Лекция 7: Тестирование Backend-приложений. Строим "сеть безопасности" для нашего кода

Введение: Почему хороший код — это протестированный код

Здравствуйте! На протяжении последних лекций мы с вами строили наше backend-приложение. Оно обрело логику, научилось работать с базой данных, обзавелось системой аутентификации и даже получило красивую "инструкцию по эксплуатации" в виде Swagger-документации. Кажется, что работа сделана. Но как мы можем быть уверены, что всё это действительно работает так, как мы задумали? И, что еще важнее, как нам добавлять новые функции, не ломая то, что уже есть?

До сих пор мы проверяли работу API вручную, через Postman или Swagger UI. Этот подход годится для быстрой проверки "на лету", но он абсолютно не масштабируется и не дает никаких гарантий. Легко что-то забыть проверить, а с ростом проекта количество ручных проверок превращается в часы мучительной и однообразной работы.

Сегодня мы познакомимся с миром **автоматизированного тестирования**. Мы научимся писать код, который проверяет другой код. Для многих начинающих разработчиков это кажется скучной и лишней работой, но я постараюсь убедить вас в обратном. Автоматизированные тесты — это не обуза. Это **"сеть безопасности"**, которая дает разработчику **уверенность и свободу** — свободу проводить рефакторинг, добавлять новый функционал и не бояться, что одно изменение вызовет цепную реакцию ошибок в другой части системы.

Мы разберем классическую **"пирамиду тестирования"**, познакомимся с самым популярным фреймворком для тестирования в JavaScript — **Jest**, и на практике напишем тесты для нашего API.

1. Пирамида тестирования: Разные уровни защиты

Не все тесты одинаковы. Они различаются по своему масштабу, скорости выполнения и стоимости написания. Классическая модель, описывающая их правильное соотношение, — это **Пирамида тестирования**. Она говорит нам, каких тестов должно быть много, а каких — мало.

Уровень 1: Unit-тесты (Модульные тесты) — Основание пирамиды

- **Что тестируют:** Самые маленькие, изолированные "атомы" нашего кода — одну конкретную функцию или метод класса.
- **Принцип:** Весь внешний мир (база данных, другие модули, сеть) для такого теста "подменяется" имитациями — **моками (mocks)**. Мы проверяем логику юнита в полном вакууме.
- **Характеристики:** Они невероятно **быстрые** (выполняются за миллисекунды) и **дешевые** в написании. Их должно быть **очень много**.

Уровень 2: Интеграционные тесты — Середина пирамиды

- **Что тестируют:** Взаимодействие нескольких "атомов" друг с другом. Для нашего backend-приложения это, в первую очередь, **тестирование API эндпоинтов**.

- **Принцип:** Мы отправляем реальный HTTP-запрос на наш сервер и проверяем, что вся цепочка (роутинг, middleware, контроллер, сервис, взаимодействие с БД) отработала корректно и вернула правильный ответ.
- **Характеристики:** Они **медленнее** Unit-тестов, так как задействуют больше компонентов (включая, часто, тестовую базу данных). Они **дороже** в написании, но дают гораздо больше уверенности в том, что система работает как единое целое. Их должно быть **меньше, чем Unit-тестов**.

Уровень 3: End-to-End (E2E) тесты — Верхушка пирамиды

- **Что тестируют:** Всю систему целиком, с точки зрения конечного пользователя.
- **Принцип:** Специальный робот (например, Cypress, Playwright) запускает реальный браузер, открывает наш сайт, нажимает на кнопки, заполняет формы, отправляет запросы на наш бэкенд и проверяет, что на странице появился правильный результат.
- **Характеристики:** Они **очень медленные** (могут идти минуты), **хрупкие** (ломаются от малейших изменений в интерфейсе) и **очень дорогие** в поддержке. Их должно быть **мало**, и они должны проверять только самые критичные бизнес-сценарии (например, "пользователь может зарегистрироваться, войти и оформить заказ").

В рамках нашего курса мы сосредоточимся на двух нижних, самых важных для backend-разработчика уровнях: **Unit** и **Интеграционном**.

2. Jest и Supertest: Наши инструменты

Для написания тестов мы будем использовать два популярных npm-пакета:

- **Jest:** Самый популярный фреймворк для тестирования в JavaScript. Он предоставляет всё необходимое "из коробки": запускалку тестов, структуру (describe, it), библиотеку утверждений (expect) и мощные инструменты для создания "моков".
- **Supertest:** Библиотека, которая позволяет делать HTTP-запросы к нашему Express-приложению программно, **не запуская реальный сервер**. Она работает напрямую с объектом app, что делает интеграционные тесты очень быстрыми.

3. Практика: Тестируем наше API

Давайте напишем тесты для нашего приложения. Для этого мы создадим отдельную тестовую базу данных и настроим Jest так, чтобы он работал именно с ней и очищал ее перед каждым тестом, чтобы тесты не влияли друг на друга.

