Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

«Национальный исследовательский университет ИТМО»

Факультет Программной инженерии и компьютерной техники

Лабораторная работа №1

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

по дисциплине «Информационная безопасность»

Группа: P3430

Выполнил:

Кириллов Андрей Андреевич

Преподаватель:

Рыбаков Степан Дмитриевич

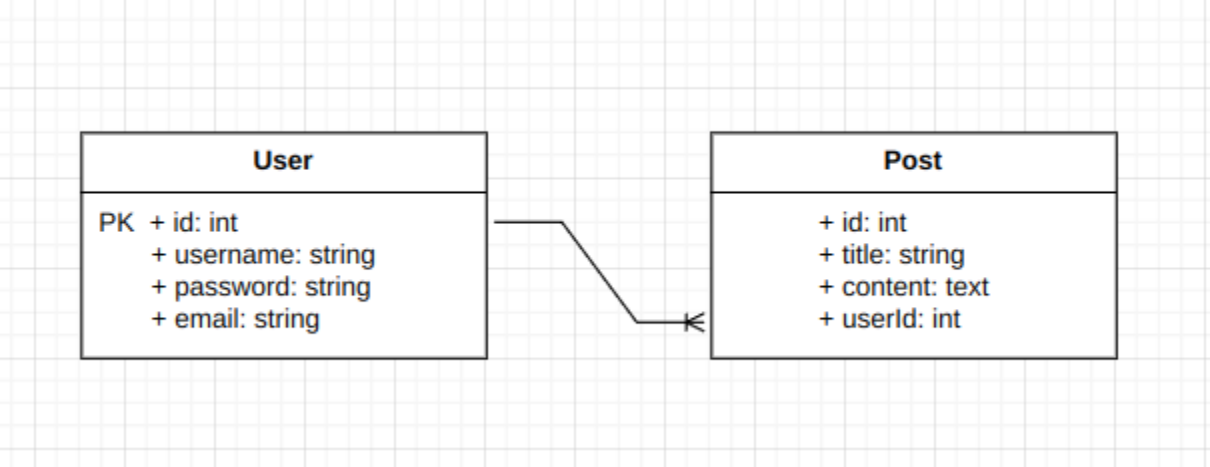
г. Санкт-Петербург

2025 г.

Цель

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



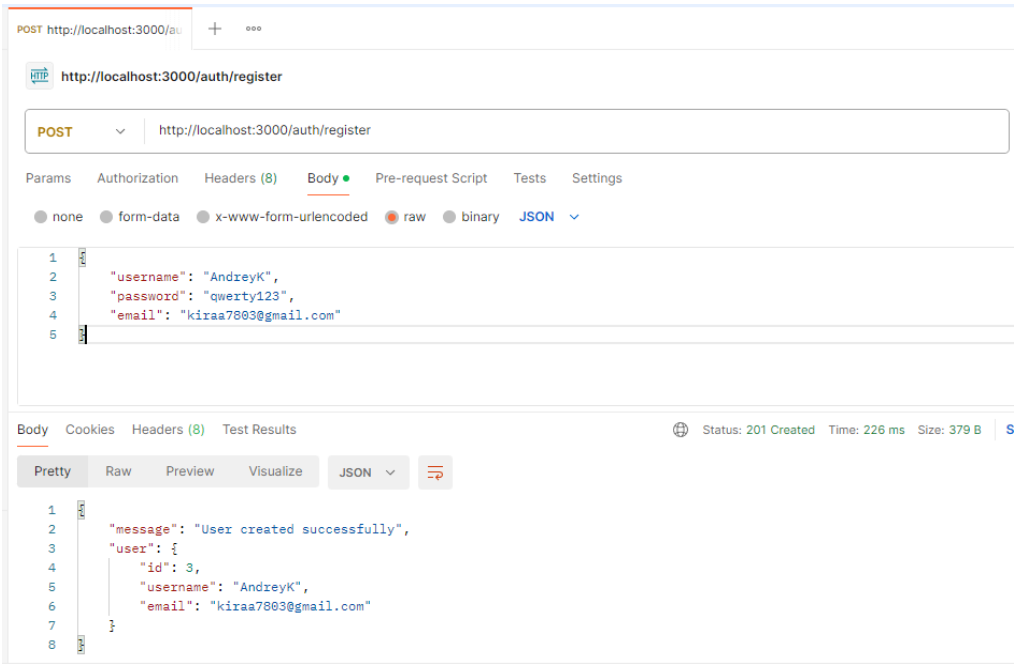
1. Таблица USER: хранение данных для аутентификации и идентификации

