МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

по выполнению практических работ
по дисциплине «Введение в бэкенд разработку» для студентов направления подготовки 09.03.04 Программная инженерия
Направленность (профиль)
«Разработка и сопровождение программного обеспечения»

Ставрополь 2026

Содержание

Введение 3
Лабораторная работа № 1 10
Лабораторная работа № 2 31
Лабораторная работа № 3 51
Лабораторная работа № 4 87
Лабораторная работа № 5 120
Лабораторная работа № 6 157
Лабораторная работа № 7 196
Лабораторная работа № 8 239
Лабораторная работа № 9 285
Лабораторная работа № 10 309
Лабораторная работа № 11 347
Лабораторная работа № 12 378
Лабораторная работа № 13 415
Лабораторная работа № 14 447
Лабораторная работа № 15 478
Лабораторная работа № 16 507
Лабораторная работа № 17 544
Лабораторная работа № 18 571

Введение

Лабораторный практикум является важнейшей составной частью дисциплины «Введение в бэкенд-разработку», изучаемой на 2 курсе в 3 семестре. Практические занятия направлены на формирование у вас навыков разработки серверной части веб-приложений, необходимых для дальнейшего успешного освоения дисциплины «Разработка Web-приложений» и последующих профессиональных курсов.

Цель лабораторного практикума:

Закрепить теоретические знания, полученные на лекциях, и сформировать практические компетенции в области:

- разработки серверных приложений на Node.js с использованием фреймворка Express;
- проектирования и работы с реляционными базами данных SQLite;
- создания REST API с поддержкой полного цикла CRUD-операций;
- реализации аутентификации пользователей (Basic Auth, сессии);
- валидации входных данных и обработки ошибок;
- логирования запросов и диагностики приложения;
- интеграции серверной части с простым фронтендом;
- развёртывания приложения на облачных платформах.

Структура лабораторных работ

Лабораторный практикум состоит из 18 работ, каждая из которых посвящена определённой теме и логически продолжает предыдущую. Работы построены по принципу «от простого к сложному»:

№	Тема	Уровень сложности
	Настройка среды и первый HTTP-сервер	Начальный
	Обработка HTTP-методов (GET, POST, PUT, DELETE)	Начальный
	Маршрутизация: параметры пути и query-параметры	Начальный
	Основы SQL: создание таблиц и наполнение данными	Начальный
	Выполнение SELECT-запросов из серверного кода	Базовый
	Полный CRUD с базой данных (INSERT, UPDATE, DELETE)	Базовый
	Фильтрация, поиск и сортировка данных в SQL	Базовый
	Связи таблиц: один-ко-многим (JOIN)	Базовый
	Проектирование REST API для предметной области	Продвинутый
0	Валидация входных данных и обработка ошибок	Продвинутый
1	Хеширование паролей и базовая аутентификация	Продвинутый

№	Тема	Уровень сложности
2	Сессионная аутентификация (в памяти сервера)	Продвинутый
3	Переменные окружения и конфигурация	Продвинутый
4	Логирование запросов и ошибок	Продвинутый
5	Тестирование API с помощью Postman	Продвинутый
6	Интеграция простого фронтенда с бэкендом	Продвинутый
7	Деплой бэкенд-приложения на облачную платформу	Продвинутый
8	Итоговая комплексная работа	Завершающий

Структура описания лабораторной работы

Каждая лабораторная работа имеет единую структуру, что облегчает её выполнение и проверку:

Тема работы - краткое название, отражающее содержание.

Цель работы - что именно вы должны освоить в ходе выполнения.

Задачи работы - конкретные шаги, которые необходимо выполнить.

Теоретическое обоснование - краткий обзор теоретических понятий, необходимых для выполнения работы.

Методика выполнения работы - подробное описание с примерами кода и их разбором.

Индивидуальные задания - 15 вариантов для каждого из трёх уровней сложности:

Базовый уровень - выполнение базовых требований (оценка «удовлетворительно»);

Средний уровень - включает требования базового уровня и дополнительные задания (оценка «хорошо»);

Продвинутый уровень - включает требования среднего уровня и задания повышенной сложности (оценка «отлично»).

Контрольные вопросы - для самопроверки и подготовки к защите.

Требования к отчёту - структура и содержание отчёта о выполненной работе.

Критерии оценивания - чёткие критерии для каждого уровня сложности.

Список литературы и интернет-ресурсов - источники для самостоятельного изучения.

Порядок выполнения лабораторных работ

Ознакомление - внимательно прочитайте теоретическое обоснование и примеры.

Выбор уровня - определите, какой уровень сложности вы будете выполнять (базовый, средний или продвинутый). Выбор уровня определяет максимальную оценку.

Выбор варианта - получите номер варианта у преподавателя. Вариант определяет предметную область вашего задания (книги, товары, студенты и т.д.).

Выполнение - напишите код в соответствии с требованиями вашего уровня и варианта.

Тестирование - проверьте работоспособность приложения с помощью Postman или браузера.

Оформление отчёта - подготовьте отчёт по установленной структуре.

Защита - продемонстрируйте работу преподавателю, объясните код и ответьте на контрольные вопросы.

Система оценивания

Каждая лабораторная работа оценивается по трём уровням сложности:

Уровень	Требования	Оценка
Базовый	Выполнение минимальных требований работы	Удовлетворительно (3)
Средний	Выполнение базового уровня + дополнительные задания	Хорошо (4)
Продвинутый	Выполнение среднего уровня + задания повышенной сложности	Отлично (5)

Важно: выбор уровня сложности не ограничивает вас в получении более высокой оценки - вы можете выполнить работу полностью на продвинутом уровне и претендовать на отличную оценку.

Требования к отчёту

Каждая лабораторная работа должна быть оформлена в виде электронного отчёта, содержащего:

Титульный лист (название работы, ФИО, группа, вариант).

Цель и задачи работы.

Теоретическое обоснование (краткий конспект по теме).

Ход выполнения работы (листинг кода с комментариями, скриншоты результатов).

Ответы на контрольные вопросы.

Заключение (выводы о проделанной работе).

Список использованных источников.

Отчёт предоставляется в электронном виде (PDF, DOCX или Markdown) и защищается устно.

Рекомендации по выполнению

Не копируйте код без понимания - старайтесь разобраться в каждом фрагменте кода. Копирование без осмысления не позволит вам успешно защитить работу.

Используйте систему контроля версий (Git) - сохраняйте каждый этап работы в репозитории. Это поможет вам отслеживать изменения и при необходимости вернуться к предыдущей версии.

Тестируйте постепенно - не пишите весь код сразу. Реализуйте один эндпоинт, протестируйте его, затем переходите к следующему.

Обращайте внимание на ошибки - изучайте сообщения об ошибках, они часто содержат подсказки о том, что пошло не так.

Ведите собственный конспект - записывайте важные моменты, типичные ошибки и способы их решения.

Не стесняйтесь задавать вопросы - если что-то непонятно, обратитесь к преподавателю или сокурсникам.

Инструментарий

Для выполнения лабораторных работ вам потребуется:

Инструмент	Назначение	Ссылка для скачивания
Node.js	Среда выполнения JavaScript на сервере	https://nodejs.org/
Visual Studio Code	Редактор кода	https://code.visualstudio.com/
Postman	Тестирование API	https://www.postman.com/
DB Browser for SQLite	Работа с базами данных SQLite	https://sqlitebrowser.org/
Git	Система контроля версий	https://git-scm.com/

Все инструменты являются бесплатными и доступны для установки на территории Российской Федерации.

Успешное выполнение всех лабораторных работ позволит вам не только получить зачёт по дисциплине, но и сформировать прочную базу для дальнейшего изучения бэкенд-разработки, включая промышленные фреймворки (Django, Spring), работу с облачными сервисами и современные практики DevOps.

Лабораторная работа № 1

Тема: Настройка среды разработки и первый HTTP-сервер

Цель работы:

Освоить установку и настройку инструментов для бэкенд-разработки, создать и запустить минимальный веб-сервер, обрабатывающий HTTP-запросы.

Задачи работы:

1. Установить необходимое программное обеспечение для разработки на Node.js.
2. Создать первый сервер на Express.js с обработкой GET-запросов.
3. Освоить отправку ответов в форматах текст и JSON.
4. Научиться запускать сервер и проверять его работу через браузер и Postman.
5. Настроить автоматический перезапуск сервера при изменениях кода.

1. Теоретическое обоснование

1.1. Что такое веб-сервер?

Веб-сервер — это программное обеспечение, которое принимает HTTP-запросы от клиентов (обычно веб-браузеров) и возвращает HTTP-ответы. В контексте бэкенд-разработки сервер выполняет бизнес-логику, обращается к базам данных и формирует ответы в понятном клиенту формате.

1.2. Node.js и его роль в бэкенде

Node.js — это среда выполнения JavaScript на сервере, построенная на движке V8 от Google. Ключевые особенности:

Асинхронная, событийно-ориентированная модель — позволяет обрабатывать тысячи одновременных соединений без создания отдельных потоков.

Единый язык на фронтенде и бэкенде — JavaScript используется везде.

Богатая экосистема npm — тысячи готовых библиотек.

1.3. Express.js — веб-фреймворк

Express.js — минималистичный и гибкий веб-фреймворк для Node.js, который предоставляет:

Удобную маршрутизацию (обработка разных URL и HTTP-методов).

Middleware-функции для обработки запросов.

Простые способы отправки ответов в форматах text, JSON, HTML.

Express позволяет быстро создать сервер без необходимости низкоуровневой работы с модулем http.

1.4. Порт и localhost

Порт - числовой идентификатор, по которому операционная система направляет сетевой трафик конкретному приложению. Веб-серверы обычно используют порт 3000, 5000, 8000 или 8080.

localhost (127.0.0.1) - специальное сетевое имя, указывающее на текущий компьютер. При разработке сервер запускается на localhost, и доступ к нему осуществляется через браузер по адресу http://localhost:порт.

2. Методика выполнения работы

2.1. Примеры и их подробный разбор

Пример 1. Установка Node.js и проверка версии

Node.js можно скачать с официального сайта https://nodejs.org/. Рекомендуется использовать LTS-версию.

После установки открыть терминал (командную строку) и выполнить команды:

bash

```
node --version
npm --version
```

Ожидаемый вывод:

text

```
v18.17.0
9.6.7
```

Разбор: node --version показывает установленную версию Node.js, npm --version — версию менеджера пакетов npm.

Пример 2. Создание папки проекта и инициализация

В терминале выполнить:

bash

```
mkdir my-first-server
cd my-first-server
npm init -y
```

Разбор:

mkdir - создаёт папку проекта.

cd - переходит в неё.

npm init -y - создаёт файл package.json с настройками по умолчанию. Этот файл хранит информацию о проекте и его зависимостях.

Пример 3. Установка Express.js

bash

```
npm install express
```

Разбор: Команда загружает пакет express из репозитория npm и добавляет его в секцию dependencies файла package.json. Также создаётся папка node_modules, где хранятся все установленные библиотеки.

Пример 4. Простейший сервер на Express

Создать файл app.js со следующим содержанием :

javascript

```
// Подключаем библиотеку express
const express = require('express');

// Создаём экземпляр приложения
const app = express();

// Определяем порт, на котором будет работать сервер
```

```
const PORT = 3000;

// Обработчик GET-запроса на корневой маршрут '/'
app.get('/', (req, res) => {
    res.send('Hello, Backend World!');
});

// Запускаем сервер
app.listen(PORT, () => {
    console.log(`Сервер запущен на http://localhost:${PORT}`);
});
```

Подробный разбор кода:

Строка	Описание
const express = require('express');	Импортируем установленный пакет express для использования в коде.
const app = express();	Создаём объект приложения — главный объект Express, через который мы настраиваем маршруты и middleware.
const PORT = 3000;	Объявляем константу с номером порта.
app.get('/', (req, res) => { ... })	Регистрируем обработчик для GET-запросов по пути /. Функция-обработчик принимает два параметра: req (объект запроса) и res (объект ответа).
res.send('Hello, Backend World!');	Отправляем текстовый ответ клиенту.

Строка	Описание
app.listen(PORT, () => { ... })	Запускаем сервер на указанном порту. Второй параметр — callback-функция, которая выполнится после успешного запуска.

Пример 5. Запуск сервера

В терминале, находясь в папке проекта, выполнить:

bash

```
node app.js
```

В браузере открыть http://localhost:3000 — должна отобразиться надпись "Hello, Backend World!".

Пример 6. Отправка JSON-ответа

Изменим обработчик корневого маршрута, чтобы он возвращал JSON:

javascript

```
app.get('/', (req, res) => {
    res.json({
        message: 'Сервер работает!',
        status: 'ok',
        timestamp: new Date().toISOString()
    });
});
```

Разбор: Метод res.json() автоматически преобразует JavaScript-объект в JSON-строку и устанавливает правильный заголовок Content-Type: application/json.

Пример 7. Установка Nodemon для автоматического перезапуска

Nodemon — утилита, которая отслеживает изменения файлов и автоматически перезапускает сервер .

Установка:

bash

```
npm install --save-dev nodemon
```

Добавление скрипта в package.json:

json

```
"scripts": {
  "start": "node app.js",
  "dev": "nodemon app.js"
}
```

Запуск в режиме разработки:

bash

```
npm run dev
```

Пример 8. Создание нескольких маршрутов

javascript

```
// Главная страница
app.get('/', (req, res) => {
  res.send('Добро пожаловать на главную страницу!');
});

// Страница "О нас"
app.get('/about', (req, res) => {
  res.send('Это учебный сервер для изучения бэкенда');
});

// API-эндпоинт, возвращающий список пользователей
app.get('/api/users', (req, res) => {
  const users = [
    { id: 1, name: 'Иван' },
    { id: 2, name: 'Мария' }
  ];
  res.json(users);
```

});

2.2. Индивидуальные задания

2.2.1. Общее задание для всех вариантов

Независимо от варианта, студент должен:

Создать папку проекта и инициализировать package.json.

Установить Express и nodemon.

Реализовать все маршруты, указанные в варианте.

Запустить сервер на порту 3000.

Проверить работу через браузер и/или Postman.

2.2.2. Базовый уровень (15 вариантов)

Для получения оценки «удовлетворительно» необходимо реализовать минимум 3 маршрута (из предложенных в варианте) с текстовыми ответами.

Вариант	Тематика	Маршруты для реализации
1	Библиотека	/ — приветствие; /books — список книг; /authors — список авторов
2	Магазин	/ — приветствие; /products — список товаров; /categories — список категорий
3	Университет	/ — приветствие; /students — список студентов; /teachers — список преподавателей
4	Кинотеатр	/ — приветствие; /movies —

Вариант	Тематика	Маршруты для реализации
		список фильмов; /halls — список залов
5	Клиника	/ — приветствие; /doctors — список врачей; /services — список услуг
6	Ресторан	/ — приветствие; /menu — список блюд; /staff — список сотрудников
7	Спортклуб	/ — приветствие; /trainings — список тренировок; /coaches — список тренеров
8	IT-компания	/ — приветствие; /employees — список сотрудников; /projects — список проектов
9	Гостиница	/ — приветствие; /rooms — список номеров; /services — список доп. услуг
10	Автосалон	/ — приветствие; /cars — список автомобилей; /brands — список брендов
11	Школа	/ — приветствие; /classes — список классов; /subjects — список

Вариант	Тематика	Маршруты для реализации
		предметов
12	Банк	/ — приветствие; /accounts — список счетов; /services — список банковских услуг
13	Турфирма	/ — приветствие; /tours — список туров; /countries — список стран
14	Фитнес-центр	/ — приветствие; /exercises — список упражнений; /trainers — список тренеров
15	Аптека	/ — приветствие; /medicines — список лекарств; /manufacturers — список производителей

Пример выполнения для варианта 1 (Библиотека):

javascript

```
const express = require('express');
const app = express();
const PORT = 3000;

app.get('/', (req, res) => {
    res.send('Добро пожаловать в библиотеку!');
});
```

```
app.get('/books', (req, res) => {
    res.send('Список книг: Война и мир, Преступление и наказание, Мастер и Маргарита');
});

app.get('/authors', (req, res) => {
    res.send('Список авторов: Толстой, Достоевский, Булгаков');
});

app.listen(PORT, () => {
    console.log(`Сервер запущен на http://localhost:${PORT}`);
});
```

2.2.3. Средний уровень (15 вариантов)

1. Для получения оценки «хорошо» необходимо дополнительно к базовому уровню:
2. Реализовать JSON-ответы с структурированными данными (массивы объектов).
3. Добавить обработку ошибок (например, 404 для несуществующих маршрутов).

Вариант	JSON-структура для маршрута /items
1	[{ "id": 1, "title": "Война и мир", "author": "Толстой", "year": 1869 }]
2	[{ "id": 1, "name": "Ноутбук", "price": 50000, "category": "Электроника" }]
3	[{ "id": 1, "fullName": "Иванов Иван", "group": "ИВТ-21", "averageGrade": 4.5 }]

Вариант	JSON-структура для маршрута /items
4	[{ "id": 1, "title": "Побег из Шоушенка", "genre": "Драма", "duration": 142 }]
5	[{ "id": 1, "fullName": "Петрова Анна", "specialization": "Терапевт", "experience": 5 }]
6	[{ "id": 1, "name": "Борщ", "price": 250, "category": "Первое" }]
7	[{ "id": 1, "name": "Йога", "duration": 60, "maxParticipants": 15 }]
8	[{ "id": 1, "name": "Иванов Петр", "position": "Разработчик", "salary": 100000 }]
9	[{ "id": 1, "number": 101, "type": "Люкс", "price": 5000 }]
10	[{ "id": 1, "model": "Tesla Model S", "year": 2023, "price": 7000000 }]
11	[{ "id": 1, "grade": 10, "letter": "A", "studentsCount": 25 }]
12	[{ "id": 1, "type": "Дебетовый", "balance": 15000, "currency": "RUB" }]
13	[{ "id": 1, "country": "Турция", "price": 50000,

Вариант	JSON-структура для маршрута /items
	"duration": 7 }]
14	[{ "id": 1, "name": "Приседания", "calories": 200, "difficulty": "Средний" }]
15	[{ "id": 1, "name": "Парацетамол", "price": 50, "prescription": false }]

Обязательный элемент для среднего уровня: обработчик для всех несуществующих маршрутов:

javascript

```
// Обработчик 404 — должен быть последним
app.use((req, res) => {
  res.status(404).json({ error: 'Маршрут не найден' });
});
```

Пример выполнения для варианта 1 (Средний уровень):

javascript

```
const express = require('express');
const app = express();
const PORT = 3000;

app.get('/', (req, res) => {
  res.send('Добро пожаловать в библиотеку!');
});

app.get('/books', (req, res) => {
  const books = [
```

```
        { id: 1, title: 'Война и мир', author: 'Толстой', year: 1869 },
        { id: 2, title: 'Преступление и наказание', author: 'Достоевский', year: 1866 },
        { id: 3, title: 'Мастер и Маргарита', author: 'Булгаков', year: 1967 }
    ];
    res.json(books);
});

app.get('/authors', (req, res) => {
    const authors = [
        { id: 1, name: 'Толстой', birthYear: 1828 },
        { id: 2, name: 'Достоевский', birthYear: 1821 },
        { id: 3, name: 'Булгаков', birthYear: 1891 }
    ];
    res.json(authors);
});

// Обработка ошибок для несуществующих маршрутов
app.use((req, res) => {
    res.status(404).json({ error: 'Страница не найдена' });
});

app.listen(PORT, () => {
    console.log(`Сервер запущен на http://localhost:${PORT}`);
});
```

2.2.4. Продвинутый уровень (15 вариантов)

Для получения оценки «отлично» необходимо дополнительно к предыдущим уровням:

1. Настроить переменные окружения через .env для хранения порта .

2. Реализовать логирование запросов с использованием библиотеки morgan .
3. Добавить динамический маршрут с параметром (например, /items/:id).

Вариант	**Дополнительный динамический маршрут**	**Формат ответа**
1	GET /books/:id	{ "id": 1, "title": "...", "author": "..." }
2	GET /products/:id	{ "id": 1, "name": "...", "price": 50000 }
3	GET /students/:id	{ "id": 1, "fullName": "...", "group": "..." }
4	GET /movies/:id	{ "id": 1, "title": "...", "genre": "..." }
5	GET /doctors/:id	{ "id": 1, "fullName": "...", "specialization": "..." }
6	GET /menu/:id	{ "id": 1, "name": "...", "price": 250 }
7	GET /trainings/:id	{ "id": 1, "name": "...", "duration": 60 }
8	GET /employees/:id	{ "id": 1, "name": "...", "position": "..." }
9	GET /rooms/:id	{ "id": 1, "number": 101, "type":

Вариант	Дополнительный динамический маршрут	Формат ответа
		"Люкс" }
10	GET /cars/:id	{ "id": 1, "model": "Tesla Model S", "year": 2023 }
11	GET /classes/:id	{ "id": 1, "grade": 10, "letter": "A" }
12	GET /accounts/:id	{ "id": 1, "type": "Дебетовый", "balance": 15000 }
13	GET /tours/:id	{ "id": 1, "country": "Турция", "price": 50000 }
14	GET /exercises/:id	{ "id": 1, "name": "Приседания", "calories": 200 }
15	GET /medicines/:id	{ "id": 1, "name": "Парацетамол", "price": 50 }

Пример выполнения для варианта 1 (Продвинутый уровень):

javascript

```
// Подключение модулей
require('dotenv').config();  // Загрузка переменных из .env
const express = require('express');
const morgan = require('morgan');  // Логирование

const app = express();
const PORT = process.env.PORT || 3000;  // Переменная окружения
```

```
// Логирование всех запросов
app.use(morgan('combined'));

app.get('/', (req, res) => {
  res.send('Добро пожаловать в библиотеку!');
});

app.get('/books', (req, res) => {
  const books = [
    { id: 1, title: 'Война и мир', author: 'Толстой', year: 1869 },
    { id: 2, title: 'Преступление и наказание', author: 'Достоевский', year: 1866 },
    { id: 3, title: 'Мастер и Маргарита', author: 'Булгаков', year: 1967 }
  ];
  res.json(books);
});

// Динамический маршрут для получения книги по ID
app.get('/books/:id', (req, res) => {
  const id = parseInt(req.params.id);
  const books = {
    1: { id: 1, title: 'Война и мир', author: 'Толстой', year: 1869 },
    2: { id: 2, title: 'Преступление и наказание', author: 'Достоевский', year: 1866 },
    3: { id: 3, title: 'Мастер и Маргарита', author: 'Булгаков', year: 1967 }
  };

  if (books[id]) {
```

```
        res.json(books[id]);
    } else {
        res.status(404).json({ error: 'Книга не найдена' });
    }
});

app.get('/authors', (req, res) => {
    res.json([
        { id: 1, name: 'Толстой', birthYear: 1828 },
        { id: 2, name: 'Достоевский', birthYear: 1821 },
        { id: 3, name: 'Булгаков', birthYear: 1891 }
    ]);
});

// Обработка ошибок
app.use((req, res) => {
    res.status(404).json({ error: 'Маршрут не найден' });
});

app.listen(PORT, () => {
    console.log(`Сервер запущен на http://localhost:${PORT}`);
});
```

Создание файла .env:

text

PORT=3000

Создание файла .gitignore (чтобы не загружать секреты и зависимости в репозиторий) :

text

node_modules/

.env

Обновление package.json (скрипты):

json

```
"scripts": {
  "start": "node app.js",
  "dev": "nodemon app.js"
}
```

Установка дополнительных пакетов для продвинутого уровня:

bash

```
npm install morgan dotenv
```

3. Контрольные вопросы

1. Что такое Node.js и для чего он используется?
2. Какую роль выполняет Express.js в веб-разработке?
3. Что такое порт и зачем он нужен при запуске сервера?
4. Чем отличается метод res.send() от res.json()?
5. Какие коды состояния HTTP вы знаете? Что означают 200, 404, 500?
6. Для чего нужен Nodemon? В чём его отличие от обычного запуска через node?
7. Что такое localhost и почему для разработки используется именно он?
8. Как обработать запрос к несуществующему маршруту?
9. Как через req.params получить параметр из URL (например, /users/5)?
10. Зачем нужны переменные окружения и файл .env?

4. Требования к отчёту

Отчёт должен быть оформлен в электронном виде и содержать:

1. Титульный лист с названием работы, ФИО студента, номером группы, вариантом.
2. Цель и задачи работы (переписать из методических указаний).
3. Теоретическое обоснование (краткий конспект по теме).
4. Ход выполнения работы:
5. Список установленного ПО с версиями.

6. Листинг кода с комментариями.
7. Скриншоты запуска сервера в терминале.
8. Скриншоты проверки работы через браузер (для каждого маршрута).
9. Скриншоты проверки работы через Postman (для JSON-маршрутов).
10. Ответы на контрольные вопросы (письменно, развёрнуто).
11. Заключение (выводы о проделанной работе, какие навыки приобретены).
12. Список использованных источников.

5. Критерии оценивания

Уровень выполнения	Требования	Оценка
Базовый	Реализованы 3 маршрута с текстовыми ответами. Сервер запускается и отвечает на запросы.	Удовлетворительно
Средний	Выполнены требования базового уровня + реализованы JSON-ответы с массивами объектов + добавлен обработчик 404.	Хорошо
Продвинутый	Выполнены требования среднего уровня + настроены переменные окружения (.env) + подключено логирование	Отлично

Уровень выполнения	Требования	Оценка
	(morgan) + реализован динамический маршрут с параметром /:id.	

Дополнительные условия:

1. Код должен быть отформатирован и снабжён комментариями.
2. Отчёт должен быть оформлен аккуратно, без грамматических ошибок.
3. При защите студент должен объяснить код и ответить на контрольные вопросы.

6. Список рекомендуемой литературы и интернет-ресурсов

Литература (доступна в РФ)

1. Браун, И. (2021). *Express.js: Разработка веб-приложений на Node.js*. ДМК Пресс. — ISBN: 978-5-97060-945-7.
2. Кантелон, М., Хартер, М., & Холстейн, С. (2020). *Node.js в действии*. Питер. — ISBN: 978-5-4461-1524-2.
3. Чиннатамби, К. (2019). *Express.js: Создание веб-приложений*. БХВ-Петербург. — ISBN: 978-5-9775-4176-7.

Интернет-ресурсы

Название	Ссылка	Описание
Официальная документация Node.js	https://nodejs.org/ru/docs/	Документация на русском языке, руководства, API
Официальная	https://expressjs.com/ru/	Документация Express на

Название	Ссылка	Описание
документация Express.js		русском языке
Документация Nodemon	https://nodemon.io/	Инструмент для автоматического перезапуска сервера
Документация Morgan	https://github.com/expressjs/morgan	HTTP-логгер для Express
Документация Dotenv	https://github.com/motdotla/dotenv	Библиотека для работы с переменными окружения
Metanit — Node.js	https://metanit.com/web/nodejs/	Учебник по Node.js на русском языке
HTML Academy — Express	https://htmlacademy.ru/blog/express	Статьи по Express.js
Postman	https://learning.postman.com/	Официальна

Название	Ссылка	Описание
Learning Center		я документация Postman (англ.)

Лабораторная работа № 2

Тема: Обработка HTTP-методов (GET, POST, PUT, DELETE). Работа с массивами в памяти сервера

Цель работы:

Научиться обрабатывать различные HTTP-методы (GET, POST, PUT, DELETE) для реализации CRUD-операций с хранением данных в памяти сервера.

Задачи работы:

1. Реализовать обработку GET-запросов для получения всех элементов и одного элемента по ID.
2. Реализовать обработку POST-запросов для создания новых элементов.
3. Реализовать обработку PUT-запросов для обновления существующих элементов.
4. Реализовать обработку DELETE-запросов для удаления элементов.
5. Освоить работу с массивом объектов как с хранилищем данных.
6. Изучить правильные коды HTTP-ответов для различных операций.

1. Теоретическое обоснование

1.1. HTTP-методы и CRUD-операции

В REST API каждому типу операции с данными соответствует определённый HTTP-метод:

HTTP-метод	CRUD операция	Описание	Идемпотентность
GET	Read	Получение данных	Да
POS	Create	Создание	Нет

HTTP-метод	CRUD операция	Описание	Идемпотентность
T		нового ресурса	
PUT	Update (замена)	Полное обновление ресурса	Да
PATCH	Update (частичное)	Частичное обновление ресурса	Нет
DELETE	Delete	Удаление ресурса	Да

Идемпотентность - свойство операции, при котором повторное выполнение того же запроса даёт тот же результат, что и первое выполнение.

1.2. Обработка POST-запросов

POST-запросы обычно содержат тело (body) с данными для создания нового ресурса. Для их обработки в Express необходимо использовать middleware express.json() для автоматического парсинга JSON-данных:

```
javascript
app use(express json()); // Подключаем парсер JSON

app post("/api/users", (req, res) => {
    const userName = req body name;
    const userAge = req body age;
    // Создание нового пользователя...
});
```

1.3. Параметры маршрута (Route Parameters)

Для получения одного элемента по идентификатору используются динамические параметры в URL. Параметры обозначаются двоеточием ::

```
javascript
// Параметр :id будет доступен как req.params.id
app get("/api/users/:id", (req, res) => {
    const id = req params id;
    // Поиск пользователя по id...
});
```

1.4. Обработка PUT-запросов

PUT-запрос обновляет существующий ресурс. Обычно идентификатор передаётся в параметрах маршрута, а новые данные — в теле запроса:

```
javascript
app put("/api/users/:id", (req, res) => {
```

```
    const id = req.params.id;
    const newName = req.body.name;
    const newAge = req.body.age;
    // Обновление пользователя...
});
```

1.5. Обработка DELETE-запросов

DELETE-запрос удаляет ресурс по идентификатору:

javascript

```
app.delete("/api/users/:id", (req, res) => {
    const id = req.params.id;
    // Удаление пользователя...
});
```

1.6. Коды HTTP-ответов для CRUD-операций

Операция	Успешный код	Описание
GET (один элемент)	200 OK	Элемент найден и возвращён
GET (элемент не найден)	404 Not Found	Элемент с таким ID отсутствует
POST (создание)	201 Created	Ресурс успешно создан
PUT (обновление)	200 OK	Ресурс успешно обновлён
PUT (ресурс не найден)	404 Not Found	Обновляемый ресурс отсутствует
DELETE (удаление)	204 No Content или 200 OK	Ресурс успешно удалён
Некорректные данные	400 Bad Request	Тело запроса отсутствует или невалидно

2. Методика выполнения работы

2.1. Примеры и их подробный разбор

Пример 1. Базовая структура приложения с парсером JSON

javascript

```
const express = require('express');
const app = express();
const PORT = 3000;

// Middleware для автоматического парсинга JSON в теле запроса
app.use(express.json());

// Массив для хранения данных (временное хранилище в памяти)
const items = [];

app.listen(PORT, () => {
  console.log(`Сервер запущен на http://localhost:${PORT}`);
});
```

Разбор кода:

app.use(express.json()) — подключает встроенный middleware, который преобразует JSON из тела запроса в JavaScript-объект, доступный через req.body.

const items = [] — массив-хранилище. Данные будут существовать только пока работает сервер.

Пример 2. GET-запросы: получение всех элементов и одного по ID

javascript

```
// Получение всех элементов
app.get('/api/items', (req, res) => {
  res.json(items);
});

// Получение одного элемента по ID
app.get('/api/items/:id', (req, res) => {
  const id = parseInt(req.params.id);
  const item = items.find(item => item.id === id);

  if (item) {
    res.json(item);
  } else {
    res.status(404).json({ error: 'Элемент не найден' });
  }
});
```

Разбор кода:

parseInt(req.params.id) — преобразует строковый параметр ID в число, так как параметры маршрута всегда приходят как строки.

items.find() — поиск элемента в массиве по id.

res.status(404).json(...) — установка кода ответа 404 и отправка JSON с сообщением об ошибке.

Пример 3. POST-запрос: создание нового элемента

javascript

```
app.post('/api/items', (req, res) => {
    // Проверка наличия тела запроса
    if (!req.body || Object.keys(req.body).length === 0) {
        return res.status(400).json({ error: 'Тело запроса не может быть пустым' });
    }

    const { name, price } = req.body;

    // Валидация обязательных полей
    if (!name) {
        return res.status(400).json({ error: 'Поле "name" обязательно' });
    }

    // Создание нового элемента
    const newItem = {
        id: items.length + 1,  // Простая генерация ID
        name: name,
        price: price || 0
    };

    items.push(newItem);
    res.status(201).json(newItem);  // 201 Created
});
```

Разбор кода:

const { name, price } = req.body — деструктуризация объекта для извлечения полей.

Проверка на наличие обязательных полей — если name отсутствует, возвращается ошибка 400.

items.length + 1 — простой способ генерации ID (для учебных целей).

res.status(201).json(...) — установка статуса 201 (Created) при успешном создании.

Пример 4. PUT-запрос: обновление элемента

javascript

```
app.put('/api/items/:id', (req, res) => {
    const id = parseInt(req.params.id);

    if (!req.body) {
        return res.status(400).json({ error: 'Тело запроса не может быть пустым' });
    }
```

```
    const index = items.findIndex(item => item.id === id);

    if (index === -1) {
        return res.status(404).json({ error: 'Элемент не найден' });
    }

    const { name, price } = req.body;

    // Обновление существующего элемента
    items[index] = {
        ...items[index],      // Сохраняем старые значения
        ...(name && { name }), // Обновляем name, если оно передано
        ...(price !== undefined && { price })  // Обновляем price, если оно передано
    };

    res.json(items[index]);
});
```

Разбор кода:

findIndex() — поиск индекса элемента в массиве. Возвращает -1, если элемент не найден.

Если индекс равен -1 → возвращаем 404.

Оператор ... (spread) используется для сохранения старых значений и частичного обновления.

price !== undefined — проверка, было ли передано поле price (даже если price = 0).

Пример 5. DELETE-запрос: удаление элемента

javascript

```
app.delete('/api/items/:id', (req, res) => {
    const id = parseInt(req.params.id);
    const index = items.findIndex(item => item.id === id);

    if (index === -1) {
        return res.status(404).json({ error: 'Элемент не найден' });
    }

    const deletedItem = items.splice(index, 1)[0];
    res.json({ message: 'Элемент удалён', deletedItem });
});
```

Разбор кода:

splice(index, 1) — удаляет из массива один элемент по указанному индексу и возвращает массив удалённых элементов.

[0] — извлекаем первый (и единственный) удалённый элемент из массива.

Пример 6. Полный пример приложения (Средний уровень)

javascript

```
const express = require('express');
const app = express();
const PORT = 3000;

app.use(express.json());

// Хранилище заметок
let notes = [
    { id: 1, title: 'Помыть посуду', completed: false },
    { id: 2, title: 'Сделать уроки', completed: false }
];

// GET все заметки
app.get('/api/notes', (req, res) => {
    res.json(notes);
});

// GET заметка по ID
app.get('/api/notes/:id', (req, res) => {
    const id = parseInt(req.params.id);
    const note = notes.find(n => n.id === id);

    if (!note) {
        return res.status(404).json({ error: 'Заметка не найдена' });
    }
    res.json(note);
});

// POST создать заметку
app.post('/api/notes', (req, res) => {
    const { title } = req.body;

    if (!title) {
        return res.status(400).json({ error: 'Поле title обязательно' });
    }

    const newNote = {
        id: notes.length + 1,
        title: title,
```

```
        completed: false
    };

    notes.push(newNote);
    res.status(201).json(newNote);
});

// PUT обновить заметку
app.put('/api/notes/:id', (req, res) => {
    const id = parseInt(req.params.id);
    const { title, completed } = req.body;
    const index = notes.findIndex(n => n.id === id);

    if (index === -1) {
        return res.status(404).json({ error: 'Заметка не найдена' });
    }

    notes[index] = {
        ...notes[index],
        ...(title && { title }),
        ...(completed !== undefined && { completed })
    };

    res.json(notes[index]);
});

// DELETE удалить заметку
app.delete('/api/notes/:id', (req, res) => {
    const id = parseInt(req.params.id);
    const index = notes.findIndex(n => n.id === id);

    if (index === -1) {
        return res.status(404).json({ error: 'Заметка не найдена' });
    }

    notes.splice(index, 1);
    res.json({ message: 'Заметка удалена' });
});

// Обработчик несуществующих маршрутов
app.use((req, res) => {
    res.status(404).json({ error: 'Маршрут не найден' });
});
```

```
app.listen(PORT, () => {
  console.log(`Сервер запущен на http://localhost:${PORT}`);
});
```

2.2. Индивидуальные задания

2.2.1. Базовый уровень (15 вариантов)

Общее задание для базового уровня: Реализовать сервер с обработкой следующих HTTP-методов для указанной предметной области:

1. GET /api/items — получение всех элементов
2. GET /api/items/:id — получение элемента по ID (с проверкой существования)
3. POST /api/items — создание нового элемента (с проверкой наличия обязательных полей)
4. Данные хранить в массиве в памяти сервера
5. Начальные данные (2-3 элемента) должны быть добавлены при старте сервера

Варианты заданий (предметные области):

Вариант	Название сущности	Обязательные поля	Дополнительные поля
1	Book (Книга)	title, author	year, genre
2	Product (Товар)	name, price	category, inStock
3	Student (Студент)	fullName, group	age, email
4	Movie (Фильм)	title, director	year, rating
5	Task (Задача)	title	completed, priority
6	Employee (Сотрудник)	name, position	salary, department
7	Car (Автомобиль)	model, brand	year, price

Вариант	Название сущности	Обязательные поля	Дополнительные поля
8	Recipe (Рецепт)	title, cuisine	cookingTime, difficulty
9	Event (Мероприятие)	name, date	location, maxParticipants
10	Course (Курс)	title, duration	price, level
11	Hotel (Отель)	name, city	stars, pricePerNight
12	Medicine (Лекарство)	name, manufacturer	price, prescriptionRequired
13	Song (Песня)	title, artist	duration, genre
14	Order (Заказ)	customerName, amount	status, date
15	Article (Статья)	title, author	publishedAt, tags

Пример выполнения для варианта 1 (Book):

javascript

```
const express = require('express');
const app = express();
app.use(express.json());

let books = [
  { id: 1, title: 'Война и мир', author: 'Толстой', year: 1869, genre: 'Роман' },
  { id: 2, title: 'Преступление и наказание', author: 'Достоевский', year: 1866, genre: 'Роман' }
];

app.get('/api/books', (req, res) => res.json(books));

app.get('/api/books/:id', (req, res) => {
```

```
    const id = parseInt(req.params.id);
    const book = books.find(b => b.id === id);
    if (!book) return res.status(404).json({ error: 'Книга не найдена' });
    res.json(book);
});

app.post('/api/books', (req, res) => {
    const { title, author } = req.body;
    if (!title || !author) {
        return res.status(400).json({ error: 'Поля title и author обязательны' });
    }
    const newBook = { id: books.length + 1, title, author, year: req.body.year,
genre: req.body.genre };
    books.push(newBook);
    res.status(201).json(newBook);
});

app.listen(3000);
```

2.2.2. Средний уровень (15 вариантов)

Общее задание для среднего уровня: Выполнить требования базового уровня, а также дополнительно реализовать:

1. PUT-запрос для обновления элемента по ID (PUT /api/items/:id) — с валидацией существования элемента
2. DELETE-запрос для удаления элемента по ID (DELETE /api/items/:id) — с валидацией существования элемента
3. Валидацию всех входящих данных (проверка типов, обязательных полей)
4. Обработчик 404 для несуществующих маршрутов

Варианты заданий (те же предметные области, что в базовом уровне):

Вариант	**PUT-обновляемые поля**	**DELETE-поведение**
1	title, author, year, genre	Удалить книгу, вернуть удалённый объект
2	name, price, category, inStock	Удалить товар, вернуть удалённый объект

Вариант	PUT-обновляемые поля	DELETE-поведение
3	fullName, group, age, email	Удалить студента, вернуть удалённый объект
4	title, director, year, rating	Удалить фильм, вернуть удалённый объект
5	title, completed, priority	Удалить задачу, вернуть удалённый объект
6	name, position, salary, department	Удалить сотрудника, вернуть удалённый объект
7	model, brand, year, price	Удалить автомобиль, вернуть удалённый объект
8	title, cuisine, cookingTime, difficulty	Удалить рецепт, вернуть удалённый объект
9	name, date, location, maxParticipants	Удалить мероприятие, вернуть удалённый объект
10	title, duration, price, level	Удалить курс, вернуть удалённый объект
11	name, city, stars, pricePerNight	Удалить отель, вернуть удалённый объект
12	name, manufacturer, price, prescriptionRequired	Удалить лекарство, вернуть удалённый объект

Вариант	PUT-обновляемые поля	DELETE-поведение
13	title, artist, duration, genre	Удалить песню, вернуть удалённый объект
14	customerName, amount, status, date	Удалить заказ, вернуть удалённый объект
15	title, author, publishedAt, tags	Удалить статью, вернуть удалённый объект

Пример выполнения для варианта 1 (Book) — дополнение к базовому уровню:

javascript

```
// PUT обновление книги
app.put('/api/books/:id', (req, res) => {
  const id = parseInt(req.params.id);
  const { title, author, year, genre } = req.body;
  const index = books.findIndex(b => b.id === id);

  if (index === -1) {
    return res.status(404).json({ error: 'Книга не найдена' });
  }

  books[index] = {
    id: id,
    title: title || books[index].title,
    author: author || books[index].author,
    year: year !== undefined ? year : books[index].year,
    genre: genre || books[index].genre
  };

  res.json(books[index]);
});

// DELETE удаление книги
app.delete('/api/books/:id', (req, res) => {
  const id = parseInt(req.params.id);
  const index = books.findIndex(b => b.id === id);
```

```
    if (index === -1) {
        return res.status(404).json({ error: 'Книга не найдена' });
    }

    const deletedBook = books.splice(index, 1)[0];
    res.json({ message: 'Книга удалена', deletedBook });
});

// Обработчик 404 (должен быть в конце файла)
app.use((req, res) => {
    res.status(404).json({ error: 'Маршрут не найден' });
});
```

2.2.3. Продвинутый уровень (15 вариантов)

Общее задание для продвинутого уровня: Выполнить требования среднего уровня, а также дополнительно реализовать:

1. Генерацию уникальных ID с использованием crypto.randomUUID() вместо length + 1
2. Валидацию типов данных (например, price должен быть числом, year — целым числом от 1900 до текущего года)
3. Частичное обновление через PATCH (обновление только переданных полей)
4. Возврат правильных кодов статуса:
    - 201 Created для POST
    - 204 No Content для DELETE (без тела ответа)
    - 400 Bad Request при невалидных данных
    - Логирование всех запросов через morgan

Установка дополнительных пакетов:

bash

```
npm install morgan dotenv
```

Варианты дополнительных условий валидации:

Вариант	Дополнительная валидация
1	year: число от 1800 до 2025
2	price: число > 0, inStock: boolean
3	age: число от 17 до 100, email: формат email
4	year: число от 1900 до 2025, rating: число от 0 до 10

Вариант	Дополнительная валидация
5	priority: 'low', 'medium', 'high'
6	salary: число > 0
7	year: число от 1900 до 2025, price: число > 0
8	cookingTime: число > 0, difficulty: 'easy', 'medium', 'hard'
9	maxParticipants: целое число > 0
10	duration: число > 0, price: число >= 0, level: 'beginner', 'intermediate', 'advanced'
11	stars: число от 1 до 5, pricePerNight: число > 0
12	price: число > 0, prescriptionRequired: boolean
13	duration: число > 0
14	amount: число > 0, status: 'pending', 'completed', 'cancelled'
15	publishedAt: дата в формате ISO

Пример выполнения для варианта 1 (Book) — продвинутый уровень:
javascript

```
require('dotenv').config();
const express = require('express');
const morgan = require('morgan');
const crypto = require('crypto');

const app = express();
const PORT = process.env.PORT || 3000;

// Middleware
app.use(express.json());
app.use(morgan('combined')); // Логирование всех запросов

// Хранилище
```

```
let books = [
    { id: crypto.randomUUID(), title: 'Война и мир', author: 'Толстой', year: 1869, genre: 'Роман' },
    { id: crypto.randomUUID(), title: 'Преступление и наказание', author: 'Достоевский', year: 1866, genre: 'Роман' }
];

// Вспомогательная функция валидации книги
function validateBook(data, isUpdate = false) {
    const errors = [];

    if (!isUpdate || data.title !== undefined) {
        if (!data.title || typeof data.title !== 'string' || data.title.trim().length === 0) {
            errors.push('title: обязательное непустое строковое поле');
        }
    }

    if (!isUpdate || data.author !== undefined) {
        if (!data.author || typeof data.author !== 'string' || data.author.trim().length === 0) {
            errors.push('author: обязательное непустое строковое поле');
        }
    }

    if (!isUpdate || data.year !== undefined) {
        const year = data.year;
        if (typeof year !== 'number' || year < 1800 || year > new Date().getFullYear()) {
            errors.push(`year: число от 1800 до ${new Date().getFullYear()}`);
        }
    }

    if (!isUpdate || data.genre !== undefined) {
        if (data.genre !== undefined && typeof data.genre !== 'string') {
            errors.push('genre: строка');
        }
    }

    return errors;
}

// GET все книги
```

```
app get('/api/books', (req, res) => {
  res json(books);
});

// GET книга по ID
app get('/api/books/:id', (req, res) => {
  const book = books find(b => b id === req params id);
  if (!book) {
    return res status(404) json({ error: 'Книга не найдена' });
  }
  res json(book);
});

// POST создание книги
app post('/api/books', (req, res) => {
  const errors = validateBook(req body);
  if (errors length > 0) {
    return res status(400) json({ errors });
  }

  const newBook = {
    id: crypto randomUUID(),
    title: req body title trim(),
    author: req body author trim(),
    year: req body year,
    genre: req body genre || null
  };

  books push(newBook);
  res status(201) json(newBook);
});

// PUT полное обновление книги
app put('/api/books/:id', (req, res) => {
  const index = books findIndex(b => b id === req params id);
  if (index === -1) {
    return res status(404) json({ error: 'Книга не найдена' });
  }

  const errors = validateBook(req body);
  if (errors length > 0) {
    return res status(400) json({ errors });
  }
```

```
    books[index] = {
        id: req.params.id,
        title: req.body.title.trim(),
        author: req.body.author.trim(),
        year: req.body.year,
        genre: req.body.genre || null
    };

    res.json(books[index]);
});

// PATCH частичное обновление книги
app.patch('/api/books/:id', (req, res) => {
    const index = books.findIndex(b => b.id === req.params.id);
    if (index === -1) {
        return res.status(404).json({ error: 'Книга не найдена' });
    }

    const errors = validateBook(req.body, true);
    if (errors.length > 0) {
        return res.status(400).json({ errors });
    }

    books[index] = {
        ...books[index],
        ...(req.body.title && { title: req.body.title.trim() }),
        ...(req.body.author && { author: req.body.author.trim() }),
        ...(req.body.year !== undefined && { year: req.body.year }),
        ...(req.body.genre !== undefined && { genre: req.body.genre })
    };

    res.json(books[index]);
});

// DELETE удаление книги
app.delete('/api/books/:id', (req, res) => {
    const index = books.findIndex(b => b.id === req.params.id);
    if (index === -1) {
        return res.status(404).json({ error: 'Книга не найдена' });
    }

    books.splice(index, 1);
```

```
    res.status(204).send(); // 204 No Content — без тела ответа
});

// Обработчик 404
app.use((req, res) => {
    res.status(404).json({ error: 'Маршрут не найден' });
});

// Глобальный обработчик ошибок
app.use((err, req, res, next) => {
    console.error(err.stack);
    res.status(500).json({ error: 'Внутренняя ошибка сервера' });
});

app.listen(PORT, () => {
    console.log(`Сервер запущен на http://localhost:${PORT}`);
});
```

Создание файла .env:

text

```
PORT=3000
```

Обновление package.json (скрипты):

json

```
"scripts": {
    "start": "node app.js",
    "dev": "nodemon app.js"
}
```

3. Контрольные вопросы

1. Какие HTTP-методы соответствуют CRUD-операциям?
2. В чём разница между PUT и PATCH методами?
3. Как в Express получить параметр из URL (например, /users/5)?
4. Как в Express получить данные из тела POST/PUT-запроса?
5. Что такое middleware express.json() и зачем он нужен?
6. Какие коды HTTP-ответов возвращаются при успешном создании ресурса?
7. Какой код возвращается при попытке получить несуществующий ресурс?
8. В чём разница между find() и findIndex() методами массивов?
9. Почему ID из req.params нужно преобразовывать в число?
10. Что такое идемпотентность и какие методы ей обладают?
11. Как удалить элемент из массива по индексу?
12. Для чего используется метод splice()?

13. Как сгенерировать уникальный ID с помощью встроенных средств Node.js?
14. Что делает morgan middleware и зачем он нужен?
15. Какой статус-код возвращается при успешном DELETE-запросе без тела ответа?

4. Требования к отчёту

Отчёт должен быть оформлен в электронном виде и содержать:

Титульный лист с названием работы, ФИО студента, номером группы, вариантом.

Цель и задачи работы (переписать из методических указаний).

Теоретическое обоснование (краткий конспект по теме).

Ход выполнения работы:

Листинг кода с комментариями.

Скриншоты проверки работы через Postman для каждого эндпоинта (GET, POST, PUT, DELETE, PATCH — для продвинутого уровня).

Демонстрация обработки ошибок (404 при несуществующем ID, 400 при невалидных данных).

Ответы на контрольные вопросы (письменно, развёрнуто).

Заключение (выводы о проделанной работе, какие навыки приобретены).

Список использованных источников.

5. Критерии оценивания

Уровень выполнения	Требования	Оценка
Базовый	Реализованы GET (все и по ID) и POST запросы. Данные хранятся в массиве. Проверка обязательных полей.	Удовлетворительно
Средний	Выполнены требования базового уровня + реализованы PUT и DELETE запросы + обработчик 404 для несуществующих маршрутов.	Хорошо

Уровень выполнения	Требования	Оценка
Продвинутый	Выполнены требования среднего уровня + генерация ID через crypto.randomUUID() + PATCH-запрос + типизированная валидация + логирование (morgan) + правильные коды статусов (201, 204).	Отлично

Дополнительные условия:

1. Код должен быть отформатирован и снабжён комментариями.
2. Отчёт должен быть оформлен аккуратно, без грамматических ошибок.
3. При защите студент должен объяснить код и ответить на контрольные вопросы.

6. Список рекомендуемой литературы и интернет-ресурсов

Литература (доступна в РФ)

1. Браун, И. (2021). *Express.js: Разработка веб-приложений на Node.js*. ДМК Пресс. — ISBN: 978-5-97060-945-7.
2. Кантелон, М., Хартер, М., & Холстейн, С. (2020). *Node.js в действии*. Питер. — ISBN: 978-5-4461-1524-2.

Интернет-ресурсы

Название	Ссылка	Описание
Официальная документация Express.js	https://expressjs.com/ru/	Документация Express на русском языке
Metanit — Создание API в Express	https://metanit.com/web/nodejs/4.11.php	Примеры REST API на Express
Metanit — Маршрутизация в Express	https://metanit.com/web/nodejs/4.3.php	Работа с параметрами маршрутов

Название	Ссылка	Описание
Документация Morgan	https://github.com/expressjs/morgan	HTTP-логгер для Express
Документация crypto.randomUUID()	https://nodejs.org/api/crypto.html	Генерация уникальных ID
Postman Learning Center	https://learning.postman.com/	Официальная документация Postman
Habr — REST API на Express	https://habr.com/ru/articles/	Статьи по REST API (поиск по сайту)

Все ресурсы доступны и легальны на территории Российской Федерации.

Лабораторная работа № 3

Тема: Маршрутизация: параметры пути и query-параметры

Цель работы:

Освоить извлечение данных из URL: параметров маршрута (route parameters) и строки запроса (query parameters) для реализации фильтрации, поиска и пагинации данных на сервере.

Задачи работы:

1. Научиться извлекать параметры пути (route parameters) из URL.
2. Освоить получение query-параметров из строки запроса.
3. Реализовать фильтрацию данных на основе query-параметров.
4. Реализовать поиск по тексту с использованием query-параметра.
5. Реализовать пагинацию и сортировку данных на сервере.
6. Научиться устанавливать значения по умолчанию для параметров.

1. Теоретическое обоснование

1.1. Параметры пути (Route Parameters)

Параметры пути - это переменные части URL, которые используются для идентификации конкретного ресурса. В Express они обозначаются двоеточием : перед именем параметра.

Синтаксис:

javascript

```
app.get('/users/:userId/books/:bookId', (req, res) => {
    const userId = req.params.userId;
    const bookId = req.params.bookId;
    // Обработка запроса
});
```

Пример URL: /users/5/books/42 → req.params = { userId: '5', bookId: '42' }

1.2. Query-параметры (Строка запроса)

Query-параметры передаются после знака ? в URL и имеют формат ключ=значение. Несколько параметров разделяются символом &. В Express они автоматически парсятся в объект req.query.

