Лекция 5: Аутентификация и авторизация. Вешаем замки и выдаем ключи

Введение: "Кто ты?" и "Что тебе можно?"

Здравствуйте! На прошлой лекции мы подключили к нашему приложению базу данных, и оно научилось надежно хранить информацию. Но сейчас наше API похоже на дом с открытыми настежь дверями: любой желающий может войти и делать все что угодно — читать, создавать, изменять и удалять данные. Это абсолютно неприемлемо для любого реального приложения.

Сегодня мы будем строить систему безопасности для нашего "дома". Мы разберем два фундаментальных понятия, которые часто путают, но которые являются основой любой защищенной системы:

1. **Аутентификация (Authentication):** Процесс проверки личности. Отвечает на вопрос **"Кто ты?"**.
2. **Авторизация (Authorization):** Процесс проверки прав доступа. Отвечает на вопрос **"Что тебе можно делать?"**.

Мы изучим, как безопасно хранить пароли, разберем современный стандарт аутентификации для API — **JSON Web Tokens (JWT)**, и на практике реализуем в нашем приложении систему регистрации и входа, а также научимся защищать эндпоинты от неавторизованного доступа.

1. Безопасность паролей: Правило №1

Прежде чем говорить о чем-либо еще, мы должны усвоить одно незыблемое правило, нарушение которого является не просто плохой практикой, а преступной халатностью:

Никогда, ни при каких обстоятельствах, не храните пароли в открытом виде!

Если произойдет утечка вашей базы данных (а это случается даже с IT-гигантами), пароли всех ваших пользователей окажутся в руках злоумышленников. А так как многие люди используют одинаковые пароли на разных сайтах, утечка из вашего, казалось бы, неважного сервиса может дать хакерам доступ к почте, соцсетям и банковским приложениям ваших пользователей.

Правильное решение: Хеширование

Мы не храним сам пароль. Мы храним только его **хеш** — результат работы односторонней математической функции.

- **Односторонняя:** легко получить хеш из пароля, но практически невозможно получить пароль из хеша. (Аналогия: из куска мяса легко сделать фарш, но из фарша собрать обратно кусок мяса — невозможно).
- **Процесс:** Когда пользователь регистрируется, мы хешируем его пароль и сохраняем в базу только хеш. Когда он пытается войти, мы снова хешируем введенный им пароль и **сравниваем два хеша**. Если они совпадают — пароль верный.

Чтобы защититься от атак с использованием "радужных таблиц" (предвычисленных хешей для популярных паролей), к каждому паролю перед хешированием добавляется **"соль" (salt)** — случайная строка, уникальная для каждого пользователя.

Для этого мы не будем "изобретать велосипед", а воспользуемся проверенной временем библиотекой **bcrypt**. Она автоматически управляет "солью" и предоставляет два простых метода: bcrypt.hash() для хеширования и bcrypt.compare() для сравнения.

2. От сессий к токенам: Эволюция аутентификации в вебе

Существует два фундаментально разных подхода к тому, как сервер "помнит" залогиненного пользователя.

- **Stateful (с состоянием): Сессии и Cookies.** Это "классический" способ. Когда пользователь логинится, сервер создает на своей стороне **сессию** (запись в хранилище, например, "сессия_XYZ связана с user_id_123") и отправляет клиенту в cookie только идентификатор этой сессии. При каждом запросе браузер автоматически присылает cookie, и сервер по ID находит сессию у себя.
    - **Проблема:** Этот подход плохо масштабируется (если у вас много серверов, им нужно общее хранилище сессий) и неудобен для не-браузерных клиентов (мобильных приложений).
- **Stateless (без состояния): Токены.** Это современный подход, идеальный для API. Серверу **не нужно ничего помнить**.
    - **Процесс:** Когда пользователь логинится, сервер генерирует **токен** — зашифрованную строку, которая сама по себе содержит всю нужную информацию (например, ID пользователя и срок действия). Токен отдается клиенту.
    - Клиент сохраняет токен и **вручную прикрепляет его к каждому запросу** в специальном заголовке (Authorization).
    - Сервер, получив запрос, просто проверяет подлинность токена, извлекает из него данные и понимает, кто к нему обратился.

3. Глубокое погружение в JWT (JSON Web Token)

Самым популярным стандартом для stateless-аутентификации является **JWT (произносится "джот")**.

Аналогия: JWT — это как **персональный бейдж** на мероприятии. На нем есть информация о вас (Payload), информация о самом бейдже (Header) и, самое главное, — **защитная цифровая подпись-голограмма (Signature)**, которую невозможно подделать. Любой охранник (сервер), просто взглянув на голограмму, поймет, что бейдж настоящий, ему не нужно сверяться со списками в центральной базе.

Токен состоит из трех частей, разделенных точками: Header.Payload.Signature.

- **Header (Заголовок):** Метаданные о токене (алгоритм подписи и т.д.).

