САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Бэк-энд разработка

Отчет

Домашняя работа №4

Выполнил:

Корчагин Вадим

Группа
К3341

Проверил:
Добряков Д. И.

Санкт-Петербург

2025 г.

Тема

Платформа для фитнес-тренировок и здоровья.

Задача

- реализовать тестирование API средствами Postman;
- написать тесты внутри Postman.

Технологии:

- Postman
- Swagger/OpenAPI (routing-controllers-openapi)
- openapi2postmanv2
- jq для упаковки коллекции
- Makefile для автоматизации
- REST-сервер на Express + TypeORM + TypeScript

Ход работы

Этап 1. Генерация документации OpenAPI

Этап 2. Генерация Postman коллекции

Этап 3. Написание тестов в Postman

Общие тесты

Вместо того, чтобы расписывать тесты на каждый запрос можно сделать общие тесты на всю коллекцию. Ниже будут приведены тесты, которые могут быть применимы на практике.

1. Этот тест проверяет, что HTTP-ответ находится в допустимом диапазоне: успешный (2xx) или контролируемо ошибочный (400, 401, 403, 404).

```
pm.test("Статус 2xx или ожидаемая ошибка", function () {
  const status = pm.response.code;
  pm.expect(
    (status >= 200 && status <= 299) || [400, 401, 403, 404].includes(status)
  ).to.be.true;
});
```

2. Этот тест гарантирует, что сервер возвращает ответ в ожидаемом формате JSON, а не text/html, text/plain или application/xml.

```
// Проверка, что ответ в формате JSON
pm.test("Ответ в формате JSON", function () {
  pm.response.to.have.header("Content-Type");
  pm.expect(pm.response.headers.get("Content-Type")).to.include("application/json");
});
```

3. Такой универсальный блок проверяет, что:

- Ошибки возвращаются в понятной и стандартизированной структуре (например, { message: "User not found" });
- Успешные ответы не пустые и содержат данные;
- API не возвращает null, "" или странные структуры, которые сломают клиент.

```
try {
  const jsonData = pm.response.json();

  if (pm.response.code >= 400) {
    pm.test("Ответ содержит поле 'message' или 'error'", function () {
      pm.expect(jsonData).to.have.any.keys("message", "error");
    });
  } else {
    if (typeof jsonData === "object" && !Array.isArray(jsonData)) {
      pm.test("Ответ содержит как минимум одно поле", function () {
        pm.expect(Object.keys(jsonData).length).to.be.above(0);
      });
    }

    if (Array.isArray(jsonData)) {
      pm.test("Ответ — массив с элементами", function () {
        pm.expect(jsonData.length).to.be.at.least(0); // разрешён пустой массив
      });
    }
  }
} catch (e) {
}
```

4. Тест измеряет, сколько миллисекунд занял ответ сервера

```
pm.test("Время ответа < 2 сек", function () {
  pm.expect(pm.response.responseTime).to.be.below(2000);
});
```

5. Автоматическая установка токена авторизации

- Проверяет, есть ли в текущем запросе заголовок Authorization.
- Если нет — берёт токен из переменной окружения token.
- Добавляет заголовок Authorization: Bearer <token> ко всем запросам.

Это устраняет необходимость вручную добавлять токен в каждый запрос.

```
1  if (!pm.request.headers.has('Authorization')) {
2    const token = pm.environment.get("token");
3    if (token) {
4      pm.request.headers.add({
5        key: "Authorization",
6        value: "Bearer " + token
7      });
8    }
9  }
```

Тесты: логин (POST /auth/login)

- Проверка HTTP-статуса:
  Убедиться, что логин прошёл успешно и сервер вернул 200 OK.
- Проверка структуры ответа:
- Ответ должен содержать ключ token
- Значение должно быть строкой (токен JWT)

Сохранение токена - полученный токен записывается в переменную окружения token, которую используют другие запросы в заголовке Authorization.

Этот блок должен быть выполнен первым, перед запуском запросов к защищённым API.

```
pm.test("Успешный логин — статус 200", function () {
    pm.response.to.have.status(200);
});

pm.test("Ответ содержит токен", function () {
    const res = pm.response.json();
    pm.expect(res).to.have.property("token");
    pm.expect(res.token).to.be.a("string");
});

pm.environment.set("token", pm.response.json().token);
```

Тесты /user

1. GET /users/id/:id

Проверяет успешный статус ответа (200 OK), то есть пользователь с указанным ID найден.