Синтаксис:

javascript

```
app.get('/products', (req, res) => {
    const category = req.query.category;
    const minPrice = req.query.minPrice;
    const maxPrice = req.query.maxPrice;
    // Фильтрация данных
});
```

Пример URL: /products?category=electronics&minPrice=1000&maxPrice=50000 → req.query = { category: 'electronics', minPrice: '1000', maxPrice: '50000' }

1.3. Отличие параметров пути от query-параметров

Характеристика	Параметры пути	Query-параметры
Назначение	Идентификация конкретного ресурса	Фильтрация, сортировка, пагинация
Пример	/users/5	/users?age=25&city=Moscow
Обязательность	Обычно обязательны	Необязательны
Место в URL	Часть пути	После знака ?
В Express	req.params	req.query

1.4. Фильтрация данных

Фильтрация позволяет возвращать только те элементы, которые соответствуют определённым критериям:

javascript

```
app.get('/products', (req, res) => {
  let result = [...products];

  // Фильтр по категории
  if (req.query.category) {
    result = result.filter(p => p.category === req.query.category);
  }
```

```
    // Фильтр по диапазону цен
    if (req.query.minPrice) {
        result = result.filter(p => p.price >= parseInt(req.query.minPrice));
    }

    res.json(result);
});
```

1.5. Поиск (Search)

Поиск обычно реализуется с помощью параметра `q` (query) и проверки вхождения подстроки:

```
javascript
app.get('/products', (req, res) => {
    let result = [...products];

    if (req.query.q) {
        const searchTerm = req.query.q.toLowerCase();
        result = result.filter(p =>
            p.name.toLowerCase().includes(searchTerm) ||
            p.description.toLowerCase().includes(searchTerm)
        );
    }

    res.json(result);
});
```

1.6. Пагинация

Пагинация позволяет разбивать большой набор данных на страницы с помощью параметров page (номер страницы) и limit (количество элементов на странице):

javascript

```
app.get('/products', (req, res) => {
  let page = parseInt(req.query.page) || 1;      // По умолчанию страница 1
  let limit = parseInt(req.query.limit) || 10;      // По умолчанию 10 элементов

  const startIndex = (page - 1) * limit;
  const endIndex = startIndex + limit;

  const paginatedResult = products.slice(startIndex, endIndex);

  res.json({
    total: products.length,
    page: page,
    limit: limit,
    totalPages: Math.ceil(products.length / limit),
    data: paginatedResult
  });
});
```

1.7. Сортировка

Сортировка реализуется с помощью параметров sort (поле для сортировки) и order (направление: asc/desc):

javascript

```
app.get('/products', (req, res) => {
  let result = [...products];
```

```
    const sortField = req.query.sort || 'id';
    const sortOrder = req.query.order === 'desc' ? -1 : 1;

    result.sort((a, b) => {
        if (a[sortField] < b[sortField]) return -1 * sortOrder;
        if (a[sortField] > b[sortField]) return 1 * sortOrder;
        return 0;
    });

    res.json(result);
});
```

1.8. Преобразование типов

Все параметры из req.params и req.query приходят как строки. При необходимости их нужно преобразовывать в числа или другие типы:

```
javascript
const id = parseInt(req.params.id);          // строка → число
const page = parseInt(req.query.page) || 1;  // с значением по умолчанию
const inStock = req.query.inStock === 'true'; // строка → boolean
```

2. Методика выполнения работы

2.1. Примеры и их подробный разбор

Пример 1. Параметры пути (один и несколько)

```
javascript
const express = require('express');
const app = express();

// Один параметр: получение пользователя по ID
```

```
app.get('/users/:id', (req, res) => {
  const userId = req.params.id;
  res.json({ message: `Получен пользователь с ID: ${userId}` });
});

// Несколько параметров: получение комментария в посте
app.get('/users/:userId/posts/:postId/comments/:commentId', (req, res) => {
  const { userId, postId, commentId } = req.params;
  res.json({
    message: `Комментарий ${commentId} пользователя ${userId} к посту ${postId}`
  });
});

app.listen(3000);
```

Проверка: GET /users/5/posts/12/comments/3 → {"message": "Комментарий 3 пользователя 5 к посту 12"}

Пример 2. Query-параметры (базовая фильтрация)

javascript

```
const express = require('express');
const app = express();

const movies = [
  { id: 1, title: 'Побег из Шоушенка', genre: 'Драма', year: 1994, rating: 9.3 },
  { id: 2, title: 'Крёстный отец', genre: 'Драма', year: 1972, rating: 9.2 },
  { id: 3, title: 'Тёмный рыцарь', genre: 'Боевик', year: 2008, rating: 9.0 },
  { id: 4, title: 'Криминальное чтиво', genre: 'Криминал', year: 1994, rating: 8.9 }
```

```
];

app.get('/movies', (req, res) => {
  let result = [...movies];

  // Фильтр по жанру
  if (req.query.genre) {
    result = result.filter(m => m.genre === req.query.genre);
  }

  // Фильтр по минимальному рейтингу
  if (req.query.minRating) {
    const minRating = parseFloat(req.query.minRating);
    result = result.filter(m => m.rating >= minRating);
  }

  // Фильтр по году
  if (req.query.year) {
    const year = parseInt(req.query.year);
    result = result.filter(m => m.year === year);
  }

  res.json(result);
});

app.listen(3000);
```

Примеры запросов:

GET /movies?genre=Драма → только драмы

GET /movies?minRating=9.0 → фильмы с рейтингом ≥ 9.0

GET /movies?genre=Драма&minRating=9.0 → комбинация фильтров

Пример 3. Поиск по тексту (query-параметр q)

javascript

```
app.get('/movies', (req, res) => {
  let result = [...movies];

  // Поиск по названию
  if (req.query.q) {
    const searchTerm = req.query.q.toLowerCase();
    result = result.filter(m =>
      m.title.toLowerCase().includes(searchTerm)
    );
  }

  res.json(result);
});
```

Пример запроса: GET /movies?q=крёстный → найдёт "Крёстный отец"

Пример 4. Пагинация (page, limit)

javascript

```
app.get('/movies', (req, res) => {
  let result = [...movies];

  // Применяем фильтры (если есть)
  if (req.query.genre) {
    result = result.filter(m => m.genre === req.query.genre);
  }

  // Пагинация
```

```
    const page = parseInt(req.query.page) || 1;
    const limit = parseInt(req.query.limit) || 2;

    const startIndex = (page - 1) * limit;
    const endIndex = startIndex + limit;
    const paginatedData = result.slice(startIndex, endIndex);

    res.json({
      total: result.length,
      page: page,
      limit: limit,
      totalPages: Math.ceil(result.length / limit),
      data: paginatedData
    });
});
```

Пример запроса: GET /movies?page=2&limit=2 → вторая страница по 2 фильма

Пример 5. Сортировка (sort, order)

javascript

```
app.get('/movies', (req, res) => {
  let result = [...movies];

  // Сортировка
  const sortField = req.query.sort || 'id';
  const sortOrder = req.query.order === 'desc' ? -1 : 1;

  result.sort((a, b) => {
    if (a[sortField] < b[sortField]) return -1 * sortOrder;
    if (a[sortField] > b[sortField]) return 1 * sortOrder;
```

```
            return 0;
        });

        res.json(result);
    });
```

Примеры запросов:

GET /movies?sort=rating&order=desc → по убыванию рейтинга

GET /movies?sort=year&order=asc → по возрастанию года

Пример 6. Полный пример с комбинацией всех параметров

javascript

```
const express = require('express');
const app = express();

// Данные о товарах
let products = [
    { id: 1, name: 'Ноутбук', category: 'electronics', price: 50000, inStock: true },
    { id: 2, name: 'Смартфон', category: 'electronics', price: 30000, inStock: true },
    { id: 3, name: 'Наушники', category: 'electronics', price: 5000, inStock: false },
    { id: 4, name: 'Книга JS', category: 'books', price: 1500, inStock: true },
    { id: 5, name: 'Книга Python', category: 'books', price: 1200, inStock: true },
    { id: 6, name: 'Книга C++', category: 'books', price: 1800, inStock: false }
];

app.get('/products', (req, res) => {
    let result = [...products];
```

```
// 1. Фильтрация по категории
if (req.query.category) {
  result = result.filter(p => p.category === req.query.category);
}

// 2. Фильтрация по диапазону цен
if (req.query.minPrice) {
  result = result.filter(p => p.price >= parseInt(req.query.minPrice));
}
if (req.query.maxPrice) {
  result = result.filter(p => p.price <= parseInt(req.query.maxPrice));
}

// 3. Фильтр по наличию
if (req.query.inStock === 'true') {
  result = result.filter(p => p.inStock === true);
} else if (req.query.inStock === 'false') {
  result = result.filter(p => p.inStock === false);
}

// 4. Поиск по названию
if (req.query.q) {
  const searchTerm = req.query.q.toLowerCase();
  result = result.filter(p => p.name.toLowerCase().includes(searchTerm));
}

// 5. Сортировка
const sortField = req.query.sort || 'id';
```

```
const sortOrder = req.query.order === 'desc' ? -1 : 1;
result.sort((a, b) => {
    if (a[sortField] < b[sortField]) return -1 * sortOrder;
    if (a[sortField] > b[sortField]) return 1 * sortOrder;
    return 0;
});

// 6. Пагинация
const page = parseInt(req.query.page) || 1;
const limit = parseInt(req.query.limit) || 10;
const startIndex = (page - 1) * limit;
const endIndex = startIndex + limit;
const paginatedData = result.slice(startIndex, endIndex);

// 7. Ответ с мета-информацией
res.json({
    total: result.length,
    page: page,
    limit: limit,
    totalPages: Math.ceil(result.length / limit),
    filters: {
        category: req.query.category || null,
        minPrice: req.query.minPrice || null,
        maxPrice: req.query.maxPrice || null,
        inStock: req.query.inStock || null,
        search: req.query.q || null
    },
    sort: { field: sortField, order: req.query.order || 'asc' },
    data: paginatedData
```

```
    });
});

// Получение конкретного товара по ID (параметр пути)
app.get('/products/:id', (req, res) => {
    const id = parseInt(req.params.id);
    const product = products.find(p => p.id === id);

    if (!product) {
        return res.status(404).json({ error: 'Товар не найден' });
    }

    res.json(product);
});

app.listen(3000, () => {
    console.log('Сервер запущен на http://localhost:3000');
});
```

2.2. Индивидуальные задания

2.2.1. Базовый уровень (15 вариантов)

Общее задание для базового уровня: Реализовать сервер с обработкой следующих эндпоинтов для указанной предметной области:

1. GET /api/items — получение всех элементов
2. GET /api/items/:id — получение элемента по ID (с проверкой существования)
3. GET /api/items/search?q=... — поиск элементов по тексту (в названии или описании)
4. Данные хранить в массиве в памяти сервера

5. Начальные данные (минимум 5 элементов) должны быть добавлены при старте сервера

Варианты заданий (предметные области):

Вариант	**Сущность**	**Поля для поиска (q)**	**Поля в ответе**
1	Книги	title, author	id, title, author, year
2	Товары	name, description	id, name, price, category
3	Фильмы	title, director	id, title, director, year
4	Студенты	fullName, group	id, fullName, group, age
5	Рецепты	title, cuisine	id, title, cuisine, cookingTime
6	Сотрудники	name, position	id, name, position, department
7	Автомобили	model, brand	id, model, brand, year
8	Песни	title, artist	id, title, artist, duration
9	Города	name, country	id, name, country, population

Вариант	Сущность	Поля для поиска (q)	Поля в ответе
10	Курсы	title, instructor	id, title, instructor, duration
11	Отели	name, city	id, name, city, stars
12	Лекарства	name, manufacturer	id, name, manufacturer, price
13	Мероприятия	name, location	id, name, location, date
14	Заказы	customerName, status	id, customerName, amount, status
15	Статьи	title, author	id, title, author, publishedAt

Пример выполнения для варианта 1 (Базовый уровень):

javascript

```
const express = require('express');
const app = express();

let books = [
  { id: 1, title: 'Война и мир', author: 'Толстой', year: 1869 },
  { id: 2, title: 'Преступление и наказание', author: 'Достоевский', year: 1866 },
  { id: 3, title: 'Мастер и Маргарита', author: 'Булгаков', year: 1967 },
  { id: 4, title: 'Анна Каренина', author: 'Толстой', year: 1877 },
```

```
    { id: 5, title: 'Идиот', author: 'Достоевский', year: 1869 }
];

// GET все книги
app.get('/api/books', (req, res) => {
    res.json(books);
});

// GET книга по ID
app.get('/api/books/:id', (req, res) => {
    const id = parseInt(req.params.id);
    const book = books.find(b => b.id === id);
    if (!book) {
        return res.status(404).json({ error: 'Книга не найдена' });
    }
    res.json(book);
});

// GET поиск по книгам
app.get('/api/books/search', (req, res) => {
    const query = req.query.q;
    if (!query) {
        return res.json(books);
    }

    const searchTerm = query.toLowerCase();
    const results = books.filter(book =>
        book.title.toLowerCase().includes(searchTerm) ||
        book.author.toLowerCase().includes(searchTerm)
```

```
    );

    res.json(results);
});

app.listen(3000);
```

2.2.2. Средний уровень (15 вариантов)

Общее задание для среднего уровня: Выполнить требования базового уровня, а также дополнительно реализовать:

1. Фильтрацию по одному или нескольким полям (например, по категории, по году)
2. Сортировку по любому полю с параметрами sort (поле) и order (asc/desc)
3. Пагинацию с параметрами page и limit (значения по умолчанию: page=1, limit=10)
4. Возврат мета-информации: total, page, limit, totalPages

Варианты полей для фильтрации и сортировки (для всех вариантов добавлены единообразно):

Вариант	Поля для фильтрации	Поля для сортировки
1	year, author	id, title, author, year
2	category, price	id, name, price
3	year, director	id, title, year, rating
4	group, age	id, fullName, group, age
5	cuisine, cookingTime	id, title, cookingTime

Вариант	Поля для фильтрации	Поля для сортировки
6	department, position	id, name, position, salary
7	brand, year	id, model, brand, year
8	artist, genre	id, title, artist, duration
9	country	id, name, country, population
10	instructor, duration	id, title, duration, level
11	city, stars	id, name, city, stars
12	manufacturer, price	id, name, price
13	location, date	id, name, date
14	status, customerName	id, customerName, amount, status
15	author, publishedAt	id, title, author, publishedAt

Пример выполнения для варианта 1 (Средний уровень):

javascript

```
const express = require('express');
const app = express();
```

```
let books = [
    { id: 1, title: 'Война и мир', author: 'Толстой', year: 1869 },
    { id: 2, title: 'Преступление и наказание', author: 'Достоевский', year: 1866 },
    { id: 3, title: 'Мастер и Маргарита', author: 'Булгаков', year: 1967 },
    { id: 4, title: 'Анна Каренина', author: 'Толстой', year: 1877 },
    { id: 5, title: 'Идиот', author: 'Достоевский', year: 1869 },
    { id: 6, title: 'Собачье сердце', author: 'Булгаков', year: 1925 }
];

app.get('/api/books', (req, res) => {
    let result = [...books];

    // 1. Поиск по тексту
    if (req.query.q) {
        const searchTerm = req.query.q.toLowerCase();
        result = result.filter(book =>
            book.title.toLowerCase().includes(searchTerm) ||
            book.author.toLowerCase().includes(searchTerm)
        );
    }

    // 2. Фильтрация по автору
    if (req.query.author) {
        result = result.filter(book => book.author === req.query.author);
    }

    // 3. Фильтрация по году
    if (req.query.year) {
```

```
    const year = parseInt(req.query.year);
    result = result.filter(book => book.year === year);
  }

  // 4. Сортировка
  const sortField = req.query.sort || 'id';
  const sortOrder = req.query.order === 'desc' ? -1 : 1;
  result.sort((a, b) => {
    if (a[sortField] < b[sortField]) return -1 * sortOrder;
    if (a[sortField] > b[sortField]) return 1 * sortOrder;
    return 0;
  });

  // 5. Пагинация
  const page = parseInt(req.query.page) || 1;
  const limit = parseInt(req.query.limit) || 10;
  const startIndex = (page - 1) * limit;
  const endIndex = startIndex + limit;
  const paginatedData = result.slice(startIndex, endIndex);

  // 6. Ответ с мета-информацией
  res.json({
    total: result.length,
    page: page,
    limit: limit,
    totalPages: Math.ceil(result.length / limit),
    data: paginatedData
  });
});
```

```
// GET книга по ID (без изменений)
app get('/api/books/:id', (req, res) => {
    const id = parseInt(req params id);
    const book = books find(b => b id === id);
    if (!book) {
        return res status(404) json({ error: 'Книга не найдена' });
    }
    res json(book);
});

// POST, PUT, DELETE (можно добавить по желанию)
app listen(3000);
```

2.2.3. Продвинутый уровень (15 вариантов)

Общее задание для продвинутого уровня: Выполнить требования среднего уровня, а также дополнительно реализовать:

1. Множественную фильтрацию — возможность фильтровать по нескольким значениям одного поля через запятую (например, ?author=Толстой,Булгаков)
2. Диапазонную фильтрацию для числовых полей (например, ?minYear=1860&maxYear=1900)
3. Сложный поиск — поиск одновременно по нескольким текстовым полям с релевантностью
4. Динамическую сортировку по нескольким полям (например, ?sort=author,-year — по автору (возрастание), затем по году (убывание))
5. Возврат выбранных полей (например, ?fields=id,title — вернуть только указанные поля)
6. Логирование всех запросов через morgan

Установка дополнительных пакетов:

```
bash
npm install morgan dotenv
```

Варианты задания (конкретные параметры для каждого варианта):

Вариант	**Сложный поиск по полям**	**Диапазонные поля**	**Множественная фильтрация**	**Сортировка по умолчанию**
1	title, author	year	author	-year, title
2	name, description	price	category	price, name
3	title, director	year	director	-year, title
4	fullName, group	age	group	-age, fullName
5	title, cuisine	cookingTime	cuisine	cookingTime, title
6	name, position	salary	department	position, name
7	model, brand	year	brand	-year, model
8	title, artist	duration	artist	-duration, title
9	name,	populat	country	-

Вариант	Сложный поиск по полям	Диапазонные поля	Множественная фильтрация	Сортировка по умолчанию
	country	ion		population, name
10	title, instructor	duration	instructor	duration, title
11	name, city	stars	city	-stars, name
12	name, manufacturer	price	manufacturer	price, name
13	name, location	date	location	date, name
14	customerName, status	amount	status	-amount, customerName
15	title, author	publishedAt	author	-publishedAt, title

Пример выполнения для варианта 1 (Продвинутый уровень):

javascript

```
require('dotenv').config();
const express = require('express');
const morgan = require('morgan');

const app = express();
const PORT = process.env.PORT || 3000;

// Middleware
app.use(express.json());
app.use(morgan('combined')); // Логирование всех запросов

let books = [
    { id: 1, title: 'Война и мир', author: 'Толстой', year: 1869, price: 500, inStock: true },
    { id: 2, title: 'Преступление и наказание', author: 'Достоевский', year: 1866, price: 450, inStock: true },
    { id: 3, title: 'Мастер и Маргарита', author: 'Булгаков', year: 1967, price: 600, inStock: false },
    { id: 4, title: 'Анна Каренина', author: 'Толстой', year: 1877, price: 550, inStock: true },
    { id: 5, title: 'Идиот', author: 'Достоевский', year: 1869, price: 400, inStock: true },
    { id: 6, title: 'Собачье сердце', author: 'Булгаков', year: 1925, price: 350, inStock: true },
    { id: 7, title: 'Белая гвардия', author: 'Булгаков', year: 1925, price: 420, inStock: false },
    { id: 8, title: 'Братья Карамазовы', author: 'Достоевский', year: 1880, price: 650, inStock: true }
];
```

```
// Вспомогательная функция для фильтрации по нескольким значениям
function filterByMultipleValues(items, field, valueParam) {
    if (!valueParam) return items;
    const values = valueParam.split(',');
    return items.filter(item => values.includes(item[field]));
}

// Вспомогательная функция для диапазонной фильтрации
function filterByRange(items, field, minParam, maxParam) {
    let result = [...items];
    if (minParam) {
        const min = parseInt(minParam);
        result = result.filter(item => item[field] >= min);
    }
    if (maxParam) {
        const max = parseInt(maxParam);
        result = result.filter(item => item[field] <= max);
    }
    return result;
}

// Вспомогательная функция для сложного поиска (по нескольким полям с релевантностью)
function searchBooks(items, searchTerm) {
    if (!searchTerm) return items.map(item => ({ ...item, relevance: 0 }));

    const term = searchTerm.toLowerCase();
    return items.map(item => {
```

```
            let relevance = 0;

            // Поиск в названии (более высокий вес)
            if (item.title.toLowerCase().includes(term)) {
                relevance += 10;
                if (item.title.toLowerCase() === term) relevance += 5;
            }

            // Поиск в авторе
            if (item.author.toLowerCase().includes(term)) {
                relevance += 5;
            }

            return { ...item, relevance };
        }).filter(item => item.relevance > 0)
          .sort((a, b) => b.relevance - a.relevance);
    }

    // Вспомогательная функция для динамической сортировки по нескольким полям
    function multiSort(items, sortParam) {
        if (!sortParam) return items;

        const sortFields = sortParam.split(',');
        return [...items].sort((a, b) => {
            for (const field of sortFields) {
                let order = 1;
                let actualField = field;
```

```
            if (field.startsWith('-')) {
                order = -1;
                actualField = field.slice(1);
            }

            if (a[actualField] < b[actualField]) return -1 * order;
            if (a[actualField] > b[actualField]) return 1 * order;
        }
        return 0;
    });
}

// Вспомогательная функция для выбора полей
function selectFields(items, fieldsParam) {
    if (!fieldsParam) return items;

    const fields = fieldsParam.split(',');
    return items.map(item => {
        const newItem = {};
        fields.forEach(field => {
            if (item.hasOwnProperty(field)) {
                newItem[field] = item[field];
            }
        });
        return newItem;
    });
}

// Основной эндпоинт с комбинированной фильтрацией
```

```
app.get('/api/books', (req, res) => {
    let result = [...books];

    // 1. Сложный поиск (q)
    if (req.query.q) {
        result = searchBooks(result, req.query.q);
    }

    // 2. Множественная фильтрация по автору
    if (req.query.author) {
        result = filterByMultipleValues(result, 'author', req.query.author);
    }

    // 3. Диапазонная фильтрация по году
    result = filterByRange(result, 'year', req.query.minYear,
req.query.maxYear);

    // 4. Множественная сортировка
    result = multiSort(result, req.query.sort);

    // 5. Выбор полей
    result = selectFields(result, req.query.fields);

    // 6. Пагинация (применяется после всех фильтраций)
    const page = parseInt(req.query.page) || 1;
    const limit = parseInt(req.query.limit) || 10;
    const startIndex = (page - 1) * limit;
    const endIndex = startIndex + limit;
    const total = result.length;
```

```
    const paginatedData = result.slice(startIndex, endIndex);

    // 7. Мета-информация
    res.json({
        total: total,
        page: page,
        limit: limit,
        totalPages: Math.ceil(total / limit),
        filters: {
            search: req.query.q || null,
            author: req.query.author || null,
            minYear: req.query.minYear || null,
            maxYear: req.query.maxYear || null
        },
        sort: req.query.sort || 'id',
        fields: req.query.fields || 'все',
        data: paginatedData
    });
});

// GET книга по ID
app.get('/api/books/:id', (req, res) => {
    const id = parseInt(req.params.id);
    const book = books.find(b => b.id === id);

    if (!book) {
        return res.status(404).json({ error: 'Книга не найдена' });
    }
```

```
    // Выбор полей для конкретной книги
    if (req.query.fields) {
        const fields = req.query.fields.split(',');
        const filteredBook = {};
        fields.forEach(field => {
            if (book.hasOwnProperty(field)) {
                filteredBook[field] = book[field];
            }
        });
        return res.json(filteredBook);
    }

    res.json(book);
});

// Обработчик 404
app.use((req, res) => {
    res.status(404).json({ error: 'Маршрут не найден' });
});

// Глобальный обработчик ошибок
app.use((err, req, res, next) => {
    console.error(err.stack);
    res.status(500).json({ error: 'Внутренняя ошибка сервера' });
});

app.listen(PORT, () => {
    console.log(`Сервер запущен на http://localhost:${PORT}`);
});
```

Примеры тестовых запросов для продвинутого уровня:

bash

```
# Поиск по тексту
GET /api/books?q=война

# Множественная фильтрация по авторам
GET /api/books?author=Толстой,Булгаков

# Диапазонная фильтрация по году
GET /api/books?minYear=1860&maxYear=1900

# Комбинация всех фильтров + сортировка + выбор полей + пагинация
GET /api/books?q=достоевский&minYear=1860&maxYear=1880&sort=-year,title&fields=id,title,year&page=1&limit=5

# Получение книги с выбором полей
GET /api/books/2?fields=id,title,year
```

3. Контрольные вопросы

1. В чём разница между параметрами пути (route parameters) и query-параметрами?
2. Как в Express получить доступ к query-параметрам?
3. Как в Express получить доступ к параметрам пути?
4. Как преобразовать строковый параметр из URL в число?
5. Как реализовать поиск с учётом регистра?
6. Что такое пагинация и как она реализуется на сервере?
7. Какие параметры обычно используются для пагинации?
8. Как реализовать сортировку данных на сервере?
9. Что такое slice() и как он используется для пагинации?
10. Как установить значения по умолчанию для query-параметров?
11. В чём разница между фильтрацией и поиском?

12. Как реализовать фильтрацию по нескольким значениям одного поля (например, ?author=Толстой,Булгаков)?
13. Как реализовать диапазонную фильтрацию для числовых полей?
14. Как реализовать сортировку по нескольким полям?
15. Для чего нужен параметр fields и как он реализуется?

4. Требования к отчёту

Отчёт должен быть оформлен в электронном виде и содержать:

Титульный лист с названием работы, ФИО студента, номером группы, вариантом.

Цель и задачи работы (переписать из методических указаний).

Теоретическое обоснование (краткий конспект по теме).

Ход выполнения работы:

Листинг кода с комментариями.

Скриншоты проверки работы через Postman или браузер для каждого типа запросов:

GET с параметрами пути (/items/:id)

GET с поиском (?q=...)

GET с фильтрацией

GET с сортировкой

GET с пагинацией

GET с комбинацией параметров

Демонстрация обработки ошибок (404 при несуществующем ID).

Ответы на контрольные вопросы (письменно, развёрнуто).

Заключение (выводы о проделанной работе, какие навыки приобретены).

Список использованных источников.

5. Критерии оценивания

Уровень выполнения	Требования	Оценка
Базовый	Реализованы GET всех элементов, GET по ID и GET поиск (?q=). Поиск работает по одному текстовому полю.	Удовлетворительно
Средний	Выполнены требования базового уровня + реализованы фильтрация по 1-2 полям, сортировка (sort, order), пагинация (page, limit) с мета-информацией.	Хорошо
Продвинутый	Выполнены требования среднего уровня + множественная фильтрация, диапазонная фильтрация, сортировка по нескольким полям, выбор полей (fields), логирование (morgan).	Отлично

Дополнительные условия:

1. Код должен быть отформатирован и снабжён комментариями.
2. Отчёт должен быть оформлен аккуратно, без грамматических ошибок.

3. При защите студент должен объяснить код и ответить на контрольные вопросы.

6. Список рекомендуемой литературы и интернет-ресурсов

Литература (доступна в РФ)

1. Браун, И. (2021). *Express.js: Разработка веб-приложений на Node.js*. ДМК Пресс. — ISBN: 978-5-97060-945-7.
2. Кантелон, М., Хартер, М., & Холстейн, С. (2020). *Node.js в действии*. Питер. — ISBN: 978-5-4461-1524-2.

Интернет-ресурсы

Название	Ссылка	Описание
Официальная документация Express.js (Маршрутизация)	https://expressjs.com/ru/guide/routing.html	Полное руководство по маршрутизации
Metanit — Параметры маршрутов в Express	https://metanit.com/web/nodejs/4.3.php	Разбор параметров пути и query
Metanit — Получение параметров запроса	https://metanit.com/web/nodejs/4.4.php	Примеры работы с req.query и req.params
Документация Morgan	https://github.com/expressjs/morgan	HTTP-логгер для Express

Название	Ссылка	Описание
MDN Web Docs — Параметры URL	https://developer.mozilla.org/ru/docs/Web/API/URL	Общая теория URL и параметров
Habr — Пагинация, сортировка, фильтрация API	https://habr.com/ru/articles/	Статьи по REST API (поиск по сайту)

Все ресурсы доступны и легальны на территории Российской Федерации.

Лабораторная работа № 4

Тема: Основы SQL: создание таблиц и наполнение данными

Цель работы:

Познакомиться с языком SQL на примере SQLite, научиться создавать таблицы, определять типы данных, добавлять ограничения (первичный ключ, NOT NULL, UNIQUE) и вставлять данные в таблицу.

Задачи работы:

1. Установить и освоить работу с SQLite и DB Browser for SQLite.
2. Изучить синтаксис команды CREATE TABLE.
3. Изучить основные типы данных SQLite.
4. Освоить вставку данных с помощью INSERT INTO.
5. Научиться выполнять простые SELECT-запросы для проверки данных.
6. Понять концепцию первичного ключа (PRIMARY KEY) и его роль.

7. Научиться создавать таблицы с ограничениями (NOT NULL, UNIQUE).

1. Теоретическое обоснование

1.1. Реляционные базы данных

Реляционная база данных (РБД) - это база данных, основанная на реляционной модели данных, где информация хранится в виде таблиц, состоящих из строк и столбцов. Каждая таблица представляет собой множество записей (строк) с одинаковой структурой (столбцами).

Основные термины:

1. Таблица — множество записей с одинаковой структурой.
2. Строка (запись, кортеж) — один элемент данных в таблице.
3. Столбец (поле, атрибут) — характеристика (свойство) записи.
4. Первичный ключ (Primary Key) — столбец (или комбинация столбцов), уникально идентифицирующий каждую строку.

Схема базы данных — описание структуры таблиц, их связей и ограничений.

1.2. SQL (Structured Query Language)

SQL - декларативный язык программирования, предназначенный для управления данными в реляционных базах данных.

Основные категории команд SQL:

DDL (Data Definition Language) — определение структуры данных:

CREATE TABLE — создание таблицы.

ALTER TABLE — изменение структуры таблицы.

DROP TABLE — удаление таблицы.

DML (Data Manipulation Language) — манипуляция данными:

INSERT — вставка данных.

SELECT — выборка данных.

UPDATE — обновление данных.

DELETE — удаление данных.

DQL (Data Query Language) — запросы данных (обычно выделяют SELECT).

DCL (Data Control Language) — управление доступом (GRANT, REVOKE).

1.3. SQLite

SQLite — это встраиваемая реляционная база данных, не требующая отдельного сервера. Данные хранятся в одном файле (обычно с расширением .db или .sqlite). Используется в мобильных приложениях, небольших веб-проектах и при обучении.

Преимущества SQLite:

Не требует установки и настройки отдельного сервера.

База данных — обычный файл, легко копируется и перемещается.

Поддерживает стандартный SQL.

Идеален для обучения и небольших проектов.

1.4. Типы данных в SQLite

SQLite использует динамическую типизацию, но поддерживает следующие типы:

Тип	Описание	Пример
INTEGER	Целое число	42, -5, 0
REAL	Число с плавающей точкой	3.14, -0.5, 2.0
TEXT	Текстовая строка	'Иван', 'Hello'

Тип	Описание	Пример
BLOB	Бинарные данные	изображения, файлы
NUMERIC	Число (INTEGER или REAL)	10, 3.14
BOOLEAN	Логический тип (хранится как INTEGER 0/1)	TRUE, FALSE

1.5. Ограничения столбцов (Constraints)

Ограничение	**Описание**
PRIMARY KEY	Первичный ключ - уникальный идентификатор записи
NOT NULL	Значение не может быть пустым (NULL)
UNIQUE	Все значения в столбце должны быть уникальными
DEFAULT значение	Значение по умолчанию, если не указано
CHECK (условие)	Проверка условия при вставке/обновлении

1.6. Создание таблицы (CREATE TABLE)

Синтаксис:

sql

```
CREATE TABLE имя_таблицы (
```

```
    имя_столбца1 тип1 ограничения1,
    имя_столбца2 тип2 ограничения2,
    ...
    имя_столбцаN типN ограниченияN,
    PRIMARY KEY (столбец)
);
```

Примеры:

sql

```
-- Простая таблица пользователей
CREATE TABLE users (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    age INTEGER,
    email TEXT UNIQUE
);


-- Таблица книг с несколькими ограничениями
CREATE TABLE books (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    title TEXT NOT NULL,
    author TEXT NOT NULL,
    year INTEGER CHECK (year >= 0),
    price REAL DEFAULT 0
);
```

AUTOINCREMENT автоматически увеличивает значение первичного ключа при вставке новой записи.

1.7. Вставка данных (INSERT INTO)

Синтаксис:

sql

```sql
-- Вставка со списком столбцов
INSERT INTO имя_таблицы (столбец1, столбец2, ...)
VALUES (значение1, значение2, ...);

-- Вставка без списка столбцов (значения для всех столбцов)
INSERT INTO имя_таблицы
VALUES (значение1, значение2, ...);

-- Вставка нескольких записей
INSERT INTO имя_таблицы (столбец1, столбец2)
VALUES (значение1, значение2),
       (значение3, значение4),
       (значение5, значение6);
```

Примеры:

sql

```sql
-- Вставка одной записи
INSERT INTO users (name, age, email)
VALUES ('Иван Петров', 25, 'ivan@example.com');

-- Вставка нескольких записей
INSERT INTO books (title, author, year, price)
VALUES ('Война и мир', 'Толстой', 1869, 500),
       ('Преступление и наказание', 'Достоевский', 1866, 450);
```

1.8. Выборка данных (SELECT)

Синтаксис:

sql

```sql
SELECT столбец1, столбец2, ...
```

```
FROM имя_таблицы
WHERE условие;
```

Примеры:

sql

```
-- Выбрать все записи
SELECT * FROM books;

-- Выбрать только названия и авторов
SELECT title, author FROM books;

-- Выбрать книги Толстого
SELECT * FROM books WHERE author = 'Толстой';

-- Выбрать книги дороже 400 рублей
SELECT * FROM books WHERE price > 400;

-- Выбрать книги, отсортированные по году
SELECT * FROM books ORDER BY year DESC;
```

1.9. DB Browser for SQLite

DB Browser for SQLite - это бесплатная программа с графическим интерфейсом для работы с базами данных SQLite. Позволяет:

1. Создавать и открывать файлы баз данных.
2. Выполнять SQL-запросы.
3. Просматривать и редактировать данные в табличном виде.
4. Экспортировать данные в CSV, JSON, Excel.
5. Создавать резервные копии.

2. Методика выполнения работы

2.1. Примеры и их подробный разбор

Пример 1. Установка и настройка SQLite и DB Browser

Шаг 1. Скачать и установить DB Browser for SQLite

Перейти на официальный сайт: https://sqlitebrowser.org/

Скачать установщик для Windows (или версию для своей ОС)

Установить программу, следуя инструкциям установщика

Шаг 2. Проверка установки SQLite (опционально — через командную строку)

bash

```
# Проверка версии SQLite
sqlite3 --version
```

Если команда не найдена, SQLite уже встроен в DB Browser.

Шаг 3. Создание новой базы данных в DB Browser

Запустить DB Browser for SQLite.

Нажать кнопку «Новая база данных» или меню File → New Database.

Выбрать место для сохранения файла и указать имя (например, my_database.db).

Нажать «Сохранить». Откроется окно «Edit Table Definition».

Пример 2. Создание таблицы через интерфейс DB Browser

В окне «Edit Table Definition» выбрать вкладку «Table definition».

В поле «Table name» ввести имя таблицы (например, students).

Добавить столбцы кнопкой «Add field»:

id — тип INTEGER, флаги PK (Primary Key) и AI (Autoincrement)

name — тип TEXT, флаг NOT NULL

age — тип INTEGER

group — тип TEXT

Нажать «OK» для создания таблицы.

Пример 3. Создание таблицы через SQL-запрос

В DB Browser перейти на вкладку «Execute SQL» и выполнить:

sql

```
-- Создание таблицы студентов
CREATE TABLE students (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    age INTEGER CHECK (age >= 0 AND age <= 120),
    group_name TEXT,
    email TEXT UNIQUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Разбор запроса:

Часть запроса	Пояснение
CREATE TABLE students	Создание таблицы с именем students
id INTEGER PRIMARY KEY AUTOINCREMENT	Автоинкрементный первичный ключ
name TEXT NOT NULL	Имя не может быть пустым
age INTEGER CHECK (age >= 0 AND age <= 120)	Возраст от 0 до 120
group_name TEXT	Название группы (может быть пустым)
email TEXT UNIQUE	Email должен быть уникальным
created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP	Время создания, по умолчанию текущее

Пример 4. Вставка данных через SQL-запрос

sql

```sql
-- Вставка одного студента
INSERT INTO students (name, age, group_name, email)
VALUES ('Иван Петров', 20, 'ИВТ-21', 'ivan@university.ru');

-- Вставка нескольких студентов
INSERT INTO students (name, age, group_name, email)
VALUES
    ('Мария Иванова', 19, 'ИВТ-21', 'maria@university.ru'),
    ('Петр Сидоров', 21, 'ИВТ-22', 'petr@university.ru'),
    ('Анна Козлова', 20, 'ИВТ-21', 'anna@university.ru');

-- Вставка без указания id (он сгенерируется автоматически)
INSERT INTO students (name, age, group_name)
VALUES ('Ольга Смирнова', 22, 'ИВТ-22');
-- email будет NULL, так как не указан
```

Пример 5. Вставка данных через интерфейс DB Browser

Перейти на вкладку «Browse Data».

Выбрать созданную таблицу.

Нажать кнопку «New Record» (зелёная иконка с плюсом).

Заполнить поля.

Нажать «Write Changes» (сохранить).

Пример 6. Проверка данных через SELECT

sql

```sql
-- Выбрать всех студентов
SELECT * FROM students;

-- Выбрать только имена и возраст
SELECT name, age FROM students;
```

```sql
-- Выбрать студентов группы ИВТ-21
SELECT * FROM students WHERE group_name = 'ИВТ-21';

-- Выбрать студентов старше 20 лет
SELECT * FROM students WHERE age > 20;

-- Отсортировать по возрасту (по возрастанию)
SELECT * FROM students ORDER BY age ASC;

-- Отсортировать по имени (по убыванию)
SELECT * FROM students ORDER BY name DESC;

-- Выбрать первые 3 записи
SELECT * FROM students LIMIT 3;
```

Пример 7. Полный пример: создание таблицы книг и наполнение данными

sql

```sql
-- Создание таблицы книг
CREATE TABLE books (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    title TEXT NOT NULL,
    author TEXT NOT NULL,
    year INTEGER CHECK (year >= 0 AND year <= 2025),
    price REAL DEFAULT 0,
    in_stock INTEGER DEFAULT 1
);

-- Вставка данных
```

```sql
INSERT INTO books (title, author, year, price, in_stock)
VALUES
    ('Война и мир', 'Лев Толстой', 1869, 599, 1),
    ('Преступление и наказание', 'Фёдор Достоевский', 1866, 499, 1),
    ('Мастер и Маргарита', 'Михаил Булгаков', 1967, 699, 0),
    ('Анна Каренина', 'Лев Толстой', 1877, 549, 1),
    ('Идиот', 'Фёдор Достоевский', 1869, 399, 1),
    ('Собачье сердце', 'Михаил Булгаков', 1925, 349, 1),
    ('Белая гвардия', 'Михаил Булгаков', 1925, 429, 0),
    ('Братья Карамазовы', 'Фёдор Достоевский', 1880, 699, 1),
    ('Двенадцать стульев', 'Илья Ильф и Евгений Петров', 1928, 450, 1),
    ('Золотой телёнок', 'Илья Ильф и Евгений Петров', 1931, 470, 0);

-- Проверка
SELECT * FROM books;
SELECT COUNT(*) FROM books;  -- Количество книг
SELECT author, COUNT(*) FROM books GROUP BY author;  -- Количество книг по авторам
SELECT * FROM books WHERE in_stock = 1;  -- Доступные книги
```

Пример 8. Создание таблицы с внешним ключом (связь)

sql

```sql
-- Таблица авторов
CREATE TABLE authors (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL UNIQUE,
    birth_year INTEGER
);

-- Таблица книг с внешним ключом на автора
```

```
CREATE TABLE books_v2 (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    title TEXT NOT NULL,
    author_id INTEGER NOT NULL,
    year INTEGER,
    price REAL,
    FOREIGN KEY (author_id) REFERENCES authors(id)
);


-- Вставка авторов
INSERT INTO authors (name, birth_year)
VALUES ('Лев Толстой', 1828),
       ('Фёдор Достоевский', 1821),
       ('Михаил Булгаков', 1891);

-- Вставка книг с указанием author_id
INSERT INTO books_v2 (title, author_id, year, price)
VALUES ('Война и мир', 1, 1869, 599),
       ('Преступление и наказание', 2, 1866, 499),
       ('Мастер и Маргарита', 3, 1967, 699);

-- Выборка с JOIN (получение автора для каждой книги)
SELECT books_v2 title, authors name, books_v2 year
FROM books_v2
JOIN authors ON books_v2 author_id = authors id;
```

2.2. Индивидуальные задания

2.2.1. Базовый уровень (15 вариантов)

Общее задание для базового уровня: Создать базу данных SQLite с одной таблицей в соответствии с вариантом. Таблица должна содержать:

Поле id — INTEGER PRIMARY KEY AUTOINCREMENT

4-5 полей (из таблицы ниже)

Минимум одно ограничение NOT NULL

Минимум одно ограничение UNIQUE (кроме id)

Вставить не менее 5 записей в таблицу

Выполнить SELECT запросы: все записи, фильтрация по одному условию, сортировка

Варианты заданий (предметные области и поля таблицы):

	Сущность	Поля	NOT NULL	UNIQUE
	Книги	title, author, year, price, genre	title, author	title
	Студенты	full_name, group, age, email, course	full_name, email	email
	Товары	name, category, price, quantity, description	name	name
	Фильм	title, director,	title	title

	Сущность	Поля	NOT NULL	UNIQUE
	ы	year, rating, genre		
	Сотрудники	full_name, position, salary, department, phone	full_name, phone	phone
	Рецепты	title, cuisine, cooking_time, difficulty, ingredients	title	title
	Автомобили	model, brand, year, price, engine	model, brand	model
	Песни	title, artist, album, duration, genre	title, artist	title
	Отели	name, city, stars, price_per_night, address	name	name
0	Курсы	title, instructor, duration, price, level	title	title
	Лекарс	name,	name	name

	Сущность	Поля	NOT NULL	UNIQUE
1	тва	manufacturer, price, prescription, stock		
2	Мероприятия	name, location, date, price, max_participants	name	name
3	Заказы	customer_name, product, quantity, amount, status	customer_name, product	order_number
4	Статьи	title, author, published_at, views, content	title	title
5	Спортсмены	full_name, sport, country, medals, birth_year	full_name	full_name

Пример выполнения для варианта 1 (Базовый уровень):

sql

```
-- Создание базы данных books.db

-- 1. Создание таблицы книг
CREATE TABLE books (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
```

```
    title TEXT NOT NULL UNIQUE,
    author TEXT NOT NULL,
    year INTEGER,
    price REAL,
    genre TEXT
);

-- 2. Вставка данных (минимум 5 записей)
INSERT INTO books (title, author, year, price, genre)
VALUES
    ('Война и мир', 'Лев Толстой', 1869, 599, 'Роман'),
    ('Преступление и наказание', 'Фёдор Достоевский', 1866, 499,
'Роман'),
    ('Мастер и Маргарита', 'Михаил Булгаков', 1967, 699, 'Фантастика'),
    ('Анна Каренина', 'Лев Толстой', 1877, 549, 'Роман'),
    ('Собачье сердце', 'Михаил Булгаков', 1925, 349, 'Повесть');

-- 3. Проверка: выборка всех записей
SELECT * FROM books;

-- 4. Фильтрация: книги Толстого
SELECT * FROM books WHERE author = 'Лев Толстой';

-- 5. Сортировка: книги по убыванию года
SELECT * FROM books ORDER BY year DESC;
```

2.2.2. Средний уровень (15 вариантов)

Общее задание для среднего уровня: Выполнить требования базового уровня, а также дополнительно:

Добавить CHECK-ограничение для одного числового поля (например, year ≥ 0, price > 0, age от 0 до 120).

Добавить DEFAULT значение для одного поля.

Вставить не менее 10 записей.

Выполнить SELECT-запросы с:

Условием WHERE ... AND ... (два условия)

Условием WHERE ... OR ...

Условием WHERE ... LIKE ... (поиск по шаблону)

Агрегатной функцией COUNT(*)

Создать индекс для одного поля (для ускорения поиска).

Дополнительные требования для среднего уровня (к вариантам из базового):

	CHECK-ограничение	**DEFAULT поле**	**Индекс на поле**
	year >= 0 AND year <= 2025	price DEFAULT 0	author
	age >= 0 AND age <= 120	course DEFAULT 1	group
	price >= 0	quantity DEFAULT 0	category
	year >= 1900 AND year <= 2025	rating DEFAULT 0	director
	salary >= 0	department DEFAULT 'Общий'	position

	CHECK-ограничение	DEFAULT поле	Индекс на поле
	cooking_time >= 0	difficulty DEFAULT 'Средний'	cuisine
	year >= 1900 AND year <= 2025	engine DEFAULT 'Бензин'	brand
	duration >= 0	album DEFAULT 'Сингл'	artist
	stars >= 1 AND stars <= 5	city DEFAULT 'Москва'	city
0	duration >= 0	level DEFAULT 'Начальный'	instructor
1	price >= 0	prescription DEFAULT 0	manufacturer
2	price >= 0	max_participants DEFAULT 100	location
3	quantity > 0 AND amount >= 0	status DEFAULT 'Новый'	customer_name
4	views >= 0	published_at DEFAULT CURRENT_TIMESTAMP	author

	CHECK-ограничение	DEFAULT поле	Индекс на поле
5	medals >= 0	country DEFAULT 'Россия'	sport

Пример выполнения для варианта 1 (Средний уровень):

sql

```
-- 1. Создание таблицы с CHECK и DEFAULT
CREATE TABLE books (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    title TEXT NOT NULL UNIQUE,
    author TEXT NOT NULL,
    year INTEGER CHECK (year >= 0 AND year <= 2025),
    price REAL DEFAULT 0,
    genre TEXT
);


-- 2. Создание индекса для поля author
CREATE INDEX idx_books_author ON books(author);


-- 3. Вставка 10+ записей
INSERT INTO books (title, author, year, price, genre)
VALUES
    ('Война и мир', 'Лев Толстой', 1869, 599, 'Роман'),
    ('Преступление и наказание', 'Фёдор Достоевский', 1866, 499, 'Роман'),
    ('Мастер и Маргарита', 'Михаил Булгаков', 1967, 699, 'Фантастика'),
    ('Анна Каренина', 'Лев Толстой', 1877, 549, 'Роман'),
    ('Идиот', 'Фёдор Достоевский', 1869, 399, 'Роман'),
```

```
('Собачье сердце', 'Михаил Булгаков', 1925, 349, 'Повесть'),
('Белая гвардия', 'Михаил Булгаков', 1925, 429, 'Роман'),
('Братья Карамазовы', 'Фёдор Достоевский', 1880, 699, 'Роман'),
('Двенадцать стульев', 'Илья Ильф и Евгений Петров', 1928, 450, 'Роман'),
('Золотой телёнок', 'Илья Ильф и Евгений Петров', 1931, 470, 'Роман');

-- 4. SELECT с двумя условиями (AND)
SELECT * FROM books WHERE author = 'Лев Толстой' AND price > 500;

-- 5. SELECT с OR
SELECT * FROM books WHERE author = 'Михаил Булгаков' OR author = 'Фёдор Достоевский';

-- 6. SELECT с LIKE (поиск книг, в названии которых есть "мир")
SELECT * FROM books WHERE title LIKE '%мир%';

-- 7. Агрегатная функция COUNT
SELECT author, COUNT(*) as book_count FROM books GROUP BY author;
```

2.2.3. Продвинутый уровень (15 вариантов)

Общее задание для продвинутого уровня: Выполнить требования среднего уровня, а также дополнительно:

Создать две связанные таблицы (один-ко-многим) с использованием внешнего ключа (FOREIGN KEY).

Вставить данные в обе таблицы (не менее 5 авторов/категорий/жанров и не менее 15 записей в основной таблице).

Выполнить SELECT с JOIN (объединение таблиц) для получения связанных данных.

Создать представление (VIEW) для упрощения частого запроса.

Выполнить обновление (UPDATE) данных по условию.

Выполнить удаление (DELETE) данных по условию.

Экспортировать результат SELECT в CSV-файл (через DB Browser).

Структура связанных таблиц (для всех вариантов):

Вариант	**Таблица 1 (справочник)**	**Таблица 2 (основная)**	**Связь**	**Внешний ключ**
1	authors (id, name, birth_year)	books (id, title, author_id, year, price)	один автор → много книг	author_id REFERENCES authors(id)
2	groups (id, group_name, course)	students (id, name, group_id, age, email)	одна группа → много студентов	group_id REFERENCES groups(id)
3	categories (id, name, parent_category)	products (id, name, category_id, price, stock)	одна категория → много товаров	category_id REFERENCES categories(id)
4	directors (id, name,	movies (id, title,	один режиссёр →	director_id

Вариант	**Таблица 1 (справочник)**	**Таблица 2 (основная)**	**Связь**	**Внешний ключ**
	birth_year)	director_id, year, rating)	много фильмов	REFERENCES directors(id)
5	departments (id, name, floor)	employees (id, name, department_id, position, salary)	один отдел → много сотрудников	department_id REFERENCES departments(id)
6	cuisines (id, name, country)	recipes (id, title, cuisine_id, cooking_time, difficulty)	одна кухня → много рецептов	cuisine_id REFERENCES cuisines(id)
7	brands (id, name, country)	cars (id, model, brand_id, year, price)	один бренд → много автомобилей	brand_id REFERENCES brands(id)
8	artists (id, name, country)	songs (id, title, artist_id, album, duration)	один исполнитель → много песен	artist_id REFERENCES artists(id)

Вариант	**Таблица 1 (справочник)**	**Таблица 2 (основная)**	**Связь**	**Внешний ключ**
9	cities (id, name, country)	hotels (id, name, city_id, stars, price)	один город → много отелей	city_id REFERENCES cities(id)
10	instructors (id, name, specialization)	courses (id, title, instructor_id, duration, price)	один инструктор → много курсов	instructor_id REFERENCES instructors(id)
11	manufacturers (id, name, country)	medicines (id, name, manufacturer_id, price, prescription)	один производитель → много лекарств	manufacturer_id REFERENCES manufacturers(id)
12	locations (id, name, address)	events (id, name, location_id, date, price)	одно место → много мероприятий	location_id REFERENCES locations(id)
13	customers (id, name,	orders (id,	один заказчик →	customer_id

Вариант	Таблица 1 (справочник)	Таблица 2 (основная)	Связь	Внешний ключ
	phone)	customer_id, order_date, amount, status)	много заказов	REFERENCES customers(id)
4	1	authors (id, name, email)	articles (id, title, author_id, published_at, views)	один автор → много статей	author_id REFERENCES authors(id)
5	1	sports (id, name, category)	athletes (id, name, sport_id, country, medals)	один вид спорта → много спортсменов	sport_id REFERENCES sports(id)

Пример выполнения для варианта 1 (Продвинутый уровень):

sql

```
-- ============================================
-- 1. Создание таблиц с внешними ключами
-- ============================================

-- Таблица авторов (справочник)
CREATE TABLE authors (
```

```
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL UNIQUE,
    birth_year INTEGER CHECK (birth_year >= 1800 AND birth_year <= 2025)
);

-- Таблица книг с внешним ключом на автора
CREATE TABLE books (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    title TEXT NOT NULL UNIQUE,
    author_id INTEGER NOT NULL,
    year INTEGER CHECK (year >= 0 AND year <= 2025),
    price REAL DEFAULT 0,
    in_stock INTEGER DEFAULT 1,
    FOREIGN KEY (author_id) REFERENCES authors(id) ON DELETE CASCADE
);

-- ============================================
-- 2. Создание индексов для ускорения запросов
-- ============================================

CREATE INDEX idx_books_author_id ON books(author_id);
CREATE INDEX idx_books_year ON books(year);

-- ============================================
-- 3. Вставка данных (авторы)
-- ============================================
```

```sql
INSERT INTO authors (name, birth_year) VALUES
    ('Лев Толстой', 1828),
    ('Фёдор Достоевский', 1821),
    ('Михаил Булгаков', 1891),
    ('Илья Ильф и Евгений Петров', 1897);

-- ============================================
-- 4. Вставка данных (книги, не менее 15)
-- ============================================

INSERT INTO books (title, author_id, year, price, in_stock) VALUES
    ('Война и мир', 1, 1869, 599, 1),
    ('Анна Каренина', 1, 1877, 549, 1),
    ('Казаки', 1, 1863, 399, 0),
    ('Преступление и наказание', 2, 1866, 499, 1),
    ('Идиот', 2, 1869, 399, 1),
    ('Братья Карамазовы', 2, 1880, 699, 1),
    ('Бесы', 2, 1872, 449, 0),
    ('Мастер и Маргарита', 3, 1967, 699, 1),
    ('Собачье сердце', 3, 1925, 349, 1),
    ('Белая гвардия', 3, 1925, 429, 0),
    ('Роковые яйца', 3, 1925, 299, 1),
    ('Двенадцать стульев', 4, 1928, 450, 1),
    ('Золотой телёнок', 4, 1931, 470, 1),
    ('Одноэтажная Америка', 4, 1937, 350, 0),
    ('Война и мир. Том 2', 1, 1869, 399, 1);

-- ============================================
-- 5. SELECT с JOIN (объединение таблиц)
```

```
-- ============================================

-- Получение списка книг с именами авторов
SELECT
    books.title,
    authors.name AS author,
    books.year,
    books.price,
    CASE WHEN books.in_stock = 1 THEN 'Да' ELSE 'Нет' END AS
in_stock
FROM books
JOIN authors ON books.author_id = authors.id
ORDER BY books.year DESC;

-- Получение количества книг по каждому автору
SELECT
    authors.name AS author,
    COUNT(books.id) AS book_count,
    SUM(CASE WHEN books.in_stock = 1 THEN 1 ELSE 0 END) AS
in_stock_count,
    ROUND(AVG(books.price), 0) AS avg_price
FROM authors
LEFT JOIN books ON authors.id = books.author_id
GROUP BY authors.id
ORDER BY book_count DESC;

-- ============================================
-- 6. Создание представления (VIEW)
-- ============================================
```

```
CREATE VIEW available_books AS
SELECT
    books id,
    books title,
    authors name AS author,
    books.year,
    books price
FROM books
JOIN authors ON books author_id = authors id
WHERE books in_stock = 1;

-- Использование представления
SELECT * FROM available_books WHERE price <= 500;

-- ============================================
-- 7. UPDATE (обновление данных по условию)
-- ============================================

-- Увеличить цену всех книг Толстого на 10%
UPDATE books
SET price = price * 1.1
WHERE author_id = (SELECT id FROM authors WHERE name = 'Лев Толстой');

-- Установить статус "в наличии" для книг с ценой до 400
UPDATE books SET in_stock = 1 WHERE price < 400;

-- ============================================
```

```sql
-- 8. DELETE (удаление данных по условию)
-- ============================================

-- Удалить книги с ценой больше 1000 (если есть)
DELETE FROM books WHERE price > 1000;

-- Удалить книги автора, у которого нет книг в наличии (пример с проверкой)
DELETE FROM books
WHERE author_id = (SELECT id FROM authors WHERE name = 'Михаил Булгаков')
AND in_stock = 0;

-- ============================================
-- 9. Дополнительные запросы для проверки
-- ============================================

-- Найти книги, изданные между 1860 и 1900 годами
SELECT * FROM books WHERE year BETWEEN 1860 AND 1900;

-- Найти книги по подстроке в названии
SELECT * FROM books WHERE title LIKE '%война%' OR title LIKE '%мир%';

-- Самые дорогие книги (топ 3)
SELECT title, price FROM books ORDER BY price DESC LIMIT 3;

-- Средняя цена книг по годам
SELECT year, ROUND(AVG(price), 2) as avg_price
```

```
FROM books
GROUP BY year
HAVING COUNT(*) >= 2
ORDER BY year;
```

Экспорт в CSV через DB Browser:

Выполнить SELECT-запрос.

Нажать на таблицу с результатами правой кнопкой мыши.

Выбрать «Export to CSV».

Указать имя файла и настройки экспорта.

3. Контрольные вопросы

1. Что такое реляционная база данных?
2. Что такое SQL и для чего он используется?
3. Чем SQLite отличается от других СУБД (MySQL, PostgreSQL)?
4. Какие типы данных поддерживает SQLite?
5. Что такое первичный ключ (PRIMARY KEY)? Зачем он нужен?
6. Что такое AUTOINCREMENT и для чего он используется?
7. Какие ограничения (CONSTRAINTS) вы знаете?
8. В чём разница между NOT NULL и UNIQUE?
9. Что такое CHECK-ограничение? Приведите пример.
10. Как вставить несколько записей одной командой INSERT?
11. Как выполнить выборку с сортировкой (ORDER BY)?
12. Как выполнить фильтрацию с частичным совпадением (LIKE)?
13. Что такое внешний ключ (FOREIGN KEY)? Зачем он нужен?
14. Что такое JOIN и для чего он используется?
15. В чём разница между LEFT JOIN и INNER JOIN?
16. Что такое представление (VIEW)?
17. Как выполнить обновление данных (UPDATE)?
18. Как выполнить удаление данных (DELETE)?
19. Что такое индекс и для чего он нужен?
20. Как экспортировать данные из SQLite в CSV?

4. Требования к отчёту

Отчёт должен быть оформлен в электронном виде и содержать:

Титульный лист с названием работы, ФИО студента, номером группы, вариантом.

Цель и задачи работы (переписать из методических указаний).

Теоретическое обоснование (краткий конспект по теме).

Ход выполнения работы:

Скриншоты установки и запуска DB Browser for SQLite.

SQL-скрипты создания таблиц (с комментариями).

SQL-скрипты вставки данных (с комментариями).

Скриншоты DB Browser с результатами SELECT-запросов.

Для среднего уровня: скриншоты с GROUP BY, LIKE, COUNT.

Для продвинутого уровня: скриншоты JOIN, VIEW, UPDATE, DELETE, экспорта в CSV.

Ответы на контрольные вопросы (письменно, развёрнуто).

Заключение (выводы о проделанной работе, какие навыки приобретены).

Список использованных источников.

5. Критерии оценивания

Уровень выполнения	Требования	Оценка
Базовый	Создана одна таблица с PRIMARY KEY, NOT NULL, UNIQUE. Вставлено не менее 5 записей. Выполнены SELECT: все записи, фильтрация, сортировка.	Удовлетворительно

Уровень выполнения	Требования	Оценка
Средний	Выполнены требования базового уровня + CHECK, DEFAULT, не менее 10 записей, SELECT с AND/OR/LIKE, COUNT, GROUP BY, создан индекс.	Хорошо
Продвинутый	Выполнены требования среднего уровня + две связанные таблицы с FOREIGN KEY, не менее 15 записей, SELECT с JOIN, создано VIEW, выполнены UPDATE и DELETE, экспорт в CSV.	Отлично

Дополнительные условия:

1. SQL-код должен быть оформлен аккуратно, с отступами и комментариями.
2. Отчёт должен содержать скриншоты всех этапов работы.
3. При защите студент должен объяснить структуру таблиц и синтаксис использованных запросов.

6. Список рекомендуемой литературы и интернет-ресурсов

Литература (доступна в РФ)

1. Дэвид, М. (2020). *SQL для начинающих*. Эксмо. — ISBN: 978-5-04-112385-7.
2. Форт, Б. (2019). *SQL за 10 минут*. Вильямс. — ISBN: 978-5-8459-2137-6.

3. Аллен, Г. (2021). *SQLite. Базы данных для небольших проектов*. ДМК Пресс. — ISBN: 978-5-97060-890-0.

Интернет-ресурсы

Название	**Ссылка**	**Описание**
Официальный сайт SQLite	https://sqlite.org/	Документация и инструменты SQLite
DB Browser for SQLite	https://sqlitebrowser.org/	Скачивание программы
Metanit — SQLite	https://metanit.com/sql/sqlite/	Учебник по SQLite на русском
SQL-Ex.ru	https://sql-ex.ru/	Тренажёр по SQL (бесплатно)
Хабр — SQL для начинающих	https://habr.com/ru/articles/	Статьи по SQL (поиск по сайту)
W3Schools SQL Tutorial	https://www.w3schools.com/sql/	Учебник SQL на английском (доступен из РФ)

Все ресурсы доступны и легальны на территории Российской Федерации.

Лабораторная работа № 5

Тема: Выполнение SELECT-запросов из серверного кода

Цель работы:

Научиться подключать базу данных SQLite к серверному приложению на Node.js/Express и выполнять SELECT-запросы для получения данных из базы с последующим возвратом их в формате JSON.

Задачи работы:

1. Установить и настроить библиотеку sqlite3 для работы с SQLite в Node.js.
2. Освоить создание соединения с файлом базы данных.
3. Научиться выполнять SELECT-запросы из серверного кода.
4. Реализовать GET-маршруты для получения всех записей и записи по ID.
5. Освоить обработку ошибок при работе с базой данных.
6. Научиться преобразовывать результаты запросов в JSON.

1. Теоретическое обоснование

1.1. Подключение базы данных к серверному приложению

В реальных веб-приложениях данные хранятся в базах данных, а не в памяти сервера (как в первых лабораторных работах). Это позволяет:

1. Сохранять данные между перезапусками сервера.
2. Работать с большими объёмами данных.
3. Обеспечивать конкурентный доступ к данным.
4. Использовать мощь SQL для выборки и фильтрации.

1.2. Библиотека sqlite3 для Node.js

sqlite3 - это официальная библиотека для работы с SQLite из Node.js. Она предоставляет как синхронный, так и асинхронный API для выполнения SQL-запросов.

Установка:

bash

