с помощью BCrypt).

В коде:

```

public void register(String username, String password, String nickname, String email) { 1 usage 2 kkettch *
    if (userRepository.findByUsername(username).isPresent()) {
        throw new RuntimeException("Username already exists");
    }
    User user = User.builder().username(username).password(passwordEncoder.encode(password)).build();
    userRepository.save(user);
    Profile profile = Profile.builder().nickname(nickname).email(email).user(user).build();
    profileRepository.save(profile);
}

```

Bean PasswordEncoder создаёт BCrypt-хешер паролей для безопасного хранения и проверки. Пароли не хранятся в открытом виде, а проверяются через matches.



# Отчеты SAST/SCA

## SpotBugs Report

### Project Information

Project: secure-api

SpotBugs version: 4.9.4

Code analyzed:

- /home/runner/work/information-security-semester-7/information-security-semester-7/target/classes

### Metrics

340 lines of code analyzed, in 21 classes, in 7 packages.

Metric	Total	Density*
High Priority Warnings		0.00
Medium Priority Warnings		0.00
<b>Total Warnings</b>	<b>0</b>	<b>0.00</b>

(\* Defects per Thousand lines of non-commenting source statements)

Рисунок 6. Прохождение SAST

```
✓ Run Snyk test (SCA)

1  ▶ Run snyk test --all-projects --show-vulnerable-paths=all
8
9  Testing /home/runner/work/information-security-semester-7/information-security-semester-7...
10
11 Organization:    kkettch
12 Package manager: maven
13 Target file:     pom.xml
14 Project name:    com.kkettch:secure-api
15 Open source:     no
16 Project path:    /home/runner/work/information-security-semester-7/information-security-semester-7
17 Licenses:        enabled
18
19 ✓ Tested 76 dependencies for known issues, no vulnerable paths found.
```

Рисунок 7. Прохождение Snyk (SCA)

Ссылка на успешно пройденный pipeline: <https://github.com/kkettch/information-security-semester-7/actions/runs/17979456301/job/51141314225#logs>

Ссылка на GitHub репозиторий: <https://github.com/kkettch/information-security-semester-7>

## **Вывод**

В ходе выполнения данной лабораторной работы я вспомнила как работать со Spring Boot и использовать инструменты для разработки backend'а, защищенного от уязвимостей, такие как: JWT для безопасного управления сессиями пользователей, BCryptPasswordEncoder для безопасного хранения паролей, JPA/Hibernate для работы с базой данных с использованием параметризованных запросов и других.

Кроме того, я опробовала такие методы тестирования безопасности как SAST и SCA (через Snyk), которые подтвердили, что мой backend защищен от главных широко известных уязвимостей.