2. Таблица POST: хранение постов, созданных пользователями

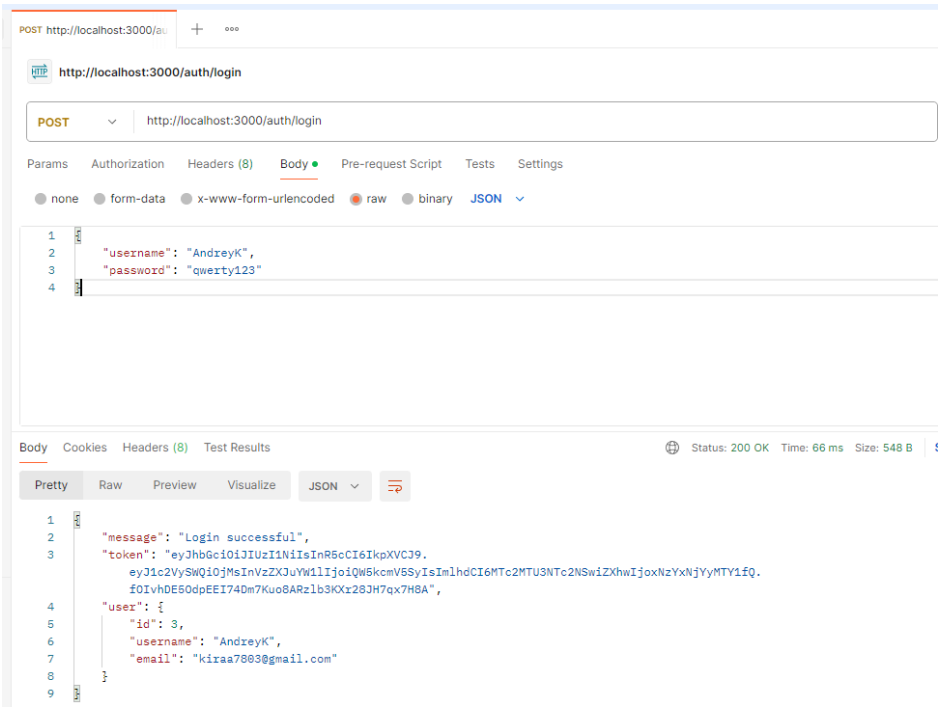
Для реализации был выбран nodejs + Express. База данных - sqlite.

Эндпоинты

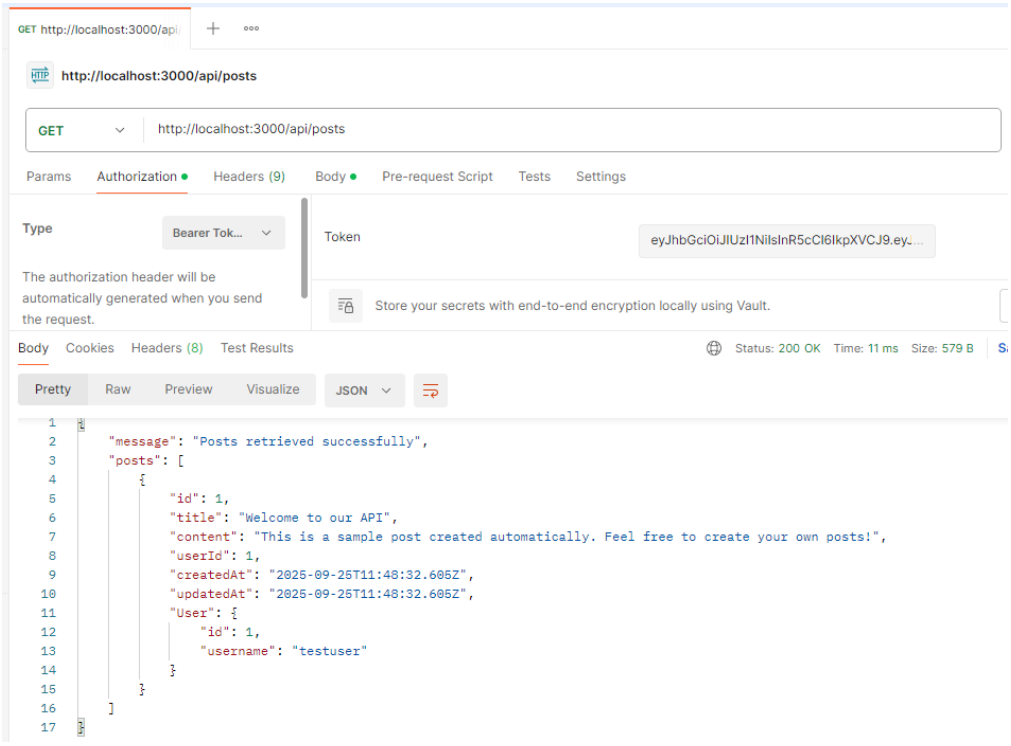
1. Регистрация пользователя - URL: POST /auth/register