```
npm install sqlite3
```

1.3. Создание соединения с базой данных

javascript

```
const sqlite3 = require('sqlite3').verbose();
const db = new sqlite3.Database('./database.db');
```

.verbose() — включает подробный вывод ошибок и предупреждений.

new sqlite3.Database('./database.db') — открывает (или создаёт) файл базы данных.

1.4. Асинхронные методы выполнения запросов

Библиотека sqlite3 предоставляет три основных метода для выполнения запросов:

Метод	Описание	Когда использовать
db.run(sql, params, callback)	Выполняет запрос без возврата данных (INSERT, UPDATE, DELETE)	Для операций изменения данных
db.get(sql, params, callback)	Возвращает одну строку	Для получения одной записи
db.all(sql, params, callback)	Возвращает все строки	Для получения всех записей

Пример использования db.all:

javascript

```
db.all("SELECT * FROM users", [], (err, rows) => {
```

```
    if (err) {
        console.error(err);
        return;
    }
    console.log(rows);
});
```

1.5. Параметризованные запросы

Для безопасности и предотвращения SQL-инъекций необходимо использовать параметризованные запросы. В библиотеке sqlite3 параметры передаются в виде массива, а в SQL-запросе используются плейсхолдеры ?:

javascript

```
// Безопасный запрос с параметрами
db.get("SELECT * FROM users WHERE id = ?", [userId], (err, user) => {
    // ...
});
```

1.6. Обработка ошибок

При работе с базой данных важно корректно обрабатывать ошибки:

1. Ошибки подключения к БД.
2. Ошибки синтаксиса SQL.
3. Ошибки целостности данных.
4. Ситуации, когда запись не найдена.

javascript

```
db.get("SELECT * FROM users WHERE id = ?", [id], (err, user) => {
    if (err) {
        return res.status(500).json({ error: 'Ошибка базы данных' });
    }
    if (!user) {
        return res.status(404).json({ error: 'Пользователь не найден' });
```

```
    }
    res.json(user);
});
```

1.7. Закрытие соединения с базой данных

При завершении работы приложения рекомендуется закрывать соединение с БД:

javascript

```
process.on('SIGINT', () => {
    db.close((err) => {
        if (err) console.error(err);
        process.exit(0);
    });
});
```

2. Методика выполнения работы

2.1. Примеры и их подробный разбор

Пример 1. Установка и настройка проекта

Шаг 1. Создание папки проекта

bash

```
mkdir lab5-select
cd lab5-select
npm init -y
```

Шаг 2. Установка зависимостей

bash

```
npm install express sqlite3
npm install --save-dev nodemon
```

Шаг 3. Подготовка базы данных

Создать файл init_db.js для инициализации базы данных:

```javascript
const sqlite3 = require('sqlite3').verbose();

// Подключаемся к базе данных (создаст файл)
const db = new sqlite3.Database('./library.db');

// Создаём таблицу книг
db.run(`
    CREATE TABLE IF NOT EXISTS books (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        title TEXT NOT NULL,
        author TEXT NOT NULL,
        year INTEGER,
        price REAL,
        in_stock INTEGER DEFAULT 1
    )
`, (err) => {
    if (err) {
        console.error('Ошибка создания таблицы:', err.message);
    } else {
        console.log('Таблица books создана или уже существует');
    }
});

// Заполняем тестовыми данными
db.run(`INSERT INTO books (title, author, year, price, in_stock) VALUES
    ('Война и мир', 'Толстой', 1869, 599, 1),
    ('Преступление и наказание', 'Достоевский', 1866, 499, 1),
    ('Мастер и Маргарита', 'Булгаков', 1967, 699, 0)
```

```
`, (err) => {
    if (err && !err.message.includes('UNIQUE')) {
        console.error('Ошибка вставки:', err.message);
    }
});

// Закрываем соединение
db.close();
```

Запуск инициализации:

```
bash
node init_db.js
```

Пример 2. Базовый сервер с подключением к БД

```
javascript
const express = require('express');
const sqlite3 = require('sqlite3').verbose();

const app = express();
const PORT = 3000;

// Подключение к базе данных
const db = new sqlite3.Database('./library.db', (err) => {
    if (err) {
        console.error('Ошибка подключения к БД:', err.message);
    } else {
        console.log('Подключено к базе данных library.db');
    }
});

// GET /api/books — получение всех книг
```

```
app get('/api/books', (req, res) => {
    db all('SELECT * FROM books', [], (err, rows) => {
        if (err) {
            return res status(500) json({ error: err message });
        }
        res json(rows);
    });
});

// GET /api/books/:id — получение книги по ID
app get('/api/books/:id', (req, res) => {
    const id = req params id

    db get('SELECT * FROM books WHERE id = ?', [id], (err, row) => {
        if (err) {
            return res status(500) json({ error: err message });
        }
        if (!row) {
            return res status(404) json({ error: 'Книга не найдена' });
        }
        res json(row);
    });
});

// Обработка закрытия приложения
process on('SIGINT', () => {
    db close((err) => {
        if (err) console error(err);
        process exit(0);
```

```
    });
});

app.listen(PORT, () => {
    console.log(`Сервер запущен на http://localhost:${PORT}`);
});
```

Разбор кода:

Строка	Описание
const db = new sqlite3.Database('./library.db', (err) => {...})	Открытие соединения с БД. Callback вызывается после завершения операции.
db.all('SELECT * FROM books', [], (err, rows) => {...})	Выполняет SELECT и возвращает все строки в виде массива rows.
db.get('SELECT * FROM books WHERE id = ?', [id], (err, row) => {...})	Возвращает только первую строку результата.
if (!row) { return res.status(404).json(...) }	Проверка существования записи — если row равен undefined, возвращаем 404.

Пример 3. GET с параметрами (фильтрация) из серверного кода

javascript

```
// GET /api/books?author=Толстой&minYear=1800&maxYear=1900&inStock=true
app.get('/api/books', (req, res) => {
    let sql = 'SELECT * FROM books WHERE 1=1';
    const params = [];
```

```
// Фильтрация по автору
if (req.query.author) {
    sql += ' AND author = ?';
    params.push(req.query.author);
}

// Фильтрация по году (диапазон)
if (req.query.minYear) {
    sql += ' AND year >= ?';
    params.push(req.query.minYear);
}
if (req.query.maxYear) {
    sql += ' AND year <= ?';
    params.push(req.query.maxYear);
}

// Фильтрация по наличию на складе
if (req.query.inStock === 'true') {
    sql += ' AND in_stock = 1';
} else if (req.query.inStock === 'false') {
    sql += ' AND in_stock = 0';
}

// Сортировка
if (req.query.sort) {
    const order = req.query.order === 'desc' ? 'DESC' : 'ASC';
    sql += ` ORDER BY ${req.query.sort} ${order}`;
}
```

```
// Пагинация
const page = parseInt(req.query.page) || 1;
const limit = parseInt(req.query.limit) || 10;
const offset = (page - 1) * limit;
sql += ' LIMIT ? OFFSET ?';
params.push(limit, offset);

db.all(sql, params, (err, rows) => {
    if (err) {
        return res.status(500).json({ error: err.message });
    }

    // Получение общего количества для пагинации
    let countSql = 'SELECT COUNT(*) as total FROM books WHERE 1=1';
    const countParams = [];
    // Повторяем условия фильтрации для подсчёта
    if (req.query.author) {
        countSql += ' AND author = ?';
        countParams.push(req.query.author);
    }
    if (req.query.minYear) {
        countSql += ' AND year >= ?';
        countParams.push(req.query.minYear);
    }
    if (req.query.maxYear) {
        countSql += ' AND year <= ?';
        countParams.push(req.query.maxYear);
```

```
    }
    if (req.query.inStock === 'true') {
      countSql += ' AND in_stock = 1';
    } else if (req.query.inStock === 'false') {
      countSql += ' AND in_stock = 0';
    }

    db.get(countSql, countParams, (err, countRow) => {
      if (err) {
        return res.status(500).json({ error: err.message });
      }

      res.json({
        data: rows,
        pagination: {
          total: countRow.total,
          page: page,
          limit: limit,
          totalPages: Math.ceil(countRow.total / limit)
        }
      });
    });
  });
});
```

Пример 4. GET с поиском по тексту (LIKE)

javascript

```
// GET /api/books/search?q=война
app.get('/api/books/search', (req, res) => {
  const searchTerm = req.query.q;
```

```
    if (!searchTerm) {
        return res.status(400).json({ error: 'Параметр q обязателен' });
    }

    const sql = `
        SELECT * FROM books
        WHERE title LIKE ? OR author LIKE ?
    `;
    const params = [`%${searchTerm}%`, `%${searchTerm}%`];

    db.all(sql, params, (err, rows) => {
        if (err) {
            return res.status(500).json({ error: err.message });
        }
        res.json({
            query: searchTerm,
            count: rows.length,
            results: rows
        });
    });
});
```

Пример 5. Полный пример сервера (базовый)

javascript

```
const express = require('express');
const sqlite3 = require('sqlite3').verbose();
const path = require('path');

const app = express();
```

```
const PORT = 3000;

// Путь к базе данных
const dbPath = path.join(__dirname, 'library.db');

// Создание/открытие базы данных
const db = new sqlite3.Database(dbPath, (err) => {
    if (err) {
        console.error('Ошибка подключения к БД:', err.message);
        process.exit(1);
    }
    console.log('Подключено к БД:', dbPath);

    // Создание таблицы, если не существует
    db.run(`
        CREATE TABLE IF NOT EXISTS books (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            title TEXT NOT NULL,
            author TEXT NOT NULL,
            year INTEGER,
            price REAL DEFAULT 0,
            in_stock INTEGER DEFAULT 1
        )
    `, (err) => {
        if (err) {
            console.error('Ошибка создания таблицы:', err.message);
        } else {
            // Проверяем, есть ли данные
```

```
        db.get('SELECT COUNT(*) as count FROM books', [], (err, row) =>
{
            if (err) return;
            if (row.count === 0) {
                // Заполняем тестовыми данными
                const insertStmt = db.prepare(`
                    INSERT INTO books (title, author, year, price, in_stock)
                    VALUES (?, ?, ?, ?, ?)
                `);
                const testData = [
                    ['Война и мир', 'Лев Толстой', 1869, 599, 1],
                    ['Преступление и наказание', 'Фёдор Достоевский', 1866,
499, 1],
                    ['Мастер и Маргарита', 'Михаил Булгаков', 1967, 699, 0],
                    ['Анна Каренина', 'Лев Толстой', 1877, 549, 1],
                    ['Идиот', 'Фёдор Достоевский', 1869, 399, 1]
                ];

                testData.forEach(book => {
                    insertStmt.run(book, (err) => {
                        if (err) console.error('Ошибка вставки:', err);
                    });
                });
                insertStmt.finalize();
                console.log('Добавлены тестовые данные');
            }
        });
    }
});
```

```
});

// GET все книги
app.get('/api/books', (req, res) => {
  db.all('SELECT * FROM books ORDER BY id', [], (err, rows) => {
    if (err) {
      return res.status(500).json({ error: err.message });
    }
    res.json(rows);
  });
});

// GET книга по ID
app.get('/api/books/:id', (req, res) => {
  const id = parseInt(req.params.id);

  if (isNaN(id)) {
    return res.status(400).json({ error: 'ID должен быть числом' });
  }

  db.get('SELECT * FROM books WHERE id = ?', [id], (err, row) => {
    if (err) {
      return res.status(500).json({ error: err.message });
    }
    if (!row) {
      return res.status(404).json({ error: 'Книга не найдена' });
    }
    res.json(row);
  });
```

```
});

// GET поиск книг
app.get('/api/books/search', (req, res) => {
    const query = req.query.q;

    if (!query) {
        return res.status(400).json({ error: 'Параметр q обязателен' });
    }

    const searchPattern = `%${query}%`;
    db.all(
        'SELECT * FROM books WHERE title LIKE ? OR author LIKE ?',
        [searchPattern, searchPattern],
        (err, rows) => {
            if (err) {
                return res.status(500).json({ error: err.message });
            }
            res.json({
                query: query,
                count: rows.length,
                results: rows
            });
        }
    );
});

// Обработчик несуществующих маршрутов
app.use((req, res) => {
```

```
    res.status(404).json({ error: 'Маршрут не найден' });
});

// Закрытие соединения при завершении
process.on('SIGINT', () => {
    console.log('\nЗакрытие соединения с БД...');
    db.close((err) => {
        if (err) console.error(err);
        console.log('Соединение закрыто');
        process.exit(0);
    });
});

app.listen(PORT, () => {
    console.log(`Сервер запущен на http://localhost:${PORT}`);
    console.log('Доступные эндпоинты:');
    console.log('  GET  /api/books          - все книги');
    console.log('  GET  /api/books/:id      - книга по ID');
    console.log('  GET  /api/books/search?q= - поиск книг');
});
```

2.2. Индивидуальные задания

2.2.1. Базовый уровень (15 вариантов)

Общее задание для базового уровня:

Реализовать сервер на Express с подключением к SQLite-базе данных, содержащей таблицу согласно варианту. Сервер должен предоставлять:

1. GET /api/items — получение всех записей из таблицы.
2. GET /api/items/:id — получение записи по ID (с проверкой существования).

Обработку ошибок: 500 при ошибке БД, 404 при отсутствии записи.

Варианты заданий (структура таблицы):

№	Сущность	Поля таблицы
1	Книги	id, title, author, year, price
2	Студенты	id, full_name, group_name, age, email
3	Товары	id, name, category, price, quantity
4	Фильмы	id, title, director, year, rating
5	Сотрудники	id, full_name, position, salary, department
6	Рецепты	id, title, cuisine, cooking_time, difficulty
7	Автомобили	id, model, brand, year, price
8	Песни	id, title, artist, album, duration
9	Отели	id, name, city, stars, price_per_night
10	Курсы	id, title, instructor, duration, price
11	Лекарства	id, name, manufacturer, price, in_stock
12	Мероприятия	id, name, location, date, max_participants

№	Сущность	Поля таблицы
13	Заказы	id, customer_name, product_name, amount, status
14	Статьи	id, title, author, published_at, views
15	Спортсмены	id, full_name, sport, country, medals

Пример выполнения для варианта 1 (Базовый уровень):

javascript

```
const express = require('express');
const sqlite3 = require('sqlite3').verbose();
const app = express();
const PORT = 3000;

// Подключение к БД
const db = new sqlite3.Database('./books.db', (err) => {
    if (err) console.error('Ошибка БД:', err);
});

// GET все книги
app.get('/api/books', (req, res) => {
    db.all('SELECT * FROM books', [], (err, rows) => {
        if (err) return res.status(500).json({ error: err.message });
        res.json(rows);
    });
});
```

```
// GET книга по ID
app.get('/api/books/:id', (req, res) => {
  const id = parseInt(req.params.id);
  db.get('SELECT * FROM books WHERE id = ?', [id], (err, row) => {
    if (err) return res.status(500).json({ error: err.message });
    if (!row) return res.status(404).json({ error: 'Не найдено' });
    res.json(row);
  });
});

app.listen(PORT);
```

2.2.2. Средний уровень (15 вариантов)

Общее задание для среднего уровня:

Выполнить требования базового уровня, а также дополнительно реализовать:

Фильтрацию по одному полю (например, ?author=Толстой или ?category=electronics).

Сортировку по любому полю с параметрами sort (поле) и order (asc/desc).

Пагинацию с параметрами page и limit (значения по умолчанию: page=1, limit=10).

Возврат **мета-информации** в ответе: total, page, limit, totalPages.

Варианты полей для фильтрации и сортировки (к вариантам из базового уровня):

№	Поле для фильтрации	Поля для сортировки
1	author	id, title, author, year, price
2	group_name	id, full_name, group_name, age
3	category	id, name, category, price
4	director	id, title, director, year, rating
5	department	id, full_name, position, salary
6	cuisine	id, title, cuisine, cooking_time
7	brand	id, model, brand, year, price
8	artist	id, title, artist, duration
9	city	id, name, city, stars
10	instructor	id, title, instructor, duration
11	manufacturer	id, name, manufacturer, price
12	location	id, name, location, date

№	Поле для фильтрации	Поля для сортировки
13	status	id, customer_name, amount, status
14	author	id, title, author, published_at
15	sport	id, full_name, sport, medals

Пример выполнения для варианта 1 (Средний уровень):

javascript

```
const express = require('express');
const sqlite3 = require('sqlite3').verbose();
const app = express();
const PORT = 3000;

const db = new sqlite3.Database('./books.db');

// GET /api/books с фильтрацией, сортировкой и пагинацией
app.get('/api/books', (req, res) => {
    let sql = 'SELECT * FROM books WHERE 1=1';
    const params = [];

    // Фильтрация по автору
    if (req.query.author) {
        sql += ' AND author = ?';
        params.push(req.query.author);
    }
```

```
    // Сортировка
    const sortField = req.query.sort || 'id';
    const sortOrder = req.query.order === 'desc' ? 'DESC' : 'ASC';
    sql += ` ORDER BY ${sortField} ${sortOrder}`;

    // Пагинация
    const page = parseInt(req.query.page) || 1;
    const limit = parseInt(req.query.limit) || 10;
    const offset = (page - 1) * limit;
    sql += ' LIMIT ? OFFSET ?';
    params.push(limit, offset);

    // Запрос данных
    db.all(sql, params, (err, rows) => {
        if (err) return res.status(500).json({ error: err.message });

        // Запрос общего количества
        let countSql = 'SELECT COUNT(*) as total FROM books WHERE
1=1';
        const countParams = [];
        if (req.query.author) {
            countSql += ' AND author = ?';
            countParams.push(req.query.author);
        }

        db.get(countSql, countParams, (err, countRow) => {
            if (err) return res.status(500).json({ error: err.message });
```

```
            res.json({
                data: rows,
                pagination: {
                    total: countRow.total,
                    page: page,
                    limit: limit,
                    totalPages: Math.ceil(countRow.total / limit)
                }
            });
        });
    });
});

app.listen(PORT);
```

2.2.3. Продвинутый уровень (15 вариантов)

Общее задание для продвинутого уровня:

Выполнить требования среднего уровня, а также дополнительно реализовать:

Связанные таблицы (две таблицы: справочник и основная) с внешним ключом.

GET /api/items/:id — возвращает данные из основной таблицы + подтягивает название из справочника через JOIN.

GET /api/items?expand=related — возможность получить данные с расширенной информацией из связанной таблицы.

Поиск (q) одновременно по нескольким текстовым полям основной таблицы и справочника.

Логирование всех запросов через morgan.

Переменные окружения (.env) для пути к БД и порта.

Варианты связанных таблиц (к вариантам из базового уровня):

	Основная таблица	Справочник (FK)	Поля для JOIN
	books (author_id)	authors (id, name)	author_name
	students (group_id)	groups (id, name)	group_name
	products (category_id)	categories (id, name)	category_name
	movies (director_id)	directors (id, name)	director_name
	employees (department_id)	departments (id, name)	department_name
	recipes (cuisine_id)	cuisines (id, name)	cuisine_name
	cars (brand_id)	brands (id, name)	brand_name
	songs (artist_id)	artists (id, name)	artist_name
	hotels (city_id)	cities (id, name)	city_name
0	courses (instructor_id)	instructors (id, name)	instructor_name
	medicines	manufacturers	manufacturer

	Основная таблица	Справочник (FK)	Поля для JOIN
1	(manufacturer_id)	(id, name)	_name
2	events (location_id)	locations (id, name)	location_name
3	orders (customer_id)	customers (id, name)	customer_name
4	articles (author_id)	article_authors (id, name)	author_name
5	athletes (sport_id)	sports (id, name)	sport_name

Пример выполнения для варианта 1 (Продвинутый уровень):

javascript

```
require('dotenv').config();
const express = require('express');
const morgan = require('morgan');
const sqlite3 = require('sqlite3').verbose();

const app = express();
const PORT = process.env.PORT || 3000;
const DB_PATH = process.env.DB_PATH || './library.db';

// Middleware
app.use(express.json());
app.use(morgan('combined'));
```

```
// Подключение к БД
const db = new sqlite3.Database(DB_PATH, (err) => {
    if (err) {
        console.error('Ошибка подключения к БД:', err);
        process.exit(1);
    }
    console.log('Подключено к БД:', DB_PATH);

    // Создание таблиц, если не существуют
    db.run(`CREATE TABLE IF NOT EXISTS authors (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT NOT NULL UNIQUE
    )`);

    db.run(`CREATE TABLE IF NOT EXISTS books (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        title TEXT NOT NULL,
        author_id INTEGER NOT NULL,
        year INTEGER,
        price REAL,
        in_stock INTEGER DEFAULT 1,
        FOREIGN KEY (author_id) REFERENCES authors(id)
    )`);

    // Заполнение тестовыми данными (если пусто)
    db.get('SELECT COUNT(*) as count FROM authors', [], (err, row) => {
        if (err) return;
        if (row.count === 0) {
```

```
            db.run(`INSERT INTO authors (name) VALUES
                ('Лев Толстой'), ('Фёдор Достоевский'), ('Михаил Булгаков')`);
            db.run(`INSERT INTO books (title, author_id, year, price) VALUES
                ('Война и мир', 1, 1869, 599),
                ('Анна Каренина', 1, 1877, 549),
                ('Преступление и наказание', 2, 1866, 499),
                ('Мастер и Маргарита', 3, 1967, 699)`);
        }
    });
});

// GET /api/books — все книги (с возможностью expand)
app.get('/api/books', (req, res) => {
    const expand = req.query.expand === 'true';

    let sql;
    if (expand) {
        sql = `
            SELECT books.*, authors.name as author_name
            FROM books
            JOIN authors ON books.author_id = authors.id
            WHERE 1=1
        `;
    } else {
        sql = 'SELECT * FROM books WHERE 1=1';
    }

    const params = [];
```

```
// Поиск (q) по названию книги и имени автора
if (req.query.q && expand) {
    sql += ' AND (books.title LIKE ? OR authors.name LIKE ?)';
    const searchTerm = `%${req.query.q}%`;
    params.push(searchTerm, searchTerm);
} else if (req.query.q) {
    sql += ' AND title LIKE ?';
    params.push(`%${req.query.q}%`);
}

// Фильтрация по автору
if (req.query.author_id) {
    sql += ' AND author_id = ?';
    params.push(req.query.author_id);
}

// Сортировка
const sortField = req.query.sort || 'id';
const sortOrder = req.query.order === 'desc' ? 'DESC' : 'ASC';
sql += ` ORDER BY ${sortField} ${sortOrder}`;

// Пагинация
const page = parseInt(req.query.page) || 1;
const limit = parseInt(req.query.limit) || 10;
const offset = (page - 1) * limit;
sql += ' LIMIT ? OFFSET ?';
params.push(limit, offset);

db.all(sql, params, (err, rows) => {
```

```
    if (err) return res.status(500).json({ error: err.message });

    // Получение общего количества
    let countSql = 'SELECT COUNT(*) as total FROM books';
    if (req.query.author_id) {
      countSql += ' WHERE author_id = ?';
      db.get(countSql, [req.query.author_id], (err, countRow) => {
        if (err) return res.status(500).json({ error: err.message });
        res.json({
          data: rows,
          pagination: {
            total: countRow.total,
            page: page,
            limit: limit,
            totalPages: Math.ceil(countRow.total / limit)
          }
        });
      });
    } else {
      db.get(countSql, [], (err, countRow) => {
        if (err) return res.status(500).json({ error: err.message });
        res.json({
          data: rows,
          pagination: {
            total: countRow.total,
            page: page,
            limit: limit,
            totalPages: Math.ceil(countRow.total / limit)
          }
```

```
            });
        });
    }
  });
});

// GET /api/books/:id — книга по ID (с JOIN)
app.get('/api/books/:id', (req, res) => {
  const id = parseInt(req.params.id);
  const expand = req.query.expand === 'true';

  let sql;
  if (expand) {
    sql = `
      SELECT books.*, authors.name as author_name
      FROM books
      JOIN authors ON books.author_id = authors.id
      WHERE books.id = ?
    `;
  } else {
    sql = 'SELECT * FROM books WHERE id = ?';
  }

  db.get(sql, [id], (err, row) => {
    if (err) return res.status(500).json({ error: err.message });
    if (!row) return res.status(404).json({ error: 'Книга не найдена' });
    res.json(row);
  });
});
```

```
// GET /api/books/search — поиск с ранжированием
app.get('/api/books/search', (req, res) => {
  const query = req.query.q;
  if (!query) {
    return res.status(400).json({ error: 'Параметр q обязателен' });
  }

  const searchTerm = `%${query}%`;
  const sql = `
    SELECT
      books.*,
      authors.name as author_name,
      CASE
        WHEN books.title LIKE ? THEN 10
        WHEN authors.name LIKE ? THEN 5
        ELSE 0
      END as relevance
    FROM books
    JOIN authors ON books.author_id = authors.id
    WHERE books.title LIKE ? OR authors.name LIKE ?
    ORDER BY relevance DESC
  `;

  db.all(sql, [searchTerm, searchTerm, searchTerm, searchTerm], (err, rows) => {
    if (err) return res.status(500).json({ error: err.message });
    res.json({
      query: query,
```

```
            count: rows.length,
            results: rows
        });
    });
});

// GET /api/authors — список авторов (для связи)
app.get('/api/authors', (req, res) => {
    db.all('SELECT * FROM authors ORDER BY name', [], (err, rows) => {
        if (err) return res.status(500).json({ error: err.message });
        res.json(rows);
    });
});

// Обработчик 404
app.use((req, res) => {
    res.status(404).json({ error: 'Маршрут не найден' });
});

// Закрытие БД
process.on('SIGINT', () => {
    db.close(() => process.exit(0));
});

app.listen(PORT, () => {
    console.log(`Сервер запущен на http://localhost:${PORT}`);
});
```

Файл .env:

```
PORT=3000
```

DB_PATH=./library.db

Установка дополнительных пакетов:

bash

npm install morgan dotenv

3. Контрольные вопросы

1. Для чего используется библиотека sqlite3 в Node.js?
2. Как открыть соединение с базой данных SQLite?
3. В чём разница между методами db.all() и db.get()?
4. Как выполнить параметризованный запрос для защиты от SQL-инъекций?
5. Как обработать ошибку при выполнении запроса к БД?
6. Как проверить, существует ли запись с указанным ID?
7. Какой HTTP-код возвращается, если запись не найдена?
8. Какой HTTP-код возвращается при ошибке базы данных?
9. Как закрыть соединение с БД при завершении приложения?
10. Как реализовать поиск с использованием оператора LIKE?
11. Как реализовать пагинацию с помощью LIMIT и OFFSET?
12. Как реализовать сортировку с помощью ORDER BY?
13. Что такое JOIN и как его выполнить из кода Node.js?
14. Как использовать переменные окружения для хранения настроек БД?
15. Для чего используется библиотека morgan?

4. Требования к отчёту

Отчёт должен быть оформлен в электронном виде и содержать:

Титульный лист с названием работы, ФИО студента, номером группы, вариантом.

Цель и задачи работы (переписать из методических указаний).

Теоретическое обоснование (краткий конспект по теме).

Ход выполнения работы:

Листинг кода сервера с комментариями.

Скриншоты запуска сервера в терминале.

Скриншоты проверки работы через Postman или браузер:

GET /api/items

GET /api/items/:id

Для среднего уровня: GET /api/items?sort=price&order=desc&page=1&limit=5

Для продвинутого уровня: GET /api/items?expand=true&q=поиск&author_id=1

Демонстрация обработки ошибок (404, 500).

Ответы на контрольные вопросы (письменно, развёрнуто).

Заключение (выводы о проделанной работе, какие навыки приобретены).

Список использованных источников.

5. Критерии оценивания

Уровень выполнения	Требования	Оценка
Базовый	Реализованы GET всех записей и GET по ID с подключением к SQLite. Корректная обработка ошибок 404 и 500.	Удовлетворительно
Средний	Выполнены требования базового уровня + фильтрация по одному полю, сортировка, пагинация с мета-информацией.	Хорошо
Продвинутый	Выполнены	Отлично

Уровень выполнения	Требования	Оценка
	требования среднего уровня + две связанные таблицы (FOREIGN KEY), JOIN, поиск (q), возможность expand, логирование (morgan), переменные окружения (.env).	

Дополнительные условия:

Код должен быть структурирован и снабжён комментариями.

Отчёт должен быть оформлен аккуратно, без грамматических ошибок.

При защите студент должен объяснить структуру БД и синтаксис SQL-запросов.

6. Список рекомендуемой литературы и интернет-ресурсов

Литература (доступна в РФ)

1. Браун, И. (2021). ***Express.js: Разработка веб-приложений на Node.js***. ДМК Пресс.
2. Кантелон, М., Хартер, М., & Холстейн, С. (2020). ***Node.js в действии*** Питер.

Интернет-ресурсы

Название	Ссылка	Описание
Официальная	https://github.com/TryGhost/node-sqlite3	Документация и примеры

Название	Ссылка	Описание
документация sqlite3		
Metanit — Node.js и SQLite	**https://metanit.com/web/nodejs/5.1.php**	Работа с SQLite из Node.js
Express + SQLite примеры	**https://expressjs.com/ru/advanced/best-practice-performance.html**	Советы по производительности
Документация Morgan	**https://github.com/expressjs/morgan**	HTTP-логгер
Документация Dotenv	**https://github.com/motdotla/dotenv**	Переменные окружения
SQLite Tutorial	**https://www.sqlitetutorial.net/**	Учебник по SQLite (англ.)

Все ресурсы доступны и легальны на территории Российской Федерации.

Лабораторная работа № 6

Тема: Полный CRUD с базой данных (INSERT, UPDATE, DELETE)

Цель работы:

Реализовать полный цикл CRUD-операций (Create, Read, Update, Delete) с реальным хранением данных в базе данных SQLite, включая создание, чтение, обновление и удаление записей через REST API.

Задачи работы:

1. Реализовать POST-маршрут для создания новой записи в базе данных (INSERT).
2. Реализовать PUT/PATCH-маршрут для обновления существующей записи (UPDATE).
3. Реализовать DELETE-маршрут для удаления записи (DELETE).
4. Освоить валидацию входных данных перед вставкой/обновлением.
5. Научиться возвращать правильные коды HTTP-ответов (201, 200, 204, 400, 404).
6. Реализовать проверку существования записи перед обновлением/удалением.

1. Теоретическое обоснование

1.1. CRUD-операции и HTTP-методы

CRUD - аббревиатура от Create (создание), Read (чтение), Update (обновление), Delete (удаление). В REST API каждой операции соответствует HTTP-метод и эндпоинт:

CRUD	**HTTP-метод**	**Типичный эндпоинт**	**Действие**
Create	POST	/api/items	Создание нового ресурса
Read	GET	/api/items	Получение всех ресурсов

CRUD	**HTTP-метод**	**Типичный эндпоинт**	**Действие**
Read	GET	/api/items/:id	Получение конкретного ресурса
Update	PUT / PATCH	/api/items/:id	Обновление ресурса
Delete	DELETE	/api/items/:id	Удаление ресурса

1.2. Команды SQL для CRUD

Операция	SQL-команда	Пример
Create (INSERT)	INSERT INTO ... VALUES ...	INSERT INTO books (title, author) VALUES ('Война и мир', 'Толстой')
Read (SELECT)	SELECT ... FROM ... WHERE ...	SELECT * FROM books WHERE id = 1
Update (UPDATE)	UPDATE ... SET ... WHERE ...	UPDATE books SET price = 599 WHERE id = 1
Delete (DELETE)	DELETE FROM ... WHERE ...	DELETE FROM books WHERE id = 1

1.3. Последовательность выполнения UPDATE

При обновлении записи необходимо:

- Получить ID из параметров маршрута.
- Получить данные для обновления из тела запроса.
- Проверить, существует ли запись с таким ID.
- Если не существует — вернуть 404.
- Если существует — выполнить UPDATE.
- Вернуть обновлённую запись (или статус 204).

1.4. Получение ID созданной записи

После выполнения INSERT часто требуется получить ID созданной записи. В библиотеке sqlite3 это можно сделать через this.lastID внутри callback-функции:

javascript

```
db.run('INSERT INTO books (title, author) VALUES (?, ?)', [title, author], function(err) {
    if (err) return;
    const newId = this.lastID;  // ID последней вставленной записи
    res.status(201).json({ id: newId, title, author });
});
```

1.5. Коды HTTP-ответов для CRUD

Операция	Успешный код	Дополнительно
POST (создание)	201 Created	Возвращает созданный объект или Location-заголовок
GET (все)	200 OK	Возвращает массив объектов
GET (один)	200 OK	Возвращает объект

Операция	Успешный код	Дополнительно
PUT/PATCH	200 OK	Возвращает обновлённый объект
DELETE	204 No Content	Не возвращает тело ответа
Ошибка валидации	400 Bad Request	Возвращает сообщение об ошибке
Ресурс не найден	404 Not Found	Возвращает сообщение об ошибке
Ошибка сервера	500 Internal Server Error	-

1.6. Валидация входных данных

Перед вставкой или обновлением данных необходимо проверять:

1. Наличие обязательных полей.
2. Типы данных (числа, строки, булевы значения).
3. Допустимые значения (например, возраст не может быть отрицательным).

Уникальность (при необходимости).

javascript

```
function validateBook(data) {
  const errors = [];
  if (!data title || data title trim() length === 0) {
    errors push('Поле "title" обязательно');
  }
```

```
    if (!data.author || data.author.trim().length === 0) {
        errors.push('Поле "author" обязательно');
    }
    if (data.year && (data.year < 0 || data.year > new Date().getFullYear())) {
        errors.push('Год должен быть от 0 до текущего года');
    }
    if (data.price && data.price < 0) {
        errors.push('Цена не может быть отрицательной');
    }
    return errors;
}
```

2. Методика выполнения работы

2.1. Примеры и их подробный разбор

Пример 1. Создание таблицы (инициализация)

javascript

```
const sqlite3 = require('sqlite3').verbose();
const db = new sqlite3.Database('./database.db');

// Создание таблицы книг
db.run(`
    CREATE TABLE IF NOT EXISTS books (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        title TEXT NOT NULL,
        author TEXT NOT NULL,
        year INTEGER,
        price REAL DEFAULT 0,
        in_stock INTEGER DEFAULT 1
    )
```

```
`, (err) => {
    if (err) console.error('Ошибка создания таблицы:', err.message);
});
```

Пример 2. POST-маршрут (создание записи)

javascript

```
app.post('/api/books', (req, res) => {
    const { title, author, year, price, in_stock } = req.body;

    // Валидация обязательных полей
    if (!title || !author) {
        return res.status(400).json({
            error: 'Поля title и author обязательны'
        });
    }

    // Вставка в базу данных
    db.run(
        `INSERT INTO books (title, author, year, price, in_stock)
         VALUES (?, ?, ?, ?, ?)`,
        [title, author, year || null, price || 0, in_stock !== undefined ? in_stock : 1],
        function(err) {
            if (err) {
                return res.status(500).json({ error: err.message });
            }

            // Получение созданной записи
            db.get('SELECT * FROM books WHERE id = ?', [this.lastID], (err, row) => {
```

```
            if (err) {
                return res.status(500).json({ error: err.message });
            }
            res.status(201).json(row);
        });
    }
  );
});
```

Разбор кода:

this.lastID — свойство объекта statement, содержащее ID последней вставленной записи.

После INSERT выполняется дополнительный SELECT для получения полной записи с авто-сгенерированными полями.

Пример 3. PUT-маршрут (полное обновление записи)

javascript

```
app.put('/api/books/:id', (req, res) => {
  const id = parseInt(req.params.id);
  const { title, author, year, price, in_stock } = req.body;

  // Валидация обязательных полей
  if (!title || !author) {
    return res.status(400).json({
      error: 'Поля title и author обязательны'
    });
  }

  // Проверка существования записи
  db.get('SELECT * FROM books WHERE id = ?', [id], (err, book) => {
    if (err) return res.status(500).json({ error: err.message });
```

```
            if (!book) return res.status(404).json({ error: 'Книга не найдена' });

            // Обновление записи
            db.run(
              `UPDATE books
               SET title = ?, author = ?, year = ?, price = ?, in_stock = ?
               WHERE id = ?`,
              [title, author, year || null, price || 0, in_stock !== undefined ? in_stock : 1, id],
              function(err) {
                if (err) return res.status(500).json({ error: err.message });

                // Возврат обновлённой записи
                db.get('SELECT * FROM books WHERE id = ?', [id], (err, updatedBook) => {
                  if (err) return res.status(500).json({ error: err.message });
                  res.json(updatedBook);
                });
              }
            );
          });
        });
```

Пример 4. DELETE-маршрут (удаление записи)

javascript

```
app.delete('/api/books/:id', (req, res) => {
  const id = parseInt(req.params.id);

  // Проверка существования записи
  db.get('SELECT * FROM books WHERE id = ?', [id], (err, book) => {
```

```
        if (err) return res.status(500).json({ error: err.message });
        if (!book) return res.status(404).json({ error: 'Книга не найдена' });

        // Удаление записи
        db.run('DELETE FROM books WHERE id = ?', [id], function(err) {
            if (err) return res.status(500).json({ error: err.message });

            // 204 No Content — без тела ответа
            res.status(204).send();
        });
    });
});
```

Пример 5. PATCH-маршрут (частичное обновление)

javascript

```
app.patch('/api/books/:id', (req, res) => {
    const id = parseInt(req.params.id);
    const updates = req.body;

    // Проверка существования записи
    db.get('SELECT * FROM books WHERE id = ?', [id], (err, book) => {
        if (err) return res.status(500).json({ error: err.message });
        if (!book) return res.status(404).json({ error: 'Книга не найдена' });

        // Формирование динамического UPDATE
        const fields = [];
        const values = [];

        if (updates.title !== undefined) {
            fields.push('title = ?');
```

```
    values.push(updates.title);
}
if (updates.author !== undefined) {
    fields.push('author = ?');
    values.push(updates.author);
}
if (updates.year !== undefined) {
    fields.push('year = ?');
    values.push(updates.year);
}
if (updates.price !== undefined) {
    fields.push('price = ?');
    values.push(updates.price);
}
if (updates.in_stock !== undefined) {
    fields.push('in_stock = ?');
    values.push(updates.in_stock);
}

if (fields.length === 0) {
    return res.status(400).json({ error: 'Нет полей для обновления' });
}

values.push(id);
const sql = `UPDATE books SET ${fields.join(', ')} WHERE id = ?`;

db.run(sql, values, function(err) {
    if (err) return res.status(500).json({ error: err.message });
```

```
            db.get('SELECT * FROM books WHERE id = ?', [id], (err, updatedBook) => {
                if (err) return res.status(500).json({ error: err.message });
                res.json(updatedBook);
            });
        });
    });
});
```

Пример 6. Полный пример сервера (базовый CRUD)

```
javascript
const express = require('express');
const sqlite3 = require('sqlite3').verbose();

const app = express();
const PORT = 3000;

app.use(express.json());  // Парсинг JSON-тела запроса

// Подключение к БД
const db = new sqlite3.Database('./books.db', (err) => {
    if (err) console.error('Ошибка БД:', err.message);
});

// Создание таблицы
db.run(`
    CREATE TABLE IF NOT EXISTS books (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        title TEXT NOT NULL,
        author TEXT NOT NULL,
```

```
        year INTEGER,
        price REAL DEFAULT 0,
        in_stock INTEGER DEFAULT 1
    )
`);

// ========== READ ==========
// GET все книги
app.get('/api/books', (req, res) => {
    db.all('SELECT * FROM books ORDER BY id', [], (err, rows) => {
        if (err) return res.status(500).json({ error: err.message });
        res.json(rows);
    });
});

// GET книга по ID
app.get('/api/books/:id', (req, res) => {
    const id = parseInt(req.params.id);
    db.get('SELECT * FROM books WHERE id = ?', [id], (err, row) => {
        if (err) return res.status(500).json({ error: err.message });
        if (!row) return res.status(404).json({ error: 'Книга не найдена' });
        res.json(row);
    });
});

// ========== CREATE ==========
app.post('/api/books', (req, res) => {
    const { title, author, year, price, in_stock } = req.body;
```

```
    if (!title || !author) {
      return res.status(400).json({ error: 'title и author обязательны' });
    }

    db.run(
      `INSERT INTO books (title, author, year, price, in_stock)
       VALUES (?, ?, ?, ?, ?)`,
      [title, author, year || null, price || 0, in_stock !== undefined ? in_stock : 1],
      function(err) {
        if (err) return res.status(500).json({ error: err.message });

        db.get('SELECT * FROM books WHERE id = ?', [this.lastID], (err, newBook) => {
          if (err) return res.status(500).json({ error: err.message });
          res.status(201).json(newBook);
        });
      }
    );
  });

  // ========== UPDATE ==========
  app.put('/api/books/:id', (req, res) => {
    const id = parseInt(req.params.id);
    const { title, author, year, price, in_stock } = req.body;

    if (!title || !author) {
      return res.status(400).json({ error: 'title и author обязательны' });
    }
```

```
        db.get('SELECT * FROM books WHERE id = ?', [id], (err, book) => {
            if (err) return res.status(500).json({ error: err.message });
            if (!book) return res.status(404).json({ error: 'Книга не найдена' });

            db.run(
                `UPDATE books SET title = ?, author = ?, year = ?, price = ?, in_stock = ?
                 WHERE id = ?`,
                [title, author, year || null, price || 0, in_stock !== undefined ? in_stock : 1, id],
                function(err) {
                    if (err) return res.status(500).json({ error: err.message });

                    db.get('SELECT * FROM books WHERE id = ?', [id], (err, updatedBook) => {
                        if (err) return res.status(500).json({ error: err.message });
                        res.json(updatedBook);
                    });
                }
            );
        });
    });

    // ========== DELETE ==========
    app.delete('/api/books/:id', (req, res) => {
        const id = parseInt(req.params.id);

        db.get('SELECT * FROM books WHERE id = ?', [id], (err, book) => {
```

```
        if (err) return res.status(500).json({ error: err.message });
        if (!book) return res.status(404).json({ error: 'Книга не найдена' });

        db.run('DELETE FROM books WHERE id = ?', [id], function(err) {
            if (err) return res.status(500).json({ error: err.message });
            res.status(204).send();
        });
    });
});

// Обработчик 404
app.use((req, res) => {
    res.status(404).json({ error: 'Маршрут не найден' });
});

// Закрытие БД
process.on('SIGINT', () => {
    db.close(() => process.exit(0));
});

app.listen(PORT, () => {
    console.log(`Сервер запущен на http://localhost:${PORT}`);
});
```

2.2. Индивидуальные задания

2.2.1. Базовый уровень (15 вариантов)

Общее задание для базового уровня:

Реализовать сервер с полным CRUD для указанной сущности:

1. GET /api/items — получение всех записей.

2. GET /api/items/:id — получение записи по ID.
3. POST /api/items — создание новой записи (с валидацией обязательных полей).

Данные хранятся в SQLite, таблица создаётся автоматически при запуске.

Варианты заданий (сущности и обязательные поля):

№	Сущность	Обязательные поля	Дополнительные поля
1	Книги	title, author	year, price, in_stock
2	Студенты	full_name, group_name	age, email, phone
3	Товары	name, category	price, quantity, description
4	Фильмы	title, director	year, rating, genre
5	Сотрудники	full_name, position	salary, department, hire_date
6	Рецепты	title, cuisine	cooking_time, difficulty, ingredients
7	Автомобили	model, brand	year, price, engine
8	Песни	title, artist	album, duration, genre

№	Сущность	Обязательные поля	Дополнительные поля
9	Отели	name, city	stars, price_per_night, address
10	Курсы	title, instructor	duration, price, level
11	Лекарства	name, manufacturer	price, prescription, in_stock
12	Мероприятия	name, location	date, price, max_participants
13	Заказы	customer_name, product_name	quantity, amount, status
14	Статьи	title, author	published_at, views, content
15	Спортсмены	full_name, sport	country, medals, birth_year

Пример выполнения для варианта 1 (Базовый уровень):

javascript

```
const express = require('express');
const sqlite3 = require('sqlite3').verbose();
const app = express();
app.use(express.json());

const db = new sqlite3.Database('./books.db');
```

```
// Создание таблицы
db.run(`CREATE TABLE IF NOT EXISTS books (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    title TEXT NOT NULL,
    author TEXT NOT NULL,
    year INTEGER,
    price REAL DEFAULT 0,
    in_stock INTEGER DEFAULT 1
)`);

// GET все книги
app.get('/api/books', (req, res) => {
    db.all('SELECT * FROM books', [], (err, rows) => {
        if (err) return res.status(500).json({ error: err.message });
        res.json(rows);
    });
});

// GET книга по ID
app.get('/api/books/:id', (req, res) => {
    const id = parseInt(req.params.id);
    db.get('SELECT * FROM books WHERE id = ?', [id], (err, row) => {
        if (err) return res.status(500).json({ error: err.message });
        if (!row) return res.status(404).json({ error: 'Не найдено' });
        res.json(row);
    });
});
```

```
// POST создать книгу
app.post('/api/books', (req, res) => {
  const { title, author, year, price, in_stock } = req.body;
  if (!title || !author) {
    return res.status(400).json({ error: 'title и author обязательны' });
  }
  db.run(
    `INSERT INTO books (title, author, year, price, in_stock) VALUES (?, ?, ?, ?, ?)`,
    [title, author, year || null, price || 0, in_stock !== undefined ? in_stock : 1],
    function(err) {
      if (err) return res.status(500).json({ error: err.message });
      res.status(201).json({ id: this.lastID, title, author, year, price, in_stock });
    }
  );
});

app.listen(3000);
```

2.2.2. Средний уровень (15 вариантов)

Общее задание для среднего уровня:

Выполнить требования базового уровня, а также дополнительно реализовать:

1. PUT-маршрут для полного обновления записи (с проверкой существования).
2. DELETE-маршрут для удаления записи (с проверкой существования).

3. Валидацию всех полей перед вставкой и обновлением.
4. Возврат созданной/обновлённой записи в ответе на POST/PUT (получение через SELECT).
5. Для DELETE — возврат статуса 204 No Content.

Дополнительные требования к валидации (для каждого варианта):

№	Дополнительная валидация
1	year: 0–2025; price: ≥0
2	age: 0–120; email: формат
3	price: ≥0; quantity: ≥0
4	year: 1900–2025; rating: 0–10
5	salary: ≥0
6	cooking_time: ≥0; difficulty: 'easy','medium','hard'
7	year: 1900–2025; price: ≥0
8	duration: ≥0
9	stars: 1–5; price_per_night: ≥0
10	duration: ≥0; price: ≥0
11	price: ≥0; prescription: 0/1
12	price: ≥0; max_participants: ≥1
13	quantity: ≥1; amount: ≥0

№	Дополнительная валидация
14	published_at: формат ISO
15	medals: ≥0

Пример выполнения для варианта 1 (Средний уровень) — дополнение к базовому:

javascript

```
// PUT обновление книги
app put('/api/books/:id', (req, res) => {
    const id = parseInt(req params id);
    const { title, author, year, price, in_stock } = req body,

    if (!title || !author) {
        return res status(400) json({ error: 'title и author обязательны' });
    }

    if (year && (year < 0 || year > 2025)) {
        return res status(400) json({ error: 'year должен быть от 0 до 2025' });
    }

    if (price && price < 0) {
        return res status(400) json({ error: 'price не может быть отрицательным' });
    }

    // Проверка существования
    db get('SELECT * FROM books WHERE id = ?', [id], (err, book) => {
        if (err) return res status(500) json({ error: err message });
```

```
            if (!book) return res.status(404).json({ error: 'Книга не найдена' });

            db.run(
              `UPDATE books SET title = ?, author = ?, year = ?, price = ?, in_stock = ?
               WHERE id = ?`,
              [title, author, year || null, price || 0, in_stock !== undefined ? in_stock : 1, id],
              function(err) {
                if (err) return res.status(500).json({ error: err.message });

                db.get('SELECT * FROM books WHERE id = ?', [id], (err, updatedBook) => {
                  if (err) return res.status(500).json({ error: err.message });
                  res.json(updatedBook);
                });
              }
            );
          });
        });

        // DELETE удаление книги
        app.delete('/api/books/:id', (req, res) => {
          const id = parseInt(req.params.id);

          db.get('SELECT * FROM books WHERE id = ?', [id], (err, book) => {
            if (err) return res.status(500).json({ error: err.message });
            if (!book) return res.status(404).json({ error: 'Книга не найдена' });
```

```
        db.run('DELETE FROM books WHERE id = ?', [id], function(err) {
            if (err) return res.status(500).json({ error: err.message });
            res.status(204).send();
        });
    });
});
```

2.2.3. Продвинутый уровень (15 вариантов)

Общее задание для продвинутого уровня:

Выполнить требования среднего уровня, а также дополнительно реализовать:

1. Две связанные таблицы (справочник + основная) с внешним ключом.
2. PATCH-маршрут для частичного обновления записи (обновление только переданных полей).
3. Каскадное удаление (при удалении из справочника — удаление связанных записей или установка NULL).
4. Логирование всех CRUD-операций (в файл или консоль).
5. Переменные окружения (.env) для порта и пути к БД.
6. Обработчик глобальных ошибок.

Структура связанных таблиц (для всех вариантов):

	Основная таблица	**Справочник**	**Поля для связи**
	books	authors	author_id → authors.id
	students	groups	group_id → groups.id
	products	categorie	category_id → categories.id

	Основная таблица	Справочник	Поля для связи
		s	
	movies	directors	director_id → directors.id
	employees	departments	department_id → departments.id
	recipes	cuisines	cuisine_id → cuisines.id
	cars	brands	brand_id → brands.id
	songs	artists	artist_id → artists.id
	hotels	cities	city_id → cities.id
0	courses	instructors	instructor_id → instructors.id
1	medicines	manufacturers	manufacturer_id → manufacturers.id
2	events	locations	location_id → locations.id
3	orders	customers	customer_id → customers.id
4	articles	authors_articles	author_id → authors_articles.id

	Основная таблица	Справочник	Поля для связи
5	athletes	sports	sport_id → sports.id

Пример выполнения для варианта 1 (Продвинутый уровень):

javascript

```
require('dotenv').config();
const express = require('express');
const sqlite3 = require('sqlite3').verbose();
const fs = require('fs');

const app = express();
const PORT = process.env.PORT || 3000;
const DB_PATH = process.env.DB_PATH || './library.db';

app.use(express.json());

// Логирование CRUD-операций
const logStream = fs.createWriteStream('./crud.log', { flags: 'a' });
function logCRUD(operation, endpoint, data) {
  const logEntry = `${new Date().toISOString()} | ${operation} | ${endpoint} | ${JSON.stringify(data)}\n`;
  logStream.write(logEntry);
  console.log(logEntry.trim());
}

// Подключение к БД
const db = new sqlite3.Database(DB_PATH, (err) => {
```

```
    if (err) {
        console.error('Ошибка БД:', err.message);
        process.exit(1);
    }
    console.log('Подключено к БД:', DB_PATH);

    // Создание таблиц
    db.run(`CREATE TABLE IF NOT EXISTS authors (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT NOT NULL UNIQUE
    )`);

    db.run(`CREATE TABLE IF NOT EXISTS books (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        title TEXT NOT NULL,
        author_id INTEGER NOT NULL,
        year INTEGER CHECK (year >= 0 AND year <= 2025),
        price REAL DEFAULT 0,
        in_stock INTEGER DEFAULT 1,
        FOREIGN KEY (author_id) REFERENCES authors(id) ON DELETE CASCADE
    )`);

    // Заполнение тестовыми данными
    db.get('SELECT COUNT(*) as count FROM authors', [], (err, row) => {
        if (row.count === 0) {
            db.run(`INSERT INTO authors (name) VALUES ('Лев Толстой'), ('Фёдор Достоевский')`);
            db.run(`INSERT INTO books (title, author_id, year, price) VALUES
```

```
            ('Война и мир', 1, 1869, 599),
            ('Анна Каренина', 1, 1877, 549),
            ('Преступление и наказание', 2, 1866, 499)`);
        }
    });
});

// ========== CRUD для авторов (справочник) ==========
app.get('/api/authors', (req, res) => {
    db.all('SELECT * FROM authors ORDER BY name', [], (err, rows) => {
        if (err) return res.status(500).json({ error: err.message });
        res.json(rows);
    });
});

app.post('/api/authors', (req, res) => {
    const { name } = req.body;
    if (!name) return res.status(400).json({ error: 'name обязателен' });

    db.run('INSERT INTO authors (name) VALUES (?)', [name], function(err) {
        if (err) return res.status(500).json({ error: err.message });
        res.status(201).json({ id: this.lastID, name });
    });
});

app.delete('/api/authors/:id', (req, res) => {
    const id = parseInt(req.params.id);
    db.run('DELETE FROM authors WHERE id = ?', [id], function(err) {
```

```
        if (err) return res.status(500).json({ error: err.message });
        res.status(204).send();
    });
});

// ========== CRUD для книг (основная таблица) ==========
// GET все книги (с JOIN)
app.get('/api/books', (req, res) => {
    const sql = `
        SELECT books.*, authors.name as author_name
        FROM books
        JOIN authors ON books.author_id = authors.id
        ORDER BY books.id
    `;
    db.all(sql, [], (err, rows) => {
        if (err) return res.status(500).json({ error: err.message });
        res.json(rows);
    });
});

// GET книга по ID
app.get('/api/books/:id', (req, res) => {
    const id = parseInt(req.params.id);
    const sql = `
        SELECT books.*, authors.name as author_name
        FROM books
        JOIN authors ON books.author_id = authors.id
        WHERE books.id = ?
    `;
```

```
        db.get(sql, [id], (err, row) => {
            if (err) return res.status(500).json({ error: err.message });
            if (!row) return res.status(404).json({ error: 'Книга не найдена' });
            res.json(row);
        });
    });

    // POST создание книги
    app.post('/api/books', (req, res) => {
        const { title, author_id, year, price, in_stock } = req.body;

        if (!title || !author_id) {
            return res.status(400).json({ error: 'title и author_id обязательны' });
        }

        // Проверка существования автора
        db.get('SELECT * FROM authors WHERE id = ?', [author_id], (err, author) => {
            if (err) return res.status(500).json({ error: err.message });
            if (!author) return res.status(400).json({ error: 'Автор с таким ID не существует' });

            db.run(
                `INSERT INTO books (title, author_id, year, price, in_stock)
                 VALUES (?, ?, ?, ?, ?)`,
                [title, author_id, year || null, price || 0, in_stock !== undefined ? in_stock : 1],
                function(err) {
                    if (err) return res.status(500).json({ error: err.message });
```

```
                logCRUD('CREATE', '/api/books', { id: this.lastID, title,
author_id });

                db.get(
                    `SELECT books.*, authors.name as author_name
                     FROM books
                     JOIN authors ON books.author_id = authors.id
                     WHERE books.id = ?`,
                    [this.lastID],
                    (err, newBook) => {
                        if (err) return res.status(500).json({ error: err.message });
                        res.status(201).json(newBook);
                    }
                );
            }
        );
    });
});

// PUT полное обновление книги
app.put('/api/books/:id', (req, res) => {
    const id = parseInt(req.params.id);
    const { title, author_id, year, price, in_stock } = req.body;

    if (!title || !author_id) {
        return res.status(400).json({ error: 'title и author_id обязательны' });
    }
```

```
        // Проверка существования книги
        db.get('SELECT * FROM books WHERE id = ?', [id], (err, book) => {
            if (err) return res.status(500).json({ error: err.message });
            if (!book) return res.status(404).json({ error: 'Книга не найдена' });