Проверяет содержимое тела ответа, что объект содержит ключи id, email и name.

```
pm.test("Пользователь получен", function () {
  pm.response.to.have.status(200);
});

pm.test("Поля пользователя присутствуют", function () {
  const user = pm.response.json();
  pm.expect(user).to.have.property("email");
  pm.expect(user).to.have.property("name");
  pm.expect(user).to.have.property("id");
});
```

2. GET /users/email/:email

Проверяет, что пользователь успешно получен по email и сервер вернул статус 200.

```
pm.test("Получение пользователя по email", function () {
  pm.response.to.have.status(200);
});
```

3. PATCH /users/:id

Проверяет HTTP-статус, что пользователь успешно обновлён (200 OK).

Проверяет, что в теле ответа пришло обновлённое имя - сравнивает значение поля name в ответе с ожидаемым ("Updated User").

```
pm.test("Пользователь обновлён", function () {
  pm.response.to.have.status(200);
});

pm.test("Имя обновлено", function () {
  const user = pm.response.json();
  pm.expect(user.name).to.eql("Updated User");
});
```

4. POST /users

Проверяет статус ответа, как 200 OK, так и 201 Created.

Проверяет, что тело ответа содержит ключ id, это подтверждает, что пользователь действительно был сохранён и сериализован обратно, сохраняет id нового пользователя в переменную окружения user_id, это позволяет автоматически использовать user_id в последующих запросах (например, GET, PATCH, DELETE /users/:id).

```
pm.test("Пользователь создан", function () {
  pm.expect(pm.response.code).to.be.oneOf([200, 201]);
});

const res = pm.response.json();
pm.test("Ответ содержит ID пользователя", function () {
  pm.expect(res).to.have.property("id");
});

pm.environment.set("user_id", res.id);
```

Результаты

- POST User login 6 | 0
  - Pass Статус 2xx или ожидаемая ошибка
  - Pass Ответ в формате JSON
  - Pass Ответ содержит как минимум одно поле
  - Pass Время ответа < 2 сек
  - Pass Успешный логин — статус 200
  - Pass Ответ содержит токен
- GET Get user by ID 6 | 0
  - Pass Статус 2xx или ожидаемая ошибка
  - Pass Ответ в формате JSON
  - Pass Ответ содержит как минимум одно поле
  - Pass Время ответа < 2 сек
  - Pass Пользователь получен
  - Pass Поля пользователя присутствуют
- GET Get user by email 5 | 0
- PATCH Update a user 6 | 0
  - Pass Статус 2xx или ожидаемая ошибка
  - Pass Ответ в формате JSON
  - Pass Ответ содержит как минимум одно поле
  - Pass Время ответа < 2 сек
  - Pass Пользователь обновлён
  - Pass Имя обновлено
- DELETE Delete a user 6 | 0
- GET Get all users 4 | 0
- POST Create a new user 3 | 3 ✕
  - Fail Статус 2xx или ожидаемая ошибка ✕
  - Pass Ответ в формате JSON
  - Pass Ответ содержит поле 'message' или 'error'
  - Pass Время ответа < 2 сек
  - Fail Пользователь создан ✕
  - Fail Ответ содержит ID пользователя ✕

- GET Get role by ID — 4 | 0
  - Pass Статус 2xx или ожидаемая ошибка
  - Pass Ответ в формате JSON
  - Pass Ответ содержит как минимум одно поле
  - Pass Время ответа < 2 сек
- PATCH Update a role — 4 | 0
- DELETE Delete a role — 4 | 0
- GET Get all roles — 4 | 0
- POST Create a new role — 3 | 1
  - Fail Статус 2xx или ожидаемая ошибка
  - Pass Ответ в формате JSON
  - Pass Ответ содержит поле 'message' или 'error'
  - Pass Время ответа < 2 сек

Выводы

В ходе выполнения задания были разработаны и протестированы автоматические сценарии для API платформы, реализованной на Express + TypeORM + TypeScript.

Было выяснено, как писать тесты для коллекции и для отдельных запросов. Были сделаны тесты для login, users и общие тесты для всех ручек.

По результатам можно увидеть корректную работу тестов, где одни проходят, другие - нет. (тесты, которые словили ошибки, вызваны неправильными параметрами запроса).