- **Payload (Полезная нагрузка):** Сами данные, которые мы хотим "зашить" в токен. Обычно это userId, может быть role (роль пользователя), а также стандартные поля: iat (время выдачи) и exp (время истечения срока действия).
    - **Важно:** Header и Payload не зашифрованы, а просто закодированы в Base64. Их может прочитать любой. **Никогда не храните в Payload конфиденциальную информацию!**
- **Signature (Подпись):** Самая важная часть. Она создается путем хеширования заголовка и полезной нагрузки с использованием **секретного ключа**, который известен **только серверу**. Именно подпись гарантирует, что данные в токене не были изменены после его выдачи.

Для работы с JWT в Node.js мы будем использовать популярную библиотеку jsonwebtoken.

4. Практика: Реализация регистрации и входа

Теперь давайте применим эти знания на практике и создадим эндпоинты для аутентификации в нашем приложении.

Шаг 1: Установка зависимостей

```
npm install bcrypt jsonwebtoken
```

Шаг 2: Создание роута /auth/register

Этот эндпоинт будет принимать email и password, хешировать пароль и создавать нового пользователя в базе данных.

```
// routes/auth.js
const bcrypt = require('bcrypt');

router.post('/register', async (req, res, next) => {
  try {
    const { email, password } = req.body;
    // ... здесь должна быть валидация данных ...

    // Проверяем, не занят ли email
    const existingUser = await prisma.user.findUnique({ where: { email } });
    if (existingUser) {
      return res.status(400).json({ message: 'Пользователь уже существует' });
    }

    // Хешируем пароль
    const hashedPassword = await bcrypt.hash(password, 12); // 12 - 'сложность'

    // Создаем пользователя
    const user = await prisma.user.create({
      data: { email, password: hashedPassword },
    });

    res.status(201).json({ message: 'Пользователь успешно создан' });
  } catch (e) {
    next(e);
  }
});
```

Шаг 3: Создание роута /auth/login

Этот эндпоинт будет проверять учетные данные и, в случае успеха, генерировать и возвращать JWT.

```
// routes/auth.js
const jwt = require('jsonwebtoken');

router.post('/login', async (req, res, next) => {
  try {
    const { email, password } = req.body;
    const user = await prisma.user.findUnique({ where: { email } });
    if (!user) {
      return res.status(401).json({ message: 'Неверные учетные данные' });
    }

    // Сравниваем введенный пароль с хешем в базе
    const isPasswordValid = await bcrypt.compare(password, user.password);
    if (!isPasswordValid) {
      return res.status(401).json({ message: 'Неверные учетные данные' });
    }

    // Генерируем токен
    const token = jwt.sign(
      { userId: user.id },      // Полезная нагрузка (Payload)
      process.env.JWT_SECRET,   // Секретный ключ из .env!
      { expiresIn: '1h' }      // Срок жизни токена
    );

    res.json({ token, userId: user.id });
  } catch (e) {
    next(e);
  }
});
```

5. Защита роутов с помощью Middleware

Теперь, когда пользователь может получить токен, нам нужно создать "фейс-контроль" для эндпоинтов, которые должны быть доступны только залогиненным пользователям (например, создание новой задачи). Для этого идеально подходит **Middleware**.

Мы создадим authMiddleware.js, который будет:

1. Извлекать токен из заголовка Authorization: Bearer <token>.
2. Проверять его подлинность и срок действия с помощью jwt.verify().
3. Если токен валидный, извлекать из него userId и добавлять его в объект req (например, req.userData = { userId: ... }), чтобы последующие обработчики могли его использовать.
4. Если токен невалидный или отсутствует, отправлять ошибку 401 Unauthorized.

```
// middleware/auth.js
const jwt = require('jsonwebtoken');

module.exports = (req, res, next) => {
  try {
    const token = req.headers.authorization.split(' ')[1]; // Bearer TOKEN
    const decodedToken = jwt.verify(token, process.env.JWT_SECRET);
    req.userData = { userId: decodedToken.userId };
    next(); // Пропускаем дальше
```

```
  } catch (error) {
    res.status(401).json({ message: 'Аутентификация не удалась!' });
  }
};
```

Теперь мы можем легко защитить любой роут, просто добавив это middleware перед основным обработчиком:

```
// routes/tasks.js
const authMiddleware = require('../middleware/auth');

// Этот эндпоинт теперь защищен
router.post('/', authMiddleware, async (req, res) => {
  // Благодаря middleware, у нас здесь есть req.userData.userId
  const userId = req.userData.userId;
  // ... логика создания задачи с привязкой к автору ...
});
```

Заключение

Сегодня мы построили полноценную систему безопасности для нашего приложения. Мы научились безопасно хранить пароли, разобрали современный стандарт аутентификации на основе JWT и на практике реализовали регистрацию, вход и защиту эндпоинтов.

Наш "дом" теперь под надежным замком. На следующей лекции мы создадим для него понятную "инструкцию по эксплуатации", чтобы другие разработчики (например, фронтендеры) могли легко понять, как с ним взаимодействовать. Мы поговорим о **документировании API с помощью Swagger/OpenAPI**.