            // Проверка существования автора
            db.get('SELECT * FROM authors WHERE id = ?', [author_id], (err, author) => {
                if (err) return res.status(500).json({ error: err.message });
                if (!author) return res.status(400).json({ error: 'Автор с таким ID не существует' });

                db.run(
                    `UPDATE books SET title = ?, author_id = ?, year = ?, price = ?, in_stock = ?
                     WHERE id = ?`,
                    [title, author_id, year || null, price || 0, in_stock !== undefined ? in_stock : 1, id],
                    function(err) {
                        if (err) return res.status(500).json({ error: err.message });

                        logCRUD('UPDATE', `/api/books/${id}`, { title, author_id });

                        db.get(
                            `SELECT books.*, authors.name as author_name
                             FROM books
                             JOIN authors ON books.author_id = authors.id
                             WHERE books.id = ?`,
                            [id],
```

```
            (err, updatedBook) => {
                if (err) return res.status(500).json({ error: err.message });
                res.json(updatedBook);
            }
        );
    }
  );
 });
});

// PATCH частичное обновление книги
app.patch('/api/books/:id', (req, res) => {
  const id = parseInt(req.params.id);
  const updates = req.body;

  db.get('SELECT * FROM books WHERE id = ?', [id], (err, book) => {
    if (err) return res.status(500).json({ error: err.message });
    if (!book) return res.status(404).json({ error: 'Книга не найдена' });

    const fields = [];
    const values = [];

    if (updates.title !== undefined) {
      fields.push('title = ?');
      values.push(updates.title);
    }
    if (updates.author_id !== undefined) {
      fields.push('author_id = ?');
```

```
    values.push(updates.author_id);
  }
  if (updates.year !== undefined) {
    fields.push('year = ?');
    values.push(updates.year);
  }
  if (updates.price !== undefined) {
    fields.push('price = ?');
    values.push(updates.price);
  }
  if (updates.in_stock !== undefined) {
    fields.push('in_stock = ?');
    values.push(updates.in_stock);
  }

  if (fields.length === 0) {
    return res.status(400).json({ error: 'Нет полей для обновления' });
  }

  values.push(id);
  const sql = `UPDATE books SET ${fields.join(', ')} WHERE id = ?`;

  db.run(sql, values, function(err) {
    if (err) return res.status(500).json({ error: err.message });

    logCRUD('PATCH', `/api/books/${id}`, updates);

    db.get(
      `SELECT books.*, authors.name as author_name
```

```
            FROM books
            JOIN authors ON books.author_id = authors.id
            WHERE books.id = ?`,
            [id],
            (err, updatedBook) => {
              if (err) return res.status(500).json({ error: err.message });
              res.json(updatedBook);
            }
          );
        });
    });
});

// DELETE удаление книги
app.delete('/api/books/:id', (req, res) => {
    const id = parseInt(req.params.id);

    db.get('SELECT * FROM books WHERE id = ?', [id], (err, book) => {
        if (err) return res.status(500).json({ error: err.message });
        if (!book) return res.status(404).json({ error: 'Книга не найдена' });

        db.run('DELETE FROM books WHERE id = ?', [id], function(err) {
            if (err) return res.status(500).json({ error: err.message });

            logCRUD('DELETE', `/api/books/${id}`, { id });
            res.status(204).send();
        });
    });
});
```

```
// Глобальный обработчик ошибок
app.use((err, req, res, next) => {
    console.error('Глобальная ошибка:', err.stack);
    res.status(500).json({ error: 'Внутренняя ошибка сервера' });
});

// Обработчик 404
app.use((req, res) => {
    res.status(404).json({ error: 'Маршрут не найден' });
});

process.on('SIGINT', () => {
    logStream.end();
    db.close(() => process.exit(0));
});

app.listen(PORT, () => {
    console.log(`Сервер запущен на http://localhost:${PORT}`);
});
```

Файл .env:

```
text
PORT=3000
DB_PATH=./library.db
```

Установка дополнительных пакетов:

```
bash
npm install dotenv
```

3. Контрольные вопросы

1. Какие HTTP-методы соответствуют CRUD-операциям?
2. В чём разница между PUT и PATCH?
3. Какой HTTP-код возвращается при успешном создании ресурса (POST)?
4. Какой HTTP-код возвращается при успешном удалении ресурса?
5. Как получить ID последней вставленной записи в SQLite?
6. Зачем нужна проверка существования записи перед UPDATE/DELETE?
7. Как выполнить валидацию входных данных перед вставкой в БД?
8. Как реализовать частичное обновление (PATCH) с динамическим SQL?
9. Что такое каскадное удаление (ON DELETE CASCADE)?
10. Для чего нужны переменные окружения (.env)?
11. Как организовать логирование CRUD-операций?
12. Что такое внешний ключ (FOREIGN KEY) и зачем он нужен?
13. Как выполнить JOIN для получения связанных данных?
14. Какой код возвращается при невалидных входных данных?
15. Что такое глобальный обработчик ошибок в Express?

4. Требования к отчёту

Отчёт должен быть оформлен в электронном виде и содержать:

Титульный лист с названием работы, ФИО студента, номером группы, вариантом.

Цель и задачи работы (переписать из методических указаний).

Теоретическое обоснование (краткий конспект по теме).

Ход выполнения работы:

SQL-скрипты создания таблицы/таблиц.

Листинг кода сервера с комментариями.

- Скриншоты проверки работы через Postman:

- POST /api/items (создание)
- GET /api/items (чтение всех)
- GET /api/items/:id (чтение одного)
- PUT /api/items/:id (полное обновление)
- PATCH /api/items/:id (частичное обновление — для продвинутого уровня)
- DELETE /api/items/:id (удаление)
- Скриншоты обработки ошибок (400, 404).

Ответы на контрольные вопросы (письменно, развёрнуто).

Заключение (выводы о проделанной работе, какие навыки приобретены).

Список использованных источников.

5. Критерии оценивания

Уровень выполнения	Требования	Оценка
Базовый	Реализованы GET (все и по ID), POST с созданием записи в БД. Валидация обязательных полей.	Удовлетворительно
Средний	Выполнены требования базового уровня + PUT и DELETE с проверкой существования записи + расширенная валидация.	Хорошо
Продвинутый	Выполнены требования среднего уровня + две связанные	Отлично

Уровень выполнения	Требования	Оценка
	таблицы + PATCH-частичное обновление + логирование + переменные окружения + каскадное удаление.	

Дополнительные условия:

Код должен быть структурирован и снабжён комментариями.

Отчёт должен содержать скриншоты всех CRUD-операций.

При защите студент должен объяснить структуру БД и логику CRUD-операций.

6. Список рекомендуемой литературы и интернет-ресурсов

Литература (доступна в РФ)

1. Браун, И. (2021). *Express.js: Разработка веб-приложений на Node.js*. ДМК Пресс.
2. Кантелон, М., Хартер, М., & Холстейн, С. (2020). *Node.js в действии*. Питер.

Интернет-ресурсы

Название	Ссылка	Описание
Официальная документация sqlite3	https://github.com/TryGhost/node-sqlite3	Документация и примеры

Название	Ссылка	Описание
Metanit — Express + SQLite CRUD	https://metanit.com/web/nodejs/5.3.php	Примеры CRUD на Express
Документация Express — обработка запросов	https://expressjs.com/ru/api.html	API Express
SQLite FOREIGN KEY	https://sqlite.org/foreignkeys.html	Документация по внешним ключам
Документация Dotenv	https://github.com/motdotla/dotenv	Переменные окружения
REST API — коды статусов	https://developer.mozilla.org/ru/docs/Web/HTTP/Status	Коды HTTP

Все ресурсы доступны и легальны на территории Российской Федерации.

Лабораторная работа № 7

Тема: Фильтрация, поиск и сортировка данных в SQL

Цель работы:

Научиться реализовывать фильтрацию, поиск и сортировку данных на стороне сервера с использованием SQL-запросов, а также передавать параметры фильтрации через query-параметры URL.

Задачи работы:

1. Реализовать фильтрацию данных по одному и нескольким полям через query-параметры.
2. Освоить операторы WHERE, AND, OR, LIKE для фильтрации данных.
3. Реализовать полнотекстовый поиск по нескольким полям с использованием LIKE.
4. Реализовать сортировку данных с использованием ORDER BY ASC/DESC.
5. Реализовать пагинацию с использованием LIMIT и OFFSET.
6. Научиться комбинировать фильтрацию, поиск, сортировку и пагинацию в одном запросе.

1. Теоретическое обоснование

1.1. Фильтрация данных с WHERE

Оператор WHERE используется для фильтрации строк по условию.

Синтаксис:

sql

```
SELECT * FROM table_name WHERE condition;
```

Примеры условий:

Условие	Оператор	Пример
Равенство	=	WHERE author = 'Толстой'
Неравенство	!= или <>	WHERE price != 0

Условие	Оператор	Пример
Больше	>	WHERE year > 2000
Меньше	<	WHERE price < 500
Больше или равно	>=	WHERE rating >= 4.5
Меньше или равно	<=	WHERE year <= 2020
Диапазон	BETWEEN	WHERE year BETWEEN 1800 AND 1900
Список значений	IN	WHERE author IN ('Толстой', 'Достоевский')
Частичное совпадение	LIKE	WHERE title LIKE '%война%'

1.2. Составные условия (AND, OR)

sql

```
-- AND: все условия должны быть истинны
SELECT * FROM books WHERE year > 1800 AND year < 1900 AND price < 500;

-- OR: хотя бы одно условие истинно
SELECT * FROM books WHERE author = 'Толстой' OR author = 'Достоевский';

-- Комбинация AND и OR (порядок важен, используйте скобки)
```

```
SELECT * FROM books WHERE (author = 'Толстой' OR author = 'Достоевский') AND price < 600;
```

1.3. Поиск с LIKE

Оператор LIKE используется для поиска по шаблону:

% — любое количество любых символов (включая ноль)

_ — ровно один любой символ

sql

```
-- Начинается с "Война"
SELECT * FROM books WHERE title LIKE 'Война%';

-- Содержит "мир"
SELECT * FROM books WHERE title LIKE '%мир%';

-- Заканчивается на "ение"
SELECT * FROM books WHERE title LIKE '%ение';

-- Название из 5 букв
SELECT * FROM books WHERE title LIKE '_____';

-- Поиск по нескольким полям
SELECT * FROM books WHERE title LIKE '%война%' OR author LIKE '%толстой%';
```

1.4. Сортировка ORDER BY

sql

```
-- Сортировка по возрастанию (ASC — по умолчанию)
SELECT * FROM books ORDER BY year;
```

```
-- Сортировка по убыванию
SELECT * FROM books ORDER BY price DESC;

-- Сортировка по нескольким полям
SELECT * FROM books ORDER BY author ASC, year DESC;

-- Сортировка по вычисляемому полю
SELECT *, price * quantity AS total FROM orders ORDER BY total DESC;
```

1.5. Пагинация LIMIT и OFFSET

sql

```
-- Первые 10 записей
SELECT * FROM books LIMIT 10;

-- Записи с 11 по 20 (страница 2, лимит 10)
SELECT * FROM books LIMIT 10 OFFSET 10;

-- Альтернативный синтаксис
SELECT * FROM books LIMIT 10 OFFSET 10;  -- то же самое
```

Формулы:

OFFSET = (page - 1) * limit

page — номер страницы (начиная с 1)

limit — количество записей на странице

1.6. Подсчёт количества записей

Для пагинации необходимо знать общее количество записей:

sql

```
-- Общее количество записей в таблице
```

```
SELECT COUNT(*) as total FROM books;

-- Количество записей с учётом фильтров
SELECT COUNT(*) as total FROM books WHERE author = 'Толстой';
```

1.7. Динамическое построение SQL-запросов

При фильтрации из серверного кода SQL-запрос строится динамически в зависимости от переданных параметров:

```
javascript
app.get('/api/books', (req, res) => {
  let sql = 'SELECT * FROM books WHERE 1=1';  // 1=1 упрощает добавление условий
  const params = [];

  if (req.query.author) {
    sql += ' AND author = ?';
    params.push(req.query.author);
  }

  if (req.query.minYear) {
    sql += ' AND year >= ?';
    params.push(req.query.minYear);
  }

  db.all(sql, params, (err, rows) => { ... });
});
```

2. Методика выполнения работы

2.1. Примеры и их подробный разбор

Пример 1. Базовая фильтрация по одному полю

javascript

```
app.get('/api/books', (req, res) => {
    let sql = 'SELECT * FROM books WHERE 1=1';
    const params = [];

    // Фильтр по автору
    if (req.query.author) {
        sql += ' AND author = ?';
        params.push(req.query.author);
    }

    // Фильтр по году
    if (req.query.year) {
        sql += ' AND year = ?';
        params.push(req.query.year);
    }

    // Фильтр по цене (меньше или равно)
    if (req.query.maxPrice) {
        sql += ' AND price <= ?';
        params.push(req.query.maxPrice);
    }

    db.all(sql, params, (err, rows) => {
        if (err) return res.status(500).json({ error: err.message });
        res.json(rows);
    });
});
```

Примеры запросов:

GET /api/books?author=Толстой — книги Толстого

GET /api/books?year=1869 — книги 1869 года

GET /api/books?maxPrice=500 — книги до 500 рублей

Пример 2. Диапазонная фильтрация (min, max)

javascript

```
app.get('/api/books', (req, res) => {
    let sql = 'SELECT * FROM books WHERE 1=1';
    const params = [];

    // Диапазон лет
    if (req.query.minYear) {
        sql += ' AND year >= ?';
        params.push(req.query.minYear);
    }
    if (req.query.maxYear) {
        sql += ' AND year <= ?';
        params.push(req.query.maxYear);
    }

    // Диапазон цен
    if (req.query.minPrice) {
        sql += ' AND price >= ?';
        params.push(req.query.minPrice);
    }
    if (req.query.maxPrice) {
        sql += ' AND price <= ?';
        params.push(req.query.maxPrice);
    }
```

```
    db.all(sql, params, (err, rows) => {
        if (err) return res.status(500).json({ error: err.message });
        res.json(rows);
    });
});
```

Пример запроса:

GET /api/books?minYear=1860&maxYear=1900&minPrice=400&maxPrice=600

Пример 3. Поиск с LIKE (по нескольким полям)

javascript

```
app.get('/api/books/search', (req, res) => {
    const query = req.query.q;

    if (!query) {
        return res.status(400).json({ error: 'Параметр q обязателен' });
    }

    const searchPattern = `%${query}%`;
    const sql = `
        SELECT * FROM books
        WHERE title LIKE ? OR author LIKE ? OR genre LIKE ?
    `;

    db.all(sql, [searchPattern, searchPattern, searchPattern], (err, rows) => {
        if (err) return res.status(500).json({ error: err.message });
        res.json({
            query: query,
            count: rows.length,
```

```
            results: rows
        });
    });
});
```

Пример запроса: GET /api/books/search?q=война

Пример 4. Сортировка

javascript

```
app.get('/api/books', (req, res) => {
    let sql = 'SELECT * FROM books WHERE 1=1';
    const params = [];

    // Добавляем фильтры (как в примере 1)
    if (req.query.author) {
        sql += ' AND author = ?';
        params.push(req.query.author);
    }

    // Сортировка
    const sortField = req.query.sort || 'id';
    const sortOrder = req.query.order === 'desc' ? 'DESC' : 'ASC';

    // Белый список полей для сортировки (защита от SQL-инъекций)
    const allowedSortFields = ['id', 'title', 'author', 'year', 'price'];
    const finalSortField = allowedSortFields.includes(sortField) ? sortField :
'id';

    sql += ` ORDER BY ${finalSortField} ${sortOrder}`;

    db.all(sql, params, (err, rows) => {
```

```
        if (err) return res.status(500).json({ error: err.message });
        res.json(rows);
    });
});
```

Примеры запросов:

GET /api/books?sort=price&order=desc — по убыванию цены

GET /api/books?author=Толстой&sort=year&order=asc — книги Толстого по возрастанию года

Пример 5. Пагинация

javascript

```
app.get('/api/books', (req, res) => {
    let sql = 'SELECT * FROM books WHERE 1=1';
    const params = [];

    // Фильтры (как в примере 1)
    if (req.query.author) {
        sql += ' AND author = ?';
        params.push(req.query.author);
    }

    // Сортировка
    const sortField = req.query.sort || 'id';
    const sortOrder = req.query.order === 'desc' ? 'DESC' : 'ASC';
    sql += ` ORDER BY ${sortField} ${sortOrder}`;

    // Пагинация
    const page = parseInt(req.query.page) || 1;
    const limit = parseInt(req.query.limit) || 10;
    const offset = (page - 1) * limit;
```

```
    sql += ' LIMIT ? OFFSET ?';
    params.push(limit, offset);

    db.all(sql, params, (err, rows) => {
      if (err) return res.status(500).json({ error: err.message });

      // Подсчёт общего количества (без LIMIT/OFFSET)
      let countSql = 'SELECT COUNT(*) as total FROM books WHERE
1=1';
      const countParams = [];

      if (req.query.author) {
        countSql += ' AND author = ?';
        countParams.push(req.query.author);
      }

      db.get(countSql, countParams, (err, countRow) => {
        if (err) return res.status(500).json({ error: err.message });

        res.json({
          data: rows,
          pagination: {
            total: countRow.total,
            page: page,
            limit: limit,
            totalPages: Math.ceil(countRow.total / limit)
          }
        });
      });
```

```
    });
});
```

Пример 6. Полный пример сервера

```
javascript
const express = require('express');
const sqlite3 = require('sqlite3').verbose();

const app = express();
const PORT = 3000;

// Подключение к БД
const db = new sqlite3.Database('./library.db', (err) => {
    if (err) console.error('Ошибка БД:', err.message);
});

// Создание таблицы
db.run(`
    CREATE TABLE IF NOT EXISTS books (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        title TEXT NOT NULL,
        author TEXT NOT NULL,
        year INTEGER,
        price REAL,
        genre TEXT,
        in_stock INTEGER DEFAULT 1
    )
`);

// Заполнение тестовыми данными
```

```
db.get('SELECT COUNT(*) as count FROM books', [], (err, row) => {
    if (row.count === 0) {
        const books = [
            ['Война и мир', 'Лев Толстой', 1869, 599, 'Роман', 1],
            ['Анна Каренина', 'Лев Толстой', 1877, 549, 'Роман', 1],
            ['Преступление и наказание', 'Фёдор Достоевский', 1866, 499, 'Роман', 1],
            ['Идиот', 'Фёдор Достоевский', 1869, 399, 'Роман', 1],
            ['Мастер и Маргарита', 'Михаил Булгаков', 1967, 699, 'Фантастика', 0],
            ['Собачье сердце', 'Михаил Булгаков', 1925, 349, 'Повесть', 1],
            ['Белая гвардия', 'Михаил Булгаков', 1925, 429, 'Роман', 0],
            ['Двенадцать стульев', 'Илья Ильф и Евгений Петров', 1928, 450, 'Роман', 1],
            ['Золотой телёнок', 'Илья Ильф и Евгений Петров', 1931, 470, 'Роман', 1],
            ['Тихий Дон', 'Михаил Шолохов', 1940, 650, 'Роман', 1]
        ];

        const stmt = db.prepare(`
            INSERT INTO books (title, author, year, price, genre, in_stock)
            VALUES (?, ?, ?, ?, ?, ?)
        `);
        books.forEach(book => stmt.run(book));
        stmt.finalize();
        console.log('Добавлено 10 тестовых книг');
    }
});
```

```
// Белый список полей для сортировки
const allowedSortFields = ['id', 'title', 'author', 'year', 'price', 'genre'];

// GET /api/books — фильтрация, сортировка, пагинация
app.get('/api/books', (req, res) => {
  let sql = 'SELECT * FROM books WHERE 1=1';
  const params = [];

  // ===== Фильтрация =====
  // По автору
  if (req.query.author) {
    sql += ' AND author = ?';
    params.push(req.query.author);
  }

  // По жанру
  if (req.query.genre) {
    sql += ' AND genre = ?';
    params.push(req.query.genre);
  }

  // По диапазону годов
  if (req.query.minYear) {
    sql += ' AND year >= ?';
    params.push(req.query.minYear);
  }
  if (req.query.maxYear) {
    sql += ' AND year <= ?';
    params.push(req.query.maxYear);
```

```
    }

    // По диапазону цен
    if (req.query.minPrice) {
      sql += ' AND price >= ?';
      params.push(req.query.minPrice);
    }
    if (req.query.maxPrice) {
      sql += ' AND price <= ?';
      params.push(req.query.maxPrice);
    }

    // По наличию на складе
    if (req.query.inStock === 'true') {
      sql += ' AND in_stock = 1';
    } else if (req.query.inStock === 'false') {
      sql += ' AND in_stock = 0';
    }

    // ===== Сортировка =====
    const sortField = req.query.sort || 'id';
    const sortOrder = req.query.order === 'desc' ? 'DESC' : 'ASC';
    const finalSortField = allowedSortFields.includes(sortField) ? sortField :
'id';

    sql += ` ORDER BY ${finalSortField} ${sortOrder}`;

    // ===== Пагинация =====
    const page = parseInt(req.query.page) || 1;
    const limit = parseInt(req.query.limit) || 10;
```

```
    const offset = (page - 1) * limit;
    sql += ' LIMIT ? OFFSET ?';
    params.push(limit, offset);

    // Выполнение основного запроса
    db.all(sql, params, (err, rows) => {
      if (err) return res.status(500).json({ error: err.message });

      // Подсчёт общего количества (для пагинации)
      let countSql = 'SELECT COUNT(*) as total FROM books WHERE
1=1';
      const countParams = [];

      if (req.query.author) {
        countSql += ' AND author = ?';
        countParams.push(req.query.author);
      }
      if (req.query.genre) {
        countSql += ' AND genre = ?';
        countParams.push(req.query.genre);
      }
      if (req.query.minYear) {
        countSql += ' AND year >= ?';
        countParams.push(req.query.minYear);
      }
      if (req.query.maxYear) {
        countSql += ' AND year <= ?';
        countParams.push(req.query.maxYear);
      }
```

```
if (req.query.minPrice) {
  countSql += ' AND price >= ?';
  countParams.push(req.query.minPrice);
}
if (req.query.maxPrice) {
  countSql += ' AND price <= ?';
  countParams.push(req.query.maxPrice);
}
if (req.query.inStock === 'true') {
  countSql += ' AND in_stock = 1';
} else if (req.query.inStock === 'false') {
  countSql += ' AND in_stock = 0';
}

db.get(countSql, countParams, (err, countRow) => {
  if (err) return res.status(500).json({ error: err.message });

  res.json({
    data: rows,
    filters: {
      author: req.query.author || null,
      genre: req.query.genre || null,
      yearRange: {
        min: req.query.minYear || null,
        max: req.query.maxYear || null
      },
      priceRange: {
        min: req.query.minPrice || null,
        max: req.query.maxPrice || null
```

```
            },
            inStock: req.query.inStock || null
        },
        sort: {
            field: finalSortField,
            order: sortOrder
        },
        pagination: {
            total: countRow.total,
            page: page,
            limit: limit,
            totalPages: Math.ceil(countRow.total / limit)
        }
    });
  });
});

// GET /api/books/search — поиск по тексту
app.get('/api/books/search', (req, res) => {
  const query = req.query.q;

  if (!query) {
    return res.status(400).json({ error: 'Параметр q обязателен' });
  }

  const searchPattern = `%${query}%`;
  const sql = `
    SELECT *,
```

```
            CASE
                WHEN title LIKE ? THEN 10
                WHEN author LIKE ? THEN 5
                WHEN genre LIKE ? THEN 2
                ELSE 0
            END as relevance
        FROM books
        WHERE title LIKE ? OR author LIKE ? OR genre LIKE ?
        ORDER BY relevance DESC
    `;

    db.all(sql, [searchPattern, searchPattern, searchPattern, searchPattern, searchPattern, searchPattern],
        (err, rows) => {
            if (err) return res.status(500).json({ error: err.message });
            res.json({
                query: query,
                count: rows.length,
                results: rows
            });
        }
    );
});

// GET /api/books/filters — получение доступных фильтров
app.get('/api/books/filters', (req, res) => {
    const sql = `
        SELECT
            GROUP_CONCAT(DISTINCT author) as authors,
```

```
            GROUP_CONCAT(DISTINCT genre) as genres,
            MIN(year) as minYear,
            MAX(year) as maxYear,
            MIN(price) as minPrice,
            MAX(price) as maxPrice
        FROM books
    `;

    db.get(sql, [], (err, row) => {
        if (err) return res.status(500).json({ error: err.message });
        res.json({
            authors: row.authors ? row.authors.split(',') : [],
            genres: row.genres ? row.genres.split(',') : [],
            yearRange: { min: row.minYear, max: row.maxYear },
            priceRange: { min: row.minPrice, max: row.maxPrice }
        });
    });
});

app.listen(PORT, () => {
    console.log(`Сервер запущен на http://localhost:${PORT}`);
    console.log('Доступные эндпоинты:');
    console.log('  GET /api/books — фильтрация, сортировка, пагинация');
    console.log('  GET /api/books/search?q= — поиск');
    console.log('  GET /api/books/filters — доступные фильтры');
});
```

2.2. Индивидуальные задания

2.2.1. Базовый уровень (15 вариантов)

Общее задание для базового уровня: Реализовать сервер с базой данных SQLite, содержащей таблицу согласно варианту. Реализовать эндпоинт GET /api/items с возможностью:

Фильтрации по одному полю (по выбору из таблицы вариантов).

Поиска (q) по одному текстовому полю с использованием LIKE.

Варианты заданий:

	Сущность	Поля таблицы	Поле для фильтрации	Поле для поиска (q)
	Книги	id, title, author, year, price	author	title
	Студенты	id, full_name, group, age, email	group	full_name
	Товары	id, name, category, price, stock	category	name
	Фильмы	id, title, director, year, rating	director	title
	Сотрудники	id, name, position, salary, department	department	name
	Рецепты	id, title, cuisine, cooking_time, difficulty	cuisine	title

	Сущность	Поля таблицы	Поле для фильтрации	Поле для поиска (q)
	Автомобили	id, model, brand, year, price	brand	model
	Песни	id, title, artist, album, duration	artist	title
	Отели	id, name, city, stars, price	city	name
0	Курсы	id, title, instructor, duration, price	instructor	title
1	Лекарства	id, name, manufacturer, price, prescription	manufacturer	name
2	Мероприятия	id, name, location, date, price	location	name
3	Заказы	id, customer, product, amount, status	status	customer
4	Статьи	id, title, author, published_at, views	author	title

	Сущность	Поля таблицы	Поле для фильтрации	Поле для поиска (q)
5	Спортсмены	id, name, sport, country, medals	sport	name

Пример выполнения для варианта 1 (Базовый уровень):

javascript

```
const express = require('express');
const sqlite3 = require('sqlite3').verbose();
const app = express();

const db = new sqlite3.Database('./books.db');

// Создание таблицы и тестовых данных
db.run(`CREATE TABLE IF NOT EXISTS books (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    title TEXT NOT NULL,
    author TEXT NOT NULL,
    year INTEGER,
    price REAL
)`);

// GET /api/books — фильтрация и поиск
app.get('/api/books', (req, res) => {
    let sql = 'SELECT * FROM books WHERE 1=1';
    const params = [];

    // Фильтр по автору
```

```
    if (req.query.author) {
        sql += ' AND author = ?';
        params.push(req.query.author);
    }

    // Поиск по названию
    if (req.query.q) {
        sql += ' AND title LIKE ?';
        params.push(`%${req.query.q}%`);
    }

    db.all(sql, params, (err, rows) => {
        if (err) return res.status(500).json({ error: err.message });
        res.json(rows);
    });
});

app.listen(3000);
```

2.2.2. Средний уровень (15 вариантов)

Общее задание для среднего уровня: Выполнить требования базового уровня, а также дополнительно реализовать:

Диапазонную фильтрацию для числового поля (min/max) — например, minYear, maxYear или minPrice, maxPrice.

Сортировку по любому полю с параметрами sort (поле) и order (asc/desc).

Пагинацию с параметрами page и limit (значения по умолчанию: page=1, limit=10).

Возврат мета-информации (total, page, limit, totalPages).

Дополнительные параметры для каждого варианта (средний уровень):

	Поле для диапазонной фильтрации	Поля для сортировки (allowedSortFields)
	year, price	id, title, author, year, price
	age	id, full_name, group, age
	price, stock	id, name, category, price
	year, rating	id, title, director, year, rating
	salary	id, name, position, salary
	cooking_time	id, title, cuisine, cooking_time
	year, price	id, model, brand, year, price
	duration	id, title, artist, duration
	stars, price	id, name, city, stars
0	duration, price	id, title, instructor, duration
1	price	id, name, manufacturer, price
2	price	id, name, location, date
	amount	id, customer, amount, status

	Поле для диапазонной фильтрации	Поля для сортировки (allowedSortFields)
3		
4	views	id, title, author, published_at
5	medals	id, name, sport, medals

Пример выполнения для варианта 1 (Средний уровень):

javascript

```
const express = require('express');
const sqlite3 = require('sqlite3').verbose();
const app = express();

const db = new sqlite3.Database('./books.db');

// Белый список полей для сортировки
const allowedSortFields = ['id', 'title', 'author', 'year', 'price'];

app.get('/api/books', (req, res) => {
    let sql = 'SELECT * FROM books WHERE 1=1';
    const params = [];

    // Фильтр по автору
    if (req.query.author) {
        sql += ' AND author = ?';
```

```
    params.push(req.query.author);
}

// Поиск по названию
if (req.query.q) {
    sql += ' AND title LIKE ?';
    params.push(`%${req.query.q}%`);
}

// Диапазонная фильтрация по году
if (req.query.minYear) {
    sql += ' AND year >= ?';
    params.push(req.query.minYear);
}
if (req.query.maxYear) {
    sql += ' AND year <= ?';
    params.push(req.query.maxYear);
}

// Диапазонная фильтрация по цене
if (req.query.minPrice) {
    sql += ' AND price >= ?';
    params.push(req.query.minPrice);
}
if (req.query.maxPrice) {
    sql += ' AND price <= ?';
    params.push(req.query.maxPrice);
}
```

```
        // Сортировка
        const sortField = req.query.sort || 'id';
        const sortOrder = req.query.order === 'desc' ? 'DESC' : 'ASC';
        const finalSortField = allowedSortFields.includes(sortField) ? sortField : 'id';

        sql += ` ORDER BY ${finalSortField} ${sortOrder}`;

        // Пагинация
        const page = parseInt(req.query.page) || 1;
        const limit = parseInt(req.query.limit) || 10;
        const offset = (page - 1) * limit;
        sql += ' LIMIT ? OFFSET ?';
        params.push(limit, offset);

        db.all(sql, params, (err, rows) => {
            if (err) return res.status(500).json({ error: err.message });

            // Подсчёт общего количества
            let countSql = 'SELECT COUNT(*) as total FROM books WHERE 1=1';
            const countParams = [];

            if (req.query.author) {
                countSql += ' AND author = ?';
                countParams.push(req.query.author);
            }
            if (req.query.q) {
                countSql += ' AND title LIKE ?';
                countParams.push(`%${req.query.q}%`);
```

```
}
if (req.query.minYear) {
  countSql += ' AND year >= ?';
  countParams.push(req.query.minYear);
}
if (req.query.maxYear) {
  countSql += ' AND year <= ?';
  countParams.push(req.query.maxYear);
}
if (req.query.minPrice) {
  countSql += ' AND price >= ?';
  countParams.push(req.query.minPrice);
}
if (req.query.maxPrice) {
  countSql += ' AND price <= ?';
  countParams.push(req.query.maxPrice);
}

db.get(countSql, countParams, (err, countRow) => {
  if (err) return res.status(500).json({ error: err.message });

  res.json({
    data: rows,
    pagination: {
      total: countRow.total,
      page: page,
      limit: limit,
      totalPages: Math.ceil(countRow.total / limit)
    }
```

```
    });
  });
});
});

app.listen(3000);
```

2.2.3. Продвинутый уровень (15 вариантов)

Общее задание для продвинутого уровня: Выполнить требования среднего уровня, а также дополнительно реализовать:

Фильтрацию с IN (по нескольким значениям через запятую) — например, ?author=Толстой,Достоевский.

Сложный поиск по нескольким текстовым полям с ранжированием (релевантностью).

Фильтрацию по нескольким категориям (например, жанр может быть выбран несколько раз).

Эндпоинт для получения доступных фильтров (GET /api/items/filters).

Логирование запросов через morgan.

Переменные окружения (.env) для порта и пути к БД.

Дополнительные параметры для продвинутого уровня (к каждому варианту):

	Поля для сложного поиска (q)	Поля для IN-фильтрации	Поля для filters-эндпоинта
	title, author, genre	author	authors, genres, yearRange, priceRange
	full_name, group, email	group	groups, ageRange

	Поля для сложного поиска (q)	Поля для IN-фильтрации	Поля для filters-эндпоинта
	name, category, description	category	categories, priceRange
	title, director, genre	director	directors, yearRange, ratingRange
	name, position, department	department	departments, salaryRange
	title, cuisine, ingredients	cuisine	cuisines, cookingTimeRange
	model, brand, engine	brand	brands, yearRange, priceRange
	title, artist, album	artist	artists, durationRange
	name, city, address	city	cities, starsRange, priceRange
0	title, instructor, level	instructor	instructors, durationRange
1	name, manufacturer, description	manufacturer	manufacturers, priceRange

	Поля для сложного поиска (q)	Поля для IN-фильтрации	Поля для filters-эндпоинта
2	name, location, description	location	locations, priceRange
3	customer, product, status	status	statuses, amountRange
4	title, author, tags	author	authors, viewsRange
5	name, sport, country	sport	sports, medalsRange

Пример выполнения для варианта 1 (Продвинутый уровень):

javascript

```
require('dotenv').config();
const express = require('express');
const morgan = require('morgan');
const sqlite3 = require('sqlite3').verbose();

const app = express();
const PORT = process.env.PORT || 3000;
const DB_PATH = process.env.DB_PATH || './library.db';

app.use(express.json());
app.use(morgan('combined'));

const db = new sqlite3.Database(DB_PATH, (err) => {
  if (err) console.error('Ошибка БД:', err.message);
```

```
});

// Белый список полей для сортировки
const allowedSortFields = ['id', 'title', 'author', 'year', 'price', 'genre'];

// GET /api/books — расширенная фильтрация
app.get('/api/books', (req, res) => {
    let sql = 'SELECT * FROM books WHERE 1=1';
    const params = [];

    // ===== IN-фильтрация (несколько авторов через запятую) =====
    if (req.query.author) {
        const authors = req.query.author.split(',');
        const placeholders = authors.map(() => '?').join(',');
        sql += ` AND author IN (${placeholders})`;
        params.push(...authors);
    }

    // IN-фильтрация по жанрам
    if (req.query.genre) {
        const genres = req.query.genre.split(',');
        const placeholders = genres.map(() => '?').join(',');
        sql += ` AND genre IN (${placeholders})`;
        params.push(...genres);
    }

    // ===== Диапазонная фильтрация =====
    if (req.query.minYear) {
        sql += ' AND year >= ?';
```

```
        params.push(req.query.minYear);
    }
    if (req.query.maxYear) {
        sql += ' AND year <= ?';
        params.push(req.query.maxYear);
    }
    if (req.query.minPrice) {
        sql += ' AND price >= ?';
        params.push(req.query.minPrice);
    }
    if (req.query.maxPrice) {
        sql += ' AND price <= ?';
        params.push(req.query.maxPrice);
    }

    // ===== Наличие на складе =====
    if (req.query.inStock === 'true') {
        sql += ' AND in_stock = 1';
    } else if (req.query.inStock === 'false') {
        sql += ' AND in_stock = 0';
    }

    // ===== Сложный поиск с релевантностью =====
    // Если есть поисковый запрос, используем отдельный SQL с ранжированием
    if (req.query.q) {
        const searchPattern = `%${req.query.q}%`;
        sql = `
            SELECT *,
```

```
                CASE
                    WHEN title LIKE ? THEN 10
                    WHEN author LIKE ? THEN 5
                    WHEN genre LIKE ? THEN 2
                    ELSE 0
                END as relevance
            FROM books
            WHERE (title LIKE ? OR author LIKE ? OR genre LIKE ?)
        `;
        const searchParams = [searchPattern, searchPattern, searchPattern, searchPattern, searchPattern, searchPattern];

        // Добавляем остальные фильтры
        if (req.query.author && !req.query.q) {
            // IN-фильтрация
        }
        // ... (логика объединения фильтров с поиском)
    }

    // ===== Сортировка =====
    const sortField = req.query.sort || 'id';
    const sortOrder = req.query.order === 'desc' ? 'DESC' : 'ASC';
    const finalSortField = allowedSortFields.includes(sortField) ? sortField : 'id';
    sql += ` ORDER BY ${finalSortField} ${sortOrder}`;

    // ===== Пагинация =====
    const page = parseInt(req.query.page) || 1;
    const limit = parseInt(req.query.limit) || 10;
```

```
    const offset = (page - 1) * limit;
    sql += ' LIMIT ? OFFSET ?';
    params.push(limit, offset);

    db.all(sql, params, (err, rows) => {
      if (err) return res.status(500).json({ error: err.message });

      // Подсчёт общего количества (упрощённо)
      let countSql = 'SELECT COUNT(*) as total FROM books WHERE
1=1';
      const countParams = [];
      if (req.query.author) {
        const authors = req.query.author.split(',');
        const placeholders = authors.map(() => '?').join(',');
        countSql += ` AND author IN (${placeholders})`;
        countParams.push(...authors);
      }
      if (req.query.genre) {
        const genres = req.query.genre.split(',');
        const placeholders = genres.map(() => '?').join(',');
        countSql += ` AND genre IN (${placeholders})`;
        countParams.push(...genres);
      }

      db.get(countSql, countParams, (err, countRow) => {
        if (err) return res.status(500).json({ error: err.message });

        res.json({
          data: rows,
```

```
            pagination: {
                total: countRow.total,
                page: page,
                limit: limit,
                totalPages: Math.ceil(countRow.total / limit)
            }
        });
    });
});

// GET /api/books/filters — доступные фильтры
app.get('/api/books/filters', (req, res) => {
    const sql = `
        SELECT
            GROUP_CONCAT(DISTINCT author) as authors,
            GROUP_CONCAT(DISTINCT genre) as genres,
            MIN(year) as minYear,
            MAX(year) as maxYear,
            MIN(price) as minPrice,
            MAX(price) as maxPrice,
            MIN(in_stock) as hasOutOfStock
        FROM books
    `;

    db.get(sql, [], (err, row) => {
        if (err) return res.status(500).json({ error: err.message });
        res.json({
            authors: row.authors ? row.authors.split(',') : [],
```

```
            genres: row.genres ? row.genres.split(',') : [],
            yearRange: { min: row.minYear, max: row.maxYear },
            priceRange: { min: row.minPrice, max: row.maxPrice },
            inStockOptions: ['true', 'false']
        });
    });
});

// GET /api/books/search — выделенный эндпоинт для поиска с релевантностью
app.get('/api/books/search', (req, res) => {
    const query = req.query.q;

    if (!query) {
        return res.status(400).json({ error: 'Параметр q обязателен' });
    }

    const searchPattern = `%${query}%`;
    const sql = `
        SELECT *,
            CASE
                WHEN title LIKE ? THEN 10
                WHEN author LIKE ? THEN 5
                WHEN genre LIKE ? THEN 2
                ELSE 0
            END as relevance
        FROM books
        WHERE title LIKE ? OR author LIKE ? OR genre LIKE ?
        ORDER BY relevance DESC
```

```
    `;

    db.all(sql, [searchPattern, searchPattern, searchPattern, searchPattern, searchPattern, searchPattern],
      (err, rows) => {
        if (err) return res.status(500).json({ error: err.message });
        res.json({
          query: query,
          count: rows.length,
          results: rows
        });
      }
    );
  });

  app.listen(PORT, () => {
    console.log(`Сервер запущен на http://localhost:${PORT}`);
  });
```

Файл .env:

```
PORT=3000
DB_PATH=./library.db
```

Установка дополнительных пакетов:

```
bash
npm install morgan dotenv
```

3. Контрольные вопросы

1. Какой оператор SQL используется для фильтрации строк?
2. Как выполнить поиск по частичному совпадению в SQL?

3. В чём разница между операторами AND и OR?
4. Как выполнить сортировку результатов в SQL?
5. Что такое ORDER BY и как задать направление сортировки?
6. Как реализовать пагинацию с помощью LIMIT и OFFSET?
7. Как получить общее количество записей для пагинации?
8. Как защититься от SQL-инъекций при динамическом построении запросов?
9. Что такое белый список (whitelist) полей для сортировки и зачем он нужен?
10. Как реализовать фильтрацию по нескольким значениям одного поля (IN)?
11. Как реализовать диапазонную фильтрацию (min/max)?
12. Что такое релевантность (relevance) при поиске и как её реализовать?
13. Какой эндпоинт можно создать для получения доступных фильтров?
14. Для чего используется библиотека morgan?
15. Как объединить фильтрацию, поиск, сортировку и пагинацию в одном запросе?

4. Требования к отчёту

Отчёт должен быть оформлен в электронном виде и содержать:

Титульный лист с названием работы, ФИО студента, номером группы, вариантом.

Цель и задачи работы (переписать из методических указаний).

Теоретическое обоснование (краткий конспект по теме).

Ход выполнения работы:

SQL-скрипты создания таблицы.

Листинг кода сервера с комментариями.

Скриншоты проверки работы через Postman:

Фильтрация по одному полю

Диапазонная фильтрация (min/max)

Поиск (q)

Сортировка (sort, order)

Пагинация (page, limit)

Комбинация параметров

(Для продвинутого) IN-фильтрация, filters-эндпоинт

Ответы на контрольные вопросы (письменно, развёрнуто).

Заключение (выводы о проделанной работе, какие навыки приобретены).

Список использованных источников.

5. Критерии оценивания

Уровень выполнения	Требования	Оценка
Базовый	Реализованы фильтрация по одному полю и поиск (q) по одному текстовому полю.	Удовлетвори тельно
Средний	Выполнены требования базового уровня + диапазонная фильтрация (min/max), сортировка, пагинация с мета-информацией.	Хорошо
Продвинутый	Выполнены требования среднего уровня + IN-фильтрация (несколько значений), сложный поиск с	Отлично

Уровень выполнения	Требования	Оценка
	релевантностью, эндпоинт /filters, логирование (morgan), переменные окружения.	

Дополнительные условия:

1. Код должен быть структурирован и снабжён комментариями.
2. Отчёт должен содержать скриншоты всех типов запросов.
3. При защите студент должен объяснить SQL-логику фильтрации.

6. Список рекомендуемой литературы и интернет-ресурсов

Литература (доступна в РФ)

1. Браун, И. (2021). *Express.js: Разработка веб-приложений на Node.js*. ДМК Пресс.
2. Кантелон, М., Хартер, М., & Холстейн, С. (2020). *Node.js в действии*. Питер.

Интернет-ресурсы

Название	**Ссылка**	**Описание**
SQL WHERE — Metanit	https://metanit.com/sql/sqlite/2.4.php	Условная выборка в SQLite
SQL ORDER BY — Metanit	https://metanit.com/sql/sqlite/2.5.php	Сортировка в SQLite

Название	Ссылка	Описание
SQL LIMIT — Metanit	https://metanit.com/sql/sqlite/2.6.php	Пагинация в SQLite
SQL LIKE — Metanit	https://metanit.com/sql/sqlite/2.7.php	Поиск по шаблону
SQLite Query Language	https://www.sqlite.org/lang.html	Официальная документация
Express + SQLite фильтрация	https://expressjs.com/ru/advanced/best-practice-performance.html	Best practices
Документация Morgan	https://github.com/expressjs/morgan	HTTP-логгер

Все ресурсы доступны и легальны на территории Российской Федерации.

Лабораторная работа № 8

Тема: Связи таблиц: один-ко-многим (JOIN)

Цель работы:

Научиться проектировать связанные таблицы с отношением «один-ко-многим», реализовывать внешние ключи (FOREIGN KEY) и выполнять JOIN-запросы для получения связанных данных через REST API.

Задачи работы:

1. Спроектировать две связанные таблицы с отношением «один-ко-многим».
2. Реализовать внешний ключ (FOREIGN KEY) для обеспечения ссылочной целостности.
3. Освоить выполнение JOIN-запросов (INNER JOIN, LEFT JOIN) для получения данных из нескольких таблиц.
4. Реализовать API для получения связанных данных:
5. Получение всех записей основной таблицы с подтягиванием данных из справочника.
6. Получение записи по ID с расширенной информацией.
7. Получение всех связанных записей для конкретного элемента справочника.
8. Научиться вставлять данные в связанные таблицы с проверкой существования внешнего ключа.

1. Теоретическое обоснование

1.1. Отношение «один-ко-многим» (One-to-Many)

Отношение «один-ко-многим» — это тип связи между двумя таблицами, при котором одной записи в первой таблице (справочнике) соответствует много записей во второй таблице (основной).

Примеры:

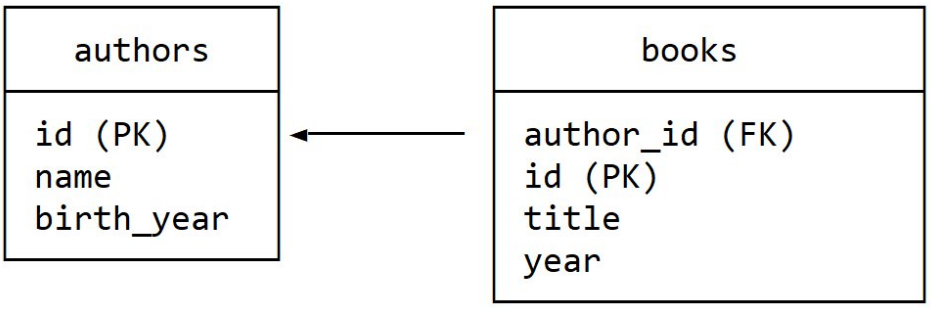
Один автор → много книг

Одна категория → много товаров

Один студент → много оценок

Одна группа → много студентов

Схема:



PK — Primary Key (первичный ключ)

FK — Foreign Key (внешний ключ)

1.2. Внешний ключ (FOREIGN KEY)

Внешний ключ — это поле (или комбинация полей) в дочерней таблице, которое ссылается на первичный ключ родительской таблицы.

Синтаксис создания внешнего ключа:

sql

```
CREATE TABLE books (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    title TEXT NOT NULL,
    author_id INTEGER NOT NULL,
    FOREIGN KEY (author_id) REFERENCES authors(id) ON DELETE CASCADE
);
```

Опции ON DELETE:

Опция	Поведение при удалении родительской записи
CASCADE	Удалить все связанные дочерние записи
SET NULL	Установить внешний ключ в NULL
RESTRICT	Запретить удаление, если есть связанные

Опция	Поведение при удалении родительской записи
	записи
NO ACTION	Аналогично RESTRICT

1.3. Типы JOIN

INNER JOIN — возвращает только те записи, для которых есть соответствие в обеих таблицах.

sql

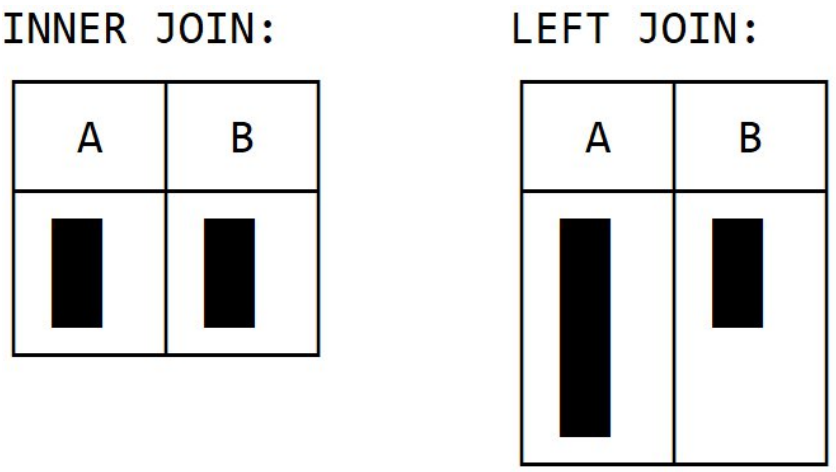
```
SELECT books.title, authors.name
FROM books
INNER JOIN authors ON books.author_id = authors.id;
```

LEFT JOIN (LEFT OUTER JOIN) — возвращает все записи из левой таблицы и соответствующие записи из правой. Если соответствия нет, поля из правой таблицы будут NULL.

sql

```
SELECT books.title, authors.name
FROM books
LEFT JOIN authors ON books.author_id = authors.id;
```

Визуальное представление JOIN:



1.4. Добавление данных в связанные таблицы

Порядок вставки:

Сначала вставляем данные в родительскую таблицу (справочник), получаем ID.

Затем вставляем данные в дочернюю таблицу, используя полученный ID.

javascript

```
// Сначала создаём автора
db.run('INSERT INTO authors (name) VALUES (?)', [name], function(err) {
  const authorId = this.lastID;

  // Затем создаём книгу этого автора
  db.run('INSERT INTO books (title, author_id) VALUES (?, ?)', [title, authorId]);
});
```

1.5. Получение связанных данных

Все книги с именами авторов:

sql

```
SELECT books.*, authors.name as author_name
```

```
FROM books
JOIN authors ON books author_id = authors id
ORDER BY books id;
```

Книги конкретного автора:

sql

```
SELECT * FROM books WHERE author_id = ?;
```

Авторы, у которых есть книги в наличии:

sql

```
SELECT DISTINCT authors.*
FROM authors
JOIN books ON authors id = books author_id
WHERE books in_stock = 1;
```

2. Методика выполнения работы

2.1. Примеры и их подробный разбор

Пример 1. Создание связанных таблиц (авторы и книги)

sql

```
-- Таблица авторов (справочник, родительская)
CREATE TABLE IF NOT EXISTS authors (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL UNIQUE,
    birth_year INTEGER,
    country TEXT
);

-- Таблица книг (основная, дочерняя)
CREATE TABLE IF NOT EXISTS books (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    title TEXT NOT NULL,
```

```
    author_id INTEGER NOT NULL,
    year INTEGER,
    price REAL,
    in_stock INTEGER DEFAULT 1,
    FOREIGN KEY (author_id) REFERENCES authors(id) ON DELETE CASCADE
);
```

Пример 2. Вставка данных в связанные таблицы

javascript

```
// Вставка авторов
const authors = [
    'Лев Толстой',
    'Фёдор Достоевский',
    'Михаил Булгаков'
];

authors.forEach(author => {
    db.run('INSERT INTO authors (name) VALUES (?)', [author]);
});

// Вставка книг с указанием author_id
const books = [
    ['Война и мир', 1, 1869, 599, 1],
    ['Анна Каренина', 1, 1877, 549, 1],
    ['Преступление и наказание', 2, 1866, 499, 1],
    ['Идиот', 2, 1869, 399, 1],
    ['Мастер и Маргарита', 3, 1967, 699, 0],
    ['Собачье сердце', 3, 1925, 349, 1]
];
```

```
books.forEach(book => {
  db.run(
    'INSERT INTO books (title, author_id, year, price, in_stock) VALUES (?, ?, ?, ?, ?)',
    book
  );
});
```

Пример 3. GET все книги с JOIN (с именами авторов)

javascript

```
app.get('/api/books', (req, res) => {
  const sql = `
    SELECT
      books.*,
      authors.name as author_name,
      authors.birth_year as author_birth_year,
      authors.country as author_country
    FROM books
    JOIN authors ON books.author_id = authors.id
    ORDER BY books.id
  `;

  db.all(sql, [], (err, rows) => {
    if (err) return res.status(500).json({ error: err.message });
    res.json(rows);
  });
});
```

Пример ответа:

json

```
[
  {
    "id": 1,
    "title": "Война и мир",
    "author_id": 1,
    "year": 1869,
    "price": 599,
    "in_stock": 1,
    "author_name": "Лев Толстой",
    "author_birth_year": 1828,
    "author_country": null
  }
]
```


Пример 4. GET книга по ID с JOIN

javascript

```
app.get('/api/books/:id', (req, res) => {
  const id = req.params.id;
  const sql = `
    SELECT
      books.*,
      authors.name as author_name,
      authors.birth_year as author_birth_year
    FROM books
    JOIN authors ON books.author_id = authors.id
    WHERE books.id = ?
  `;

  db.get(sql, [id], (err, row) => {
    if (err) return res.status(500).json({ error: err.message });
```

```
        if (!row) return res.status(404).json({ error: 'Книга не найдена' });
        res.json(row);
    });
});
```

Пример 5. GET все книги с LEFT JOIN (включая книги без авторов)

```
javascript
app.get('/api/books/all', (req, res) => {
    const sql = `
        SELECT
            books.*,
            authors.name as author_name
        FROM books
        LEFT JOIN authors ON books.author_id = authors.id
    `;

    db.all(sql, [], (err, rows) => {
        if (err) return res.status(500).json({ error: err.message });
        res.json(rows);
    });
});
```

Пример 6. GET книги конкретного автора

```
javascript
app.get('/api/authors/:id/books', (req, res) => {
    const authorId = req.params.id;

    // Проверка существования автора
    db.get('SELECT * FROM authors WHERE id = ?', [authorId], (err, author) => {
        if (err) return res.status(500).json({ error: err.message });
```

```
            if (!author) return res.status(404).json({ error: 'Автор не найден' });

            // Получение книг автора
            db.all(
                'SELECT * FROM books WHERE author_id = ? ORDER BY year',
                [authorId],
                (err, books) => {
                    if (err) return res.status(500).json({ error: err.message });
                    res.json({
                        author: author,
                        books_count: books.length,
                        books: books
                    });
                }
            );
        });
    });
```

Пример 7. GET все авторы с количеством их книг

```
javascript
app.get('/api/authors', (req, res) => {
    const sql = `
        SELECT
            authors.*,
            COUNT(books.id) as books_count,
            SUM(CASE WHEN books.in_stock = 1 THEN 1 ELSE 0 END) as books_in_stock,
            ROUND(AVG(books.price), 2) as avg_price
        FROM authors
        LEFT JOIN books ON authors.id = books.author_id
```

```
    GROUP BY authors.id
    ORDER BY authors.name
  `;

  db.all(sql, [], (err, rows) => {
    if (err) return res.status(500).json({ error: err.message });
    res.json(rows);
  });
});
```

Пример 8. POST создание книги с проверкой существования автора

```
javascript
app.post('/api/books', (req, res) => {
  const { title, author_id, year, price, in_stock } = req.body;

  if (!title || !author_id) {
    return res.status(400).json({ error: 'title и author_id обязательны' });
  }

  // Проверка, существует ли автор
  db.get('SELECT * FROM authors WHERE id = ?', [author_id], (err, author) => {
    if (err) return res.status(500).json({ error: err.message });
    if (!author) {
      return res.status(400).json({ error: 'Автор с таким ID не существует' });
    }

    db.run(
      `INSERT INTO books (title, author_id, year, price, in_stock)
```

```
                VALUES (?, ?, ?, ?, ?)`,
                [title, author_id, year || null, price || 0, in_stock !== undefined ?
in_stock : 1],
                function(err) {
                    if (err) return res.status(500).json({ error: err.message });

                    // Возвращаем созданную книгу с именем автора
                    const selectSql = `
                        SELECT books.*, authors.name as author_name
                        FROM books
                        JOIN authors ON books.author_id = authors.id
                        WHERE books.id = ?
                    `;
                    db.get(selectSql, [this.lastID], (err, newBook) => {
                        if (err) return res.status(500).json({ error: err.message });
                        res.status(201).json(newBook);
                    });
                }
            );
        });
    });
```

Пример 9. Полный пример сервера

```
javascript
const express = require('express');
const sqlite3 = require('sqlite3').verbose();

const app = express();
const PORT = 3000;
```

```
app.use(express.json());

// Подключение к БД
const db = new sqlite3.Database('./library.db', (err) => {
    if (err) console.error('Ошибка БД:', err.message);
});

// Создание таблиц
db.serialize(() => {
    // Таблица авторов
    db.run(`
        CREATE TABLE IF NOT EXISTS authors (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            name TEXT NOT NULL UNIQUE,
            birth_year INTEGER,
            country TEXT
        )
    `);

    // Таблица книг с внешним ключом
    db.run(`
        CREATE TABLE IF NOT EXISTS books (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            title TEXT NOT NULL,
            author_id INTEGER NOT NULL,
            year INTEGER,
            price REAL,
            in_stock INTEGER DEFAULT 1,
```

```
            FOREIGN KEY (author_id) REFERENCES authors(id) ON DELETE CASCADE
        )
    `);