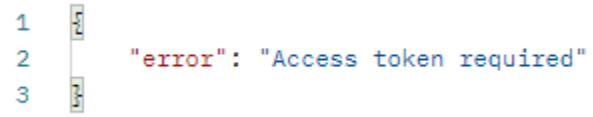
2. Авторизация пользователя - URL: POST /auth/login



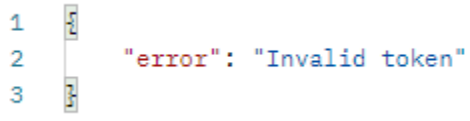
3. Получение постов - URL: GET /api/posts



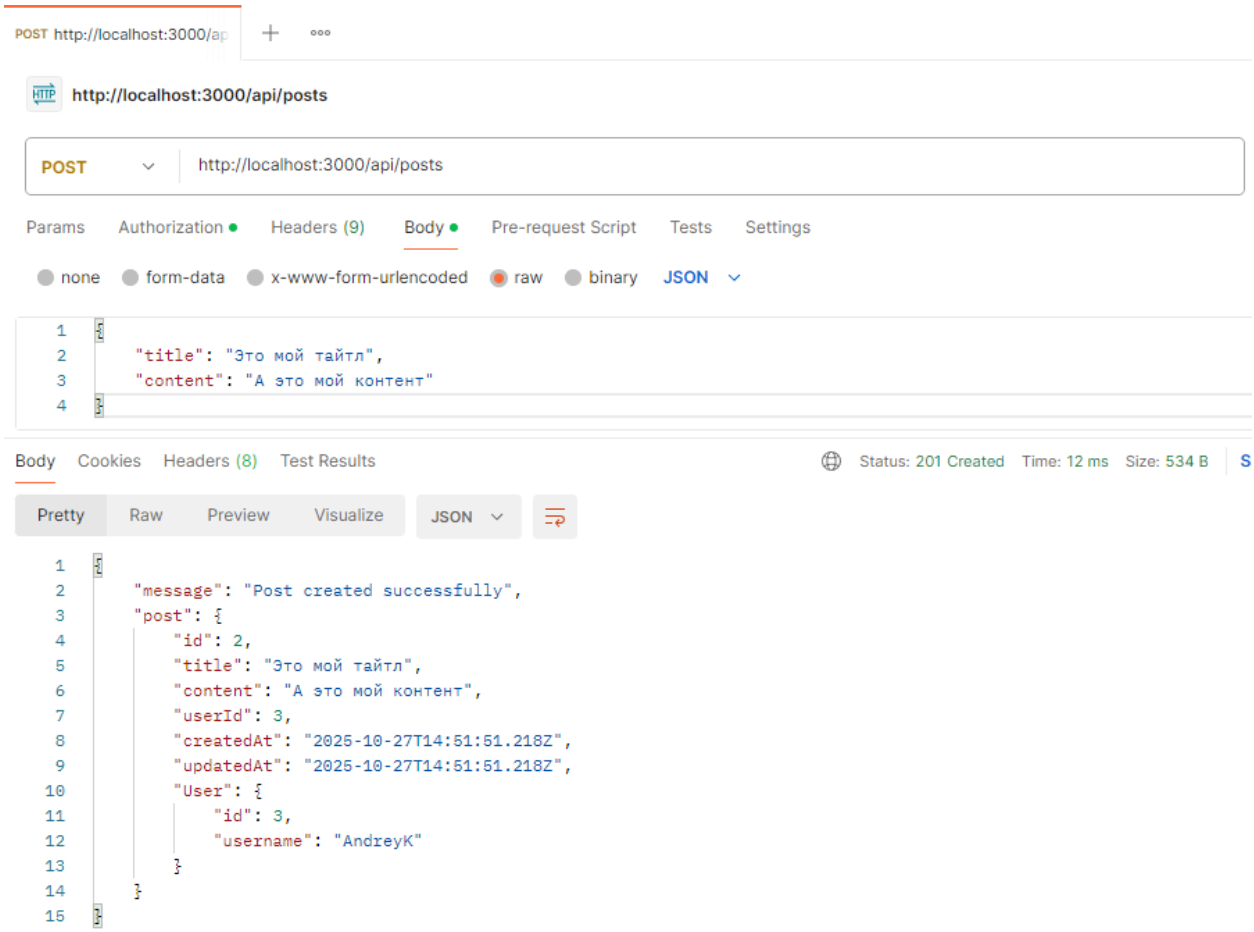
Полученный в п.2 токен вставляем в bearer. Теперь можем попросить и получить данные, иначе получим ошибку:



Или если токен некорректный:



4. Создание поста - URL: POST /api/posts



Тут также нужно указать токен, тк только авторизованные пользователи могут создавать посты.

Реализованные меры защиты

1. Защита от SQLi (SQL-инъекций) - я использую базу базу данных sqlite вместе с sequalize, которой обеспечивает маппинг объектов на таблицу (ОРМ). Благодаря такому подходу мы решаем проблему sql инъекций путем конкатенации строк.

Пример описания объекта User с помощью sequelize

```
const User = sequelize.define('User', {
  id: {
    type: DataTypes.INTEGER,
    primaryKey: true,
    autoIncrement: true
  },
  username: {
    type: DataTypes.STRING,
    allowNull: false,
    unique: true
  },
  password: {
    type: DataTypes.STRING,
    allowNull: false
  },
  email: {
    type: DataTypes.STRING,
    allowNull: false,
    unique: true
  }
}, {
  hooks: {
    beforeCreate: async (user) => {
      user.password = await bcrypt.hash(user.password, 10);
    }
  }
});

User.prototype.validatePassword = async function(password) {
  return await bcrypt.compare(password, this.password);
};
```

Пример запроса к бд с помощью sequelize:

```
const user = await User.findOne({ where: { username } });
```

2. Защита от XSS - я реализовал санитизацию всех пользовательских данных перед отправкой в ответах API.

```
const sanitizeString = (str) => {
  if (typeof str !== 'string') return '';
  return escapeHtml(str);
};

const sanitizeObject = (obj) => {
  if (!obj || typeof obj !== 'object') return obj;

  const sanitized = {};
  for (const [key, value] of Object.entries(obj)) {
    if (typeof value === 'string') {
      sanitized[key] = sanitizeString(value);
    } else if (typeof value === 'object' && value !== null) {
      sanitized[key] = sanitizeObject(value);
    } else {
      sanitized[key] = value;
    }
  }
  return sanitized;
};
```

Теперь перед отправкой ответа использую написанную функцию

```
const sanitizedUser = sanitizeObject({
  id: user.id,
  username: user.username,
  email: user.email
});

res.json({
  message: 'Login successful',
  token,
  user: sanitizedUser
});
```

3. Аутентификация реализована с использованием JWT-токенов. Для каждого входящего запроса выполняется проверка наличия и валидности токена доступа. Запросы без корректного токена не проходят. Приведу код моего middleware:

```
const jwt = require('jsonwebtoken');
const { User } = require('../models');

const authenticateToken = async (req, res, next) => {
  try {
    const authHeader = req.headers['authorization'];
    const token = authHeader && authHeader.split(' ')[1];

    if (!token) {
      return res.status(401).json({ error: 'Access token required' });
    }

    const decoded = jwt.verify(token, process.env.JWT_SECRET || 'fallback-secret');
    const user = await User.findByPk(decoded.userId);

    if (!user) {
      return res.status(401).json({ error: 'Invalid token' });
    }

    req.user = user;
    next();
  } catch (error) {
    console.error('Auth error:', error);
    return res.status(403).json({ error: 'Invalid token' });
  }
};

module.exports = { authenticateToken };
```