Шаг 1: Установка и настройка

```
npm install jest supertest --save-dev
```

В package.json мы добавляем скрипт "test": "jest", который будет запускать все наши тесты.

Шаг 2: Тестирование публичного эндпоинта

Давайте напишем простой интеграционный тест для эндпоинта, который не требует аутентификации, например, GET /users.

```
// __tests__/users.test.js
```

```
const request = require('supertest');
const app = require('../app'); // Наше Express-приложение
const { PrismaClient } = require('@prisma/client');
const prisma = new PrismaClient();

describe('Users API', () => {
  // Перед каждым тестом очищаем таблицы
  beforeEach(async () => {
    await prisma.task.deleteMany({});
    await prisma.user.deleteMany({});
  });

  it('should return an empty array if no users exist', async () => {
    const response = await request(app)
      .get('/users')
      .expect('Content-Type', /json/) // Проверяем заголовок
      .expect(200); // Проверяем статус-код

    // Проверяем тело ответа с помощью Jest
    expect(response.body).toBeInstanceOf(Array);
    expect(response.body).toHaveLength(0);
  });

  it('should return a list of users', async () => {
    // Сначала создадим тестового пользователя напрямую в БД
    await prisma.user.create({ data: { email: 'test@test.com', password: '...' }
});

    const response = await request(app).get('/users').expect(200);

    expect(response.body).toHaveLength(1);
    expect(response.body[0].email).toBe('test@test.com');
  });
});
```

Здесь мы используем supertest (request(app)) для отправки запроса и проверки статуса/заголовков, а jest (expect(...)) для детальной проверки тела ответа.

Шаг 3: Тестирование защищенного эндпоинта

Как протестировать роут, который требует JWT? Очень просто: мы должны в тесте симулировать полный цикл — зарегистрировать пользователя, залогиниться, получить токен и затем использовать этот токен для доступа к защищенному ресурсу.

```
// __tests__/tasks.test.js

describe('Tasks API', () => {
  let token; // Переменная для хранения токена

  // Перед тестами регистрируемся и логинимся, чтобы получить токен
  beforeAll(async () => {
    await request(app).post('/auth/register').send({ email: 'taskuser@test.com',
password: 'password' });
    const loginRes = await request(app).post('/auth/login').send({ email:
'taskuser@test.com', password: 'password' });
    token = loginRes.body.token;
  });

  it('should return 401 if no token is provided', async () => {
    // Негативный сценарий: пытаемся создать задачу без токена
    await request(app)
```

```
      .post('/tasks')
      .send({ text: 'This should fail' })
      .expect(401);
  });

  it('should create a task for an authenticated user', async () => {
    // Позитивный сценарий: создаем задачу с токеном
    const response = await request(app)
      .post('/tasks')
      .set('Authorization', `Bearer ${token}`) // <-- Устанавливаем заголовок!
      .send({ text: 'My first protected task' })
      .expect(201);

    expect(response.body.text).toBe('My first protected task');
  });
});
```

4. Unit-тестирование и "Моки"

Интеграционные тесты прекрасны, но что если у нас есть сложная бизнес-логика, не связанная напрямую с HTTP? Например, функция, которая рассчитывает стоимость доставки на основе веса, расстояния и скидок пользователя.

```
// services/deliveryService.js
function calculateDeliveryCost(user, cart) {
  // 100 строк сложной логики...
}
```

Тестировать такую функцию через API — долго и неудобно. Для этого существуют **Unit-тесты**. Мы тестируем эту функцию в **полной изоляции**. Если она зависит от других модулей (например, от модуля для получения тарифов), мы **"мокаем"** (подменяем) эти зависимости, чтобы контролировать их поведение.

```
// services/deliveryService.test.js
const { calculateDeliveryCost } = require('./deliveryService');

// "Мокаем" зависимость
jest.mock('../utils/tariffProvider', () => ({
  getTariff: jest.fn().mockReturnValue(1.5),
}));

it('should calculate cost correctly for VIP user', () => {
  const vipUser = { isVip: true };
  const heavyCart = { weight: 10 };

  const cost = calculateDeliveryCost(vipUser, heavyCart);

  // Проверяем только логику нашей функции, зная, что тариф всегда будет 1.5
  expect(cost).toBe(/* ... ожидаемое значение ... */);
});
```

Заключение

Автоматизированное тестирование — это не опция, а неотъемлемая часть профессиональной разработки. Это ваша "сеть безопасности", которая позволяет разрабатывать быстро и уверенно.

- **Интеграционные тесты** (Jest + Supertest) дают уверенность, что ваше API работает как единое целое.
- **Unit-тесты** (Jest) позволяют детально проверить сложную бизнес-логику в изоляции.

На следующей лекции мы наденем "шляпу хакера" и поговорим о последнем аспекте качественного кода — **безопасности**. Мы разберем самые популярные уязвимости из списка **OWASP Top 10** и научимся защищать наше приложение от них.