    // Заполнение тестовыми данными, если таблицы пустые
    db.get('SELECT COUNT(*) as count FROM authors', [], (err, row) => {
        if (row.count === 0) {
            // Добавляем авторов
            db.run('INSERT INTO authors (name, birth_year, country) VALUES (?, ?, ?)',
                ['Лев Толстой', 1828, 'Россия']);
            db.run('INSERT INTO authors (name, birth_year, country) VALUES (?, ?, ?)',
                ['Фёдор Достоевский', 1821, 'Россия']);
            db.run('INSERT INTO authors (name, birth_year, country) VALUES (?, ?, ?)',
                ['Михаил Булгаков', 1891, 'Россия']);

            // Добавляем книги
            db.run(`INSERT INTO books (title, author_id, year, price, in_stock) VALUES
                ('Война и мир', 1, 1869, 599, 1),
                ('Анна Каренина', 1, 1877, 549, 1),
                ('Преступление и наказание', 2, 1866, 499, 1),
                ('Идиот', 2, 1869, 399, 1),
                ('Мастер и Маргарита', 3, 1967, 699, 0),
                ('Собачье сердце', 3, 1925, 349, 1)`);
        }
```

```
    });
});

// ========== ЭНДПОИНТЫ ==========

// GET все книги (с информацией об авторах)
app.get('/api/books', (req, res) => {
    const sql = `
        SELECT
            b.*,
            a.name as author_name,
            a.birth_year as author_birth_year,
            a.country as author_country
        FROM books b
        INNER JOIN authors a ON b.author_id = a.id
        ORDER BY b.id
    `;

    db.all(sql, [], (err, rows) => {
        if (err) return res.status(500).json({ error: err.message });
        res.json(rows);
    });
});

// GET книга по ID (с информацией об авторе)
app.get('/api/books/:id', (req, res) => {
    const id = req.params.id
    const sql = `
        SELECT
```

```
            b.*,
            a.name as author_name,
            a.birth_year as author_birth_year,
            a.country as author_country
        FROM books b
        INNER JOIN authors a ON b.author_id = a.id
        WHERE b.id = ?
    `;

    db.get(sql, [id], (err, row) => {
        if (err) return res.status(500).json({ error: err.message });
        if (!row) return res.status(404).json({ error: 'Книга не найдена' });
        res.json(row);
    });
});

// POST создать книгу
app.post('/api/books', (req, res) => {
    const { title, author_id, year, price, in_stock } = req.body;

    if (!title || !author_id) {
        return res.status(400).json({ error: 'title и author_id обязательны' });
    }

    // Проверка существования автора
    db.get('SELECT * FROM authors WHERE id = ?', [author_id], (err, author) => {
        if (err) return res.status(500).json({ error: err.message });
        if (!author) {
```

```
            return res.status(400).json({ error: 'Автор с таким ID не существует' });
        }

        db.run(
          `INSERT INTO books (title, author_id, year, price, in_stock)
           VALUES (?, ?, ?, ?, ?)`,
          [title, author_id, year || null, price || 0, in_stock !== undefined ? in_stock : 1],
          function(err) {
            if (err) return res.status(500).json({ error: err.message });

            db.get(`
              SELECT b.*, a.name as author_name
              FROM books b
              JOIN authors a ON b.author_id = a.id
              WHERE b.id = ?
            `, [this.lastID], (err, newBook) => {
              if (err) return res.status(500).json({ error: err.message });
              res.status(201).json(newBook);
            });
          }
        );
      });
    });

    // PUT обновить книгу
    app.put('/api/books/:id', (req, res) => {
      const id = req.params.id;
```

```
    const { title, author_id, year, price, in_stock } = req.body;

    if (!title || !author_id) {
      return res.status(400).json({ error: 'title и author_id обязательны' });
    }

    db.get('SELECT * FROM books WHERE id = ?', [id], (err, book) => {
      if (err) return res.status(500).json({ error: err.message });
      if (!book) return res.status(404).json({ error: 'Книга не найдена' });

      db.get('SELECT * FROM authors WHERE id = ?', [author_id], (err, author) => {
        if (err) return res.status(500).json({ error: err.message });
        if (!author) {
          return res.status(400).json({ error: 'Автор с таким ID не существует' });
        }

        db.run(
          `UPDATE books SET title = ?, author_id = ?, year = ?, price = ?, in_stock = ?
           WHERE id = ?`,
          [title, author_id, year || null, price || 0, in_stock !== undefined ? in_stock : 1, id],
          function(err) {
            if (err) return res.status(500).json({ error: err.message });

            db.get(`
              SELECT b.*, a.name as author_name
```

```
                FROM books b
                JOIN authors a ON b.author_id = a.id
                WHERE b.id = ?
            `, [id], (err, updatedBook) => {
                if (err) return res.status(500).json({ error: err.message });
                res.json(updatedBook);
            });
        }
    );
  });
});

// DELETE удалить книгу
app.delete('/api/books/:id', (req, res) => {
  const id = req.params.id;

  db.get('SELECT * FROM books WHERE id = ?', [id], (err, book) => {
    if (err) return res.status(500).json({ error: err.message });
    if (!book) return res.status(404).json({ error: 'Книга не найдена' });

    db.run('DELETE FROM books WHERE id = ?', [id], function(err) {
      if (err) return res.status(500).json({ error: err.message });
      res.status(204).send();
    });
  });
});

// ========== ЭНДПОИНТЫ ДЛЯ АВТОРОВ ==========
```

```
// GET все авторы (с количеством книг)
app.get('/api/authors', (req, res) => {
  const sql = `
    SELECT
      a.*,
      COUNT(b.id) as books_count,
      SUM(CASE WHEN b.in_stock = 1 THEN 1 ELSE 0 END) as books_in_stock,
      ROUND(AVG(b.price), 2) as avg_book_price
    FROM authors a
    LEFT JOIN books b ON a.id = b.author_id
    GROUP BY a.id
    ORDER BY a.name
  `;

  db.all(sql, [], (err, rows) => {
    if (err) return res.status(500).json({ error: err.message });
    res.json(rows);
  });
});

// GET автор по ID с его книгами
app.get('/api/authors/:id', (req, res) => {
  const id = req.params.id;

  db.get('SELECT * FROM authors WHERE id = ?', [id], (err, author) => {
    if (err) return res.status(500).json({ error: err.message });
    if (!author) return res.status(404).json({ error: 'Автор не найден' });
```

```
            db.all('SELECT * FROM books WHERE author_id = ? ORDER BY year', [id], (err, books) => {
                if (err) return res.status(500).json({ error: err.message });
                res.json({
                    ...author,
                    books_count: books.length,
                    books: books
                });
            });
        });
    });

    // GET книги конкретного автора
    app.get('/api/authors/:id/books', (req, res) => {
        const authorId = req.params.id;

        db.get('SELECT * FROM authors WHERE id = ?', [authorId], (err, author) => {
            if (err) return res.status(500).json({ error: err.message });
            if (!author) return res.status(404).json({ error: 'Автор не найден' });

            db.all('SELECT * FROM books WHERE author_id = ? ORDER BY year', [authorId], (err, books) => {
                if (err) return res.status(500).json({ error: err.message });
                res.json({
                    author: author,
                    books: books
                });
```

```
    });
  });
});

// POST создать автора
app.post('/api/authors', (req, res) => {
  const { name, birth_year, country } = req.body;

  if (!name) {
    return res.status(400).json({ error: 'name обязателен' });
  }

  db.run(
    'INSERT INTO authors (name, birth_year, country) VALUES (?, ?, ?)',
    [name, birth_year || null, country || null],
    function(err) {
      if (err) return res.status(500).json({ error: err.message });
      res.status(201).json({ id: this.lastID, name, birth_year, country });
    }
  );
});

// DELETE удалить автора (каскадно удалятся его книги)
app.delete('/api/authors/:id', (req, res) => {
  const id = req.params.id;

  db.get('SELECT * FROM authors WHERE id = ?', [id], (err, author) => {
    if (err) return res.status(500).json({ error: err.message });
    if (!author) return res.status(404).json({ error: 'Автор не найден' });
```

```
        db.run('DELETE FROM authors WHERE id = ?', [id], function(err) {
            if (err) return res.status(500).json({ error: err.message });
            res.status(204).send();
        });
    });
});

app.listen(PORT, () => {
    console.log(`Сервер запущен на http://localhost:${PORT}`);
});
```

2.2. Индивидуальные задания

2.2.1. Базовый уровень (15 вариантов)

Общее задание для базового уровня: Создать две связанные таблицы с отношением «один-ко-многим» согласно варианту. Реализовать:

GET /api/items — получение всех записей основной таблицы с JOIN (подтягивание названия/имени из справочника).

GET /api/items/:id — получение записи по ID с расширенной информацией из справочника.

POST /api/items — создание записи в основной таблице (с проверкой существования внешнего ключа).

Варианты заданий:

	Справочник (один)	Основная таблица (много)	Поля для JOIN
	authors (id,	books (id, title,	author_name

	Справочник (один)	Основная таблица (много)	Поля для JOIN
	name)	author_id, year)	
	groups (id, name)	students (id, full_name, group_id, age)	group_name
	categories (id, name)	products (id, name, category_id, price)	category_name
	directors (id, name)	movies (id, title, director_id, year)	director_name
	departments (id, name)	employees (id, name, department_id, salary)	department_name
	cuisines (id, name)	recipes (id, title, cuisine_id, cooking_time)	cuisine_name
	brands (id, name)	cars (id, model, brand_id, year)	brand_name
	artists (id, name)	songs (id, title, artist_id, duration)	artist_name
	cities (id, name)	hotels (id, name, city_id, stars)	city_name
0	instructors (id, name)	courses (id, title, instructor_id, duration)	instructor_name

	Справочник (один)	Основная таблица (много)	Поля для JOIN
1	manufacturers (id, name)	medicines (id, name, manufacturer_id, price)	manufacturer_name
2	locations (id, name)	events (id, name, location_id, date)	location_name
3	customers (id, name)	orders (id, customer_id, amount, status)	customer_name
4	article_authors (id, name)	articles (id, title, author_id, published_at)	author_name
5	sports (id, name)	athletes (id, name, sport_id, medals)	sport_name

Пример выполнения для варианта 1 (Базовый уровень):

javascript

```
const express = require('express');
const sqlite3 = require('sqlite3').verbose();
const app = express();
app.use(express.json());

const db = new sqlite3.Database('./library.db');

// Создание таблиц
db.run(`CREATE TABLE IF NOT EXISTS authors (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL UNIQUE
```

```
)`);

db.run(`CREATE TABLE IF NOT EXISTS books (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    title TEXT NOT NULL,
    author_id INTEGER NOT NULL,
    year INTEGER,
    FOREIGN KEY (author_id) REFERENCES authors(id)
)`);

// GET все книги с JOIN
app.get('/api/books', (req, res) => {
    const sql = `
        SELECT books.*, authors.name as author_name
        FROM books
        JOIN authors ON books.author_id = authors.id
    `;
    db.all(sql, [], (err, rows) => {
        if (err) return res.status(500).json({ error: err.message });
        res.json(rows);
    });
});

// GET книга по ID с JOIN
app.get('/api/books/:id', (req, res) => {
    const sql = `
        SELECT books.*, authors.name as author_name
        FROM books
        JOIN authors ON books.author_id = authors.id
```

```
    WHERE books.id = ?
  `;
  db.get(sql, [req.params.id], (err, row) => {
    if (err) return res.status(500).json({ error: err.message });
    if (!row) return res.status(404).json({ error: 'Книга не найдена' });
    res.json(row);
  });
});

// POST создать книгу
app.post('/api/books', (req, res) => {
  const { title, author_id, year } = req.body;
  if (!title || !author_id) {
    return res.status(400).json({ error: 'title и author_id обязательны' });
  }

  db.get('SELECT * FROM authors WHERE id = ?', [author_id], (err, author) => {
    if (err) return res.status(500).json({ error: err.message });
    if (!author) return res.status(400).json({ error: 'Автор не существует' });

    db.run(
      'INSERT INTO books (title, author_id, year) VALUES (?, ?, ?)',
      [title, author_id, year || null],
      function(err) {
        if (err) return res.status(500).json({ error: err.message });
        res.status(201).json({ id: this.lastID, title, author_id, year });
      }
```

```
    );
  });
});

app.listen(3000);
```

2.2.2. Средний уровень (15 вариантов)

Общее задание для среднего уровня: Выполнить требования базового уровня, а также дополнительно реализовать:

GET /api/parents — получение всех записей из справочника с количеством связанных записей (COUNT).

GET /api/parents/:id/children — получение всех дочерних записей для конкретного элемента справочника.

PUT и DELETE маршруты для основной таблицы (с проверками).

POST маршрут для добавления записей в справочник.

Пример выполнения для варианта 1 (Средний уровень) — дополнение к базовому:

javascript

```
// GET все авторы с количеством книг
app.get('/api/authors', (req, res) => {
  const sql = `
    SELECT authors.*, COUNT(books.id) as books_count
    FROM authors
    LEFT JOIN books ON authors.id = books.author_id
    GROUP BY authors.id
  `;
  db.all(sql, [], (err, rows) => {
    if (err) return res.status(500).json({ error: err.message });
    res.json(rows);
```

```
    });
  });

  // GET книги конкретного автора
  app.get('/api/authors/:id/books', (req, res) => {
    const authorId = req.params.id;

    db.get('SELECT * FROM authors WHERE id = ?', [authorId], (err, author) => {
      if (err) return res.status(500).json({ error: err.message });
      if (!author) return res.status(404).json({ error: 'Автор не найден' });

      db.all('SELECT * FROM books WHERE author_id = ?', [authorId], (err, books) => {
        if (err) return res.status(500).json({ error: err.message });
        res.json({ author, books });
      });
    });
  });

  // PUT обновить книгу
  app.put('/api/books/:id', (req, res) => {
    const id = req.params.id;
    const { title, author_id, year } = req.body;

    if (!title || !author_id) {
      return res.status(400).json({ error: 'title и author_id обязательны' });
    }
```

```
        db.get('SELECT * FROM books WHERE id = ?', [id], (err, book) => {
            if (err) return res.status(500).json({ error: err.message });
            if (!book) return res.status(404).json({ error: 'Книга не найдена' });

            db.get('SELECT * FROM authors WHERE id = ?', [author_id], (err, author) => {
                if (err) return res.status(500).json({ error: err.message });
                if (!author) return res.status(400).json({ error: 'Автор не существует' });

                db.run(
                    'UPDATE books SET title = ?, author_id = ?, year = ? WHERE id = ?',
                    [title, author_id, year || null, id],
                    function(err) {
                        if (err) return res.status(500).json({ error: err.message });
                        res.json({ id, title, author_id, year });
                    }
                );
            });
        });
    });

    // POST создать автора
    app.post('/api/authors', (req, res) => {
        const { name } = req.body;
        if (!name) return res.status(400).json({ error: 'name обязателен' });
```

```
        db.run('INSERT INTO authors (name) VALUES (?)', [name], function(err) {
            if (err) return res.status(500).json({ error: err.message });
            res.status(201).json({ id: this.lastID, name });
        });
    });
```

2.2.3. Продвинутый уровень (15 вариантов)

Общее задание для продвинутого уровня: Выполнить требования среднего уровня, а также дополнительно реализовать:

Дополнительные агрегатные функции в запросе списка справочника: SUM, AVG, MIN, MAX по связанным записям.

Фильтрацию, сортировку и пагинацию для основного списка GET /api/items (с учётом JOIN).

Каскадное удаление (ON DELETE CASCADE) — при удалении из справочника удаляются все связанные записи.

Эндпоинт для поиска по полям основной таблицы и справочника одновременно (поиск книг по названию и имени автора).

Логирование запросов через morgan и переменные окружения (.env).

Дополнительные агрегации для каждого варианта (продвинутый уровень):

№	Дополнительные поля в ответе справочника
1	books_count, books_in_stock, avg_price, min_year, max_year
2	students_count, avg_age, min_age, max_age
3	products_count, avg_price, total_stock

№	Дополнительные поля в ответе справочника
4	movies_count, avg_rating, min_year, max_year
5	employees_count, avg_salary, min_salary, max_salary
6	recipes_count, avg_cooking_time
7	cars_count, avg_price, min_year, max_year
8	songs_count, total_duration, avg_duration
9	hotels_count, avg_stars, avg_price
10	courses_count, avg_duration, avg_price
11	medicines_count, avg_price
12	events_count, avg_price
13	orders_count, total_amount, avg_amount
14	articles_count, total_views, avg_views
15	athletes_count, total_medals, avg_medals

Пример выполнения для варианта 1 (Продвинутый уровень):

javascript

```
require('dotenv').config();
const express = require('express');
const morgan = require('morgan');
const sqlite3 = require('sqlite3').verbose();

const app = express();
```

```
const PORT = process.env.PORT || 3000;
const DB_PATH = process.env.DB_PATH || './library.db';

app.use(express.json());
app.use(morgan('combined'));

const db = new sqlite3.Database(DB_PATH);

// Создание таблиц с CASCADE
db.run(`CREATE TABLE IF NOT EXISTS authors (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL UNIQUE,
    birth_year INTEGER,
    country TEXT
)`);

db.run(`CREATE TABLE IF NOT EXISTS books (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    title TEXT NOT NULL,
    author_id INTEGER NOT NULL,
    year INTEGER,
    price REAL,
    in_stock INTEGER DEFAULT 1,
    FOREIGN KEY (author_id) REFERENCES authors(id) ON DELETE CASCADE
)`);

// ========== ЭНДПОИНТЫ ДЛЯ КНИГ (с фильтрацией и пагинацией) ==========
```

```
// GET /api/books — с фильтрацией, сортировкой, пагинацией и JOIN
app.get('/api/books', (req, res) => {
    let sql = `
        SELECT b.*, a.name as author_name, a.country as author_country
        FROM books b
        INNER JOIN authors a ON b.author_id = a.id
        WHERE 1=1
    `;
    const params = [];

    // Фильтрация по автору
    if (req.query.author_id) {
        sql += ' AND b.author_id = ?';
        params.push(req.query.author_id);
    }

    // Фильтрация по наличию
    if (req.query.inStock === 'true') {
        sql += ' AND b.in_stock = 1';
    } else if (req.query.inStock === 'false') {
        sql += ' AND b.in_stock = 0';
    }

    // Диапазон цен
    if (req.query.minPrice) {
        sql += ' AND b.price >= ?';
        params.push(req.query.minPrice);
    }
```

```
    if (req.query.maxPrice) {
        sql += ' AND b.price <= ?';
        params.push(req.query.maxPrice);
    }

    // Поиск по названию книги и имени автора
    if (req.query.q) {
        sql += ' AND (b.title LIKE ? OR a.name LIKE ?)';
        const searchPattern = `%${req.query.q}%`;
        params.push(searchPattern, searchPattern);
    }

    // Сортировка
    const sortField = req.query.sort || 'b.id';
    const sortOrder = req.query.order === 'desc' ? 'DESC' : 'ASC';
    const allowedSortFields = ['b.id', 'b.title', 'b.year', 'b.price', 'a.name'];
    const finalSortField = allowedSortFields.includes(sortField) ? sortField :
'b.id';
    sql += ` ORDER BY ${finalSortField} ${sortOrder}`;

    // Пагинация
    const page = parseInt(req.query.page) || 1;
    const limit = parseInt(req.query.limit) || 10;
    const offset = (page - 1) * limit;
    sql += ' LIMIT ? OFFSET ?';
    params.push(limit, offset);

    db.all(sql, params, (err, rows) => {
        if (err) return res.status(500).json({ error: err.message });
```

```
// Подсчёт общего количества
let countSql = `
    SELECT COUNT(*) as total
    FROM books b
    INNER JOIN authors a ON b.author_id = a.id
    WHERE 1=1
`;
const countParams = [];
if (req.query.author_id) {
    countSql += ' AND b.author_id = ?';
    countParams.push(req.query.author_id);
}
if (req.query.inStock === 'true') {
    countSql += ' AND b.in_stock = 1';
} else if (req.query.inStock === 'false') {
    countSql += ' AND b.in_stock = 0';
}
if (req.query.minPrice) {
    countSql += ' AND b.price >= ?';
    countParams.push(req.query.minPrice);
}
if (req.query.maxPrice) {
    countSql += ' AND b.price <= ?';
    countParams.push(req.query.maxPrice);
}
if (req.query.q) {
    countSql += ' AND (b.title LIKE ? OR a.name LIKE ?)';
    const searchPattern = `%${req.query.q}%`;
```

```
            countParams.push(searchPattern, searchPattern);
        }

        db.get(countSql, countParams, (err, countRow) => {
            if (err) return res.status(500).json({ error: err.message });
            res.json({
                data: rows,
                pagination: {
                    total: countRow.total,
                    page: page,
                    limit: limit,
                    totalPages: Math.ceil(countRow.total / limit)
                }
            });
        });
    });
});

// ========== ЭНДПОИНТЫ ДЛЯ АВТОРОВ (с агрегациями) ==========

// GET /api/authors — все авторы с агрегатными данными по книгам
app.get('/api/authors', (req, res) => {
    const sql = `
        SELECT
            a.*,
            COUNT(b.id) as books_count,
            SUM(CASE WHEN b.in_stock = 1 THEN 1 ELSE 0 END) as books_in_stock,
```

```
            ROUND(AVG(b.price), 2) as avg_price,
            MIN(b.year) as min_year,
            MAX(b.year) as max_year,
            MIN(b.price) as min_price,
            MAX(b.price) as max_price
        FROM authors a
        LEFT JOIN books b ON a.id = b.author_id
        GROUP BY a.id
        ORDER BY a.name
    `;

    db.all(sql, [], (err, rows) => {
        if (err) return res.status(500).json({ error: err.message });
        res.json(rows);
    });
});

// GET /api/authors/:id/books — книги конкретного автора
app.get('/api/authors/:id/books', (req, res) => {
    const authorId = req.params.id;

    db.get('SELECT * FROM authors WHERE id = ?', [authorId], (err, author) => {
        if (err) return res.status(500).json({ error: err.message });
        if (!author) return res.status(404).json({ error: 'Автор не найден' });

        db.all('SELECT * FROM books WHERE author_id = ? ORDER BY year', [authorId], (err, books) => {
            if (err) return res.status(500).json({ error: err.message });
```

```
            res.json({
                author: author,
                books_count: books.length,
                books: books
            });
        });
    });
});

// ========== CRUD ДЛЯ АВТОРОВ ==========

app.post('/api/authors', (req, res) => {
    const { name, birth_year, country } = req.body;
    if (!name) return res.status(400).json({ error: 'name обязателен' });

    db.run(
        'INSERT INTO authors (name, birth_year, country) VALUES (?, ?, ?)',
        [name, birth_year || null, country || null],
        function(err) {
            if (err) return res.status(500).json({ error: err.message });
            res.status(201).json({ id: this.lastID, name, birth_year, country });
        }
    );
});

app.put('/api/authors/:id', (req, res) => {
    const id = req.params.id;
    const { name, birth_year, country } = req.body;
```

```
    db.get('SELECT * FROM authors WHERE id = ?', [id], (err, author) => {
      if (err) return res.status(500).json({ error: err.message });
      if (!author) return res.status(404).json({ error: 'Автор не найден' });

      db.run(
        'UPDATE authors SET name = ?, birth_year = ?, country = ? WHERE id = ?',
        [name || author.name, birth_year !== undefined ? birth_year : author.birth_year,
         country !== undefined ? country : author.country, id],
        function(err) {
          if (err) return res.status(500).json({ error: err.message });
          res.json({ id, name, birth_year, country });
        }
      );
    });
  });

  app.delete('/api/authors/:id', (req, res) => {
    const id = req.params.id;

    db.get('SELECT * FROM authors WHERE id = ?', [id], (err, author) => {
      if (err) return res.status(500).json({ error: err.message });
      if (!author) return res.status(404).json({ error: 'Автор не найден' });

      db.run('DELETE FROM authors WHERE id = ?', [id], function(err) {
        if (err) return res.status(500).json({ error: err.message });
        res.status(204).send();
      });
```

```
    });
  });

  // ========== CRUD ДЛЯ КНИГ (базовые операции) ==========

  app.post('/api/books', (req, res) => {
    const { title, author_id, year, price, in_stock } = req.body;
    if (!title || !author_id) {
      return res.status(400).json({ error: 'title и author_id обязательны' });
    }

    db.get('SELECT * FROM authors WHERE id = ?', [author_id], (err, author) => {
      if (err) return res.status(500).json({ error: err.message });
      if (!author) return res.status(400).json({ error: 'Автор не существует' });

      db.run(
        `INSERT INTO books (title, author_id, year, price, in_stock)
         VALUES (?, ?, ?, ?, ?)`,
        [title, author_id, year || null, price || 0, in_stock !== undefined ? in_stock : 1],
        function(err) {
          if (err) return res.status(500).json({ error: err.message });
          res.status(201).json({ id: this.lastID, title, author_id, year, price, in_stock });
        }
      );
    });
```

```
});

app.delete('/api/books/:id', (req, res) => {
    const id = req.params.id;

    db.get('SELECT * FROM books WHERE id = ?', [id], (err, book) => {
        if (err) return res.status(500).json({ error: err.message });
        if (!book) return res.status(404).json({ error: 'Книга не найдена' });

        db.run('DELETE FROM books WHERE id = ?', [id], function(err) {
            if (err) return res.status(500).json({ error: err.message });
            res.status(204).send();
        });
    });
});

app.listen(PORT, () => {
    console.log(`Сервер запущен на http://localhost:${PORT}`);
});
```

Файл .env:

text

```
PORT=3000
DB_PATH=./library.db
```

Установка дополнительных пакетов:

bash

```
npm install morgan dotenv
```

3. Контрольные вопросы

1. Что такое отношение «один-ко-многим» в базах данных?

2. Для чего нужен внешний ключ (FOREIGN KEY)?
3. Какие опции ON DELETE существуют и в чём их разница?
4. В чём разница между INNER JOIN и LEFT JOIN?
5. Как получить все записи из основной таблицы с данными из справочника?
6. Как получить все дочерние записи для конкретной родительской записи?
7. Как получить количество дочерних записей для каждого родителя?
8. Как выполнить проверку существования внешнего ключа при вставке?
9. Что произойдёт при попытке вставить запись с несуществующим внешним ключом?
10. Как реализовать поиск одновременно по полям основной таблицы и справочника?
11. Как выполнить агрегацию (SUM, AVG, COUNT) по связанным записям?
12. Что такое каскадное удаление и когда оно полезно?
13. В каком порядке нужно вставлять данные в связанные таблицы?
14. Как получить общее количество записей с учётом JOIN для пагинации?
15. Как защитить запросы JOIN от SQL-инъекций?

4. Требования к отчёту

Отчёт должен быть оформлен в электронном виде и содержать:

Титульный лист с названием работы, ФИО студента, номером группы, вариантом.

Цель и задачи работы (переписать из методических указаний).

Теоретическое обоснование (краткий конспект по теме).

Ход выполнения работы:

SQL-скрипты создания связанных таблиц (схема БД).

Листинг кода сервера с комментариями.

Скриншоты проверки работы через Postman:

GET /api/items (с JOIN)

GET /api/items/:id (с JOIN)

POST /api/items (с проверкой FK)

GET /api/parents (с агрегацией)

GET /api/parents/:id/children

Для продвинутого: фильтрация, поиск, пагинация

Ответы на контрольные вопросы (письменно, развёрнуто).

Заключение (выводы о проделанной работе, какие навыки приобретены).

Список использованных источников.

5. Критерии оценивания

Уровень выполнения	Требования	Оценка
Базовый	Созданы две связанные таблицы с FK. Реализованы GET (все и по ID) с JOIN, POST с проверкой FK.	Удовлетворительно
Средний	Выполнены требования базового уровня + GET /parents (с COUNT), GET /parents/:id/children, PUT и DELETE для основной таблицы, POST для справочника.	Хорошо
Продвинутый	Выполнены требования среднего уровня + агрегации (SUM, AVG, MIN, MAX) в	Отлично

Уровень выполнения	Требования	Оценка
	списке справочника, фильтрация и пагинация для основного списка, поиск (q) по объединённым полям, каскадное удаление, логирование, переменные окружения.	

Дополнительные условия:

1. Код должен быть структурирован и снабжён комментариями.
2. Отчёт должен содержать скриншоты всех типов запросов.
3. При защите студент должен объяснить схему БД и SQL-запросы с JOIN.

6. Список рекомендуемой литературы и интернет-ресурсов

Литература (доступна в РФ)

1. Браун, И. (2021). *Express.js: Разработка веб-приложений на Node.js*. ДМК Пресс.
2. Кантелон, М., Хартер, М., & Холстейн, С. (2020). *Node.js в действии*. Питер.

Интернет-ресурсы

Название	Ссылка	Описание
SQLite FOREIGN KEY	https://sqlite.org/foreignkeys.html	Официальная документация внешних ключей

Название	Ссылка	Описание
SQL JOIN — Metanit	https://metanit.com/sql/sqlite/3.1.php	Объединение таблиц в SQLite
INNER JOIN vs LEFT JOIN	https://metanit.com/sql/sqlite/3.2.php	Различия типов JOIN
SQLite ON DELETE CASCADE	https://sqlite.org/foreignkeys.html#fk_actions	Действия при удалении
Express + SQLite JOIN	https://expressjs.com/ru/advanced/best-practice-performance.html	Best practices
Документация Morgan	https://github.com/expressjs/morgan	HTTP-логгер

Все ресурсы доступны и легальны на территории Российской Федерации.

Лабораторная работа № 9

Тема: Проектирование REST API для предметной области

Цель работы:

Научиться самостоятельно проектировать REST API для заданной предметной области, разрабатывать схему базы данных и реализовывать все CRUD-операции для связанных таблиц.

Задачи работы:

1. Проанализировать предметную область и выделить основные сущности.
2. Спроектировать схему базы данных (не менее 3-4 таблиц).
3. Разработать структуру REST API (ресурсы, эндпоинты, методы).
4. Реализовать все CRUD-операции для каждой сущности.
5. Реализовать необходимые связи между таблицами (один-ко-многим, многие-ко-многим).
6. Обеспечить валидацию входных данных.
7. Подготовить документацию API.

1. Теоретическое обоснование

1.1. Проектирование REST API

REST API проектируется вокруг ресурсов -объектов или сущностей предметной области. Каждый ресурс идентифицируется уникальным URL.

Принципы проектирования:

Используйте существительные для названий ресурсов (/users, /orders, /products), а не глаголы.

Используйте HTTP-методы для обозначения действий:

GET - получение ресурса(ов)

POST - создание нового ресурса

PUT/PATCH - обновление ресурса

DELETE - удаление ресурса

Используйте вложенные ресурсы для связей (/users/5/orders -заказы конкретного пользователя).

Возвращайте правильные коды HTTP-статусов.

1.2. Этапы проектирования API

Анализ предметной области - выделение сущностей и их свойств.

Определение связей - один-ко-многим, многие-ко-многим.

Проектирование эндпоинтов - URL, методы, параметры, тело запроса, ответы.

Проектирование базы данных - создание схемы, таблиц, связей.

Реализация - кодирование эндпоинтов, валидация, обработка ошибок.

Документирование - описание API для пользователей.

1.3. Пример проектирования (Интернет-магазин)

Сущности:

1. categories -категории товаров
2. products -товары (принадлежат категории, один-ко-многим)
3. customers -покупатели
4. orders -заказы (принадлежат покупателю)

5. order_items -товары в заказе (связь многие-ко-многим)

Эндпоинты:

Метод	Эндпоинт	Описание
GET	/api/categories	Список категорий
GET	/api/categories/:id/products	Товары категории
GET	/api/products	Список товаров (с фильтрацией)
GET	/api/products/:id	Товар по ID
POST	/api/products	Создание товара
PUT	/api/products/:id	Обновление товара
DELETE	/api/products/:id	Удаление товара
GET	/api/customers	Список покупателей
POST	/api/customers	Регистрация покупателя
GET	/api/customers/:id/orders	Заказы покупателя
POST	/api/orders	Создание заказа

1.4. Схема базы данных (пример)

```
sql
-- Категории
CREATE TABLE categories (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL UNIQUE,
    description TEXT
);

-- Товары
CREATE TABLE products (
```

```
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    category_id INTEGER NOT NULL,
    price REAL NOT NULL CHECK(price >= 0),
    stock INTEGER DEFAULT 0,
    description TEXT,
    FOREIGN KEY (category_id) REFERENCES categories(id)
);

-- Покупатели
CREATE TABLE customers (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    email TEXT UNIQUE NOT NULL,
    phone TEXT,
    registered_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Заказы
CREATE TABLE orders (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    customer_id INTEGER NOT NULL,
    order_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    status TEXT DEFAULT 'new',
    total REAL DEFAULT 0,
    FOREIGN KEY (customer_id) REFERENCES customers(id)
);

-- Товары в заказе
CREATE TABLE order_items (
    order_id INTEGER NOT NULL,
    product_id INTEGER NOT NULL,
    quantity INTEGER NOT NULL CHECK(quantity > 0),
    price_at_moment REAL NOT NULL,
    FOREIGN KEY (order_id) REFERENCES orders(id) ON DELETE
CASCADE,
    FOREIGN KEY (product_id) REFERENCES products(id),
    PRIMARY KEY (order_id, product_id)
);
```

1.5. Документирование API

Документация API должна включать:

Описание каждого эндпоинта

Параметры запроса (path, query, body)

Формат ответа (успех и ошибки)
Примеры запросов и ответов
Пример документации в README.md:
markdown
GET /api/products

Получение списка товаров с возможностью фильтрации.

Параметры query:
- `category_id` (number, optional) -фильтр по категории
- `min_price` (number, optional) -минимальная цена
- `max_price` (number, optional) -максимальная цена
- `page` (number, default: 1) -номер страницы
- `limit` (number, default: 10) -элементов на странице

Ответ (200 OK):
```json
{
  "data": [...],
  "pagination": {
    "total": 42,
    "page": 1,
    "limit": 10,
    "totalPages": 5
  }
}
text

---

## 2. Методика выполнения работы

### 2.1. Примеры и их подробный разбор

#### Пример 1. Анализ предметной области (Библиотека)

**Сущности:**
1. `authors` -авторы (id, name, birth_year, country)
2. `genres` -жанры (id, name)
3. `books` -книги (id, title, author_id, year, price, in_stock)
4. `books_genres` -связь книг и жанров (book_id, genre_id)

**Связи:**
```

- Один автор → много книг (один-ко-многим)
- Много книг ↔ много жанров (многие-ко-многим)

Эндпоинты (план):

Метод	Эндпоинт	Описание
GET	`/api/books`	Список книг (с автором и жанрами)
GET	`/api/books/:id`	Книга по ID
POST	`/api/books`	Создание книги
PUT	`/api/books/:id`	Обновление книги
DELETE	`/api/books/:id`	Удаление книги
GET	`/api/authors`	Список авторов
POST	`/api/authors`	Создание автора
GET	`/api/authors/:id/books`	Книги автора
GET	`/api/genres`	Список жанров
POST	`/api/genres`	Создание жанра
GET	`/api/books/:id/genres`	Жанры книги
POST	`/api/books/:id/genres`	Добавление жанра книге
DELETE	`/api/books/:id/genres/:genreId`	Удаление жанра у книги

Пример 2. Создание схемы базы данных (Библиотека)

```sql
-- Авторы
CREATE TABLE IF NOT EXISTS authors (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL UNIQUE,
    birth_year INTEGER,
    country TEXT
);

-- Жанры
CREATE TABLE IF NOT EXISTS genres (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL UNIQUE
);

-- Книги
CREATE TABLE IF NOT EXISTS books (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    title TEXT NOT NULL,
    author_id INTEGER NOT NULL,
```

```
    year INTEGER,
    price REAL DEFAULT 0,
    in_stock INTEGER DEFAULT 1,
    FOREIGN KEY (author_id) REFERENCES authors(id) ON DELETE
CASCADE
);

-- Связь книг и жанров (многие-ко-многим)
CREATE TABLE IF NOT EXISTS book_genres (
    book_id INTEGER NOT NULL,
    genre_id INTEGER NOT NULL,
    FOREIGN KEY (book_id) REFERENCES books(id) ON DELETE
CASCADE,
    FOREIGN KEY (genre_id) REFERENCES genres(id) ON DELETE
CASCADE,
    PRIMARY KEY (book_id, genre_id)
);
```

Пример 3. Реализация эндпоинтов (Библиотека)

```
javascript
const express = require('express');
const sqlite3 = require('sqlite3').verbose();
const app = express();
app.use(express.json());

const db = new sqlite3.Database('./library.db');

// ========== КНИГИ ==========

// GET все книги с автором и жанрами
app.get('/api/books', (req, res) => {
    const sql = `
        SELECT
            b.*,
            a.name as author_name,
            GROUP_CONCAT(g.name, ', ') as genres
        FROM books b
        LEFT JOIN authors a ON b.author_id = a.id
        LEFT JOIN book_genres bg ON b.id = bg.book_id
        LEFT JOIN genres g ON bg.genre_id = g.id
        GROUP BY b.id
        ORDER BY b.id
    `;
```

```
    db.all(sql, [], (err, rows) => {
        if (err) return res.status(500).json({ error: err.message });
        res.json(rows);
    });
});

// GET книга по ID
app.get('/api/books/:id', (req, res) => {
    const sql = `
        SELECT
            b.*,
            a.name as author_name,
            GROUP_CONCAT(g.id || ':' || g.name, ',') as genres
        FROM books b
        LEFT JOIN authors a ON b.author_id = a.id
        LEFT JOIN book_genres bg ON b.id = bg.book_id
        LEFT JOIN genres g ON bg.genre_id = g.id
        WHERE b.id = ?
        GROUP BY b.id
    `;

    db.get(sql, [req.params.id], (err, row) => {
        if (err) return res.status(500).json({ error: err.message });
        if (!row) return res.status(404).json({ error: 'Книга не найдена' });

        // Парсинг жанров
        if (row.genres) {
            row.genres = row.genres.split(',').map(g => {
                const [id, name] = g.split(':');
                return { id: parseInt(id), name };
            });
        } else {
            row.genres = [];
        }

        res.json(row);
    });
});

// POST создать книгу
app.post('/api/books', (req, res) => {
    const { title, author_id, year, price, in_stock, genre_ids } = req.body;
```

```
        if (!title || !author_id) {
            return res.status(400).json({ error: 'title и author_id обязательны' });
        }

        // Проверка существования автора
        db.get('SELECT * FROM authors WHERE id = ?', [author_id], (err, author) => {
            if (err) return res.status(500).json({ error: err.message });
            if (!author) return res.status(400).json({ error: 'Автор не существует' });

            // Вставка книги
            db.run(
                `INSERT INTO books (title, author_id, year, price, in_stock)
                 VALUES (?, ?, ?, ?, ?)`,
                [title, author_id, year || null, price || 0, in_stock !== undefined ? in_stock : 1],
                function(err) {
                    if (err) return res.status(500).json({ error: err.message });

                    const bookId = this.lastID;

                    // Добавление жанров
                    if (genre_ids && genre_ids.length > 0) {
                        const stmt = db.prepare('INSERT INTO book_genres (book_id, genre_id) VALUES (?, ?)');
                        genre_ids.forEach(genre_id => {
                            stmt.run([bookId, genre_id]);
                        });
                        stmt.finalize();
                    }

                    res.status(201).json({ id: bookId, title, author_id, year, price, in_stock });
                }
            );
        });
    });

    // PUT обновить книгу
    app.put('/api/books/:id', (req, res) => {
        const id = req.params.id;
        const { title, author_id, year, price, in_stock, genre_ids } = req.body;
```

```
        if (!title || !author_id) {
            return res.status(400).json({ error: 'title и author_id обязательны' });
        }

        db.get('SELECT * FROM books WHERE id = ?', [id], (err, book) => {
            if (err) return res.status(500).json({ error: err.message });
            if (!book) return res.status(404).json({ error: 'Книга не найдена' });

            db.get('SELECT * FROM authors WHERE id = ?', [author_id], (err, author) => {
                if (err) return res.status(500).json({ error: err.message });
                if (!author) return res.status(400).json({ error: 'Автор не существует' });

                // Обновление книги
                db.run(
                    `UPDATE books SET title = ?, author_id = ?, year = ?, price = ?, in_stock = ?
                     WHERE id = ?`,
                    [title, author_id, year || null, price || 0, in_stock !== undefined ? in_stock : 1, id],
                    function(err) {
                        if (err) return res.status(500).json({ error: err.message });

                        // Обновление жанров (удаляем старые и добавляем новые)
                        if (genre_ids !== undefined) {
                            db.run('DELETE FROM book_genres WHERE book_id = ?', [id], (err) => {
                                if (err) return;

                                const stmt = db.prepare('INSERT INTO book_genres (book_id, genre_id) VALUES (?, ?)');
                                genre_ids.forEach(genre_id => {
                                    stmt.run([id, genre_id]);
                                });
                                stmt.finalize();
                            });
                        }

                        res.json({ id, title, author_id, year, price, in_stock });
                    }
                );
```

```
        });
    });
});

// DELETE удалить книгу
app.delete('/api/books/:id', (req, res) => {
    const id = req.params.id;

    db.get('SELECT * FROM books WHERE id = ?', [id], (err, book) => {
        if (err) return res.status(500).json({ error: err.message });
        if (!book) return res.status(404).json({ error: 'Книга не найдена' });

        db.run('DELETE FROM books WHERE id = ?', [id], function(err) {
            if (err) return res.status(500).json({ error: err.message });
            res.status(204).send();
        });
    });
});

// ========== АВТОРЫ ==========

app.get('/api/authors', (req, res) => {
    const sql = `
        SELECT a.*, COUNT(b.id) as books_count
        FROM authors a
        LEFT JOIN books b ON a.id = b.author_id
        GROUP BY a.id
        ORDER BY a.name
    `;

    db.all(sql, [], (err, rows) => {
        if (err) return res.status(500).json({ error: err.message });
        res.json(rows);
    });
});

app.get('/api/authors/:id', (req, res) => {
    const id = req.params.id;

    db.get('SELECT * FROM authors WHERE id = ?', [id], (err, author) => {
        if (err) return res.status(500).json({ error: err.message });
        if (!author) return res.status(404).json({ error: 'Автор не найден' });
        res.json(author);
```

```
    });
});

app.post('/api/authors', (req, res) => {
    const { name, birth_year, country } = req.body;

    if (!name) return res.status(400).json({ error: 'name обязателен' });

    db.run(
        'INSERT INTO authors (name, birth_year, country) VALUES (?, ?, ?)',
        [name, birth_year || null, country || null],
        function(err) {
            if (err) return res.status(500).json({ error: err.message });
            res.status(201).json({ id: this.lastID, name, birth_year, country });
        }
    );
});

app.get('/api/authors/:id/books', (req, res) => {
    const authorId = req.params.id;

    db.get('SELECT * FROM authors WHERE id = ?', [authorId], (err,
author) => {
        if (err) return res.status(500).json({ error: err.message });
        if (!author) return res.status(404).json({ error: 'Автор не найден' });

        db.all('SELECT * FROM books WHERE author_id = ?', [authorId],
(err, books) => {
            if (err) return res.status(500).json({ error: err.message });
            res.json({ author, books });
        });
    });
});

// ========== ЖАНРЫ ==========

app.get('/api/genres', (req, res) => {
    db.all('SELECT * FROM genres ORDER BY name', [], (err, rows) => {
        if (err) return res.status(500).json({ error: err.message });
        res.json(rows);
    });
});
```

```
app.post('/api/genres', (req, res) => {
  const { name } = req.body;

  if (!name) return res.status(400).json({ error: 'name обязателен' });

  db.run('INSERT INTO genres (name) VALUES (?)', [name], function(err)
{
    if (err) return res.status(500).json({ error: err.message });
    res.status(201).json({ id: this.lastID, name });
  });
});

// Жанры книги
app.get('/api/books/:id/genres', (req, res) => {
  const bookId = req.params.id;

  const sql = `
    SELECT g.*
    FROM genres g
    JOIN book_genres bg ON g.id = bg.genre_id
    WHERE bg.book_id = ?
    ORDER BY g.name
  `;

  db.all(sql, [bookId], (err, rows) => {
    if (err) return res.status(500).json({ error: err.message });
    res.json(rows);
  });
});

// Добавление жанра книге
app.post('/api/books/:id/genres', (req, res) => {
  const bookId = req.params.id;
  const { genre_id } = req.body;

  if (!genre_id) return res.status(400).json({ error: 'genre_id обязателен' });

  db.get('SELECT * FROM books WHERE id = ?', [bookId], (err, book) =>
{
    if (err) return res.status(500).json({ error: err.message });
    if (!book) return res.status(404).json({ error: 'Книга не найдена' });
```

```
            db.get('SELECT * FROM genres WHERE id = ?', [genre_id], (err, genre) => {
                if (err) return res.status(500).json({ error: err.message });
                if (!genre) return res.status(404).json({ error: 'Жанр не найден' });

                db.run(
                    'INSERT INTO book_genres (book_id, genre_id) VALUES (?, ?)',
                    [bookId, genre_id],
                    function(err) {
                        if (err) return res.status(500).json({ error: err.message });
                        res.status(201).json({ message: 'Жанр добавлен книге' });
                    }
                );
            });
        });
    });

    // Удаление жанра у книги
    app.delete('/api/books/:id/genres/:genreId', (req, res) => {
        const { id, genreId } = req.params;

        db.run(
            'DELETE FROM book_genres WHERE book_id = ? AND genre_id = ?',
            [id, genreId],
            function(err) {
                if (err) return res.status(500).json({ error: err.message });
                if (this.changes === 0) {
                    return res.status(404).json({ error: 'Связь не найдена' });
                }
                res.status(204).send();
            }
        );
    });

    app.listen(3000);
```

2.2. Индивидуальные задания

2.2.1. Базовый уровень

Общее задание для базового уровня:

1. Разработать REST API для заданной предметной области. Минимальные требования:

2. 2-3 сущности (одна из них -главная).
3. Связь один-ко-многим между сущностями.
4. CRUD-операции для главной сущности (GET всех, GET по ID, POST, PUT, DELETE).
5. GET-запрос для получения всех записей из справочника.
6. Валидация обязательных полей.

Варианты заданий:

	Предметная область	Главная сущность	Справочник (один-ко-многим)	Дополнительные поля
	Библиотека	Книги	Авторы	год, цена ISBN
	Интернет-магазин	Товары	Категории	цена, количество, описание
	Университет	Студенты	Группы	возраст, email телефон
	Кинотеатр	Фильмы	Режиссёры	год, рейтинг длительность
	IT-компания	Сотрудники	Департаменты	должность, зарплата, дата найма
	Ресторан	Блюда	Категории блюд	цена, время готовки, калории
	Автосалон	Автомобили	Бренды	год, цена, цвет
	Музыкальный сервис	Песни	Исполнители	альбом, длительность, жанр
	Туризм	Отели	Города	звёзды, цена за ночь, адрес
0	Образование	Курсы	Инструкторы	длительность, цена, уровень
	Аптека	Лекарст	Производи	цена, рецепт,

	Предметная область	Главная сущность	Справочник (один-ко-многим)	Дополнительные поля
1		ва	тели	наличии
2	Клуб	Мероприятия	Локации	дата, цена макс. участников
3	Спорт	Спортсмены	Виды спорта	страна, медали, возраст
4	Блог	Статьи	Авторы	дата публикации, просмотры
5	CRM	Заказы	Клиенты	сумма, статус дата заказа

Пример выполнения для варианта 1 (Базовый уровень) -структура API:

markdown
API Библиотека

Книги
- GET /api/books -список всех книг
- GET /api/books/:id -книга по ID
- POST /api/books -создать книгу (поля: title, author_id, year, price)
- PUT /api/books/:id -обновить книгу
- DELETE /api/books/:id -удалить книгу

Авторы
- GET /api/authors -список авторов
- POST /api/authors -создать автора (поля: name, birth_year)

2.2.2. Средний уровень (15 вариантов)

Общее задание для среднего уровня:

1. Выполнить требования базового уровня, а также дополнительно реализовать:
2. Третью сущность (второй справочник) или дополнительную связь.

3. GET /api/main/:id/details -получение главной сущности со всеми связанными данными (с JOIN).
4. GET /api/ref/:id/items -получение всех записей главной сущности для конкретного элемента справочника.
5. Фильтрацию по одному из полей (query-параметр).
6. Сортировку по одному из полей (sort, order).

Дополнительные сущности для каждого варианта (средний уровень):

	Основная + первый справочник	Вторая сущность/связь
	Книги → Авторы	Жанры (многие-ко-многим)
	Товары → Категории	Поставщики (многие-ко-многим)
	Студенты → Группы	Дисциплины (многие-ко-многим с оценками)
	Фильмы → Режиссёры	Жанры (многие-ко-многим)
	Сотрудники → Департаменты	Проекты (многие-ко-многим)
	Блюда → Категории	Ингредиенты (многие-ко-многим)
	Автомобили → Бренды	Дилеры (многие-ко-многим)
	Песни → Исполнители	Альбомы (один-ко-многим)
	Отели → Города	Удобства (многие-ко-многим)
0	Курсы → Инструкторы	Студенты (многие-ко-многим)
1	Лекарства → Производители	Болезни (многие-ко-многим)

	Основная + первый справочник	Вторая сущность/связь
2	Мероприятия → Локации	Организаторы (один-ко-многим)
3	Спортсмены → Виды спорта	Тренировки (один-ко-многим)
4	Статьи → Авторы	Теги (многие-ко-многим)
5	Заказы → Клиенты	Товары (многие-ко-многим через order_items)

2.2.3. Продвинутый уровень (15 вариантов)

Общее задание для продвинутого уровня:

1. Выполнить требования среднего уровня, а также дополнительно реализовать:
2. Полноценную связь многие-ко-многим (с промежуточной таблицей).
3. Пагинацию (page, limit) для всех GET-списков.
4. Поиск (q) по нескольким текстовым полям.
5. Фильтрацию по нескольким критериям (минимум 3 фильтра).
6. Агрегацию данных (COUNT, SUM, AVG) в ответах.
7. Логирование через morgan и переменные окружения.
8. Документацию API в формате OpenAPI (Swagger) или подробный README.

Дополнительные требования для продвинутого уровня:

№	Дополнительная функциональность
1	Подсчёт количества книг по жанрам, фильтрация книг по цене и наличию
2	Анализ продаж по категориям, популярные товары
3	Средний балл студентов по дисциплинам
4	Статистика фильмов по годам и режиссёрам
5	Загрузка сотрудников по проектам

№	Дополнительная функциональность
6	Популярность блюд, средняя калорийность
7	Статистика продаж по брендам и годам
8	Популярность исполнителей, длительность плейлистов
9	Средняя цена отелей по городам
10	Популярность курсов, средняя оценка
11	Статистика назначений лекарств по болезням
12	Анализ посещаемости мероприятий
13	Медальный зачёт по видам спорта
14	Популярность статей по тегам
15	Аналитика по клиентам и суммам заказов

Пример выполнения для варианта 1 (Продвинутый уровень) - расширенный функционал:

javascript

```
// GET /api/books с фильтрацией, пагинацией и агрегацией
app.get('/api/books', (req, res) => {
    let sql = `
        SELECT
            b.*,
            a.name as author_name,
            GROUP_CONCAT(DISTINCT g.name) as genres
        FROM books b
        LEFT JOIN authors a ON b.author_id = a.id
        LEFT JOIN book_genres bg ON b.id = bg.book_id
        LEFT JOIN genres g ON bg.genre_id = g.id
        WHERE 1=1
    `;
    const params = [];

    // Фильтр по автору
    if (req.query.author_id) {
```

```
            sql += ' AND b.author_id = ?';
            params.push(req.query.author_id);
        }

        // Фильтр по жанру
        if (req.query.genre_id) {
            sql += ' AND EXISTS (SELECT 1 FROM book_genres bg2 WHERE
bg2.book_id = b.id AND bg2.genre_id = ?)';
            params.push(req.query.genre_id);
        }

        // Диапазон цен
        if (req.query.minPrice) {
            sql += ' AND b.price >= ?';
            params.push(req.query.minPrice);
        }
        if (req.query.maxPrice) {
            sql += ' AND b.price <= ?';
            params.push(req.query.maxPrice);
        }

        // Поиск
        if (req.query.q) {
            sql += ' AND (b.title LIKE ? OR a.name LIKE ?)';
            const searchPattern = `%${req.query.q}%`;
            params.push(searchPattern, searchPattern);
        }

        // Группировка (для агрегации жанров)
        sql += ' GROUP BY b.id';

        // Сортировка
        const sortField = req.query.sort || 'b.id';
        const sortOrder = req.query.order === 'desc' ? 'DESC' : 'ASC';
        sql += ` ORDER BY ${sortField} ${sortOrder}`;

        // Пагинация
        const page = parseInt(req.query.page) || 1;
        const limit = parseInt(req.query.limit) || 10;
        const offset = (page - 1) * limit;
        sql += ' LIMIT ? OFFSET ?';
        params.push(limit, offset);
```

```
        db.all(sql, params, (err, rows) => {
            if (err) return res.status(500).json({ error: err.message });

            // Преобразование строки жанров в массив
            rows = rows.map(row => ({
                ...row,
                genres: row.genres ? row.genres.split(',') : []
            }));

            // Подсчёт общего количества
            let countSql = `
                SELECT COUNT(DISTINCT b.id) as total
                FROM books b
                LEFT JOIN authors a ON b.author_id = a.id
                WHERE 1=1
            `;
            const countParams = [];
            if (req.query.author_id) {
                countSql += ' AND b.author_id = ?';
                countParams.push(req.query.author_id);
            }
            if (req.query.genre_id) {
                countSql += ' AND EXISTS (SELECT 1 FROM book_genres bg2
WHERE bg2.book_id = b.id AND bg2.genre_id = ?)';
                countParams.push(req.query.genre_id);
            }
            if (req.query.minPrice) {
                countSql += ' AND b.price >= ?';
                countParams.push(req.query.minPrice);
            }
            if (req.query.maxPrice) {
                countSql += ' AND b.price <= ?';
                countParams.push(req.query.maxPrice);
            }
            if (req.query.q) {
                countSql += ' AND (b.title LIKE ? OR a.name LIKE ?)';
                const searchPattern = `%${req.query.q}%`;
                countParams.push(searchPattern, searchPattern);
            }

            db.get(countSql, countParams, (err, countRow) => {
                if (err) return res.status(500).json({ error: err.message });
                res.json({
```

```
                data: rows,
                pagination: {
                    total: countRow.total,
                    page: page,
                    limit: limit,
                    totalPages: Math.ceil(countRow.total / limit)
                }
            });
        });
    });
});