Теперь в эндпоинтах аргументом до функции обработчика указываем эту функцию

```
router.post('/posts', authenticateToken, async (req, res) => {
  try {
    let { title, content } = req.body;
```

Прежде, чем перейти к обработке запроса, выполняется функция аутентификации, чтобы убедиться, что имеем дело с авторизованным пользователем. В ином случае возвращается ошибка и в функцию-обработчик уже не попадем.

Пароли храню в зашифрованном виде с помощью bcrypt:

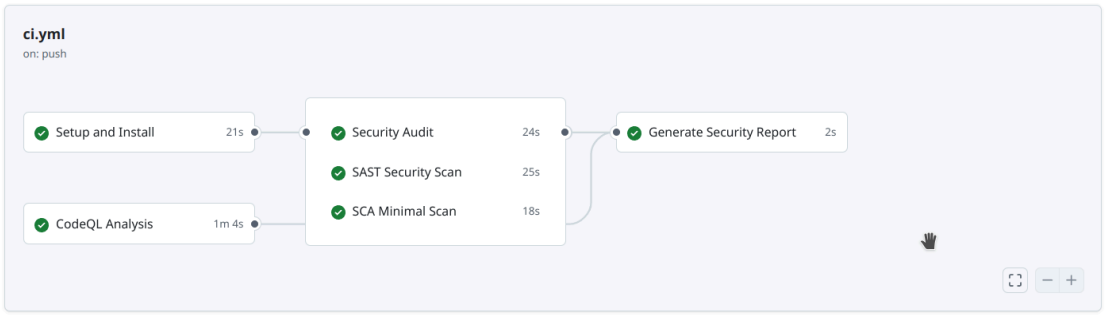
```
, {
  hooks: {
    beforeCreate: async (user) => {
      user.password = await bcrypt.hash(user.password, 10);
    }
  }
);
```

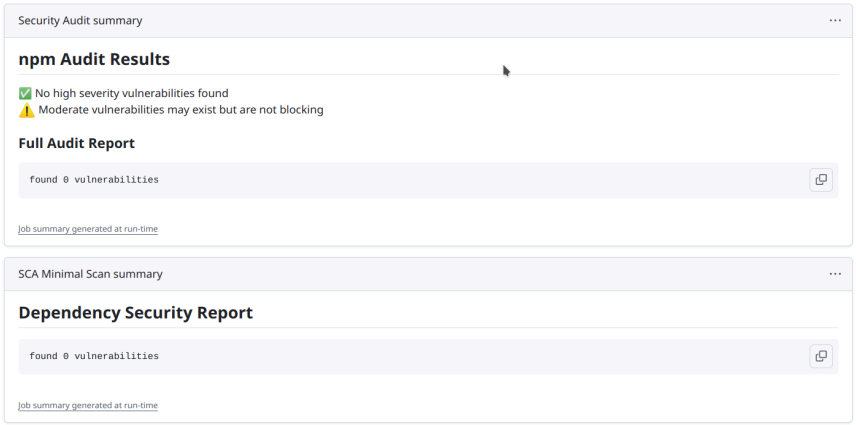
И также функция для сравнения паролей при авторизации:

```
User.prototype.validatePassword = async function(password) {
  return await bcrypt.compare(password, this.password);
};
```

Отчеты SAST/SCA

Pipeline пройден:





Ссылка: https://github.com/cemetiere/information_security_lab1/actions/runs/18881253107

Ссылка на репозиторий с кодом: https://github.com/cemetiere/information_security_lab1

Вывод

В ходе лабораторной работы я разработал защищенный backend на Node.js, реализовав основные механизмы безопасности: JWT для аутентификации, bcrypt для хеширования паролей и Sequelize с параметризованными запросами против SQL-инъекций.

Для тестирования безопасности применил SAST через ESLint и CodeQL, а также SCA через Snyk. Все проверки в CI/CD пайплайне прошли успешно, подтвердив защищенность приложения от основных уязвимостей OWASP Top 10.