// GET /api/statistics -статистика по библиотеке
app.get('/api/statistics', (req, res) => {
    const sql = `
        SELECT
            (SELECT COUNT(*) FROM books) as total_books,
            (SELECT COUNT(*) FROM authors) as total_authors,
            (SELECT COUNT(*) FROM genres) as total_genres,
            (SELECT ROUND(AVG(price), 2) FROM books) as avg_price,
            (SELECT COUNT(*) FROM books WHERE in_stock = 1) as books_in_stock,
            (SELECT MIN(price) FROM books) as min_price,
            (SELECT MAX(price) FROM books) as max_price,
            (SELECT MIN(year) FROM books) as oldest_book,
            (SELECT MAX(year) FROM books) as newest_book
    `;

    db.get(sql, [], (err, row) => {
        if (err) return res.status(500).json({ error: err.message });
        res.json(row);
    });
});
```

3. Контрольные вопросы

1. Какие этапы включает проектирование REST API?
2. Как выбрать правильные имена для эндпоинтов?
3. Какой HTTP-метод используется для каждой CRUD-операции?
4. Как проектировать эндпоинты для связанных ресурсов?
5. Что такое связь «многие-ко-многим» и как она реализуется в SQL?
6. Какую информацию должна содержать документация API?
7. Как организовать пагинацию в REST API?

8. Как реализовать фильтрацию с несколькими критериями?
9. Какие коды HTTP-статусов нужно возвращать для разных ситуаций?
10. Как обрабатывать ошибки валидации в API?
11. Что такое промежуточная таблица и зачем она нужна?
12. Как спроектировать эндпоинты для вложенных ресурсов (/users/5/orders)?
13. Как реализовать поиск по API с релевантностью?
14. Какую информацию нужно включать в ответ API для пагинации?
15. Чем отличается проектирование API для мобильного приложения от веб-приложения?

4. Требования к отчёту

Отчёт должен быть оформлен в электронном виде и содержать:

Титульный лист с названием работы, ФИО студента, номером группы, вариантом.

Анализ предметной области:

Описание предметной области.

Выделенные сущности и их атрибуты.

Связи между сущностями.

Проектирование API:

Таблица эндпоинтов (метод, URL, описание).

Примеры запросов и ответов для каждого эндпоинта.

Схема базы данных:

SQL-скрипты создания всех таблиц.

Описание связей (FOREIGN KEY).

Реализация:

Полный листинг кода сервера с комментариями.

Скриншоты проверки работы через Postman.

Документация API (README или OpenAPI).

Ответы на контрольные вопросы.

Заключение.

Список использованных источников.

5. Критерии оценивания

Уровень выполнения	Требования	Оценка
Базовый	2-3 сущности, связь один-ко-многим, CRUD для главной сущности, GET для справочника,	Удовлетворительно

Уровень выполнения	Требования	Оценка
	валидация.	
Средний	Выполнены требования базового уровня + третья сущность, GET /main/:id/details, GET /ref/:id/items, фильтрация, сортировка.	Хорошо
Продвинутый	Выполнены требования среднего уровня + связь многие-ко-многим, пагинация, поиск (q), многокритериальная фильтрация, агрегация, логирование, переменные окружения, документация.	Отлично

Дополнительные условия:

1. Код должен быть структурирован и снабжён комментариями.
2. API должен быть документирован.
3. При защите студент должен обосновать архитектурные решения.

6. Список рекомендуемой литературы и интернет-ресурсов

Литература (доступна в РФ)

1. Браун, И. (2021). *Express.js: Разработка веб-приложений на Node.js*. ДМК Пресс.
2. Кантелон, М., Хартер, М., & Холстейн, С. (2020). *Node.js в действии*. Питер.
3. Ричардсон, Л., & Руби, С. (2019). *RESTful Web APIs*. Питер.

Интернет-ресурсы

Название	Ссылка	Описание
REST API Tutorial	https://restfulapi.net/	Основы REST (англ.)
OpenAPI Specification	https://swagger.io/specification/	Спецификация OpenAPI

Название	Ссылка	Описание
SQLite Documentation	https://www.sqlite.org/docs.html	Официальная документация
Express.js Routing	https://expressjs.com/ru/guide/routing.html	Маршрутизация в Express
Habr - Проектирование API	https://habr.com/ru/articles/	Статьи по API (поиск)
Postman Learning Center	https://learning.postman.com/	Документация Postman

Все ресурсы доступны и легальны на территории Российской Федерации.

Лабораторная работа № 10

Тема: Валидация входных данных и обработка ошибок

Цель работы:

Научиться проверять корректность входящих данных, реализовывать валидацию на стороне сервера, обрабатывать ошибки и возвращать осмысленные сообщения об ошибках с соответствующими HTTP-кодами.

Задачи работы:

1. Реализовать валидацию обязательных полей при создании и обновлении ресурсов.
2. Реализовать проверку типов данных (число, строка, boolean).
3. Реализовать проверку допустимых значений (диапазоны, перечисления, форматы).
4. Реализовать проверку уникальности полей (при необходимости).
5. Создать унифицированный формат ответа для ошибок валидации.
6. Реализовать глобальный обработчик ошибок.
7. Освоить различные HTTP-коды для разных типов ошибок (400, 404, 409, 422, 500).

1. Теоретическое обоснование

1.1. Почему нужна валидация?

Валидация входных данных -критически важный аспект безопасности и надёжности бэкенд-приложения:

Защита от некорректных данных -предотвращение ошибок в бизнес-логике.

Безопасность -защита от SQL-инъекций, XSS-атак.

Целостность данных -гарантия, что в базу попадают только корректные данные.

Пользовательский опыт -понятные сообщения об ошибках.

1.2. Уровни валидации

Уровень	Описание	Пример
Синтаксическая	Проверка наличия и типов полей	title обязателен, price -число
Семантическая	Проверка смысловой корректности	age от 0 до 120, email имеет формат
Бизнес-логика	Проверка правил предметной области	сумма заказа не превышает баланс
Уникальность	Проверка отсутствия дубликатов	email не занят, название товара уникально

1.3. HTTP-коды для ошибок

од	Название	Когда использовать
00	Bad Request	Синтаксическая ошибка, отсутствие обязательного поля
01	Unauthorized	Требуется аутентификация
03	Forbidden	Нет прав для доступа
04	Not Found	Ресурс не найден
09	Conflict	Конфликт данных (например, дубликат email)
22	Unprocessable Entity	Данные валидны по синтаксису, но не проходят семантику
29	Too Many Requests	Превышен лимит запросов
00	Internal Server Error	Непредвиденная ошибка сервера

1.4. Унифицированный формат ошибок

Рекомендуемый формат ответа для ошибок валидации:

json

```
{
  "errors": [
    {
      "field": "email",
      "message": "Email должен быть в формате user@example.com",
      "code": "invalid_format"
    },
    {
      "field": "age",
      "message": "Возраст должен быть от 0 до 120",
      "code": "out_of_range",
      "value": 150
    }
  ]
}
```

Для простых ошибок можно использовать упрощённый формат:

json

```
{
  "error": "Поле 'title' обязательно для заполнения",
  "code": "missing_field"
}
```

1.5. Подходы к валидации в Express

1. Встроенная валидация в обработчиках (ручная):

javascript

```
app post('/api/users', (req, res) => {
  const errors = [];

  if (!req body name) {
```

```
    errors.push({ field: 'name', message: 'Имя обязательно' });
  }

  if (req.body.age && (req.body.age < 0 || req.body.age > 120)) {
    errors.push({ field: 'age', message: 'Возраст от 0 до 120' });
  }

  if (errors.length > 0) {
    return res.status(400).json({ errors });
  }

  // Продолжаем...
});
```

2. Валидация через middleware-функции:

javascript

```
function validateUser(req, res, next) {
  const errors = [];
  // ... валидация
  if (errors.length > 0) {
    return res.status(400).json({ errors });
  }
  next();
}

app.post('/api/users', validateUser, (req, res) => {
  // Обработка после валидации
});
```

3. Валидация через библиотеки (express-validator, Joi):

```
javascript
const { body, validationResult } = require('express-validator');

app.post('/api/users',
  body('name').notEmpty().withMessage('Имя обязательно'),
  body('email').isEmail().withMessage('Некорректный email'),
  (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.status(400).json({ errors: errors.array() });
    }
    // Продолжаем...
  }
);
```

1.6. Глобальный обработчик ошибок

В Express можно создать глобальный обработчик для перехвата всех необработанных ошибок:

```
javascript
// Глобальный обработчик ошибок (должен быть в конце)
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).json({
    error: 'Внутренняя ошибка сервера',
    message: process.env.NODE_ENV === 'development' ? err.message : undefined
  });
});
```

2. Методика выполнения работы

2.1. Примеры и их подробный разбор

Пример 1. Базовая валидация обязательных полей

javascript

```
// POST /api/books -создание книги
app.post('/api/books', (req, res) => {
    const { title, author, year, price, in_stock } = req.body;
    const errors = [];

    // Проверка обязательных полей
    if (!title || title.trim().length === 0) {
        errors.push({ field: 'title', message: 'Название книги обязательно' });
    }

    if (!author || author.trim().length === 0) {
        errors.push({ field: 'author', message: 'Автор обязателен' });
    }

    if (errors.length > 0) {
        return res.status(400).json({ errors });
    }

    // Создание книги...
    res.status(201).json({ message: 'Книга создана' });
});
```

Пример 2. Валидация типов и диапазонов

javascript

```
app.post('/api/products', (req, res) => {
```

```
    const { name, price, quantity, category } = req.body;
    const errors = [];

    // Проверка имени
    if (!name || name.trim().length === 0) {
      errors.push({ field: 'name', message: 'Название товара обязательно' });
    } else if (name.length < 3) {
      errors.push({ field: 'name', message: 'Название должно содержать минимум 3 символа' });
    } else if (name.length > 100) {
      errors.push({ field: 'name', message: 'Название не может превышать 100 символов' });
    }

    // Проверка цены
    if (price === undefined) {
      errors.push({ field: 'price', message: 'Цена обязательна' });
    } else if (typeof price !== 'number') {
      errors.push({ field: 'price', message: 'Цена должна быть числом' });
    } else if (price < 0) {
      errors.push({ field: 'price', message: 'Цена не может быть отрицательной' });
    } else if (price > 1000000) {
      errors.push({ field: 'price', message: 'Цена не может превышать 1 000 000' });
    }

    // Проверка количества
```

```
    if (quantity !== undefined) {
      if (!Number.isInteger(quantity)) {
        errors.push({ field: 'quantity', message: 'Количество должно быть целым числом' });
      } else if (quantity < 0) {
        errors.push({ field: 'quantity', message: 'Количество не может быть отрицательным' });
      }
    }

    // Проверка категории
    const validCategories = ['electronics', 'clothing', 'books', 'food'];
    if (category && !validCategories.includes(category)) {
      errors.push({
        field: 'category',
        message: `Категория должна быть одной из: ${validCategories.join(', ')}`,
        code: 'invalid_enum_value'
      });
    }

    if (errors.length > 0) {
      return res.status(422).json({ errors });
    }

    // Создание товара...
    res.status(201).json({ message: 'Товар создан' });
  });
```

Пример 3. Валидация формата email и телефона

```
javascript
// Функция проверки email
function isValidEmail(email) {
    const emailRegex = /^[^\s@]+@([^\s@.,]+\.)+[^\s@.,]{2,}$/;
    return emailRegex test(email);
}

// Функция проверки телефона (пример: +7 XXX XXX XX XX)
function isValidPhone(phone) {
    const phoneRegex = /^\+7\d{10}$/;
    return phoneRegex.test(phone);
}

app post('/api/customers', (req, res) => {
    const { name, email, phone, age } = req body;
    const errors = [];

    // Валидация имени
    if (!name || name trim() length < 2) {
        errors push({ field: 'name', message: 'Имя должно содержать минимум 2 символа' });
    }

    // Валидация email
    if (!email) {
        errors push({ field: 'email', message: 'Email обязателен' });
    } else if (!isValidEmail(email)) {
        errors push({ field: 'email', message: 'Введите корректный email (например, user@example.com)' });
```

```
    }

    // Валидация телефона (опционально)
    if (phone && !isValidPhone(phone)) {
      errors.push({ field: 'phone', message: 'Телефон должен быть в формате +7XXXXXXXXXX' });
    }

    // Валидация возраста
    if (age !== undefined) {
      if (typeof age !== 'number') {
        errors.push({ field: 'age', message: 'Возраст должен быть числом' });
      } else if (age < 18) {
        errors.push({ field: 'age', message: 'Минимальный возраст -18 лет' });
      } else if (age > 120) {
        errors.push({ field: 'age', message: 'Максимальный возраст -120 лет' });
      }
    }

    if (errors.length > 0) {
      return res.status(422).json({ errors });
    }

    res.status(201).json({ message: 'Покупатель создан' });
  });
```

Пример 4. Проверка уникальности (с запросом к БД)

```
javascript
app.post('/api/users', async (req, res) => {
    const { username, email } = req.body;
    const errors = [];

    // Проверка уникальности username
    db.get('SELECT * FROM users WHERE username = ?', [username], (err, existingUser) => {
        if (err) {
            return res.status(500).json({ error: 'Ошибка базы данных' });
        }

        if (existingUser) {
            errors.push({ field: 'username', message: 'Пользователь с таким именем уже существует' });
        }

        // Проверка уникальности email
        db.get('SELECT * FROM users WHERE email = ?', [email], (err, existingEmail) => {
            if (err) {
                return res.status(500).json({ error: 'Ошибка базы данных' });
            }

            if (existingEmail) {
                errors.push({ field: 'email', message: 'Пользователь с таким email уже существует' });
            }
```

```
    if (errors.length > 0) {
      return res.status(409).json({ errors });
    }

    // Создание пользователя...
    res.status(201).json({ message: 'Пользователь создан' });
  });
});
```

Пример 5. Валидация при обновлении (PATCH)

javascript

```
app.patch('/api/books/:id', (req, res) => {
  const id = parseInt(req.params.id);
  const updates = req.body;
  const errors = [];
  const allowedFields = ['title', 'author', 'year', 'price', 'in_stock'];

  // Проверка, что обновляются только разрешённые поля
  const invalidFields = Object.keys(updates).filter(field => !allowedFields.includes(field));
  if (invalidFields.length > 0) {
    errors.push({
      field: 'fields',
      message: `Недопустимые поля: ${invalidFields.join(', ')}`,
      code: 'invalid_fields'
    });
  }

  // Валидация title
```

```
    if (updates.title !== undefined) {
      if (!updates.title || updates.title.trim().length === 0) {
        errors.push({ field: 'title', message: 'Название не может быть пустым' });
      } else if (updates.title.length > 200) {
        errors.push({ field: 'title', message: 'Название не может превышать 200 символов' });
      }
    }

    // Валидация year
    if (updates.year !== undefined) {
      if (typeof updates.year !== 'number') {
        errors.push({ field: 'year', message: 'Год должен быть числом' });
      } else if (updates.year < 0 || updates.year > 2025) {
        errors.push({ field: 'year', message: 'Год должен быть от 0 до 2025' });
      }
    }

    // Валидация price
    if (updates.price !== undefined) {
      if (typeof updates.price !== 'number') {
        errors.push({ field: 'price', message: 'Цена должна быть числом' });
      } else if (updates.price < 0) {
        errors.push({ field: 'price', message: 'Цена не может быть отрицательной' });
      }
    }
```

```
    if (errors.length > 0) {
      return res.status(422).json({ errors });
    }

    // Проверка существования записи
    db.get('SELECT * FROM books WHERE id = ?', [id], (err, book) => {
      if (err) return res.status(500).json({ error: err.message });
      if (!book) return res.status(404).json({ error: 'Книга не найдена' });

      // Обновление книги...
      res.json({ message: 'Книга обновлена' });
    });
  });
```

Пример 6. Полный пример с глобальным обработчиком ошибок

```
javascript
require('dotenv').config();
const express = require('express');
const sqlite3 = require('sqlite3').verbose();

const app = express();
app.use(express.json());

const db = new sqlite3.Database('./store.db');

// ========== ВСПОМОГАТЕЛЬНЫЕ ФУНКЦИИ ДЛЯ ВАЛИДАЦИИ ==========

const validCategories = ['electronics', 'clothing', 'books', 'home', 'toys'];
```

```
function validateProduct(data, isUpdate = false) {
    const errors = [];

    if (!isUpdate || data.name !== undefined) {
        if (!data.name || data.name.trim().length === 0) {
            errors.push({ field: 'name', message: 'Название товара обязательно' });
        } else if (data.name.length < 3) {
            errors.push({ field: 'name', message: 'Название должно содержать минимум 3 символа' });
        } else if (data.name.length > 100) {
            errors.push({ field: 'name', message: 'Название не может превышать 100 символов' });
        }
    }

    if (!isUpdate || data.price !== undefined) {
        if (data.price === undefined) {
            if (!isUpdate) errors.push({ field: 'price', message: 'Цена обязательна' });
        } else if (typeof data.price !== 'number') {
            errors.push({ field: 'price', message: 'Цена должна быть числом' });
        } else if (data.price < 0) {
            errors.push({ field: 'price', message: 'Цена не может быть отрицательной' });
        } else if (data.price > 1000000) {
            errors.push({ field: 'price', message: 'Цена не может превышать 1 000 000' });
```

```
        }
      }

      if (!isUpdate || data.category !== undefined) {
        if (data.category && !validCategories.includes(data.category)) {
          errors.push({
            field: 'category',
            message: `Категория должна быть одной из: ${validCategories.join(', ')}`,
            code: 'invalid_category'
          });
        }
      }

      if (!isUpdate || data.stock !== undefined) {
        if (data.stock !== undefined) {
          if (!Number.isInteger(data.stock)) {
            errors.push({ field: 'stock', message: 'Количество должно быть целым числом' });
          } else if (data.stock < 0) {
            errors.push({ field: 'stock', message: 'Количество не может быть отрицательным' });
          }
        }
      }

      return errors;
    }
```

```
// ========== ЭНДПОИНТЫ ==========

// POST /api/products -создание товара
app.post('/api/products', (req, res) => {
  const { name, price, category, stock, description } = req.body;

  const errors = validateProduct({ name, price, category, stock });
  if (errors.length > 0) {
    return res.status(422).json({ errors });
  }

  // Проверка уникальности названия (если нужно)
  db.get('SELECT * FROM products WHERE name = ?', [name], (err, existing) => {
    if (err) {
      return res.status(500).json({ error: 'Ошибка базы данных' });
    }

    if (existing) {
      return res.status(409).json({
        errors: [{ field: 'name', message: 'Товар с таким названием уже существует' }]
      });
    }

    db.run(
      `INSERT INTO products (name, price, category, stock, description)
       VALUES (?, ?, ?, ?, ?)`,
      [name, price, category || null, stock || 0, description || null],
```

```
            function(err) {
                if (err) {
                    return res.status(500).json({ error: 'Ошибка при создании товара' });
                }
                res.status(201).json({
                    id: this.lastID,
                    message: 'Товар успешно создан'
                });
            }
        );
    });
});

// PUT /api/products/:id -полное обновление товара
app.put('/api/products/:id', (req, res) => {
    const id = parseInt(req.params.id);
    const { name, price, category, stock, description } = req.body;

    // Валидация всех полей (при PUT они все обязательны)
    if (!name || price === undefined) {
        return res.status(400).json({
            error: 'При PUT-запросе все поля (name, price) обязательны'
        });
    }

    const errors = validateProduct({ name, price, category, stock });
    if (errors.length > 0) {
        return res.status(422).json({ errors });
```

```
    }

    // Проверка существования товара
    db.get('SELECT * FROM products WHERE id = ?', [id], (err, product) =>
{
      if (err) return res.status(500).json({ error: 'Ошибка базы данных' });
      if (!product) return res.status(404).json({ error: 'Товар не найден' });

      db.run(
        `UPDATE products SET name = ?, price = ?, category = ?, stock = ?,
description = ?
         WHERE id = ?`,
        [name, price, category || null, stock || 0, description || null, id],
        function(err) {
          if (err) return res.status(500).json({ error: 'Ошибка при
обновлении' });
          res.json({ message: 'Товар успешно обновлён' });
        }
      );
    });
  });

  // PATCH /api/products/:id -частичное обновление
  app.patch('/api/products/:id', (req, res) => {
    const id = parseInt(req.params.id);
    const updates = req.body;

    // Проверка, что тело запроса не пустое
    if (Object.keys(updates).length === 0) {
```

```
            return res.status(400).json({ error: 'Тело запроса не может быть пустым' });
        }

        const errors = validateProduct(updates, true);
        if (errors.length > 0) {
            return res.status(422).json({ errors });
        }

        // Проверка существования товара
        db.get('SELECT * FROM products WHERE id = ?', [id], (err, product) => {
            if (err) return res.status(500).json({ error: 'Ошибка базы данных' });
            if (!product) return res.status(404).json({ error: 'Товар не найден' });

            const fields = [];
            const values = [];

            if (updates.name !== undefined) {
                fields.push('name = ?');
                values.push(updates.name);
            }
            if (updates.price !== undefined) {
                fields.push('price = ?');
                values.push(updates.price);
            }
            if (updates.category !== undefined) {
                fields.push('category = ?');
                values.push(updates.category);
```

```
    }
    if (updates.stock !== undefined) {
      fields.push('stock = ?');
      values.push(updates.stock);
    }
    if (updates.description !== undefined) {
      fields.push('description = ?');
      values.push(updates.description);
    }

    values.push(id);
    const sql = `UPDATE products SET ${fields.join(', ')} WHERE id = ?`;

    db.run(sql, values, function(err) {
      if (err) return res.status(500).json({ error: 'Ошибка при обновлении' });
      res.json({ message: 'Товар успешно обновлён' });
    });
  });
});

// DELETE /api/products/:id
app.delete('/api/products/:id', (req, res) => {
  const id = parseInt(req.params.id);

  db.get('SELECT * FROM products WHERE id = ?', [id], (err, product) => {
    if (err) return res.status(500).json({ error: 'Ошибка базы данных' });
    if (!product) return res.status(404).json({ error: 'Товар не найден' });
```

```
            db.run('DELETE FROM products WHERE id = ?', [id], function(err) {
                if (err) return res.status(500).json({ error: 'Ошибка при удалении' });
                res.status(204).send();
            });
        });
    });

    // ========== ГЛОБАЛЬНЫЙ ОБРАБОТЧИК ОШИБОК ==========

    // Обработчик несуществующих маршрутов (404)
    app.use((req, res) => {
        res.status(404).json({
            error: 'Маршрут не найден',
            path: req.originalUrl,
            method: req.method
        });
    });

    // Глобальный обработчик ошибок (500)
    app.use((err, req, res, next) => {
        console.error('Ошибка:', err.stack);
        res.status(500).json({
            error: 'Внутренняя ошибка сервера',
            message: process.env.NODE_ENV === 'development' ? err.message : undefined
        });
    });
```

```
// ========== ЗАПУСК СЕРВЕРА ==========

const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
    console.log(`Сервер запущен на http://localhost:${PORT}`);
});
```

2.2. Индивидуальные задания

2.2.1. Базовый уровень (15 вариантов)

Общее задание для базового уровня:

1. Реализовать валидацию для существующего API (из предыдущих работ) или создать новый CRUD API. Требования:
2. Валидация обязательных полей при POST и PUT запросах.
3. Проверка длины строковых полей (минимум и максимум).
4. Проверка числовых полей на диапазон (цена ≥ 0, год в разумных пределах).
5. Возврат ошибок с кодом 400 или 422.
6. Унифицированный формат ошибок (например, { "field": "name", "message": "..." }).

Варианты сущностей для валидации:

	Сущность	Обязательные поля	Длина (мин/макс)	Числовые диапазоны
	Книга	title, author	title: 1-200, author: 2-100	year: 0-2025, price: ≥0
	Товар	name, price	name: 3-100, category: 2-50	price: ≥0, stock: ≥0

	Сущность	Обязательные поля	Длина (мин/макс)	Числовые диапазоны
	Студент	full_name, group	full_name: 2-100, email: 5-100	age: 17-100
	Фильм	title, director	title: 1-150, director: 2-100	year: 1900-2025, rating: 0-10
	Сотрудник	name, position	name: 2-100, position: 2-50	salary: ≥0
	Рецепт	title, cuisine	title: 3-100, cuisine: 2-50	cooking_time: ≥1
	Автомобиль	model, brand	model: 1-100, brand: 2-50	year: 1900-2025, price: ≥0
	Песня	title, artist	title: 1-150, artist: 2-100	duration: 10-600
	Отель	name, city	name: 2-100, city: 2-50	stars: 1-5, price: ≥0
	Курс	title,	title: 3-	duration: 1-

	Сущность	Обязательные поля	Длина (мин/макс)	Числовые диапазоны
0		instructor	100, instructor: 2-50	500, price: ≥0
1	Лекарство	name, manufacturer	name: 2-100, manufacturer: 2-50	price: ≥0
2	Мероприятие	name, location	name: 3-100, location: 2-50	price: ≥0, max_participants: 1-10000
3	Заказ	customer_name, product	customer: 2-100, product: 1-100	quantity: ≥1, amount: ≥0
4	Статья	title, author	title: 5-200, author: 2-50	views: ≥0
5	Спортсмен	name, sport	name: 2-100, sport: 2-50	medals: ≥0, age: 14-50

Пример выполнения для варианта 1 (Базовый уровень):

javascript

```
// POST /api/books -создание книги с валидацией
```

```
app post('/api/books', (req, res) => {
    const { title, author, year, price } = req body;
    const errors = [];

    // Валидация title
    if (!title) {
        errors push({ field: 'title', message: 'Название книги обязательно' });
    } else if (title length < 1) {
        errors push({ field: 'title', message: 'Название не может быть пустым' });
    } else if (title length > 200) {
        errors push({ field: 'title', message: 'Название не может превышать 200 символов' });
    }

    // Валидация author
    if (!author) {
        errors push({ field: 'author', message: 'Автор обязателен' });
    } else if (author length < 2) {
        errors push({ field: 'author', message: 'Имя автора должно содержать минимум 2 символа' });
    } else if (author length > 100) {
        errors push({ field: 'author', message: 'Имя автора не может превышать 100 символов' });
    }

    // Валидация year
    if (year !== undefined) {
        if (typeof year !== 'number') {
```

```
            errors.push({ field: 'year', message: 'Год должен быть числом' });
        } else if (year < 0 || year > 2025) {
            errors.push({ field: 'year', message: 'Год должен быть от 0 до 2025' });
        }
    }

    // Валидация price
    if (price !== undefined) {
        if (typeof price !== 'number') {
            errors.push({ field: 'price', message: 'Цена должна быть числом' });
        } else if (price < 0) {
            errors.push({ field: 'price', message: 'Цена не может быть отрицательной' });
        }
    }

    if (errors.length > 0) {
        return res.status(400).json({ errors });
    }

    // Создание книги...
    res.status(201).json({ message: 'Книга создана' });
});
```

2.2.2. Средний уровень (15 вариантов)

Общее задание для среднего уровня:

1. Выполнить требования базового уровня, а также дополнительно реализовать:

2. Валидацию форматов (email, телефон, дата, URL).
3. Валидацию перечислений (enum) для полей с ограниченным набором значений.
4. Валидацию уникальности (проверка через БД).
5. Разные форматы ошибок для разных типов валидации (400 - синтаксис, 409 -дубликат).
6. Middleware для валидации (вынос логики валидации в отдельные функции).

Дополнительные поля для валидации (по вариантам):

	Дополнительное поле	Тип валидации	Допустимые значения / формат
	email (автора)	email	user@example.com
	status (товара)	enum	'available', 'out_of_stock', 'discontinued'
	email (студента) + phone	email + phone	email, +7XXXXXXXXXX
	genre (фильма)	enum	'action', 'drama', 'comedy', 'horror', 'sci-fi'
	status (сотрудника)	enum	'active', 'vacation', 'fired'
	difficulty (рецепта)	enum	'easy', 'medium', 'hard'
	fuel_type (автомобиля)	enum	'petrol', 'diesel', 'electric', 'hybrid'
	genre (песни)	enum	'rock', 'pop', 'jazz', 'classical',

	Дополнительное поле	Тип валидации	Допустимые значения / формат
			'hip-hop'
	status (отеля)	enum	'open', 'closed', 'renovation'
0	level (курса)	enum	'beginner', 'intermediate', 'advanced'
1	prescription (лекарства)	boolean	true/false
2	status (мероприятия)	enum	'planned', 'ongoing', 'completed', 'cancelled'
3	status (заказа)	enum	'new', 'processing', 'shipped', 'delivered', 'cancelled'
4	status (статьи)	enum	'draft', 'published', 'archived'
5	category (спортсмена)	enum	'team', 'individual', 'paired'

Пример middleware для валидации (средний уровень):

javascript

```
// Middleware для валидации книги
function validateBook(req, res, next) {
    const { title, author, year, price, email, status } = req.body;
    const errors = [];
```

```
    // Базовая валидация (как в базовом уровне)
    // ...

    // Валидация email
    if (email) {
      const emailRegex = /^[^\s@]+@([^\s@.,]+\.)+[^\s@.,]{2,}$/;
      if (!emailRegex.test(email)) {
        errors.push({ field: 'email', message: 'Некорректный формат email' });
      }
    }

    // Валидация статуса
    const validStatuses = ['draft', 'published', 'archived'];
    if (status && !validStatuses.includes(status)) {
      errors.push({
        field: 'status',
        message: `Статус должен быть одним из: ${validStatuses.join(', ')}`,
        code: 'invalid_enum'
      });
    }

    if (errors.length > 0) {
      return res.status(422).json({ errors });
    }

    next();
  }
```

```
app.post('/api/books', validateBook, (req, res) => {
    // Создание книги после успешной валидации
    res.status(201).json({ message: 'Книга создана' });
});
```

2.2.3. Продвинутый уровень (15 вариантов)

Общее задание для продвинутого уровня:

1. Выполнить требования среднего уровня, а также дополнительно реализовать:
2. Использование библиотеки валидации (express-validator или Joi).
3. Кастомные валидаторы для сложной бизнес-логики.
4. Валидацию зависимых полей (например, если is_company = true, то поле company_name обязательно).
5. Санитизацию входных данных (trim, escape, приведение типов).
6. Подробное логирование ошибок с уровнем warn/error.
7. Глобальный обработчик ошибок с разными форматами для dev/prod окружений.

Установка express-validator:

bash

```
npm install express-validator
```

Пример использования express-validator (продвинутый уровень):

javascript

```
const { body, param, query, validationResult, oneOf } = require('express-validator');

// Правила валидации для создания книги
const createBookValidation = [
    body('title')
        .trim()
        .notEmpty().withMessage('Название книги обязательно')
```

```
        .isLength({ min: 1, max: 200 }).withMessage('Название должно быть от 1 до 200 символов')
        .escape(),

    body('author')
        .trim()
        .notEmpty().withMessage('Автор обязателен')
        .isLength({ min: 2, max: 100 }).withMessage('Имя автора от 2 до 100 символов')
        .escape(),

    body('year')
        .optional()
        .isInt({ min: 0, max: 2025 }).withMessage('Год должен быть от 0 до 2025')
        .toInt(),

    body('price')
        .optional()
        .isFloat({ min: 0 }).withMessage('Цена должна быть неотрицательным числом')
        .toFloat(),

    body('email')
        .optional()
        .isEmail().withMessage('Некорректный формат email')
        .normalizeEmail(),

    body('status')
```

```
        .optional()
        .isIn(['draft', 'published', 'archived']).withMessage('Недопустимый статус')
    ];

    // Middleware для обработки результатов валидации
    function handleValidationErrors(req, res, next) {
        const errors = validationResult(req);
        if (!errors.isEmpty()) {
            const formattedErrors = errors.array().map(err => ({
                field: err.path,
                message: err.msg,
                value: err.value
            }));
            return res.status(422).json({ errors: formattedErrors });
        }
        next();
    }

    // Использование в маршруте
    app.post('/api/books', createBookValidation, handleValidationErrors, (req, res) => {
        // После валидации req.body уже содержит обработанные данные (trim, toInt, toFloat)
        const { title, author, year, price, email, status } = req.body;

        // Создание книги...
        res.status(201).json({ message: 'Книга создана' });
    });
```

```
// Валидация параметров пути и query
app.get('/api/books/:id',
  param('id').isInt({ min: 1 }).withMessage('ID должен быть положительным числом'),
  query('page').optional().isInt({ min: 1 }).withMessage('Страница должна быть ≥ 1').toInt(),
  query('limit').optional().isInt({ min: 1, max: 100 }).withMessage('Лимит от 1 до 100').toInt(),
  handleValidationErrors,
  (req, res) => {
    const id = parseInt(req.params.id);
    const page = req.query.page || 1;
    const limit = req.query.limit || 10;
    // Получение книги...
  }
);

// Кастомный валидатор для проверки уникальности
const isUniqueTitle = async (title) => {
  if (!title) return true;
  return new Promise((resolve, reject) => {
    db.get('SELECT * FROM books WHERE title = ?', [title], (err, row) => {
      if (err) reject(err);
      if (row) reject(new Error('Книга с таким названием уже существует'));
      resolve(true);
    });
```

```
    });
};

body('title').custom(isUniqueTitle).withMessage('Книга с таким названием уже существует');

// Кастомный валидатор для зависимых полей
body('is_company').isBoolean(),
body('company_name').custom((value, { req }) => {
  if (req.body.is_company === true && (!value || value.trim().length === 0)) {
    throw new Error('Для юрлица обязательно указать название компании');
  }
  return true;
});
```

3. Контрольные вопросы

1. Какие типы валидации существуют?
2. Какой HTTP-код возвращается при отсутствии обязательного поля?
3. Какой HTTP-код возвращается при дубликате уникального поля?
4. В чём разница между кодами 400 и 422?
5. Как организовать унифицированный формат ошибок?
6. Как реализовать проверку уникальности поля с запросом к БД?
7. Какие существуют библиотеки для валидации в Express?
8. Что такое санитизация данных и зачем она нужна?
9. Как защитить приложение от XSS-атак через валидацию?
10. Как создать кастомный валидатор для проверки формата телефона?
11. Как организовать middleware для валидации?
12. Как обрабатывать ошибки валидации при PATCH-запросах?
13. Что такое глобальный обработчик ошибок и как его создать?

14. Как сделать разные форматы ошибок для development и production?
15. Как валидировать параметры пути (:id) и query-параметры?

4. Требования к отчёту

Отчёт должен быть оформлен в электронном виде и содержать:

Титульный лист с названием работы, ФИО студента, номером группы, вариантом.

Цель и задачи работы (переписать из методических указаний).

Теоретическое обоснование (краткий конспект по теме).

Ход выполнения работы:

Листинг кода с функциями валидации.

Скриншоты проверки работы через Postman для каждого типа ошибок:

Отсутствие обязательного поля → 400/422

Неверный тип данных → 422

Выход за диапазон → 422

Неверный формат (email, телефон) → 422

Дубликат уникального поля → 409 (если реализовано)

Успешный запрос → 200/201

Ответы на контрольные вопросы (письменно, развёрнуто).

Заключение (выводы о проделанной работе).

Список использованных источников.

5. Критерии оценивания

Уровень выполнения	**Требования**	**Оценка**
Базовый	Валидация обязательных полей, длины	Удовлетворительно

Уровень выполнения	Требования	Оценка
	строк, числовых диапазонов. Возврат ошибок 400/422 с унифицированным форматом.	
Средний	Выполнены требования базового уровня + валидация форматов (email, enum), проверка уникальности (409), вынос валидации в middleware.	Хорошо
Продвинутый	Выполнены требования среднего уровня + использование библиотеки (express-validator/Joi), кастомные и зависимые валидаторы, санитизация, логирование, глобальный обработчик ошибок.	Отлично

Дополнительные условия:

1. Код должен быть структурирован и снабжён комментариями.
2. Отчёт должен содержать скриншоты всех сценариев ошибок.
3. При защите студент должен объяснить логику валидации.

6. Список рекомендуемой литературы и интернет-ресурсов

Литература (доступна в РФ)

1. Браун, И. (2021). *Express.js: Разработка веб-приложений на Node.js*. ДМК Пресс.
2. Кантелон, М., Хартер, М., & Холстейн, С. (2020). *Node.js в действии*. Питер.

Интернет-ресурсы

Название	Ссылка	Описание
express-validator	https://express-validator.github.io/	Документация библиотеки
Joi validation	https://joi.dev/	Схемная валидация
HTTP Status Codes	https://developer.mozilla.org/ru/docs/Web/HTTP/Status	Коды HTTP
Обработка ошибок в Express	https://expressjs.com/ru/guide/error-handling.html	Официальная документация
Санитизация данных	https://cheatsheetseries.owasp.org/	OWASP рекомендации
Регулярные выражения	https://regex101.com/	Проверка и тестирование regex

Все ресурсы доступны и легальны на территории Российской Федерации.

Лабораторная работа № 11

Тема: Хеширование паролей и базовая аутентификация (Basic Auth)

Цель работы:

Реализовать регистрацию пользователей с безопасным хешированием паролей и защиту маршрутов API с помощью Basic Authentication.

Задачи работы:

1. Реализовать эндпоинт регистрации пользователя с хешированием пароля.
2. Реализовать эндпоинт входа (проверка пароля).
3. Освоить использование библиотеки bcrypt для хеширования паролей.
4. Реализовать защиту маршрутов с помощью Basic Authentication.
5. Научиться извлекать и проверять учетные данные из заголовка Authorization.
6. Реализовать middleware для аутентификации.

1. Теоретическое обоснование

1.1. Хеширование паролей

Хеширование -это преобразование исходных данных (пароля) в строку фиксированной длины (хеш), которое является необратимым. Это означает, что из хеша невозможно восстановить исходный пароль.

Зачем нужно хеширование?

При утечке базы данных злоумышленник получит хеши, а не реальные пароли.

Даже администратор базы данных не может узнать пароль пользователя.

1.2. Что такое соль (salt)

Соль - это случайная строка, добавляемая к паролю перед хешированием. Она делает одинаковые пароли разных пользователей уникальными в хешированном виде и защищает от атак по радужным таблицам.

Без соли: hash("password123") → всегда одинаковый хеш

Со солью: hash("password123" + "случайная_строка_1") → уникальный хеш

1.3. bcrypt

bcrypt - это современный алгоритм хеширования паролей, который:

Автоматически генерирует и добавляет соль.

Имеет параметр "стоимость работы" (cost factor), замедляющий подбор паролей.

Является стандартом де-факто для хеширования паролей в веб-приложениях.

Установка:

bash

```
npm install bcrypt
```

Использование:

javascript

```
const bcrypt = require('bcrypt');
const saltRounds = 10;  // стоимость работы (чем выше, тем медленнее, но безопаснее)

// Хеширование пароля
const hash = await bcrypt.hash(password, saltRounds);
```

```
// Проверка пароля
const isMatch = await bcrypt.compare(password, hash);  // true/false
```

1.4. Basic Authentication

Basic Authentication - это простейший метод аутентификации, встроенный в протокол HTTP.

Принцип работы:

Клиент отправляет заголовок: Authorization: Basic <credentials>

<credentials> - это username:password, закодированные в Base64

Сервер декодирует и проверяет учетные данные

Пример заголовка:

Authorization: Basic dXNlcm5hbWU6cGFzc3dvcmQ=

(где dXNlcm5hbWU6cGFzc3dvcmQ= -это base64 от username:password)

Важно: Basic Auth следует использовать только с HTTPS, так как пароль передается в открытом виде (хоть и в Base64, но это не шифрование).

1.5. Схема базы данных для пользователей

sql

```
CREATE TABLE IF NOT EXISTS users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    username TEXT NOT NULL UNIQUE,
    email TEXT NOT NULL UNIQUE,
    password_hash TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    role TEXT DEFAULT 'user'  -- 'user' или 'admin'
);
```

1.6. Middleware для аутентификации

javascript

```
function authenticate(req, res, next) {
    const authHeader = req.headers.authorization;

    if (!authHeader || !authHeader.startsWith('Basic ')) {
        res.setHeader('WWW-Authenticate', 'Basic');
        return res.status(401).json({ error: 'Требуется аутентификация' });
    }

    const base64Credentials = authHeader.split(' ')[1];
    const credentials = Buffer.from(base64Credentials, 'base64').toString('utf-8');
    const [username, password] = credentials.split(':');

    // Поиск пользователя и проверка пароля
    db.get('SELECT * FROM users WHERE username = ?', [username], async (err, user) => {
        if (err || !user) {
            return res.status(401).json({ error: 'Неверное имя пользователя или пароль' });
        }

        const isValid = await bcrypt.compare(password, user.password_hash);
        if (!isValid) {
            return res.status(401).json({ error: 'Неверное имя пользователя или пароль' });
        }

        req.user = user;  // Сохраняем пользователя в объекте запроса
        next();
```

```
    });
}

// Использование middleware
app.get('/api/profile', authenticate, (req, res) => {
    res.json({ user: req.user });
});
```

2. Методика выполнения работы

2.1. Примеры и их подробный разбор

Пример 1. Установка и настройка bcrypt

bash

```
npm install bcrypt
```

Пример 2. Регистрация пользователя

javascript

```
const express = require('express');
const bcrypt = require('bcrypt');
const sqlite3 = require('sqlite3').verbose();
const app = express();

app.use(express.json());

const db = new sqlite3.Database('./users.db');

// Создание таблицы пользователей
db.run(`
    CREATE TABLE IF NOT EXISTS users (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        username TEXT NOT NULL UNIQUE,
```

```
            email TEXT NOT NULL UNIQUE,
            password_hash TEXT NOT NULL,
            created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
        )
    `);

    // POST /api/register -регистрация
    app.post('/api/register', async (req, res) => {
        const { username, email, password } = req.body;

        // Базовая валидация
        if (!username || !email || !password) {
            return res.status(400).json({ error: 'Все поля обязательны' });
        }

        if (password.length < 6) {
            return res.status(400).json({ error: 'Пароль должен содержать минимум 6 символов' });
        }

        try {
            // Хеширование пароля
            const saltRounds = 10;
            const passwordHash = await bcrypt.hash(password, saltRounds);

            // Сохранение пользователя в БД
            db.run(
                'INSERT INTO users (username, email, password_hash) VALUES (?, ?, ?)',
```

```
            [username, email, passwordHash],
            function(err) {
                if (err) {
                    if (err.message.includes('UNIQUE')) {
                        return res.status(409).json({ error: 'Пользователь с таким именем или email уже существует' });
                    }
                    return res.status(500).json({ error: 'Ошибка сервера' });
                }

                res.status(201).json({
                    message: 'Пользователь успешно зарегистрирован',
                    userId: this.lastID
                });
            }
        );
    } catch (error) {
        res.status(500).json({ error: 'Ошибка при хешировании пароля' });
    }
});
```

Пример 3. Middleware для Basic Authentication

javascript

```
// Middleware для проверки Basic Auth
async function authenticate(req, res, next) {
    const authHeader = req.headers.authorization;

    // Проверка наличия заголовка
    if (!authHeader) {
        res.setHeader('WWW-Authenticate', 'Basic');
```

```
            return res.status(401).json({ error: 'Требуется аутентификация' });
        }

        // Проверка формата заголовка
        if (!authHeader.startsWith('Basic ')) {
            return res.status(401).json({ error: 'Неверный формат аутентификации' });
        }

        try {
            // Декодирование Base64
            const base64Credentials = authHeader.split(' ')[1];
            const credentials = Buffer.from(base64Credentials, 'base64').toString('utf-8');
            const [username, password] = credentials.split(':');

            if (!username || !password) {
                return res.status(401).json({ error: 'Неверные учетные данные' });
            }

            // Поиск пользователя в БД
            db.get('SELECT * FROM users WHERE username = ?', [username], async (err, user) => {
                if (err) {
                    return res.status(500).json({ error: 'Ошибка сервера' });
                }

                if (!user) {
```

```
                return res.status(401).json({ error: 'Неверное имя пользователя или пароль' });
            }

            // Проверка пароля
            const isValid = await bcrypt.compare(password, user.password_hash);
            if (!isValid) {
                return res.status(401).json({ error: 'Неверное имя пользователя или пароль' });
            }

            // Сохраняем пользователя в объекте запроса
            req.user = {
                id: user.id,
                username: user.username,
                email: user.email,
                role: user.role
            };

            next();
        });
    } catch (error) {
        res.status(500).json({ error: 'Ошибка при аутентификации' });
    }
};
```

Пример 4. Защищённые маршруты

javascript

```
// GET /api/profile -получение профиля текущего пользователя
```

```
app.get('/api/profile', authenticate, (req, res) => {
    res.json({
        user: req.user,
        message: 'Доступ к защищённому ресурсу разрешён'
    });
});

// GET /api/users -список пользователей (только для админов)
app.get('/api/users', authenticate, (req, res) => {
    // Проверка роли администратора
    if (req.user.role !== 'admin') {
        return res.status(403).json({ error: 'Доступ запрещён. Требуются права администратора' });
    }

    db.all('SELECT id, username, email, created_at FROM users', [], (err, users) => {
        if (err) {
            return res.status(500).json({ error: 'Ошибка сервера' });
        }
        res.json(users);
    });
});
```

Пример 5. POST /api/login (выдача токена -подготовка к JWT)

javascript

```
app.post('/api/login', async (req, res) => {
    const { username, password } = req.body;

    if (!username || !password) {
```

```
            return res.status(400).json({ error: 'Имя пользователя и пароль обязательны' });
        }

        db.get('SELECT * FROM users WHERE username = ?', [username], async (err, user) => {
            if (err) {
                return res.status(500).json({ error: 'Ошибка сервера' });
            }

            if (!user) {
                return res.status(401).json({ error: 'Неверное имя пользователя или пароль' });
            }

            const isValid = await bcrypt.compare(password, user.password_hash);
            if (!isValid) {
                return res.status(401).json({ error: 'Неверное имя пользователя или пароль' });
            }

            // В реальном приложении здесь возвращается JWT-токен
            res.json({
                message: 'Успешный вход',
                user: {
                    id: user.id,
                    username: user.username,
                    email: user.email,
                    role: user.role
```

```
        }
    });
});
});
```

Пример 6. Полный пример сервера с аутентификацией

```javascript
const express = require('express');
const bcrypt = require('bcrypt');
const sqlite3 = require('sqlite3').verbose();

const app = express();
app.use(express.json());

const db = new sqlite3.Database('./users.db');

// Создание таблицы пользователей
db.run(`
    CREATE TABLE IF NOT EXISTS users (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        username TEXT NOT NULL UNIQUE,
        email TEXT NOT NULL UNIQUE,
        password_hash TEXT NOT NULL,
        role TEXT DEFAULT 'user',
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    )
`);

// Добавление тестового администратора (если нет пользователей)
db.get('SELECT COUNT(*) as count FROM users', [], (err, row) => {
```

```
        if (row.count === 0) {
            bcrypt.hash('admin123', 10).then(hash => {
                db.run('INSERT INTO users (username, email, password_hash, role)
VALUES (?, ?, ?, ?)',
                    ['admin', 'admin@example.com', hash, 'admin']);
            });
        }
    });

    // ========== АУТЕНТИФИКАЦИЯ (MIDDLEWARE) ==========
    async function authenticate(req, res, next) {
        const authHeader = req.headers.authorization;

        if (!authHeader || !authHeader.startsWith('Basic ')) {
            res.setHeader('WWW-Authenticate', 'Basic');
            return res.status(401).json({ error: 'Требуется аутентификация' });
        }

        try {
            const base64Credentials = authHeader.split(' ')[1];
            const credentials = Buffer.from(base64Credentials,
'base64').toString('utf-8');
            const [username, password] = credentials.split(':');

            if (!username || !password) {
                return res.status(401).json({ error: 'Неверные учетные данные' });
            }
```

```
            db.get('SELECT * FROM users WHERE username = ?', [username],
async (err, user) => {
                if (err || !user) {
                    return res.status(401).json({ error: 'Неверное имя пользователя
или пароль' });
                }

                const isValid = await bcrypt.compare(password,
user.password_hash);
                if (!isValid) {
                    return res.status(401).json({ error: 'Неверное имя пользователя
или пароль' });
                }

                req.user = { id: user.id, username: user.username, email: user.email,
role: user.role };
                next();
            });
        } catch (error) {
            res.status(500).json({ error: 'Ошибка при аутентификации' });
        }
    }

    // ========== ПУБЛИЧНЫЕ МАРШРУТЫ ==========
    app.post('/api/register', async (req, res) => {
        const { username, email, password } = req.body;

        if (!username || !email || !password) {
            return res.status(400).json({ error: 'Все поля обязательны' });
```

```
    }

    if (password.length < 6) {
      return res.status(400).json({ error: 'Пароль должен содержать минимум 6 символов' });
    }

    try {
      const saltRounds = 10;
      const passwordHash = await bcrypt.hash(password, saltRounds);

      db.run(
        'INSERT INTO users (username, email, password_hash) VALUES (?, ?, ?)',
        [username, email, passwordHash],
        function(err) {
          if (err) {
            if (err.message.includes('UNIQUE')) {
              return res.status(409).json({ error: 'Пользователь с таким именем или email уже существует' });
            }
            return res.status(500).json({ error: 'Ошибка сервера' });
          }
          res.status(201).json({ message: 'Регистрация успешна', userId: this.lastID });
        }
      );
    } catch (error) {
      res.status(500).json({ error: 'Ошибка сервера' });
```

```
    }
  });

  app.post('/api/login', async (req, res) => {
    const { username, password } = req.body;

    if (!username || !password) {
      return res.status(400).json({ error: 'Имя пользователя и пароль обязательны' });
    }

    db.get('SELECT * FROM users WHERE username = ?', [username], async (err, user) => {
      if (err || !user) {
        return res.status(401).json({ error: 'Неверные учетные данные' });
      }

      const isValid = await bcrypt.compare(password, user.password_hash);
      if (!isValid) {
        return res.status(401).json({ error: 'Неверные учетные данные' });
      }

      res.json({
        message: 'Вход выполнен',
        user: { id: user.id, username: user.username, email: user.email, role: user.role }
      });
    });
  });
```

```
// ========== ЗАЩИЩЁННЫЕ МАРШРУТЫ ==========
app.get('/api/profile', authenticate, (req, res) => {
  res.json({ user: req.user });
});

// Административный маршрут
app.get('/api/admin/users', authenticate, (req, res) => {
  if (req.user.role !== 'admin') {
    return res.status(403).json({ error: 'Доступ запрещён. Требуются права администратора' });
  }

  db.all('SELECT id, username, email, role, created_at FROM users', [], (err, users) => {
    if (err) return res.status(500).json({ error: 'Ошибка сервера' });
    res.json(users);
  });
});

// Защищённый CRUD для товаров (пример)
app.get('/api/products', authenticate, (req, res) => {
  db.all('SELECT * FROM products', [], (err, products) => {
    if (err) return res.status(500).json({ error: 'Ошибка сервера' });
    res.json(products);
  });
});

app.post('/api/products', authenticate, (req, res) => {
```

```
    if (req.user.role !== 'admin') {
        return res.status(403).json({ error: 'Только администратор может добавлять товары' });
    }

    const { name, price } = req.body;
    if (!name || !price) {
        return res.status(400).json({ error: 'Название и цена обязательны' });
    }

    db.run('INSERT INTO products (name, price) VALUES (?, ?)', [name, price], function(err) {
        if (err) return res.status(500).json({ error: 'Ошибка сервера' });
        res.status(201).json({ id: this.lastID, name, price });
    });
});

const PORT = 3000;
app.listen(PORT, () => {
    console.log(`Сервер запущен на http://localhost:${PORT}`);
});
```

2.2. Индивидуальные задания

2.2.1. Базовый уровень (15 вариантов)

Общее задание для базового уровня:

Реализовать сервер с аутентификацией, включающую:

1. Регистрацию пользователя (POST /api/register) с хешированием пароля (bcrypt).

2. Эндпоинт входа (POST /api/login) -проверка пароля и возврат информации о пользователе.
3. Один защищённый маршрут (GET /api/profile), возвращающий информацию о текущем пользователе.
4. Базовая валидация при регистрации (обязательность полей, длина пароля ≥ 6).

Варианты расширения (базовый уровень):

	Дополнительное поле при регистрации	Специфичный эндпоинт
	full_name	/api/profile → + full_name
	phone	/api/profile → + phone
	age	/api/profile → + age
	city	/api/profile → + city
	birth_date	/api/profile → + birth_date
	position	/api/profile → + position
	department	/api/profile → + department
	student_id	/api/profile → + student_id
	group_name	/api/profile → + group_name
0	course	/api/profile → + course
	bio	/api/profile → + bio

	Дополнительное поле при регистрации	Специфичный эндпоинт
1		
2	website	/api/profile → + website
3	company	/api/profile → + company
4	github_username	/api/profile → + github_username
5	telegram_username	/api/profile → + telegram_username

Пример выполнения для варианта 1 (Базовый уровень):

javascript

```
// Добавляем поле full_name в таблицу
db run(`
  CREATE TABLE IF NOT EXISTS users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    username TEXT NOT NULL UNIQUE,
    email TEXT NOT NULL UNIQUE,
    full_name TEXT,
    password_hash TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
  )
`);
```

```
// Регистрация с full_name
app.post('/api/register', async (req, res) => {
  const { username, email, full_name, password } = req.body;

  if (!username || !email || !password) {
    return res.status(400).json({ error: 'Все поля обязательны' });
  }

  const passwordHash = await bcrypt.hash(password, 10);

  db.run(
    'INSERT INTO users (username, email, full_name, password_hash) VALUES (?, ?, ?, ?)',
    [username, email, full_name, passwordHash],
    function(err) {
      if (err) return res.status(500).json({ error: err.message });
      res.status(201).json({ message: 'Регистрация успешна' });
    }
  );
});

// GET /api/profile с полным именем
app.get('/api/profile', authenticate, (req, res) => {
  res.json({
    id: req.user.id,
    username: req.user.username,
    email: req.user.email,
    full_name: req.user.full_name
  });
```

});

2.2.2. Средний уровень (15 вариантов)

Общее задание для среднего уровня:

Выполнить требования базового уровня, а также дополнительно реализовать:

Ролевую модель (user, admin) с соответствующими правами.

1. Защищённый маршрут GET /api/admin/users (только для администраторов).
2. Защищённый CRUD для основной сущности (по варианту):
3. GET /api/items (доступ авторизованным)
4. POST /api/items (только admin)
5. PUT /api/items/:id (только admin)
6. DELETE /api/items/:id (только admin)
7. Логирование попыток входа (успешных и неудачных).

Варианты сущностей для CRUD (средний уровень):

	Сущность	Поля
	Книги (books)	title, author, year, price
	Товары (products)	name, price, stock, category
	Фильмы (movies)	title, director, year, rating
	Студенты (students)	full_name, group, age, email
	Сотрудники (employees)	name, position, salary, department

	Сущность	Поля
	Рецепты (recipes)	title, cuisine, cooking_time, difficulty
	Автомобили (cars)	model, brand, year, price
	Песни (songs)	title, artist, duration, genre
	Отели (hotels)	name, city, stars, price
0	Курсы (courses)	title, instructor, duration, price
1	Лекарства (medicines)	name, manufacturer, price, prescription
2	Мероприятия (events)	name, location, date, price
3	Заказы (orders)	customer_name, product, amount, status
4	Статьи (articles)	title, author, content, views
5	Спортсмены (athletes)	name, sport, country, medals

Пример middleware для проверки роли администратора:

javascript

```
function requireAdmin(req, res, next) {
  if (req.user.role !== 'admin') {
```

```
        return res.status(403).json({ error: 'Доступ запрещён. Требуются права администратора' });
    }
    next();
}

// Использование
app.post('/api/books', authenticate, requireAdmin, (req, res) => {
    // Только администратор может добавить книгу
});
```

2.2.3. Продвинутый уровень (15 вариантов)

Общее задание для продвинутого уровня:

Выполнить требования среднего уровня, а также дополнительно реализовать:

Rate limiting -защита от брутфорса (не более 5 неудачных попыток входа с одного IP за 15 минут).

Хранение соли отдельно или использование bcrypt с солью.

Middleware для логирования всех аутентифицированных запросов.

Эндпоинт для смены пароля (POST /api/change-password) с подтверждением старого пароля.

Восстановление пароля (генерация временного токена -упрощённо, без email).

Обновление профиля (PUT /api/profile) с валидацией.

Переменные окружения для saltRounds и других настроек.

Дополнительная функциональность по вариантам:

№	Дополнительная функция
1	Двухфакторная аутентификация (упрощённо: секретный вопрос)
2	Блокировка аккаунта после 5 неудачных попыток
3	Ограничение сессии (только одно активное устройство)
4	Логирование всех действий администратора
5	Политика паролей (заглавные, цифры, спецсимволы)
6	Истечение срока действия пароля (90 дней)
7	История паролей (нельзя использовать последние 3 пароля)
8	Captcha после 3 неудачных попыток
9	Email-подтверждение (симуляция, генерация ссылки)
10	Ограничение по времени сессии (30 минут неактивности)
11	Генерация одноразового пароля
12	Вход по email вместо username
13	Регистрация с приглашением от администратора
14	Аудит действий пользователя
15	Soft delete пользователя (флаг is_active)

Пример rate limiting и смены пароля:

```
javascript
const rateLimit = require('express-rate-limit');

// Rate limiting для входа
const loginLimiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 минут
  max: 5, // максимум 5 попыток
  message: { error: 'Слишком много попыток входа. Попробуйте позже' }
});

app.post('/api/login', loginLimiter, async (req, res) => {
  // Логика входа
});

// Смена пароля
app.post('/api/change-password', authenticate, async (req, res) => {
  const { oldPassword, newPassword } = req.body;

  if (!oldPassword || !newPassword) {
    return res.status(400).json({ error: 'Все поля обязательны' });
  }

  if (newPassword.length < 6) {
    return res.status(400).json({ error: 'Новый пароль должен содержать минимум 6 символов' });
  }

  // Проверка старого пароля
```

```
        db.get('SELECT password_hash FROM users WHERE id = ?',
[req.user.id], async (err, user) => {
            if (err) return res.status(500).json({ error: 'Ошибка сервера' });

            const isValid = await bcrypt.compare(oldPassword,
user.password_hash);
            if (!isValid) {
                return res.status(401).json({ error: 'Неверный старый пароль' });
            }

            // Хеширование нового пароля
            const saltRounds = parseInt(process.env.SALT_ROUNDS) || 10;
            const newHash = await bcrypt.hash(newPassword, saltRounds);

            db.run('UPDATE users SET password_hash = ? WHERE id = ?',
[newHash, req.user.id], (err) => {
                if (err) return res.status(500).json({ error: 'Ошибка сервера' });
                res.json({ message: 'Пароль успешно изменён' });
            });
        });
    });
```

3. Контрольные вопросы

1. Зачем нужно хешировать пароли вместо их хранения в открытом виде?
2. Что такое соль (salt) и зачем она нужна?
3. Какие преимущества у bcrypt перед простыми хеш-функциями (MD5, SHA)?
4. Как работает Basic Authentication?

5. В каком формате передаются учетные данные в Basic Auth?
6. Почему Basic Auth следует использовать только с HTTPS?
7. Как в Express получить заголовок Authorization?
8. Как декодировать Base64 в Node.js?
9. Как реализовать middleware для аутентификации?
10. Как хранить роли пользователей и проверять доступ?
11. Что такое rate limiting и зачем он нужен?
12. Как защититься от брутфорс-атак на эндпоинт входа?
13. Как реализовать смену пароля с проверкой старого?
14. Что такое «история паролей» и как её реализовать?
15. Как логировать попытки входа для аудита?

4. Требования к отчёту

Отчёт должен быть оформлен в электронном виде и содержать:

Титульный лист с названием работы, ФИО студента, номером группы, вариантом.

Цель и задачи работы (переписать из методических указаний).

Теоретическое обоснование (краткий конспект по теме).

Ход выполнения работы:

SQL-скрипты создания таблицы пользователей.

Листинг кода сервера с комментариями.

Скриншоты проверки работы через Postman:

Регистрация пользователя (201)

Попытка регистрации с существующим username (409)

Вход с верными данными (200)

Вход с неверными данными (401)

Доступ к защищённому маршруту без заголовка (401)

Доступ к защищённому маршруту с верными данными (200)

(Для среднего уровня) Доступ администратора к /api/admin/users

(Для продвинутого) Смена пароля, rate limiting

Ответы на контрольные вопросы (письменно, развёрнуто).

Заключение (выводы).

Список использованных источников.

5. Критерии оценивания

Уровень выполнения	Требования	Оценка
Базовый	Регистрация с хешированием пароля (bcrypt), эндпоинт входа, один защищённый маршрут (profile).	Удовлетворительно
Средний	Выполнены требования базового уровня + ролевая модель (user/admin), защищённый CRUD, маршрут /api/admin/users, логирование попыток входа.	Хорошо
Продвинутый	Выполнены требования среднего уровня + rate limiting, смена пароля, обновление профиля, восстановление	Отлично

Уровень выполнения	Требования	Оценка
	пароля (упрощённо), переменные окружения.	

Дополнительные условия:

1. Пароли должны хешироваться с использованием bcrypt.
2. При защите студент должен объяснить принципы хеширования и аутентификации.

6. Список рекомендуемой литературы и интернет-ресурсов

Литература (доступна в РФ)

1. Браун, И. (2021). *Express.js: Разработка веб-приложений на Node.js*. ДМК Пресс.
2. Кантелон, М., Хартер, М., & Холстейн, С. (2020). *Node.js в действии*. Питер.

Интернет-ресурсы

Название	Ссылка	Описание
bcrypt npm	https://www.npmjs.com/package/bcrypt	Документация bcrypt
Basic Authentication	https://developer.mozilla.org/ru/docs/Web/HTTP/Authentication	MDN о Basic Auth

Название	Ссылка	Описание
OWASP Password Storage	https://cheatsheetseries.owasp.org/	Рекомендации по хранению паролей
express-rate-limit	https://www.npmjs.com/package/express-rate-limit	Документация rate limiting
Base64 в Node.js	https://nodejs.org/api/buffer.html	Документация Buffer

Все ресурсы доступны и легальны на территории Российской Федерации.

Лабораторная работа № 12

Тема: Сессионная аутентификация (в памяти сервера)

Цель работы:

Реализовать механизм сессий для отслеживания авторизованных пользователей с хранением сессий в памяти сервера, без использования сторонних библиотек.

Задачи работы:

1. Реализовать генерацию уникального идентификатора сессии (session ID).
2. Организовать хранение сессий в памяти сервера (Map или объект).
3. Реализовать эндпоинт входа (POST /api/login), создающий сессию.
4. Реализовать middleware для проверки наличия активной сессии.
5. Реализовать защищённые маршруты, доступные только при наличии сессии.
6. Реализовать эндпоинт выхода (POST /api/logout), завершающий сессию.
7. Реализовать автоматическое удаление просроченных сессий.

1. Теоретическое обоснование

1.1. Что такое сессия?

Сессия — это механизм сохранения состояния между HTTP-запросами, так как HTTP сам по себе является stateless-протоколом. Сессия позволяет серверу «запоминать» пользователя между разными запросами.

Как это работает:

Пользователь успешно аутентифицируется (вводит логин и пароль).

Сервер создаёт сессию — объект, содержащий информацию о пользователе (например, { userId: 1, role: 'user' }).

Сервер генерирует уникальный идентификатор сессии (session ID) и отправляет его клиенту.

Клиент сохраняет session ID (обычно в cookie или localStorage).

При каждом следующем запросе клиент передаёт session ID серверу.

Сервер находит сессию по ID и понимает, какой пользователь выполняет запрос.

1.2. Способы передачи session ID

Способ	Описание	Преимущества	Недостатки
Cookie	Автоматически отправляется браузером	Простота, автоматическая отправка	Подвержен CSRF-атакам
Authorization header	Передаётся в заголовке X-Session-ID	Более безопасен для API	Требует ручной отправки
URL parameter	В строке запроса ?sessionId=...	Простота отладки	Небезопасно, остаётся в логах

В данной лабораторной работе будем использовать заголовок X-Session-ID, так как это наиболее подходящий способ для API.

1.3. Хранение сессий в памяти

```
javascript
// Хранилище сессий: sessionId → данные сессии
const sessions = new Map();

// Создание сессии
const sessionId = generateSessionId();
sessions.set(sessionId, {
  userId: 1,
  username: 'ivan',
  role: 'user',
```

```
    createdAt: Date.now(),
    expiresAt: Date.now() + 24 * 60 * 60 * 1000  // 24 часа
});

// Получение сессии
const session = sessions.get(sessionId);

// Удаление сессии (выход)
sessions.delete(sessionId);
```

1.4. Генерация session ID

Session ID должен быть:

1. Уникальным — вероятность коллизии близка к нулю.
2. Случайным — нельзя угадать или подобрать.
3. Достаточно длинным — чтобы исключить перебор.

javascript

```
const crypto = require('crypto');

function generateSessionId() {
    return crypto.randomBytes(32).toString('hex');
}
// Пример: 'a7f3e9d2c1b8a4f6e2d9c7b5a3f1e8d4c2b6a9f7e3d1c8b5a2f4e7d9c6b3a1f8'
```

1.5. Очистка просроченных сессий

Сессии должны автоматически удаляться через определённое время неактивности или абсолютное время жизни.

javascript

```
// Удаление просроченных сессий (каждый час)
```

```
setInterval(() => {
    const now = Date.now();
    for (const [sessionId, session] of sessions.entries()) {
        if (session.expiresAt < now) {
            sessions.delete(sessionId);
            console.log(`Сессия ${sessionId} удалена по таймауту`);
        }
    }
}, 60 * 60 * 1000);  // Каждый час
```

1.6. Схема данных сессии

```
javascript
const session = {
    sessionId: '...',          // Уникальный идентификатор
    userId: 1,                 // ID пользователя
    username: 'ivan',          // Имя пользователя
    role: 'user',              // Роль (для авторизации)
    ip: '127.0.0.1',           // IP-адрес (опционально)
    userAgent: '...',          // User-Agent (опционально)
    createdAt: 1698765432100,  // Время создания (timestamp)
    lastAccessAt: 1698765432100,// Последняя активность
    expiresAt: 1698851832100   // Время истечения
};
```

1.7. Cookie vs Header

В данной лабораторной работе используется заголовок X-Session-ID, так как он:

Не требует работы с cookie (которая может быть сложнее).

Легко тестируется через Postman.

Подходит для API-ориентированных приложений.

Клиент отправляет запрос с заголовком:

text

X-Session-ID: a7f3e9d2c1b8a4f6e2d9c7b5a3f1e8d4c2b6a9f7e3d1c8b5a2f4e7d9c6b3a1f8

2. Методика выполнения работы

2.1. Примеры и их подробный разбор

Пример 1. Создание таблицы пользователей

javascript

```
const express = require('express');
const bcrypt = require('bcrypt');
const crypto = require('crypto');
const sqlite3 = require('sqlite3').verbose();

const app = express();
app.use(express.json());

const db = new sqlite3.Database('./users.db');

// Создание таблицы пользователей
db.run(`
  CREATE TABLE IF NOT EXISTS users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    username TEXT NOT NULL UNIQUE,
    email TEXT NOT NULL UNIQUE,
    password_hash TEXT NOT NULL,
    role TEXT DEFAULT 'user',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
```

```
    )
  `);

  // Добавление тестового пользователя
  db.get('SELECT COUNT(*) as count FROM users', [], (err, row) => {
    if (row.count === 0) {
      bcrypt.hash('password123', 10, (err, hash) => {
        db.run('INSERT INTO users (username, email, password_hash) VALUES (?, ?, ?)',
          ['ivan', 'ivan@example.com', hash]);
        db.run('INSERT INTO users (username, email, password_hash, role) VALUES (?, ?, ?, ?)',
          ['admin', 'admin@example.com', hash, 'admin']);
      });
    }
  });
```

Пример 2. Хранилище сессий и генерация session ID

javascript

```
// Хранилище сессий в памяти
const sessions = new Map();

// Генерация уникального session ID
function generateSessionId() {
  return crypto.randomBytes(32).toString('hex');
}

// Создание новой сессии
function createSession(user) {
  const sessionId = generateSessionId();
```

```
    const now = Date.now();
    const SESSION_TTL = 24 * 60 * 60 * 1000; // 24 часа

    const session = {
        sessionId: sessionId,
        userId: user.id,
        username: user.username,
        role: user.role,
        createdAt: now,
        lastAccessAt: now,
        expiresAt: now + SESSION_TTL
    };

    sessions.set(sessionId, session);
    return sessionId;
}

// Получение сессии по ID
function getSession(sessionId) {
    const session = sessions.get(sessionId);
    if (!session) return null;

    // Проверка истечения срока
    if (session.expiresAt < Date.now()) {
        sessions.delete(sessionId);
        return null;
    }

    // Обновление времени последнего доступа
```

```
    session.lastAccessAt = Date.now();
    return session;
}

// Удаление сессии
function destroySession(sessionId) {
    sessions.delete(sessionId);
}

// Очистка просроченных сессий (каждый час)
setInterval(() => {
    const now = Date.now();
    let deletedCount = 0;
    for (const [sessionId, session] of sessions.entries()) {
        if (session.expiresAt < now) {
            sessions.delete(sessionId);
            deletedCount++;
        }
    }
    if (deletedCount > 0) {
        console.log(`Очищено ${deletedCount} просроченных сессий`);
    }
}, 60 * 60 * 1000);
```

Пример 3. Middleware для проверки сессии

javascript

```
// Middleware для аутентификации через сессию
function authenticateSession(req, res, next) {
    const sessionId = req.headers['x-session-id'];
```

```
    if (!sessionId) {
        return res.status(401).json({ error: 'Требуется авторизация' });
    }

    const session = getSession(sessionId);
    if (!session) {
        return res.status(401).json({ error: 'Сессия недействительна или истекла' });
    }

    // Сохраняем информацию о сессии и пользователе в объекте запроса
    req.session = session;
    req.user = {
        id: session.userId,
        username: session.username,
        role: session.role
    };

    next();
}

// Middleware для проверки роли администратора
function requireAdmin(req, res, next) {
    if (req.user.role !== 'admin') {
        return res.status(403).json({ error: 'Доступ запрещён. Требуются права администратора' });
    }
    next();
}
```

Пример 4. Эндпоинты входа и выхода

javascript

```
// POST /api/login — вход (создание сессии)
app.post('/api/login', async (req, res) => {
    const { username, password } = req.body;

    if (!username || !password) {
        return res.status(400).json({ error: 'Имя пользователя и пароль обязательны' });
    }

    // Поиск пользователя в БД
    db.get('SELECT * FROM users WHERE username = ?', [username], async (err, user) => {
        if (err) {
            return res.status(500).json({ error: 'Ошибка сервера' });
        }

        if (!user) {
            return res.status(401).json({ error: 'Неверное имя пользователя или пароль' });
        }

        // Проверка пароля
        const isValid = await bcrypt.compare(password, user.password_hash);
        if (!isValid) {
            return res.status(401).json({ error: 'Неверное имя пользователя или пароль' });
        }
```

```
        // Создание сессии
        const sessionId = createSession(user);

        res.json({
            message: 'Вход выполнен успешно',
            sessionId: sessionId,
            user: {
                id: user.id,
                username: user.username,
                email: user.email,
                role: user.role
            }
        });
    });
});

// POST /api/logout — выход (удаление сессии)
app.post('/api/logout', authenticateSession, (req, res) => {
    const sessionId = req.headers['x-session-id'];
    destroySession(sessionId);
    res.json({ message: 'Выход выполнен успешно' });
});
```

Пример 5. Защищённые маршруты

javascript

```
// GET /api/profile — получение профиля текущего пользователя
app.get('/api/profile', authenticateSession, (req, res) => {
    // Получаем полную информацию о пользователе из БД
```

```
        db.get('SELECT id, username, email, role, created_at FROM users WHERE id = ?',
            [req.user.id], (err, user) => {
                if (err) {
                    return res.status(500).json({ error: 'Ошибка сервера' });
                }
                res.json({ user, session: req.session });
            }
        );
    });

    // GET /api/users — список всех пользователей (только для админов)
    app.get('/api/users', authenticateSession, requireAdmin, (req, res) => {
        db.all('SELECT id, username, email, role, created_at FROM users', [], (err, users) => {
            if (err) {
                return res.status(500).json({ error: 'Ошибка сервера' });
            }
            res.json({ users, total: users.length });
        });
    });
```

Пример 6. Полный пример сервера с сессионной аутентификацией

```javascript
require('dotenv').config();
const express = require('express');
const bcrypt = require('bcrypt');
const crypto = require('crypto');
const sqlite3 = require('sqlite3').verbose();
```

```
const app = express();
app.use(express.json());

const db = new sqlite3.Database('./users.db');

// ========== ИНИЦИАЛИЗАЦИЯ БД ==========
db.run(`
    CREATE TABLE IF NOT EXISTS users (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        username TEXT NOT NULL UNIQUE,
        email TEXT NOT NULL UNIQUE,
        password_hash TEXT NOT NULL,
        role TEXT DEFAULT 'user',
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    )
`);

db.run(`
    CREATE TABLE IF NOT EXISTS products (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT NOT NULL,
        price REAL NOT NULL,
        created_by INTEGER,
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
        FOREIGN KEY (created_by) REFERENCES users(id)
    )
`);

// Добавление тестовых пользователей
```

```
    db.get('SELECT COUNT(*) as count FROM users', [], (err, row) => {
        if (row.count === 0) {
            bcrypt.hash('password123', 10, (err, hash) => {
                db.run('INSERT INTO users (username, email, password_hash) VALUES (?, ?, ?)',
                    ['ivan', 'ivan@example.com', hash]);
                db.run('INSERT INTO users (username, email, password_hash, role) VALUES (?, ?, ?, ?)',
                    ['admin', 'admin@example.com', hash, 'admin']);
                console.log('Тестовые пользователи созданы: ivan / password123, admin / password123');
            });
        }
    });

    // ========== ХРАНИЛИЩЕ СЕССИЙ ==========
    const sessions = new Map();

    function generateSessionId() {
        return crypto.randomBytes(32).toString('hex');
    }

    function createSession(user) {
        const sessionId = generateSessionId();
        const now = Date.now();
        const SESSION_TTL = 24 * 60 * 60 * 1000;

        const session = {
            sessionId: sessionId,
```

```
            userId: user.id,
            username: user.username,
            role: user.role,
            createdAt: now,
            lastAccessAt: now,
            expiresAt: now + SESSION_TTL
        };

        sessions.set(sessionId, session);
        console.log(`Сессия создана для пользователя ${user.username}: ${sessionId.substring(0, 16)}...`);
        return sessionId;
    }

    function getSession(sessionId) {
        const session = sessions.get(sessionId);
        if (!session) return null;

        if (session.expiresAt < Date.now()) {
            sessions.delete(sessionId);
            return null;
        }

        session.lastAccessAt = Date.now();
        return session;
    }

    function destroySession(sessionId) {
        const session = sessions.get(sessionId);
```

```
        if (session) {
            console.log(`Сессия удалена для пользователя ${session.username}`);
            sessions.delete(sessionId);
        }
    }

    // Очистка просроченных сессий
    setInterval(() => {
        const now = Date.now();
        let deletedCount = 0;
        for (const [sessionId, session] of sessions.entries()) {
            if (session.expiresAt < now) {
                sessions.delete(sessionId);
                deletedCount++;
            }
        }
        if (deletedCount > 0) {
            console.log(`Очищено ${deletedCount} просроченных сессий. Активных: ${sessions.size}`);
        }
    }, 60 * 60 * 1000);

    // ========== MIDDLEWARE ==========
    function authenticateSession(req, res, next) {
        const sessionId = req.headers['x-session-id'];

        if (!sessionId) {
```

```
        return res.status(401).json({ error: 'Требуется авторизация. Укажите заголовок X-Session-ID' });
    }

    const session = getSession(sessionId);
    if (!session) {
        return res.status(401).json({ error: 'Сессия недействительна или истекла' });
    }

    req.session = session;
    req.user = {
        id: session.userId,
        username: session.username,
        role: session.role
    };

    next();
}

function requireAdmin(req, res, next) {
    if (req.user.role !== 'admin') {
        return res.status(403).json({ error: 'Доступ запрещён. Требуются права администратора' });
    }
    next();
}

// ========== ПУБЛИЧНЫЕ ЭНДПОИНТЫ ==========
```

```
app.post('/api/register', async (req, res) => {
    const { username, email, password } = req.body;

    if (!username || !email || !password) {
        return res.status(400).json({ error: 'Все поля обязательны' });
    }

    if (password.length < 6) {
        return res.status(400).json({ error: 'Пароль должен содержать минимум 6 символов' });
    }

    try {
        const saltRounds = 10;
        const passwordHash = await bcrypt.hash(password, saltRounds);

        db.run(
            'INSERT INTO users (username, email, password_hash) VALUES (?, ?, ?)',
            [username, email, passwordHash],
            function(err) {
                if (err) {
                    if (err.message.includes('UNIQUE')) {
                        return res.status(409).json({ error: 'Пользователь с таким именем или email уже существует' });
                    }
                    return res.status(500).json({ error: 'Ошибка сервера' });
                }
```

```
                res.status(201).json({ message: 'Регистрация успешна', userId: this.lastID });
            }
        );
    } catch (error) {
        res.status(500).json({ error: 'Ошибка сервера' });
    }
});

app.post('/api/login', async (req, res) => {
    const { username, password } = req.body;

    if (!username || !password) {
        return res.status(400).json({ error: 'Имя пользователя и пароль обязательны' });
    }

    db.get('SELECT * FROM users WHERE username = ?', [username], async (err, user) => {
        if (err) {
            return res.status(500).json({ error: 'Ошибка сервера' });
        }

        if (!user) {
            return res.status(401).json({ error: 'Неверное имя пользователя или пароль' });
        }

        const isValid = await bcrypt.compare(password, user.password_hash);
```

```
            if (!isValid) {
                return res.status(401).json({ error: 'Неверное имя пользователя или пароль' });
            }

            const sessionId = createSession(user);

            res.json({
                message: 'Вход выполнен успешно',
                sessionId: sessionId,
                user: {
                    id: user.id,
                    username: user.username,
                    email: user.email,
                    role: user.role
                }
            });
        });
    });

    app.post('/api/logout', authenticateSession, (req, res) => {
        const sessionId = req.headers['x-session-id'];
        destroySession(sessionId);
        res.json({ message: 'Выход выполнен успешно' });
    });

    // ========== ЗАЩИЩЁННЫЕ ЭНДПОИНТЫ ==========
    app.get('/api/profile', authenticateSession, (req, res) => {
```

```
        db.get('SELECT id, username, email, role, created_at FROM users WHERE id = ?',
            [req.user.id], (err, user) => {
                if (err) {
                    return res.status(500).json({ error: 'Ошибка сервера' });
                }
                res.json({
                    user: user,
                    session: {
                        createdAt: req.session.createdAt,
                        lastAccessAt: req.session.lastAccessAt,
                        expiresAt: req.session.expiresAt
                    }
                });
            }
        );
    });

    // CRUD для товаров (пример защищённого ресурса)
    app.get('/api/products', authenticateSession, (req, res) => {
        db.all(`
            SELECT p.*, u.username as created_by_name
            FROM products p
            LEFT JOIN users u ON p.created_by = u.id
            ORDER BY p.created_at DESC
        `, [], (err, products) => {
            if (err) return res.status(500).json({ error: 'Ошибка сервера' });
            res.json({ products, count: products.length });
        });
```

```
});

app.post('/api/products', authenticateSession, (req, res) => {
  const { name, price } = req.body;

  if (!name || price === undefined) {
    return res.status(400).json({ error: 'Название и цена обязательны' });
  }

  if (typeof price !== 'number' || price < 0) {
    return res.status(400).json({ error: 'Цена должна быть неотрицательным числом' });
  }

  db.run(
    'INSERT INTO products (name, price, created_by) VALUES (?, ?, ?)',
    [name, price, req.user.id],
    function(err) {
      if (err) return res.status(500).json({ error: 'Ошибка сервера' });
      res.status(201).json({ id: this.lastID, name, price, created_by: req.user.id });
    }
  );
});

app.put('/api/products/:id', authenticateSession, (req, res) => {
  const id = req.params.id;
  const { name, price } = req.body;
```

```
    db.get('SELECT * FROM products WHERE id = ?', [id], (err, product) =>
{
      if (err) return res.status(500).json({ error: 'Ошибка сервера' });
      if (!product) return res.status(404).json({ error: 'Товар не найден' });

      // Проверка прав: только создатель товара или админ
      if (product.created_by !== req.user.id && req.user.role !== 'admin') {
        return res.status(403).json({ error: 'Вы можете редактировать
только свои товары' });
      }

      db.run(
        'UPDATE products SET name = ?, price = ? WHERE id = ?',
        [name || product.name, price !== undefined ? price : product.price,
id],
        function(err) {
          if (err) return res.status(500).json({ error: 'Ошибка сервера' });
          res.json({ message: 'Товар обновлён' });
        }
      );
    });
  });

  app.delete('/api/products/:id', authenticateSession, (req, res) => {
    const id = req.params.id;

    db.get('SELECT * FROM products WHERE id = ?', [id], (err, product) =>
{
      if (err) return res.status(500).json({ error: 'Ошибка сервера' });
```

```
            if (!product) return res.status(404).json({ error: 'Товар не найден' });

            // Проверка прав: только создатель товара или админ
            if (product.created_by !== req.user.id && req.user.role !== 'admin') {
                return res.status(403).json({ error: 'Вы можете удалять только свои товары' });
            }

            db.run('DELETE FROM products WHERE id = ?', [id], function(err) {
                if (err) return res.status(500).json({ error: 'Ошибка сервера' });
                res.status(204).send();
            });
        });
    });

    // Админские эндпоинты
    app.get('/api/admin/users', authenticateSession, requireAdmin, (req, res) => {
        db.all(`
            SELECT id, username, email, role, created_at,
                   (SELECT COUNT(*) FROM products WHERE created_by = users.id) as products_count
            FROM users
        `, [], (err, users) => {
            if (err) return res.status(500).json({ error: 'Ошибка сервера' });
            res.json({ users, total: users.length });
        });
    });
```

```
app.get('/api/admin/sessions', authenticateSession, requireAdmin, (req, res) => {
    const activeSessions = Array.from(sessions.values()).map(s => ({
        sessionId: s.sessionId.substring(0, 16) + '...',
        username: s.username,
        role: s.role,
        createdAt: s.createdAt,
        lastAccessAt: s.lastAccessAt,
        expiresAt: s.expiresAt
    }));
    res.json({ activeSessions: activeSessions, total: activeSessions.length });
});

// ========== ЗАПУСК СЕРВЕРА ==========
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
    console.log(`Сервер запущен на http://localhost:${PORT}`);
    console.log('Доступные эндпоинты:');
    console.log('  POST   /api/register   - регистрация');
    console.log('  POST   /api/login      - вход (получение sessionId)');
    console.log('  POST   /api/logout     - выход (требуется X-Session-ID)');
    console.log('  GET    /api/profile    - профиль (требуется X-Session-ID)');
    console.log('  GET    /api/products   - список товаров');
    console.log('  POST   /api/products   - создание товара');
    console.log('  PUT    /api/products/:id - обновление товара');
    console.log('  DELETE /api/products/:id - удаление товара');
    console.log('  GET    /api/admin/users      - список пользователей (админ)');
    console.log('  GET    /api/admin/sessions - список сессий (админ)');
```

});

2.2. Индивидуальные задания

2.2.1. Базовый уровень (15 вариантов)

Общее задание для базового уровня:

Реализовать сервер с сессионной аутентификацией, включающую:

1. Хранилище сессий в памяти (Map).
2. Генерацию session ID через crypto.randomBytes().
3. POST /api/login — создание сессии, возврат session ID.
4. POST /api/logout — удаление сессии.
5. GET /api/profile — защищённый маршрут, возвращающий информацию о пользователе.
6. Middleware authenticateSession для проверки заголовка X-Session-ID.

Варианты сущности для профиля (дополнительные поля):

	Дополнительное поле в таблице users	Описание
	full_name	Полное имя пользователя
	phone	Номер телефона
	age	Возраст
	city	Город проживания
	birth_date	Дата рождения
	position	Должность

	Дополнительное поле в таблице users	Описание
	department	Отдел
	student_id	Номер студенческого билета
	group_name	Название группы
0	course	Курс обучения
1	bio	Краткая биография
2	website	Личный сайт
3	company	Компания
4	github_username	GitHub username
5	telegram_username	Telegram username

2.2.2. Средний уровень (15 вариантов)

Общее задание для среднего уровня:

Выполнить требования базового уровня, а также дополнительно реализовать:

1. Таймаут неактивности — сессия истекает через 30 минут бездействия (проверка lastAccessAt).
2. Ограничение количества активных сессий на одного пользователя (не более 3).
3. Эндпоинт для просмотра всех активных сессий пользователя (GET /api/sessions).
4. Эндпоинт для завершения всех других сессий пользователя (POST /api/sessions/terminate-others).
5. Логирование всех действий с сессиями (создание, удаление, истечение).

Дополнительные параметры для среднего уровня:

№	Максимум сессий на пользователя	Таймаут неактивности
1	3	30 минут
2	2	1 час
3	5	15 минут
4	1	8 часов
5	3	45 минут
6	4	20 минут
7	2	12 часов
8	3	2 часа
9	5	1 час

№	Максимум сессий на пользователя	Таймаут неактивности
10	1	24 часа
11	3	10 минут
12	4	30 минут
13	2	6 часов
14	3	15 минут
15	5	4 часа

Пример реализации ограничения сессий:

javascript

```
const MAX_SESSIONS_PER_USER = 3;

function createSession(user) {
    // Проверка количества активных сессий пользователя
    let userSessions = 0;
    for (const session of sessions values()) {
        if (session userId === user id) userSessions++;
    }

    if (userSessions >= MAX_SESSIONS_PER_USER) {
        // Удаляем самую старую сессию пользователя
        let oldestSessionId = null;
        let oldestTime = Infinity;
        for (const [sid, session] of sessions entries()) {
```

```
            if (session.userId === user.id && session.createdAt < oldestTime) {
                oldestTime = session.createdAt;
                oldestSessionId = sid;
            }
        }
        if (oldestSessionId) {
            sessions.delete(oldestSessionId);
            console.log(`Удалена старая сессия пользователя ${user.username} для создания новой`);
        }
    }

    // Создание новой сессии...
}
```

2.2.3. Продвинутый уровень (15 вариантов)

Общее задание для продвинутого уровня:

1. Выполнить требования среднего уровня, а также дополнительно реализовать:
2. Привязка сессии к IP-адресу и User-Agent (защита от кражи сессии).
3. Refresh-сессии — автоматическое продление срока действия при активности.
4. Хранение сессий не только в памяти, но и в БД (опционально, для сохранения при перезапуске сервера).
5. Эндпоинт для принудительного завершения всех сессий пользователя (административный).
6. CSRF-токен для защиты форм.
7. Переменные окружения для настройки таймаутов.

Варианты дополнительной функциональности:

№	Дополнительная функция
1	Привязка к IP + User-Agent
2	Хранение сессий в SQLite
3	Автоматическое продление сессии (+5 минут после каждого запроса)
4	Эндпоинт /api/sessions/all (админ)
5	Генерация CSRF-токена при создании сессии
6	Валидация CSRF-токена для POST/PUT/DELETE
7	Шифрование session ID в cookie (httpOnly, secure)
8	Поддержка нескольких форматов передачи session ID (header + cookie)
9	Ограничение сессии по времени суток (например, только 9:00-18:00)
10	Уведомление о новом входе (логирование email)
11	История входов пользователя (таблица login_history)
12	Двухфакторная аутентификация при создании сессии
13	Капча при неверном пароле (интеграция с предыдущей лабой)

№	Дополнительная функция
14	Генерация одноразового session ID для каждого запроса
15	Валидация session ID через HMAC

Пример привязки сессии к IP и User-Agent:

javascript

```
function createSession(user, req) {
    const sessionId = generateSessionId();
    const now = Date.now();
    const SESSION_TTL = 24 * 60 * 60 * 1000;

    const session = {
        sessionId: sessionId,
        userId: user.id,
        username: user.username,
        role: user.role,
        createdAt: now,
        lastAccessAt: now,
        expiresAt: now + SESSION_TTL,
        ip: req.ip || req.connection.remoteAddress,
        userAgent: req.headers['user-agent']
    };

    sessions.set(sessionId, session);
    return sessionId;
}

function getSession(sessionId, req) {
```

```
    const session = sessions.get(sessionId);
    if (!session) return null;

    // Проверка привязки к IP и User-Agent
    const currentIp = req.ip || req.connection.remoteAddress;
    const currentUserAgent = req.headers['user-agent'];

    if (session.ip !== currentIp || session.userAgent !== currentUserAgent) {
      console.warn(`Подозрительная активность: сессия ${sessionId.substring(0, 16)}... используется с другого IP/User-Agent`);
      sessions.delete(sessionId);
      return null;
    }

    // Остальная логика...
  }
```

3. Контрольные вопросы

1. Что такое сессия и зачем она нужна в stateless-протоколе HTTP?
2. Чем сессии отличаются от JWT-токенов?
3. Какие способы передачи session ID существуют? Какие преимущества и недостатки каждого?
4. Как сгенерировать уникальный и безопасный session ID в Node.js?
5. Где в памяти сервера хранятся сессии и какие проблемы это создаёт при масштабировании?
6. Как реализовать автоматическое удаление просроченных сессий?
7. Как реализовать таймаут неактивности (отличается от абсолютного времени жизни)?

8. Как ограничить количество активных сессий на одного пользователя?
9. Как привязать сессию к IP-адресу и User-Agent для защиты от кражи?
10. В чём разница между Map и обычным объектом для хранения сессий?
11. Как организовать выход (logout) на сервере?
12. Как защититься от CSRF-атак при использовании сессий?
13. Почему хранение сессий в памяти не подходит для production-приложений?
14. Как можно хранить сессии для горизонтального масштабирования?
15. Что такое «привязка сессии» (session affinity или sticky sessions)?

4. Требования к отчёту

Отчёт должен быть оформлен в электронном виде и содержать:

Титульный лист с названием работы, ФИО студента, номером группы, вариантом.

Цель и задачи работы (переписать из методических указаний).

Теоретическое обоснование (краткий конспект по теме).

Ход выполнения работы:

Листинг кода сервера с комментариями.

Скриншоты проверки работы через Postman:

Регистрация пользователя (201)

Вход (получение session ID)

Доступ к /api/profile с правильным session ID (200)

Доступ к /api/profile без session ID (401)

Доступ к /api/profile с неверным session ID (401)

Выход (logout)

(Средний уровень) Просмотр активных сессий

(Продвинутый уровень) Проверка привязки к IP

Ответы на контрольные вопросы (письменно, развёрнуто).

Заключение (выводы).

Список использованных источников.

5. Критерии оценивания

Уровень выполнения	Требования	Оценка
Базовый	Хранилище сессий в Map, генерация session ID, эндпоинты /login (возврат sessionId) и /logout, защищённый /profile.	Удовлетворительно
Средний	Выполнены требования базового уровня + таймаут неактивности, ограничение количества сессий на пользователя, просмотр активных сессий, логирование.	Хорошо
Продвинутый	Выполнены	Отлично

Уровень выполнения	Требования	Оценка
	требования среднего уровня + привязка к IP/User-Agent, продление сессии, хранение в БД (опционально), CSRF-токен, переменные окружения.	

Дополнительные условия:

1. Код должен быть структурирован и снабжён комментариями.
2. При защите студент должен объяснить принципы работы сессионной аутентификации.

6. Список рекомендуемой литературы и интернет-ресурсов

Литература (доступна в РФ)

1. Браун, И. (2021). *Express.js: Разработка веб-приложений на Node.js*. ДМК Пресс.
2. Кантелон, М., Хартер, М., & Холстейн, С. (2020). *Node.js в действии*. Питер.

Интернет-ресурсы

Название	Ссылка	Описание
crypto.randomBytes	https://nodejs.org/api/crypto.html	Генерация случайных байтов

Название	Ссылка	Описание
HTTP Cookies	https://developer.mozilla.org/ru/docs/Web/HTTP/Cookies	MDN о cookie
Session Management OWASP	https://cheatsheetseries.owasp.org/	Рекомендации по управлению сессиями
Map в JavaScript	https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Map	Документация Map
CSRF Protection	https://owasp.org/www-community/attacks/csrf	Информация о CSRF-атаках

Все ресурсы доступны и легальны на территории Российской Федерации.

Лабораторная работа № 13

Тема: Переменные окружения и конфигурация

Цель работы:

Научиться выносить настройки приложения в переменные окружения для обеспечения безопасности, гибкости и удобства конфигурирования приложения в различных средах (разработка, тестирование, production).

Задачи работы:

Изучить назначение и принципы работы переменных окружения.

Освоить библиотеку dotenv для загрузки переменных из файла .env.

Вынести конфиденциальные данные (порт, путь к БД, секреты) в переменные окружения.

Научиться создавать разные конфигурации для разных сред (development, production, test).

Реализовать проверку наличия обязательных переменных окружения при запуске приложения.

Настроить .gitignore для исключения файлов с секретами из репозитория.

1. Теоретическое обоснование

1.1. Что такое переменные окружения?

Переменные окружения (environment variables) -это динамические именованные значения, которые могут влиять на поведение работающих процессов на компьютере. В контексте веб-приложений они используются для хранения конфигурации:

1. Порт, на котором запускается сервер
2. Строки подключения к базам данных
3. API-ключи и секреты
4. Режим работы приложения (development/production)
5. Параметры кэширования, логирования и т.д.

Примеры переменных окружения:

```
PORT=3000
DB_PATH=./database.db
JWT_SECRET=super_secret_key_123
NODE_ENV=production
```

1.2. Почему нужно использовать переменные окружения?

Причина	Объяснение
Безопасность	Пароли, ключи и секреты не попадают в исходный код и систему контроля версий
Гибкость	Одна кодовая база может работать в разных окружениях без изменений кода
Двенадцатифакторное приложение	Рекомендация методологии Twelve-Factor App -хранить конфигурацию в переменных окружения
Удобство развёртывания	На сервере можно легко изменить настройки без пересборки приложения

1.3. Библиотека dotenv

dotenv -это npm-библиотека, которая загружает переменные из файла .env в process.env.

Установка:

bash

```
npm install dotenv
```

Использование:

javascript

```
// В самом начале файла app.js
require('dotenv').config();
```

```
// После этого переменные из .env доступны через process.env
const port = process.env.PORT || 3000;
const dbPath = process.env.DB_PATH || './database.db';
```

Файл .env:

```
PORT=8080
DB_PATH=./data/app.db
JWT_SECRET=my-very-secret-key
NODE_ENV=development
```

1.4. Файл .env и .gitignore

Файл .env НЕ должен попадать в репозиторий Git, так как он содержит секреты. В репозиторий помещается файл-шаблон .env.example.

.gitignore:

```
.env
.env.local
.env.*.local
node_modules/
```

.env.example (шаблон, который можно публиковать):

```
PORT=3000
DB_PATH=./database.db
JWT_SECRET=your-secret-key-here
NODE_ENV=development
LOG_LEVEL=info
```

1.5. Проверка обязательных переменных

При запуске приложения следует проверять наличие обязательных переменных окружения:

javascript

```
function checkRequiredEnv(variables) {
  const missing = [];
  for (const variable of variables) {
    if (!process.env[variable]) {
      missing.push(variable);
    }
  }
  if (missing.length > 0) {
    console.error(`✗ Отсутствуют обязательные переменные окружения: ${missing.join(', ')}`);
    console.error('Создайте файл .env на основе .env.example');
    process.exit(1);
  }
}

// Использование
checkRequiredEnv(['PORT', 'DB_PATH', 'JWT_SECRET']);
```

1.6. Разные окружения

Часто используют разные файлы для разных окружений:

.env.development -для разработки

.env.test -для тестирования

.env.production -для продакшена

javascript

```
const env = process.env.NODE_ENV || 'development';
require('dotenv').config({ path: `.env.${env}` });
```

1.7. Типичные переменные окружения для веб-приложения

Переменная	Описание	Пример
PORT	Порт сервера	3000
NODE_ENV	Режим работы	development, production, test
DB_PATH	Путь к файлу SQLite	./data/database.db
DB_HOST	Хост БД (для PostgreSQL/MySQL)	localhost
DB_PORT	Порт БД	5432
DB_USER	Пользователь БД	postgres
DB_PASSWORD	Пароль БД	secret
DB_NAME	Имя базы данных	myapp
JWT_SECRET	Секрет для JWT	very-secret-key
SESSION_SECRET	Секрет для сессий	session-secret
LOG_LEVEL	Уровень	debug, info, warn, erro

Переменная	Описание	Пример
	логирования	r
BCRYPT_ROUNDS	Количество раундов bcrypt	10
CORS_ORIGIN	Разрешённые источники CORS	http://localhost:3001

2. Методика выполнения работы

2.1. Примеры и их подробный разбор

Пример 1. Установка и базовая настройка dotenv

Шаг 1. Установка dotenv:

bash

```
npm install dotenv
```

Шаг 2. Создание файла .env:

```
# Файл .env (не публикуется в Git)
PORT=3000
DB_PATH=./data/app.db
JWT_SECRET=super-secret-key-2024
NODE_ENV=development
LOG_LEVEL=debug
BCRYPT_ROUNDS=10
```

Шаг 3. Создание файла .env.example (публикуется в Git):

```
# Файл .env.example (шаблон)
```

```
PORT=3000
DB_PATH=./database.db
JWT_SECRET=change_this_secret_key
NODE_ENV=development
LOG_LEVEL=info
BCRYPT_ROUNDS=10
```

Шаг 4. Настройка .gitignore:

```
.env
.env.local
.env.*.local
node_modules/
data/
*.log
```

Шаг 5. Использование в коде:

javascript

```
// app.js
require('dotenv').config();

const express = require('express');
const path = require('path');

const app = express();
const PORT = process.env.PORT || 3000;
const DB_PATH = process.env.DB_PATH || './database.db';

console.log(`Запуск в режиме: ${process.env.NODE_ENV}`);
```

```
console.log(`Порт: ${PORT}`);
console.log(`Путь к БД: ${DB_PATH}`);

app.listen(PORT, () => {
    console.log(`Сервер запущен на http://localhost:${PORT}`);
});
```

Пример 2. Проверка обязательных переменных

javascript

```
// config.js -модуль конфигурации
const path = require('path');

// Загрузка переменных окружения
require('dotenv').config();

// Список обязательных переменных
const requiredEnvVars = ['PORT', 'DB_PATH', 'JWT_SECRET'];

// Проверка наличия обязательных переменных
function validateEnv() {
    const missing = [];
    for (const envVar of requiredEnvVars) {
        if (!process.env[envVar]) {
            missing.push(envVar);
        }
    }

    if (missing.length > 0) {
        console.error('✗ Ошибка: Отсутствуют обязательные переменные окружения:');
```

```
        missing.forEach(v => console.error(`   - ${v}`));
        console.error('\nСоздайте файл .env на основе .env.example');
        process.exit(1);
    }

    console.log('✅ Все обязательные переменные окружения присутствуют');
}

validateEnv();

// Экспорт конфигурации
module.exports = {
    port: parseInt(process.env.PORT, 10),
    dbPath: path.resolve(__dirname, process.env.DB_PATH),
    jwtSecret: process.env.JWT_SECRET,
    nodeEnv: process.env.NODE_ENV || 'development',
    logLevel: process.env.LOG_LEVEL || 'info',
    bcryptRounds: parseInt(process.env.BCRYPT_ROUNDS, 10) || 10,
    isProduction: process.env.NODE_ENV === 'production',
    isDevelopment: process.env.NODE_ENV === 'development',
    isTest: process.env.NODE_ENV === 'test'
};
```

Использование в основном приложении:

```
javascript
// app.js
const config = require('./config');
const express = require('express');
```

```
const app = express();

console.log('Конфигурация приложения:');
console.log(`  Режим: ${config.nodeEnv}`);
console.log(`  Порт: ${config.port}`);
console.log(`  Путь к БД: ${config.dbPath}`);
console.log(`  Уровень логирования: ${config.logLevel}`);

app.listen(config.port, () => {
    console.log(`Сервер запущен на http://localhost:${config.port}`);
});
```

Пример 3. Разные конфигурации для разных окружений

Структура файлов:

text

.env.development

.env.test

.env.production

config.js

.env.development:

text

PORT=3000

DB_PATH=./data/dev.db

JWT_SECRET=dev-secret-key

NODE_ENV=development

LOG_LEVEL=debug

.env.test:

text

PORT=3001

DB_PATH=./data/test.db

JWT_SECRET=test-secret-key

NODE_ENV=test

LOG_LEVEL=error

.env.production:

text

PORT=8080

DB_PATH=/var/lib/app/database.db

JWT_SECRET=prod-very-secret-key-change-me

NODE_ENV=production

LOG_LEVEL=info

config.js:

javascript

```
const path = require('path');

// Определение текущего окружения
const NODE_ENV = process.env.NODE_ENV || 'development';

// Загрузка соответствующего файла .env
const envFile = `.env.${NODE_ENV}`;
require('dotenv').config({ path: envFile });

console.log(`Загружена конфигурация из ${envFile}`);

module.exports = {
  port: parseInt(process.env.PORT, 10),
  dbPath: path.resolve(__dirname, process.env.DB_PATH),
  jwtSecret: process.env.JWT_SECRET,
  nodeEnv: NODE_ENV,
  logLevel: process.env.LOG_LEVEL || 'info',
```

```
    bcryptRounds: parseInt(process.env.BCRYPT_ROUNDS, 10) || 10,

    // Вспомогательные флаги
    isDev: NODE_ENV === 'development',
    isTest: NODE_ENV === 'test',
    isProd: NODE_ENV === 'production'
};
```

Запуск с разными окружениями:

bash

```
# Разработка
NODE_ENV=development node app.js

# Тестирование
NODE_ENV=test node app.js

# Продакшен
NODE_ENV=production node app.js
```

Пример 4. Полное приложение с конфигурацией

javascript

```
// config.js
const path = require('path');
const fs = require('fs');

// Загрузка переменных окружения
require('dotenv').config();

// Обязательные переменные
const REQUIRED_ENV_VARS = ['PORT', 'DB_PATH', 'JWT_SECRET'];
```

```
function validateEnv() {
    const missing = REQUIRED_ENV_VARS.filter(v => !process.env[v]);

    if (missing.length > 0) {
        console.error('\n✗ Ошибка конфигурации:');
        console.error('Отсутствуют следующие переменные окружения:');
        missing.forEach(v => console.error(`   - ${v}`));
        console.error('\nПожалуйста, создайте файл .env на основе .env.example\n');
        process.exit(1);
    }

    // Проверка существования директории для БД
    const dbDir = path.dirname(path.resolve(__dirname, process.env.DB_PATH));
    if (!fs.existsSync(dbDir)) {
        fs.mkdirSync(dbDir, { recursive: true });
        console.log(`📁 Создана директория для БД: ${dbDir}`);
    }
}

validateEnv();

// Преобразование и валидация числовых значений
const port = parseInt(process.env.PORT, 10);
if (isNaN(port) || port < 1 || port > 65535) {
    console.error(`✗ PORT должен быть числом от 1 до 65535. Текущее значение: ${process.env.PORT}`);
```

```
    process.exit(1);
  }

  const bcryptRounds = parseInt(process.env.BCRYPT_ROUNDS, 10);
  if (isNaN(bcryptRounds) || bcryptRounds < 4 || bcryptRounds > 20) {
    console.warn(`⚠ BCRYPT_ROUNDS должен быть от 4 до 20. Используется значение по умолчанию: 10`);
  }

  const config = {
    // Сервер
    port: port,
    nodeEnv: process.env.NODE_ENV || 'development',

    // База данных
    dbPath: path.resolve(__dirname, process.env.DB_PATH),

    // Безопасность
    jwtSecret: process.env.JWT_SECRET,
    bcryptRounds: bcryptRounds || 10,

    // Логирование
    logLevel: process.env.LOG_LEVEL || 'info',

    // CORS
    corsOrigin: process.env.CORS_ORIGIN || '*',

    // Флаги окружения
    isDev: process.env.NODE_ENV === 'development',
```

```
    isTest: process.env.NODE_ENV === 'test',
    isProd: process.env.NODE_ENV === 'production'
};

// Вывод конфигурации (без секретов) при запуске
if (config.isDev) {
    console.log('\n📋 Текущая конфигурация:');
    console.log(`   Режим: ${config.nodeEnv}`);
    console.log(`   Порт: ${config.port}`);
    console.log(`   Путь к БД: ${config.dbPath}`);
    console.log(`   Уровень логирования: ${config.logLevel}`);
    console.log(`   bcrypt раундов: ${config.bcryptRounds}`);
    console.log('');
}

module.exports = config;
```

app.js с использованием конфигурации:

javascript

```
const express = require('express');
const bcrypt = require('bcrypt');
const sqlite3 = require('sqlite3').verbose();
const config = require('./config');

const app = express();
app.use(express.json());

// Подключение к БД с использованием пути из конфигурации
const db = new sqlite3.Database(config.dbPath, (err) => {
```

```
    if (err) {
        console error('✕ Ошибка подключения к БД:', err message);
        process exit(1);
    }
    console log('✓ Подключено к базе данных');
});

// Создание таблиц
db run(`
    CREATE TABLE IF NOT EXISTS users (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        username TEXT NOT NULL UNIQUE,
        password_hash TEXT NOT NULL,
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    )
`);

// Регистрация с использованием bcryptRounds из конфигурации
app post('/api/register', async (req, res) => {
    const { username, password } = req body;

    if (!username || !password) {
        return res status(400) json({ error: 'Все поля обязательны' });
    }

    try {
        const hash = await bcrypt hash(password, config bcryptRounds);
```

```
            db.run('INSERT INTO users (username, password_hash) VALUES (?, ?)',
                [username, hash],
                function(err) {
                    if (err) {
                        if (err.message.includes('UNIQUE')) {
                            return res.status(409).json({ error: 'Пользователь уже существует' });
                        }
                        return res.status(500).json({ error: 'Ошибка сервера' });
                    }
                    res.status(201).json({ message: 'Регистрация успешна', userId: this.lastID });
                }
            );
        } catch (error) {
            res.status(500).json({ error: 'Ошибка сервера' });
        }
    });

    // Условный вывод в зависимости от окружения
    if (config.isDev) {
        // Только в режиме разработки
        const morgan = require('morgan');
        app.use(morgan('dev'));
        console.log('🔧 Включено детальное логирование запросов (morgan)');
    }

    if (config.isProd) {
```

```
    // В продакшене - дополнительные заголовки безопасности
    app.use((req, res, next) => {
        res.setHeader('X-Content-Type-Options', 'nosniff');
        res.setHeader('X-Frame-Options', 'DENY');
        res.setHeader('X-XSS-Protection', '1; mode=block');
        next();
    });
}

// Запуск сервера
app.listen(config.port, () => {
    console.log(`\n🚀 Сервер запущен:`);
    console.log(`   URL: http://localhost:${config.port}`);
    console.log(`   Режим: ${config.nodeEnv}`);
    console.log(`   Дата и время: ${new Date().toLocaleString()}\n`);
});

// Обработка некорректного завершения
process.on('SIGINT', () => {
    console.log('\n👋 Сервер остановлен');
    db.close(() => process.exit(0));
});
```

Пример 5. Создание скрипта для генерации секретов

scripts/generate-secrets.js:

javascript

```
const crypto = require('crypto');
const fs = require('fs');
const path = require('path');
```

```
// Генерация случайных секретов
const jwtSecret = crypto.randomBytes(32).toString('hex');
const sessionSecret = crypto.randomBytes(32).toString('hex');

// Чтение существующего .env или создание нового
const envPath = path.join(__dirname, '..', '.env');
let envContent = '';

if (fs.existsSync(envPath)) {
    envContent = fs.readFileSync(envPath, 'utf-8');
}

// Обновление или добавление секретов
const secrets = {
    JWT_SECRET: jwtSecret,
    SESSION_SECRET: sessionSecret
};

for (const [key, value] of Object.entries(secrets)) {
    const regex = new RegExp(`^${key}=.*$`, 'm');
    if (regex.test(envContent)) {
        envContent = envContent.replace(regex, `${key}=${value}`);
    } else {
        envContent += `\n${key}=${value}`;
    }
}

fs.writeFileSync(envPath, envContent.trim());
console.log('✅ Секреты сгенерированы и сохранены в .env');
```

```
console.log(`   JWT_SECRET: ${jwtSecret.substring(0, 16)}...`);
console.log(`   SESSION_SECRET: ${sessionSecret.substring(0, 16)}...`);
```

Добавление скрипта в package.json:

json

```
{
  "scripts": {
    "start": "node app.js",
    "dev": "nodemon app.js",
    "generate-secrets": "node scripts/generate-secrets.js",
    "validate-env": "node scripts/validate-env.js"
  }
}
```

2.2. Индивидуальные задания

2.2.1. Базовый уровень (15 вариантов)

Общее задание для базового уровня:

Создать приложение, которое использует переменные окружения для конфигурации. Требования:

1. Установить и настроить dotenv.
2. Создать файлы .env и .env.example.
3. Вынести в переменные окружения минимум 4 параметра:
4. Порт сервера (PORT)
5. Путь к базе данных (DB_PATH)
6. Режим работы (NODE_ENV)
7. Один параметр по варианту
8. Настроить .gitignore для исключения .env.
9. Вывести при запуске информацию о конфигурации.

Варианты четвёртого параметра:

	Переменная	Значение по умолчанию	Описание
	API_VERSION	v1	Версия API
	PAGE_SIZE	10	Количество элементов на странице
	LOG_LEVEL	info	Уровень логирования
	CORS_ORIGIN	*	Разрешённые источники CORS
	RATE_LIMIT_WINDOW	15	Окно rate limiting (минуты)
	RATE_LIMIT_MAX	100	Максимум запросов в окно
	SESSION_TIMEOUT	24	Таймаут сессии (часы)
	MAX_ITEMS_PER_PAGE	50	Максимум элементов на странице
	DEFAULT_LANGUAGE	ru	Язык по умолчанию

	Переменная	Значение по умолчанию	Описание
0	TIMEZONE	Europe/Moscow	Часовой пояс
1	DATE_FORMAT	YYYY-MM-DD	Формат даты
2	MAX_FILE_SIZE_MB	5	Максимальный размер файла (МБ)
3	UPLOAD_DIR	./uploads	Директория для загрузок
4	CACHE_TTL_SECONDS	300	Время жизни кэша (секунды)
5	MAX_RETRIES	3	Максимум повторных попыток

2.2.2. Средний уровень (15 вариантов)

Общее задание для среднего уровня:

Выполнить требования базового уровня, а также дополнительно реализовать:

1. Проверку наличия обязательных переменных с выводом понятной ошибки и завершением приложения.
2. Разные файлы конфигурации для разных окружений (.env.development, .env.production).
3. Валидацию типов (порт -число, таймаут -число и т.д.).

4. Скрипт проверки конфигурации (npm run validate-env).
5. Разные настройки в зависимости от окружения (например, в dev - детальное логирование, в prod -минимальное).

Варианты дополнительной валидации (средний уровень):

№	Валидируемое поле	Правило валидации
1	PORT	от 1024 до 49151
2	BCRYPT_ROUNDS	от 8 до 14
3	PAGE_SIZE	от 5 до 100
4	RATE_LIMIT_MAX	от 10 до 1000
5	SESSION_TIMEOUT	от 1 до 720 (часов)
6	MAX_ITEMS_PER_PAGE	от 1 до 200
7	CACHE_TTL_SECONDS	от 1 до 86400
8	MAX_FILE_SIZE_MB	от 1 до 100
9	MAX_RETRIES	от 1 до 10
10	JWT_EXPIRES_IN	формат: 1d, 2h, 30m
11	LOG_LEVEL	debug, info, warn, error

№	Валидируемое поле	Правило валидации
12	DB_POOL_SIZE	от 1 до 50
13	API_RATE_LIMIT	от 1 до 500
14	UPLOAD_DIR_PERMISSIONS	0o755, 0o775, 0o777
15	CORS_ORIGIN	валидный URL или *

2.2.3. Продвинутый уровень (15 вариантов)

Общее задание для продвинутого уровня:

1. Выполнить требования среднего уровня, а также дополнительно реализовать:
2. Использование библиотеки envalid для валидации и типизации переменных окружения.
3. Шифрование чувствительных переменных в .env (опционально).
4. Генерацию секретов при первом запуске (JWT_SECRET, SESSION_SECRET).
5. Загрузку конфигурации из JSON-файла (для сложных настроек).
6. Динамическую перезагрузку конфигурации без перезапуска (при изменении .env).
7. Поддерку Docker-переменных окружения (приоритет над .env).

Установка envalid:

bash

```
npm install envalid
```

Пример использования envalid:

javascript

```
// config.js
const path = require('path');
const { cleanEnv, port, str, num, bool, url } = require('envalid');
require('dotenv').config();

const config = cleanEnv(process.env, {
  NODE_ENV: str({ choices: ['development', 'test', 'production'], default: 'development' }),
  PORT: port({ default: 3000 }),
  DB_PATH: str({ default: './database.db' }),
  JWT_SECRET: str(),
  BCRYPT_ROUNDS: num({ default: 10, min: 8, max: 14 }),
  LOG_LEVEL: str({ choices: ['debug', 'info', 'warn', 'error'], default: 'info' }),
  CORS_ORIGIN: str({ default: '*' }),
  ENABLE_LOGGING: bool({ default: true })
});

module.exports = {
  ...config,
  dbPath: path.resolve(__dirname, config.DB_PATH),
  isDev: config.NODE_ENV === 'development',
  isProd: config.NODE_ENV === 'production',
  isTest: config.NODE_ENV === 'test'
};
```

Варианты продвинутых функций:

№	Дополнительная функция
1	Шифрование .env с помощью crypto
2	Генерация секретов при npm run setup
3	Поддержка Docker secrets (файл /run/secrets/)
4	Валидация с помощью Zod вместо envalid
5	Автоматическая документация переменных окружения
6	Загрузка конфигурации из удалённого источника (HTTP)
7	Поддержка вложенных конфигураций (YAML/JSON)
8	Горячая перезагрузка конфигурации (watch .env)
9	Маскировка секретов в логах
10	Профили конфигурации (default, production, staging)
11	Конфигурация через аргументы командной строки
12	Версионирование конфигурации
13	Валидация зависимых переменных

№	Дополнительная функция
14	Экспорт конфигурации в формате JSON для внешних сервисов
15	Интеграция с облачными секретами (AWS Secrets Manager, Yandex Lockbox)

Пример генерации секретов при первом запуске:

javascript

```
// setup.js -запускается при установке
const crypto = require('crypto');
const fs = require('fs');
const path = require('path');

function ensureEnvFile() {
  const envPath = path.join(__dirname, '.env');

  if (!fs.existsSync(envPath)) {
    console.log('📝 Файл .env не найден. Создаём...');

    const jwtSecret = crypto.randomBytes(32).toString('hex');
    const sessionSecret = crypto.randomBytes(32).toString('hex');

    const envContent = `# Сгенерировано автоматически
PORT=3000
DB_PATH=./database.db
JWT_SECRET=${jwtSecret}
SESSION_SECRET=${sessionSecret}
NODE_ENV=development
```

```
LOG_LEVEL=info
BCRYPT_ROUNDS=10
`;

        fs.writeFileSync(envPath, envContent);
        console.log('✅ Файл .env создан с сгенерированными секретами');
        console.log('⚠️ Пожалуйста, проверьте настройки в файле .env');
    }
}

ensureEnvFile();
```

Добавление в package.json:

```
json
{
  "scripts": {
    "postinstall": "node setup.js",
    "start": "node app.js",
    "dev": "nodemon app.js",
    "validate": "node scripts/validate-env.js"
  }
}
```

3. Контрольные вопросы

1. Что такое переменные окружения и для чего они используются?
2. Почему нельзя хранить секреты (пароли, ключи) непосредственно в коде?
3. Какую роль выполняет библиотека dotenv?
4. Какой файл нужно добавить в .gitignore при использовании dotenv?

5. Зачем создавать файл .env.example?
6. Как в коде получить значение переменной окружения?
7. Как задать значение по умолчанию, если переменная не определена?
8. Как организовать разные конфигурации для разработки и продакшена?
9. Как проверить наличие всех обязательных переменных при запуске?
10. Какие данные обычно хранят в переменных окружения?
11. В чём разница между process.env.PORT и process.env["PORT"]?
12. Как передать переменную окружения при запуске Node.js приложения (NODE_ENV=production node app.js)?
13. Какие проблемы могут возникнуть при хранении паролей в коде?
14. Как организовать конфигурацию для тестового окружения?
15. Какие типы данных можно хранить в переменных окружения? Как обрабатывать числа и булевы значения?

4. Требования к отчёту

Отчёт должен быть оформлен в электронном виде и содержать:

Титульный лист с названием работы, ФИО студента, номером группы, вариантом.

Цель и задачи работы (переписать из методических указаний).

Теоретическое обоснование (краткий конспект по теме).

Ход выполнения работы:

Содержание файла .env.example

Содержание файла .gitignore

Листинг кода с использованием dotenv

Скриншоты:

Запуск приложения с выводом конфигурации

Ошибка при отсутствии обязательных переменных

(Средний уровень) Запуск с разными окружениями

(Продвинутый уровень) Работа envalid, генерация секретов

Ответы на контрольные вопросы (письменно, развёрнуто).

Заключение (выводы).

Список использованных источников.

5. Критерии оценивания

Уровень выполнения	Требования	Оценка
Базовый	Установлен dotenv, созданы .env и .env.example, вынесены 4+ переменных, настроен .gitignore.	Удовлетворительно
Средний	Выполнены требования базового уровня + проверка обязательных переменных, разные файлы для окружений, валидация типов, скрипт проверки.	Хорошо
Продвинутый	Выполнены требования среднего уровня + использование envalid (или аналога), генерация секретов, поддержка Docker-переменных, шифрование (опционально).	Отлично

Дополнительные условия:

1. Код должен использовать process.env только через модуль конфигурации.
2. При защите студент должен объяснить, почему те или иные параметры вынесены в конфигурацию.

6. Список рекомендуемой литературы и интернет-ресурсов

Литература (доступна в РФ)

1. Браун, И. (2021). *Express.js: Разработка веб-приложений на Node.js*. ДМК Пресс.
2. Кантелон, М., Хартер, М., & Холстейн, С. (2020). *Node.js в действии*. Питер.

Интернет-ресурсы

Название	Ссылка	Описание
dotenv npm	https://www.npmjs.com/package/dotenv	Документация dotenv
envalid npm	https://www.npmjs.com/package/envalid	Валидация переменных окружения
Twelve-Factor App	https://12factor.net/ru/config	Хранение конфигурации

Название	Ссылка	Описание
Node.js process.env	https://nodejs.org/api/process.html	Документация
.gitignore документация	https://git-scm.com/docs/gitignore	Формат .gitignore
crypto.randomBytes	https://nodejs.org/api/crypto.html	Генерация секретов

Все ресурсы доступны и легальны на территории Российской Федерации.

Лабораторная работа № 14

Тема: Логирование запросов и ошибок

Цель работы:

Научиться настраивать логирование HTTP-запросов и системных ошибок в серверном приложении для целей отладки, мониторинга и аудита.

Задачи работы:

1. Подключить и настроить middleware для логирования HTTP-запросов (morgan).
2. Настроить вывод логов в консоль и запись в файл.
3. Реализовать логирование ошибок с разными уровнями (info, warn, error).

4. Создать формат логов с временными метками и контекстной информацией.
5. Реализовать ротацию лог-файлов (разбиение по дням или размеру).
6. Настроить разные уровни логирования в зависимости от окружения (dev/prod).
7. Интегрировать логирование в обработчики ошибок.

1. Теоретическое обоснование

1.1. Зачем нужно логирование?

Логирование - это процесс записи информации о работе приложения. Оно необходимо для:

Цель	Описание
Отладка	Понимание того, что происходит в приложении во время разработки
Мониторинг	Отслеживание состояния production-приложения
Аудит	Фиксация действий пользователей для расследования инцидентов
Анализ ошибок	Выявление причин сбоев и исключений
Анализ производительности	Измерение времени выполнения запросов

1.2. Уровни логирования

Уровень	Описание	Пример
error	Критические ошибки, требующие немедленного вмешательства	Не удалось подключиться к БД
warn	Предупреждения, которые могут привести к проблемам	Попытка доступа к несуществующему ресурсу
info	Информационные сообщения о нормальной работе	Сервер запущен, пользователь вошёл в систему
debug	Детальная информация для отладки	Параметры запроса, SQL-запросы
trace	Максимально детальные логи (для глубокой отладки)	Каждый шаг выполнения функции

1.3. Morgan -логгер HTTP-запросов

morgan -это middleware для Express, которое автоматически логирует HTTP-запросы.

Установка:

bash

```
npm install morgan
```

Использование:

javascript

```
const morgan = require('morgan');
app use(morgan('combined'));  // предустановленный формат
```

Форматы morgan:

Формат	Пример вывода
combined	::1 - - [28/Oct/2024:12:34:56 +0000] "GET /api/users HTTP/1.1" 200 1234 "-" "Mozilla/5.0..."
common	::1 - - [28/Oct/2024:12:34:56 +0000] "GET /api/users HTTP/1.1" 200 1234
dev	GET /api/users 200 12.345 ms - 1234
tiny	GET /api/users 200 1234 - 12.345 ms
short	::1 - GET /api/users HTTP/1.1 200 1234 - 12.345 ms

Кастомный формат:

javascript

```
morgan(':method :url :status :response-time ms - :res[content-length]');
```

Токены morgan:

Токен	Описание
:method	HTTP-метод (GET, POST и т.д.)
:url	URL запроса

Токен	Описание
:status	HTTP-код ответа
:response-time	Время выполнения запроса (мс)
:remote-addr	IP-адрес клиента
:user-agent	User-Agent клиента
:referrer	Referer заголовок
:date[format]	Дата и время
:req[header]	Заголовок запроса
:res[header]	Заголовок ответа

1.4. Winston -универсальный логгер

winston - мощная библиотека для логирования с поддержкой разных транспортов (консоль, файл, HTTP, и т.д.).

Установка:

bash

```
npm install winston
```

Базовое использование:

javascript

```
const winston = require('winston');

const logger = winston createLogger({
    level: 'info',
    format: winston format json(),
```

```
    transports: [
        new winston.transports.Console(),
        new winston.transports.File({ filename: 'error.log', level: 'error' }),
        new winston.transports.File({ filename: 'combined.log' })
    ]
});

logger.info('Сообщение для логирования');
logger.error('Ошибка!');
logger.warn('Предупреждение');
logger.debug('Отладочная информация');
```

1.5. Ротация логов

Лог-файлы могут очень быстро расти. Для их ротации (разбиения) используются:

1. winston-daily-rotate-file -для winston
2. logrotate -системная утилита Linux

Установка winston-daily-rotate-file:

bash

```
npm install winston-daily-rotate-file
```

Использование:

javascript

```
const DailyRotateFile = require('winston-daily-rotate-file');

const transport = new DailyRotateFile({
    filename: 'logs/application-%DATE%.log',
    datePattern: 'YYYY-MM-DD',
    maxSize: '20m',
    maxFiles: '14d'
```

```
});
```

1.6. Структурированное логирование

Структурированные логи (JSON) удобнее для анализа, чем текстовые:

javascript

```
// Вместо
logger.info('User logged in', user.id);

// Используйте
logger.info('User logged in', { userId: user.id, username: user.username, ip: req.ip });
```

2. Методика выполнения работы

2.1. Примеры и их подробный разбор

Пример 1. Установка и базовая настройка morgan

bash

```
npm install morgan
```

javascript

```
const express = require('express');
const morgan = require('morgan');
const app = express();

// Использование предустановленного формата 'dev'
app.use(morgan('dev'));

app.get('/', (req, res) => {
  res.send('Hello World!');
});

app.listen(3000);
```

Вывод в консоль:

text

```
GET / 200 2.345 ms - 12
GET /favicon.ico 404 0.512 ms - 45
```

Пример 2. Логирование с кастомным форматом

javascript

```
app.use(morgan(':method :url :status :response-time ms - :res[content-length] bytes'));

// Настройка формата с цветами (только для dev)
if (process.env.NODE_ENV !== 'production') {
  app.use(morgan('dev'));
} else {
  app.use(morgan('combined'));
}
```

Пример 3. Запись логов morgan в файл

javascript

```
const fs = require('fs');
const path = require('path');
const morgan = require('morgan');

// Создание директории для логов
const logDir = path.join(__dirname, 'logs');
if (!fs.existsSync(logDir)) {
  fs.mkdirSync(logDir);
}

// Создание потока записи в файл
const accessLogStream = fs.createWriteStream(
```

```
    path.join(logDir, 'access.log'),
    { flags: 'a' }  // 'a' -append (добавление в конец)
);

// Запись логов в файл
app.use(morgan('combined', { stream: accessLogStream }));

// Также вывод в консоль (для разработки)
if (process.env.NODE_ENV !== 'production') {
    app.use(morgan('dev'));
}
```

Пример 4. Настройка winston для логирования ошибок

bash

```
npm install winston
```

javascript

```
const winston = require('winston');
const path = require('path');
const fs = require('fs');

// Создание директории для логов
const logDir = path.join(__dirname, 'logs');
if (!fs.existsSync(logDir)) {
    fs.mkdirSync(logDir);
}

// Настройка формата логов
const logFormat = winston.format.combine(
    winston.format.timestamp({ format: 'YYYY-MM-DD HH:mm:ss' }),
    winston.format.errors({ stack: true }),
```

```
    winston.format.splat(),
    winston.format.json()
  );

  // Формат для консоли (читаемый)
  const consoleFormat = winston.format.combine(
    winston.format.colorize(),
    winston.format.timestamp({ format: 'YYYY-MM-DD HH:mm:ss' }),
    winston.format.printf(({ timestamp, level, message, ...meta }) => {
      return `${timestamp} [${level}]: ${message} ${Object.keys(meta).length ? JSON.stringify(meta) : ''}`;
    })
  );

  // Создание логгера
  const logger = winston.createLogger({
    level: process.env.LOG_LEVEL || 'info',
    format: logFormat,
    transports: [
      // Логи ошибок -отдельный файл
      new winston.transports.File({
        filename: path.join(logDir, 'error.log'),
        level: 'error',
        maxsize: 10 * 1024 * 1024, // 10 MB
        maxFiles: 5
      }),
      // Все логи -общий файл
      new winston.transports.File({
        filename: path.join(logDir, 'combined.log'),
```

```
        maxsize: 20 * 1024 * 1024,  // 20 MB
        maxFiles: 5
      })
  ]
});

// Добавление консольного вывода в режиме разработки
if (process.env.NODE_ENV !== 'production') {
  logger.add(new winston.transports.Console({
    format: consoleFormat
  }));
}

module.exports = logger;
```

Пример 5. Использование winston для логирования в приложении javascript

```
const express = require('express');
const morgan = require('morgan');
const logger = require('./logger');

const app = express();

// Настройка morgan с записью в winston
app.use(morgan('combined', {
  stream: {
    write: (message) => logger.info(message.trim())
  }
}));
```

```
// Логирование запросов с дополнительной информацией
app.use((req, res, next) => {
  const start = Date.now();

  // Перехват окончания ответа
  res.on('finish', () => {
    const duration = Date.now() - start;
    logger.info(`${req.method} ${req.url} ${res.statusCode} -
${duration}ms`, {
      method: req.method,
      url: req.url,
      status: res.statusCode,
      duration: duration,
      ip: req.ip,
      userAgent: req.get('user-agent')
    });
  });

  next();
});

// Логирование ошибок
app.use((err, req, res, next) => {
  logger.error('Ошибка обработки запроса', {
    error: err.message,
    stack: err.stack,
    method: req.method,
    url: req.url,
    body: req.body,
```

```
        query: req.query,
        params: req.params
    });

    res.status(500).json({ error: 'Внутренняя ошибка сервера' });
});

app.get('/', (req, res) => {
    logger.debug('Обработка GET /');
    res.send('Hello World!');
});

app.get('/error', (req, res) => {
    logger.warn('Сгенерирована тестовая ошибка');
    throw new Error('Тестовая ошибка');
});

app.listen(3000);
```

Пример 6. Ротация логов с winston-daily-rotate-file

bash

```
npm install winston-daily-rotate-file
```

javascript

```
const winston = require('winston');
const DailyRotateFile = require('winston-daily-rotate-file');
const path = require('path');

// Транспорт с дневной ротацией
const dailyRotateTransport = new DailyRotateFile({
    filename: path.join('logs', 'application-%DATE%.log'),
```

```
    datePattern: 'YYYY-MM-DD',
    zippedArchive: true,
    maxSize: '20m',
    maxFiles: '14d',  // хранить логи за 14 дней
    format: winston format combine(
        winston format timestamp(),
        winston format json()
    )
});

// Транспорт для ошибок (тоже с ротацией)
const errorRotateTransport = new DailyRotateFile({
    filename: path join('logs', 'error-%DATE%.log'),
    datePattern: 'YYYY-MM-DD',
    level: 'error',
    maxSize: '20m',
    maxFiles: '30d'
});

const logger = winston createLogger({
    level: 'info',
    transports: [dailyRotateTransport, errorRotateTransport]
});

// Для разработки -консоль
if (process env NODE_ENV !== 'production') {
    logger add(new winston transports Console({
        format: winston format simple()
    }));
```

```
}

module.exports = logger;
```

Пример 7. Полное приложение с логированием

```
javascript
// logger.js -модуль логирования
const winston = require('winston');
const DailyRotateFile = require('winston-daily-rotate-file');
const path = require('path');
const fs = require('fs');

const logDir = path.join(__dirname, 'logs');
if (!fs.existsSync(logDir)) {
    fs.mkdirSync(logDir, { recursive: true });
}

// Формат для JSON логов (продакшен)
const jsonFormat = winston.format.combine(
    winston.format.timestamp({ format: 'YYYY-MM-DD HH:mm:ss' }),
    winston.format.errors({ stack: true }),
    winston.format.json()
);

// Формат для читаемых логов (разработка)
const readableFormat = winston.format.combine(
    winston.format.colorize(),
    winston.format.timestamp({ format: 'YYYY-MM-DD HH:mm:ss' }),
    winston.format.printf(({ timestamp, level, message, ...meta }) => {
        let log = `${timestamp} [${level}]: ${message}`;
```

```
    if (Object.keys(meta).length > 0 && meta.stack !== undefined) {
      log += `\n${meta.stack}`;
    } else if (Object.keys(meta).length > 0) {
      log += ` ${JSON.stringify(meta)}`;
    }
    return log;
  })
);

const isProduction = process.env.NODE_ENV === 'production';

const logger = winston.createLogger({
  level: process.env.LOG_LEVEL || (isProduction ? 'info' : 'debug'),
  format: isProduction ? jsonFormat : readableFormat,
  transports: [
    // Все логи в файл
    new DailyRotateFile({
      filename: path.join(logDir, 'app-%DATE%.log'),
      datePattern: 'YYYY-MM-DD',
      maxSize: '20m',
      maxFiles: '14d',
      format: jsonFormat
    }),
    // Только ошибки в отдельный файл
    new DailyRotateFile({
      filename: path.join(logDir, 'error-%DATE%.log'),
      datePattern: 'YYYY-MM-DD',
      level: 'error',
      maxSize: '20m',
```

```
            maxFiles: '30d',
            format: jsonFormat
        })
    ]
});

// В режиме разработки -дополнительный вывод в консоль
if (!isProduction) {
    logger.add(new winston.transports.Console({
        format: readableFormat
    }));
}

module.exports = logger;
```

javascript

```
// app.js -основное приложение
const express = require('express');
const morgan = require('morgan');
const path = require('path');
const fs = require('fs');
const logger = require('./logger');

const app = express();
app.use(express.json());

// ========== НАСТРОЙКА MORGAN С ЗАПИСЬЮ В WINDSON ==========
// Создание отдельного логгера для access-логов
const accessLogger = winston.createLogger({
```

```
    format: winston format combine(
        winston format timestamp(),
        winston format json()
    ),
    transports: [
        new DailyRotateFile({
            filename: path join('logs', 'access-%DATE%.log'),
            datePattern: 'YYYY-MM-DD',
            maxSize: '20m',
            maxFiles: '14d'
        })
    ]
});

// Morgan с записью в accessLogger
app use(morgan('combined', {
    stream: {
        write: (message) => accessLogger info(message trim())
    }
}));

// В режиме разработки -дополнительный вывод morgan в консоль
if (process env NODE_ENV !== 'production') {
    app use(morgan('dev'));
}

// ========== MIDDLEWARE ДЛЯ ДЕТАЛЬНОГО ЛОГИРОВАНИЯ ЗАПРОСОВ ==========
app use((req, res, next) => {
```

```
const start = Date.now();

// Сохраняем оригинальный метод json
const originalJson = res.json;
let responseBody = null;

res.json = function(body) {
    responseBody = body;
    return originalJson.call(this, body);
};

res.on('finish', () => {
    const duration = Date.now() - start;
    const logData = {
        method: req.method,
        url: req.url,
        status: res.statusCode,
        duration: `${duration}ms`,
        ip: req.ip,
        userAgent: req.get('user-agent'),
        query: req.query,
        params: req.params
    };

    // В режиме разработки -логируем тело запроса и ответа
    if (process.env.NODE_ENV !== 'production') {
        if (Object.keys(req.body).length > 0) {
            logData.requestBody = req.body;
        }
```

```
            if (responseBody && Object keys(responseBody) length > 0) {
                logData responseBody = responseBody;
            }
        }

        const level = res statusCode >= 500 ? 'error'
                : res statusCode >= 400 ? 'warn'
                : 'info';

        logger log(level, `${req method} ${req url}`, logData);
    });

    next();
});

// ========== ОБРАБОТЧИКИ ОШИБОК С ЛОГИРОВАНИЕМ ==========
// 404 -не найден
app use((req, res) => {
    logger warn(`Маршрут не найден: ${req method} ${req url}`, {
        method: req method,
        url: req url,
        ip: req ip
    });
    res status(404) json({ error: 'Маршрут не найден' });
});

// Глобальный обработчик ошибок
app use((err, req, res, next) => {
```

```
        logger.error('Необработанная ошибка', {
            error: err.message,
            stack: err.stack,
            method: req.method,
            url: req.url,
            body: req.body,
            query: req.query,
            params: req.params,
            ip: req.ip
        });

        res.status(500).json({
            error: 'Внутренняя ошибка сервера',
            message: process.env.NODE_ENV === 'development' ? err.message : undefined
        });
    });

    // ========== ПРИМЕРЫ МАРШРУТОВ ==========
    app.get('/', (req, res) => {
        logger.info('Корневой маршрут вызван');
        res.json({ message: 'Сервер работает' });
    });

    app.get('/users/:id', (req, res) => {
        const userId = req.params.id;
        logger.debug(`Запрос пользователя ${userId}`);

        if (userId === '999') {
```

```
        logger.warn(`Пользователь ${userId} не найден`);
        return res.status(404).json({ error: 'Пользователь не найден' });
    }

    res.json({ id: userId, name: 'Иван Петров' });
});

app.post('/users', (req, res) => {
    const { name, email } = req.body;

    if (!name || !email) {
        logger.warn('Попытка создания пользователя без обязательных полей', { body: req.body });
        return res.status(400).json({ error: 'Имя и email обязательны' });
    }

    logger.info(`Создан новый пользователь: ${name} (${email})`);
    res.status(201).json({ id: Date.now(), name, email });
});

app.get('/error', (req, res) => {
    logger.error('Тестовое исключение по запросу пользователя');
    throw new Error('Тестовая ошибка!');
});

// ========== ЗАПУСК ==========
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
    logger.info(`Сервер запущен на порту ${PORT}`, {
```

```
    env: process.env.NODE_ENV || 'development',
    port: PORT
  });
});
```

2.2. Индивидуальные задания

2.2.1. Базовый уровень (15 вариантов)

Общее задание для базового уровня:

1. Настроить логирование HTTP-запросов с использованием morgan. Требования:
2. Установить и настроить morgan.
3. Выбрать и применить один из предустановленных форматов (combined, common, dev, tiny, short).
4. Настроить запись логов в файл.
5. Выводить логи также в консоль.
6. Создать директорию logs автоматически, если её нет.

Варианты форматов morgan и имён файлов:

№	Формат morgan	Имя файла лога
1	combined	access.log
2	common	requests.log
3	dev	dev.log
4	tiny	tiny.log
5	short	short.log

№	Формат morgan	Имя файла лога
6	combined	combined.log
7	common	http.log
8	dev	development.log
9	tiny	minimal.log
10	short	quick.log
11	combined	all_requests.log
12	common	server.log
13	dev	debug.log
14	tiny	requests.log
15	short	query.log

2.2.2. Средний уровень (15 вариантов)

Общее задание для среднего уровня:

1. Выполнить требования базового уровня, а также дополнительно реализовать:
2. Настройку winston для логирования ошибок и информационных сообщений.
3. Разные уровни логирования (info, warn, error, debug).
4. Логирование в файл с ротацией (winston-daily-rotate-file).
5. Логирование времени выполнения запросов.
6. Разные форматы логов для разработки и продакшена (читаемый vs JSON).

Варианты дополнительных параметров (средний уровень):

№	Дополнительный параметр	Значение
1	Максимальный размер файла	10 MB
2	Максимальный размер файла	20 MB
3	Максимальный размер файла	50 MB
4	Хранение логов (дней)	7 дней
5	Хранение логов (дней)	14 дней
6	Хранение логов (дней)	30 дней
7	Уровень логирования в dev	debug
8	Уровень логирования в dev	info

№	Дополнительный параметр	Значение
9	Уровень логирования в prod	warn
10	Уровень логирования в prod	error
11	Ротация по времени	каждый час
12	Ротация по времени	каждый день
13	Сжатие архивов	да
14	Сжатие архивов	нет
15	JSON-формат в prod	да

2.2.3. Продвинутый уровень (15 вариантов)

Общее задание для продвинутого уровня:

1. Выполнить требования среднего уровня, а также дополнительно реализовать:
2. Структурированное логирование (JSON) с rich-контекстом.
3. Логирование тела запроса и ответа (ограничение по размеру).
4. Сквозной идентификатор запроса (requestId/correlationId) для отслеживания цепочки вызовов.

5. Отдельное логирование для разных модулей приложения (с использованием child-логгеров).
6. Интеграцию с Sentry (опционально) для отслеживания ошибок.
7. Логирование производительности (медленные запросы > N мс).

Варианты продвинутых функций:

№	Дополнительная функция
1	Correlation ID (через заголовок X-Request-ID)
2	Логирование SQL-запросов (с замером времени)
3	Логирование внешних HTTP-запросов
4	Логирование с маскированием чувствительных данных (пароль, токен)
5	Отправка критических ошибок в Telegram
6	Отправка критических ошибок на email
7	Интеграция с Sentry
8	Профилирование медленных запросов (> 1 сек)
9	Логирование стека вызовов для ошибок
10	Сессионный идентификатор в логах
11	Логирование заголовков запроса
12	Логирование метрик (RPS, среднее время ответа)
13	Запись логов в базу данных (SQLite)

№	Дополнительная функция
14	Запись логов в Elasticsearch
15	Логирование с цветовой разметкой для терминала

Пример с Correlation ID:

javascript

```
const crypto = require('crypto');

// Middleware для генерации/передачи Correlation ID
app.use((req, res, next) => {
  const correlationId = req.headers['x-request-id'] || crypto.randomBytes(16).toString('hex');
  req.correlationId = correlationId;
  res.setHeader('X-Request-ID', correlationId);

  // Создание child-логгера с correlationId
  req.logger = logger.child({ correlationId });
  next();
});

// Использование
app.get('/users', (req, res) => {
  req.logger.info('Запрос списка пользователей');
  // ...
});
```

Пример логирования с маскированием:

javascript

```
function maskSensitive(data) {
```

```
  if (!data) return data;
  const masked = { ...data };
  const sensitiveFields = ['password', 'token', 'secret', 'authorization'];
  for (const field of sensitiveFields) {
    if (masked[field]) {
      masked[field] = '***MASKED***';
    }
  }
  return masked;
}

// В middleware логирования
logData.requestBody = maskSensitive(req.body);
```

3. Контрольные вопросы

1. Какие уровни логирования существуют и для чего используется каждый?
2. Какую роль выполняет morgan в Express-приложении?
3. Какие предустановленные форматы есть у morgan?
4. Как создать кастомный формат логов в morgan?
5. Как записывать логи morgan в файл, а не только в консоль?
6. Какую библиотеку можно использовать для более гибкого логирования, чем morgan?
7. Как настроить ротацию лог-файлов?
8. Что такое структурированное логирование и в чём его преимущества?
9. Как логировать ошибки с полным стеком вызовов?
10. Как настроить разные уровни логирования для разработки и продакшена?

11. Зачем нужен сквозной идентификатор запроса (correlation ID)?
12. Как избежать логирования чувствительных данных (паролей, токенов)?
13. Что такое транспорты в winston?
14. Как логировать время выполнения запросов?
15. Как интегрировать логирование с внешними системами (Sentry, Telegram)?

4. Требования к отчёту

Отчёт должен быть оформлен в электронном виде и содержать:

Титульный лист с названием работы, ФИО студента, номером группы, вариантом.

Цель и задачи работы (переписать из методических указаний).

Теоретическое обоснование (краткий конспект по теме).

Ход выполнения работы:

Листинг кода настройки логирования.

Скриншоты:

Вывод логов в консоль при запуске сервера

Логирование GET-запросов

Логирование POST-запросов с телом

Логирование ошибок (404, 500)

Содержание файлов логов

(Средний уровень) Ротация логов

(Продвинутый уровень) Correlation ID, маскирование

Ответы на контрольные вопросы (письменно, развёрнуто).

Заключение (выводы).

Список использованных источников.

5. Критерии оценивания

Уровень выполнения	Требования	Оценка
Базовый	Настроен morgan, логгирование в файл и консоль. Автоматическое создание директории logs.	Удовлетворительно
Средний	Выполнены требования базового уровня + winston, уровни логирования, ротация файлов, разные форматы для dev/prod.	Хорошо
Продвинутый	Выполнены требования среднего уровня + структурированное логирование (JSON), correlation ID, маскирование чувствительных данных, интеграция с внешними системами (опционально).	Отлично

Дополнительные условия:

1. Логи должны быть информативными и содержать контекст.
2. При защите студент должен объяснить настройку логирования и форматы логов.

6. Список рекомендуемой литературы и интернет-ресурсов

1. Литература (доступна в РФ)
2. Браун, И. (2021). *Express.js: Разработка веб-приложений на Node.js*. ДМК Пресс.
3. Кантелон, М., Хартер, М., & Холстейн, С. (2020). *Node.js в действии*. Питер.

Интернет-ресурсы

Название	Ссылка	Описание
morgan npm	https://www.npmjs.com/package/morgan	Документация morgan
winston npm	https://www.npmjs.com/package/winston	Документация winston
winston-daily-rotate-file	https://www.npmjs.com/package/winston-daily-rotate-file	Ротация логов
Express logging	https://expressjs.com/ru/guide/debugging.html	Официальное руководство
Логирование в Node.js	https://nodejs.org/api/console.html	Документация console
12 Factor Logs	https://12factor.net/ru/logs	Обработка логов

Все ресурсы доступны и легальны на территории Российской Федерации.

Лабораторная работа № 15

Тема: Тестирование API с помощью Postman

Цель работы:

Научиться проводить тестирование REST API с использованием Postman, создавать коллекции тестов, писать автоматизированные тесты на JavaScript в Postman и анализировать результаты тестирования.

Задачи работы:

1. Освоить интерфейс Postman и основные функции.
2. Создать коллекцию запросов для тестирования API.
3. Написать автоматизированные тесты на JavaScript для проверки ответов API.
4. Настроить проверки статус-кодов, структуры ответа и данных.
5. Использовать переменные окружения Postman для параметризации тестов.
6. Настроить запуск коллекции через командную строку (Newman).
7. Сформировать отчёт о результатах тестирования.

1. Теоретическое обоснование

1.1. Что такое Postman?

Postman - это популярный инструмент для тестирования API, который позволяет:

Отправлять HTTP-запросы различных типов (GET, POST, PUT, DELETE, PATCH).

Просматривать и анализировать ответы сервера.

Создавать коллекции запросов для организации тестов.

Писать автоматизированные тесты на JavaScript.

Использовать переменные и окружения для параметризации.

Генерировать документацию API.

Запускать тесты через командную строку с помощью Newman.

1.2. Коллекции Postman

Коллекция - это группа связанных запросов, объединённых общей целью (например, тестирование определённого API). Коллекции позволяют:

Организовывать запросы по папкам.

Переиспользовать переменные.

Наследовать авторизацию и заголовки.

Запускать все тесты коллекции одной кнопкой.

1.3. Написание тестов в Postman

Postman использует JavaScript для написания тестов, которые выполняются после получения ответа сервера.

Базовый синтаксис:

javascript

```
// Проверка статус-кода
pm test("Статус-код 200", function () {
    pm response to have status(200);
});

// Проверка наличия поля в ответе
pm test("Ответ содержит поле 'id'", function () {
    const jsonData = pm response json();
    pm expect(jsonData) to have property('id');
});

// Проверка значения поля
pm test("ID пользователя = 1", function () {
    const jsonData = pm response json();
    pm expect(jsonData id) to eql(1);
});
```

1.4. Переменные в Postman

Postman поддерживает несколько уровней переменных:

Уровень	Описание	Пример
Глобальные	Доступны во всех коллекциях	pm.globals.set("key", "value")
Коллекции	Доступны в рамках коллекции	pm.collectionVariables.set("key", "value")
Окружения	Зависят от выбранного окружения	pm.environment.set("key", "value")
Локальные	Только для текущего запроса	pm.variables.set("key", "value")

Использование переменных в запросах:

{{variable_name}} -в URL, заголовках, теле запроса

pm.variables.get("variable_name") -в скриптах

1.5. Автоматизация с Newman

Newman - это командная строка для запуска коллекций Postman.

Установка:

bash

npm install -g newman

Запуск коллекции:

bash

newman run collection.json -e environment.json

Генерация отчёта:

```
bash
newman run collection.json -r html,json,junit
```

1.6. Типичные тесты для API

Тип теста	Описание	Пример
Статус-код	Проверка кода ответа	pm.response.to.have.status(200)
Время ответа	Проверка производительности	pm.expect(pm.response.responseTime).to.be.below(500)
Наличие заголовка	Проверка Content-Type	pm.response.to.have.header("Content-Type")
Структура JSON	Проверка наличия полей	pm.expect(data).to.have.all.keys("id", "name")
Типы данных	Проверка типов полей	pm.expect(data.id).to.be.a('number')
Значения	Проверка конкретных значений	pm.expect(data.name).to.eql("Иван")
Схема JSON	Валидация по схеме	Использование tv4 или ajv

2. Методика выполнения работы

2.1. Примеры и их подробный разбор

Пример 1. Создание коллекции и первого теста

Шаг 1. Создание коллекции в Postman

Нажать на кнопку "New" → "Collection"

Назвать коллекцию "API Tests"

Создать папки: "Users", "Products", "Auth"

Шаг 2. Создание запроса GET /users

Нажать "New" → "Request"

Метод: GET

URL: http://localhost:3000/api/users

Сохранить в папку "Users"

Шаг 3. Написание теста для GET /users

javascript

```
// Тест: статус-код 200
pm test("Status code is 200", function () {
    pm response to have status(200);
});

// Тест: тело ответа -массив
pm test("Response body is an array", function () {
    const responseData = pm response json();
    pm expect(responseData) to be an('array');
});

// Тест: каждый элемент имеет id и name
pm test("Each user has id and name", function () {
    const responseData = pm response json();
    pm expect(responseData length) to be greaterThan(0);
```

```
    responseData.forEach(user => {
        pm.expect(user).to.have.property('id');
        pm.expect(user).to.have.property('name');
        pm.expect(user.id).to.be.a('number');
        pm.expect(user.name).to.be.a('string');
    });
});
```

Пример 2. Сохранение данных из ответа в переменные

```
javascript
// GET /users -сохранение первого пользователя
const jsonData = pm.response.json();
if (jsonData.length > 0) {
    pm.collectionVariables.set("userId", jsonData[0].id);
    pm.collectionVariables.set("userName", jsonData[0].name);
    console.log(`Сохранён пользователь: ID=${jsonData[0].id}, Name=${jsonData[0].name}`);
}
```

Использование в следующем запросе:

URL: http://localhost:3000/api/users/{{userId}}

Пример 3. Тестирование POST запроса (создание)

```
javascript
// Запрос POST /api/users
// Тело запроса (raw JSON):
// {
//     "name": "Новый пользователь",
//     "email": "newuser@example.com"
// }

// Тест: статус-код 201 (Created)
```

```
pm.test("Status code is 201", function () {
    pm.response.to.have.status(201);
});

// Тест: заголовок Content-Type application/json
pm.test("Content-Type is JSON", function () {
    pm.response.to.have.header("Content-Type");
    pm.expect(pm.response.headers.get("Content-Type")).to.include("application/json");
});

// Тест: тело ответа содержит id, name, email
pm.test("Response contains id, name, email", function () {
    const responseData = pm.response.json();
    pm.expect(responseData).to.have.property('id');
    pm.expect(responseData).to.have.property('name');
    pm.expect(responseData).to.have.property('email');
});

// Тест: значения полей соответствуют отправленным
const reqData = pm.request.body.raw ? JSON.parse(pm.request.body.raw) : {};
pm.test("Response fields match request", function () {
    const responseData = pm.response.json();
    if (reqData.name) pm.expect(responseData.name).to.eql(reqData.name);
    if (reqData.email) pm.expect(responseData.email).to.eql(reqData.email);
});

// Сохранение созданного ID
```

```
pm collectionVariables set("createdUserId", pm response json() id);
```

Пример 4. Тестирование GET /users/:id

javascript

```
// URL: http://localhost:3000/api/users/{{userId}}

pm test("Status code is 200", function () {
    pm response to have status(200);
});

pm test("Response contains user with correct ID", function () {
    const responseData = pm response json();
    const expectedId = parseInt(pm collectionVariables get("userId"));
    pm expect(responseData id) to eql(expectedId);
});

pm test("Response has required fields", function () {
    const responseData = pm response json();
    pm expect(responseData) to have all keys('id', 'name', 'email', 'created_at');
});
```

Пример 5. Тестирование PUT запроса (обновление)

javascript

```
// PUT /api/users/{{userId}}
// Тело запроса: { "name": "Обновлённое имя" }

pm test("Status code is 200", function () {
    pm response to have status(200);
});

pm test("Name was updated", function () {
```

```
    const responseData = pm.response.json();
    const reqData = JSON.parse(pm.request.body.raw);
    pm.expect(responseData.name).to.eql(reqData.name);
});

// Проверка, что ID не изменился
pm.test("ID remained the same", function () {
    const responseData = pm.response.json();
    const originalId = parseInt(pm.collectionVariables.get("userId"));
    pm.expect(responseData.id).to.eql(originalId);
});
```

Пример 6. Тестирование DELETE запроса

javascript

```
// DELETE /api/users/{{userId}}

pm.test("Status code is 204 (No Content)", function () {
    pm.response.to.have.status(204);
});

// Проверка, что тело ответа пустое
pm.test("Response body is empty", function () {
    pm.expect(pm.response.text()).to.be.empty;
});
```

Пример 7. Настройка окружения Postman

Создание окружения:

Нажать "Environments" → "New Environment"

Название: "Development"

Добавить переменные:

Переменная	Значение
baseUrl	http://localhost:3000
apiVersion	v1

Использование в запросах:

URL: {{baseUrl}}/api/{{apiVersion}}/users

Создание окружения "Production":

Переменная	Значение
baseUrl	https://api.example.com
apiVersion	v1

Пример 8. Написание тестов с использованием схемы JSON

bash

```
npm install -g ajv-cli
```

В Postman (используя библиотеку Ajv):

javascript

```
// Вкладка Pre-request Script
const Ajv = require('ajv');
const ajv = new Ajv();

// Схема для пользователя
const userSchema = {
    type: "object",
    required: ["id", "name", "email"],
    properties: {
        id: { type: "integer" },
        name: { type: "string", minLength: 2, maxLength: 100 },
```

```
        email: { type: "string", format: "email" },
        created_at: { type: "string" }
    }
};

// Тест: валидация по схеме
pm.test("Response matches schema", function () {
    const responseData = pm.response.json();
    const validate = ajv.compile(userSchema);
    const isValid = validate(responseData);
    pm.expect(isValid).to.be.true;
    if (!isValid) console.log(validate.errors);
});
```

Пример 9. Комплексный тест с цепочкой запросов

```
javascript
// Тест: полный CRUD-сценарий

// 1. Создание пользователя (POST)
pm.test("Step 1: Create user", function () {
    pm.response.to.have.status(201);
    const userId = pm.response.json().id;
    pm.collectionVariables.set("userId", userId);
});

// 2. Получение пользователя (GET)
pm.test("Step 2: Get user by ID", function () {
    pm.response.to.have.status(200);
    const userId = parseInt(pm.collectionVariables.get("userId"));
    pm.expect(pm.response.json().id).to.eql(userId);
```

```
});

// 3. Обновление пользователя (PUT)
pm.test("Step 3: Update user", function () {
    pm.response.to.have.status(200);
    const newName = JSON.parse(pm.request.body.raw).name;
    pm.expect(pm.response.json().name).to.eql(newName);
});

// 4. Удаление пользователя (DELETE)
pm.test("Step 4: Delete user", function () {
    pm.response.to.have.status(204);
});

// 5. Проверка, что пользователь удалён (GET)
pm.test("Step 5: Verify user is deleted", function () {
    pm.response.to.have.status(404);
});
```

Пример 10. Тестирование времени ответа

javascript

```
pm.test("Response time is less than 200ms", function () {
    pm.expect(pm.response.responseTime).to.be.below(200);
});

// Для медленных эндпоинтов (генерация отчёта)
pm.test("Response time is acceptable", function () {
    const maxTime = pm.collectionVariables.get("maxResponseTime") ||
1000;
    pm.expect(pm.response.responseTime).to.be.below(maxTime);
```

```
});
```

Пример 11. Интеграция с Newman (командная строка)

Экспорт коллекции и окружения:

В Postman: нажать "..." на коллекции → "Export"

Сохранить файл API_Tests.postman_collection.json

Экспортировать окружение: "..." → "Export" → Development.postman_environment.json

Запуск тестов через Newman:

bash

```
# Установка Newman
npm install -g newman

# Запуск коллекции
newman run API_Tests.postman_collection.json -e Development.postman_environment.json

# Запуск с генерацией отчёта HTML
newman run API_Tests.postman_collection.json -e Development.postman_environment.json -r html

# Запуск с несколькими отчётами
newman run API_Tests.postman_collection.json -e Development.postman_environment.json -r html,json,junit

# Запуск с итерациями (повторить 5 раз)
newman run API_Tests.postman_collection.json -n 5

# Запуск с папкой
newman run API_Tests.postman_collection.json --folder Users
```

Пример 12. Полный пример коллекции тестов

javascript

```
// pre-request скрипт (выполняется перед каждым запросом)
// Установка временной метки
pm.variables.set("timestamp", Date.now());

// Установка случайного имени
pm.variables.set("randomName", "User_" + Math.floor(Math.random() * 10000));

// request скрипт (если нужен)

// tests скрипт
// ========== ОБЩИЕ ТЕСТЫ ДЛЯ ВСЕХ ЗАПРОСОВ ==========
pm.test("Статус-код 2xx или 4xx", function () {
    pm.expect(pm.response.code).to.be.within(200, 499);
});

pm.test("Время ответа < 1 секунда", function () {
    pm.expect(pm.response.responseTime).to.be.below(1000);
});

pm.test("Content-Type содержит JSON", function () {
    pm.response.to.have.header("Content-Type");
    pm.expect(pm.response.headers.get("Content-Type")).to.include("application/json");
});

// ========== ТЕСТЫ ДЛЯ GET /api/users ==========
```

```
const usersTests = {
  "GET /api/users": () => {
    pm.test("Тело ответа -массив", () => {
      pm.expect(pm.response.json()).to.be.an('array');
    });

    pm.test("Массив не пустой", () => {
      pm.expect(pm.response.json().length).to.be.greaterThan(0);
    });

    pm.test("Каждый элемент имеет id, name, email", () => {
      const users = pm.response.json();
      users.forEach(user => {
        pm.expect(user).to.have.all.keys('id', 'name', 'email');
      });
    });

    // Сохранение ID первого пользователя
    const userId = pm.response.json()[0]?.id;
    if (userId) {
      pm.collectionVariables.set("userId", userId);
      console.log(`Сохранён userId: ${userId}`);
    }
  },

  "GET /api/users/:id": () => {
    pm.test("Статус-код 200 для существующего пользователя", () => {
      pm.response.to.have.status(200);
    });
```

```
    pm.test("ID совпадает с запрошенным", () => {
        const expectedId = parseInt(pm.variables.get("userId"));
        pm.expect(pm.response.json().id).to.eql(expectedId);
    });
},

"POST /api/users": () => {
    pm.test("Статус-код 201 Created", () => {
        pm.response.to.have.status(201);
    });

    pm.test("Ответ содержит Location заголовок", () => {
        pm.response.to.have.header("Location");
    });

    const responseData = pm.response.json();
    pm.collectionVariables.set("createdUserId", responseData.id);
},

"DELETE /api/users/:id": () => {
    pm.test("Статус-код 204", () => {
        pm.response.to.have.status(204);
    });

    pm.test("Тело ответа пустое", () => {
        pm.expect(pm.response.text()).to.be.empty;
    });
}
```

};

2.2. Индивидуальные задания

2.2.1. Базовый уровень (15 вариантов)

Общее задание для базового уровня:

Создать коллекцию Postman для тестирования существующего API (из предыдущих лабораторных работ). Требования:

1. Создать коллекцию с 5 запросами.
2. Написать минимум 3 теста на JavaScript для каждого запроса.
3. Обязательные тесты:
4. Проверка статус-кода
5. Проверка наличия обязательных полей в ответе
6. Проверка типа данных полей
7. Сохранить коллекцию в файл.
8. Варианты API для тестирования:

	API (из предыдущих лабораторных работ)	Эндпоинты для тестирования
	Books API	GET /books, GET /books/:id, POST /books, PUT /books/:id, DELETE /books/:id
	Products API	GET /products, GET /products/:id, POST /products, PUT /products/:id, DELETE /products/:id
	Students API	GET /students, GET /students/:id, POST /students, PUT /students/:id, DELETE /students/:id

	API (из предыдущих лабораторных работ)	Эндпоинты для тестирования
	Movies API	GET /movies, GET /movies/:id, POST /movies, PUT /movies/:id, DELETE /movies/:id
	Employees API	GET /employees, GET /employees/:id, POST /employees, PUT /employees/:id, DELETE /employees/:id
	Recipes API	GET /recipes, GET /recipes/:id, POST /recipes, PUT /recipes/:id, DELETE /recipes/:id
	Cars API	GET /cars, GET /cars/:id, POST /cars, PUT /cars/:id, DELETE /cars/:id
	Songs API	GET /songs, GET /songs/:id, POST /songs, PUT /songs/:id, DELETE /songs/:id
	Hotels API	GET /hotels, GET /hotels/:id, POST /hotels, PUT /hotels/:id, DELETE /hotels/:id
0	Courses API	GET /courses, GET /courses/:id, POST /courses, PUT

	API (из предыдущих лабораторных работ)	Эндпоинты для тестирования
		/courses/:id, DELETE /courses/:id
1	Medicines API	GET /medicines, GET /medicines/:id, POST /medicines, PUT /medicines/:id, DELETE /medicines/:id
2	Events API	GET /events, GET /events/:id, POST /events, PUT /events/:id, DELETE /events/:id
3	Orders API	GET /orders, GET /orders/:id, POST /orders, PUT /orders/:id, DELETE /orders/:id
4	Articles API	GET /articles, GET /articles/:id, POST /articles, PUT /articles/:id, DELETE /articles/:id
5	Athletes API	GET /athletes, GET /athletes/:id, POST /athletes, PUT /athletes/:id, DELETE /athletes/:id

2.2.2. Средний уровень (15 вариантов)

Общее задание для среднего уровня:

1. Выполнить требования базового уровня, а также дополнительно реализовать:

2. Использование переменных окружения (dev, prod).
3. Сохранение данных между запросами (например, созданный ID используется в следующих запросах).
4. Тесты для сценария E2E (создание → чтение → обновление → удаление).
5. Проверка времени ответа (максимальное время -по варианту).
6. Тесты на граничные случаи (404 для несуществующего ресурса, 400 для невалидных данных).

Варианты максимального времени ответа (средний уровень):

	Максимальное время (мс)		Максимальное время (мс)
	100		500
	150	0	600
	200	1	700
	250	2	800
	300	3	900
	350	4	1000
	400		1200

	Максимальное время (мс)		Максимальное время (мс)
		5	
	450		

Пример E2E-сценария:

javascript

```
// 1. POST /items -создание
pm.test("Создание ресурса", () => {
    pm.response.to.have.status(201);
    pm.collectionVariables.set("itemId", pm.response.json().id);
});

// 2. GET /items/:id -проверка создания
pm.test("Проверка созданного ресурса", () => {
    pm.response.to.have.status(200);

pm.expect(pm.response.json().id).to.eql(parseInt(pm.collectionVariables.get("itemId")));
});

// 3. PUT /items/:id -обновление
pm.test("Обновление ресурса", () => {
    pm.response.to.have.status(200);
});

// 4. GET /items/:id -проверка обновления
pm.test("Проверка обновления", () => {
```

```
    const newData = JSON.parse(pm.request.body.raw);
    pm.expect(pm.response.json().name).to.eql(newData.name);
});

// 5. DELETE /items/:id -удаление
pm.test("Удаление ресурса", () => {
    pm.response.to.have.status(204);
});

// 6. GET /items/:id -проверка удаления (должен вернуть 404)
pm.test("Проверка удаления", () => {
    pm.response.to.have.status(404);
});
```

2.2.3. Продвинутый уровень (15 вариантов)

Общее задание для продвинутого уровня:

Выполнить требования среднего уровня, а также дополнительно реализовать:

1. JSON Schema validation для проверки структуры ответа.
2. Параметризованные тесты (один тест на несколько наборов данных).
3. Интеграцию с Newman и генерацию отчётов (HTML, JSON, JUnit).
4. Тесты на безопасность (проверка заголовков безопасности, rate limiting).
5. Автоматизацию через CI/CD (GitHub Actions) -создание workflow для запуска тестов.
6. Нагрузочное тестирование с использованием Postman Runner.

Варианты продвинутых функций:

№	Дополнительная функция
1	JSON Schema validation (ajv)
2	Параметризованные тесты (data-driven)
3	Интеграция с Newman + HTML-отчёт
4	Интеграция с Newman + JUnit-отчёт
5	GitHub Actions workflow
6	GitLab CI pipeline
7	Тесты безопасности (XSS, SQL injection)
8	Автоматическая генерация документации через Postman
9	Тестирование авторизации (Basic Auth, JWT)
10	Тестирование rate limiting
11	Нагрузочное тестирование (100 итераций)
12	Экспорт коллекции в OpenAPI/Swagger
13	Интеграция с мониторингом (Postman Monitors)
14	Тесты на отрицательные сценарии (bad request)
15	Динамическое создание тестовых данных (pre-request)

Пример JSON Schema валидации:

javascript

```
// pre-request: загрузка библиотеки Ajv (требуется установка через Postman Console)
// Или использование встроенного валидатора

const userSchema = {
    "$schema": "http://json-schema.org/draft-07/schema#",
    "type": "object",
    "required": ["id", "name", "email"],
    "properties": {
        "id": { "type": "integer", "minimum": 1 },
        "name": { "type": "string", "minLength": 2, "maxLength": 100 },
        "email": { "type": "string", "format": "email" },
        "created_at": { "type": "string", "format": "date-time" },
        "role": { "type": "string", "enum": ["user", "admin"] }
    },
    "additionalProperties": false
};

pm.test("Response matches JSON Schema", () => {
    const responseData = pm.response.json();
    const isValid = tv4.validate(responseData, userSchema);
    pm.expect(isValid).to.be.true;
    if (!isValid) console.log(tv4.error);
});
```

Пример GitHub Actions workflow:

yaml

```
# .github/workflows/api-tests.yml
name: API Tests
```

```yaml
on:
  push:
    branches: [ main, develop ]
  pull_request:
    branches: [ main ]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3

    - name: Setup Node.js
      uses: actions/setup-node@v3
      with:
        node-version: '18'

    - name: Install Newman
      run: npm install -g newman

    - name: Start API Server
      run: |
        npm install
        npm start &
        sleep 5

    - name: Run API Tests
```

```
run: |
  newman run collections/API_Tests.json \
    -e environments/Development.json \
    -r html,json,junit \
    --reporter-html-export reports/html-report.html \
    --reporter-json-export reports/json-report.json \
    --reporter-junit-export reports/junit-report.xml

- name: Upload Test Reports
  uses: actions/upload artifact@v3
  with:
    name: test reports
    path: reports/
```

3. Контрольные вопросы

1. Что такое Postman и для чего он используется?
2. Как создать коллекцию запросов в Postman?
3. Какие типы переменных поддерживаются в Postman?
4. Как написать тест в Postman на JavaScript?
5. Как проверить статус-код ответа в Postman-тесте?
6. Как проверить структуру JSON-ответа?
7. Как сохранить значение из ответа в переменную?
8. Как использовать переменные в URL и теле запроса?
9. Что такое Newman и для чего он нужен?
10. Как запустить коллекцию тестов через Newman?
11. Какие форматы отчётов поддерживает Newman?
12. Что такое JSON Schema и как её использовать в Postman?
13. Как организовать E2E-тестирование сценария CRUD?
14. Как настроить тесты на производительность (время ответа)?

15.Как интегрировать Postman тесты в CI/CD?

4. Требования к отчёту

Отчёт должен быть оформлен в электронном виде и содержать:

Титульный лист с названием работы, ФИО студента, номером группы, вариантом.

Цель и задачи работы (переписать из методических указаний).

Теоретическое обоснование (краткий конспект по теме).

Ход выполнения работы:

Скриншот коллекции Postman.

Листинг тестов для каждого эндпоинта.

Скриншоты результатов выполнения тестов (успешное прохождение).

Скриншот настроек переменных окружения.

(Средний уровень) Скриншот результатов E2E-сценария.

(Продвинутый уровень) Команда запуска Newman и сгенерированный отчёт.

Ответы на контрольные вопросы (письменно, развёрнуто).

Заключение (выводы).

Список использованных источников.

5. Критерии оценивания

Уровень выполнения	Требования	Оценка
Базовый	Создана коллекция Postman с 5 запросами, написаны 3+ теста на каждый запрос (статус, поля, типы).	Удовлетвори тельно
Средни	Выполнены требования	Хорошо

Уровень выполнения	Требования	Оценка
й	базового уровня + переменные окружения, E2E-сценарий (create-read-update-delete), проверка времени ответа, тесты на граничные случаи.	
Продвинутый	Выполнены требования среднего уровня + JSON Schema валидация, интеграция с Newman, генерация отчётов, CI/CD (GitHub Actions), безопасность/нагрузка.	Отлично

Дополнительные условия:

1. Все тесты должны проходить успешно на корректных данных.
2. При защите студент должен объяснить написанные тесты.

6. Список рекомендуемой литературы и интернет-ресурсов

Литература (доступна в РФ)

1. Браун, И. (2021). *Express.js: Разработка веб-приложений на Node.js*. ДМК Пресс.
2. Кантелон, М., Хартер, М., & Холстейн, С. (2020). *Node.js в действии*. Питер.

Интернет-ресурсы

Название	Ссылка	Описание
Postman	https://learning.postman.com/	Официальная

Название	Ссылка	Описание
Learning Center		документация Postman
Newman	https://learning.postman.com/docs/running-collections/using-newman-cli/command-line-integration-with-newman/	CLI для Postman
JSON Schema	https://json-schema.org/	Спецификация JSON Schema
GitHub Actions	https://docs.github.com/ru/actions	Документация CI/CD
tv4 validator	https://github.com/geraintluff/tv4	JSON Schema валидатор
Postman Sandbox API	https://learning.postman.com/docs/writing-scripts/script-references/postman-sandbox-api/	API для скриптов

Все ресурсы доступны и легальны на территории Российской Федерации.

Лабораторная работа № 16

Тема: Интеграция простого фронтенда с бэкендом

Цель работы:

Научиться связывать статическую HTML-страницу с бэкенд-API через fetch, реализовать динамическое отображение данных и отправку данных на сервер без перезагрузки страницы.

Задачи работы:

1. Создать простую HTML/CSS страницу для взаимодействия с API.
2. Освоить использование fetch для выполнения HTTP-запросов к API.
3. Реализовать динамическое отображение данных из API на странице.
4. Реализовать форму для создания новых записей через POST-запрос.
5. Реализовать форму для обновления и удаления записей.
6. Настроить обработку ошибок и отображение уведомлений пользователю.
7. Настроить CORS на бэкенде для взаимодействия с фронтендом.

1. Теоретическое обоснование

1.1. Взаимодействие фронтенда и бэкенда

В современной веб-разработке фронтенд (клиентская часть) и бэкенд (серверная часть) часто разделены:

1. Фронтенд работает в браузере пользователя.
2. Бэкенд предоставляет REST API.
3. Взаимодействие происходит через HTTP-запросы.
4. Преимущества такого подхода:
5. Независимая разработка и развёртывание.
6. Возможность использовать один бэкенд для разных клиентов (веб, мобильное приложение).
7. Более чистая архитектура.

1.2. Fetch API

fetch — это современный JavaScript API для выполнения HTTP-запросов, встроенный в современные браузеры.

Базовый синтаксис:

javascript

```
fetch(url, {
    method: 'GET',          // HTTP-метод
    headers: {              // Заголовки
        'Content-Type': 'application/json'
    },
    body: JSON.stringify(data)  // Тело запроса (для POST/PUT)
})
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.error('Ошибка:', error));
```

1.3. Асинхронность с async/await

Для более удобной работы с fetch используется синтаксис async/await:

javascript

```
async function getData() {
    try {
        const response = await fetch('/api/items');
        if (!response.ok) throw new Error('Ошибка загрузки');
        const data = await response.json();
        console.log(data);
    } catch (error) {
        console.error(error);
    }
```

```
}
```

1.4. CORS (Cross-Origin Resource Sharing)

Когда фронтенд и бэкенд работают на разных портах или доменах, браузер блокирует запросы по соображениям безопасности. Для решения этой проблемы используется CORS.

Настройка CORS на бэкенде (Express):

bash

```
npm install cors
```

javascript

```
const cors = require('cors');
app.use(cors());  // Разрешить все источники

// Более строгая настройка
app.use(cors({
    origin: 'http://localhost:3001',  // Только с этого источника
    methods: ['GET', 'POST', 'PUT', 'DELETE'],
    allowedHeaders: ['Content-Type', 'Authorization']
}));
```

1.5. Динамическое обновление DOM

После получения данных от API, их нужно отобразить на странице:

javascript

```
function displayItems(items) {
    const container = document.getElementById('items-container');
    container.innerHTML = '';  // Очистка

    items.forEach(item => {
        const itemElement = document.createElement('div');
```

```
        itemElement.className = 'item';
        itemElement.innerHTML = `
            <span>${item.name}</span>
            <button onclick="editItem(${item.id})">✏</button>
            <button onclick="deleteItem(${item.id})">🗑</button>
        `;
        container.appendChild(itemElement);
    });
}
```

1.6. Обработка форм

Отправка данных из формы без перезагрузки страницы:

javascript

```
document.getElementById('create-form').addEventListener('submit', async (e) => {
    e.preventDefault();  // Отмена перезагрузки страницы

    const formData = new FormData(e.target);
    const data = Object.fromEntries(formData.entries());

    try {
        const response = await fetch('/api/items', {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify(data)
        });

        if (response.ok) {
            alert('Создано успешно!');
```

```
            e target reset();  // Очистка формы
            loadItems();       // Перезагрузка списка
        }
    } catch (error) {
        alert('Ошибка: ' + error message);
    }
});
```

2. Методика выполнения работы

2.1. Примеры и их подробный разбор

Пример 1. Базовый HTML шаблон

html

```
<!DOCTYPE html>
<html lang="ru">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Управление книгами</title>
    <style>
        * {
            margin: 0;
            padding: 0;
            box-sizing: border-box;
        }

        body {
            font-family: Arial, sans-serif;
            background: #f5f5f5;
```

```
    padding: 20px;
}

.container {
    max-width: 1200px;
    margin: 0 auto;
}

h1 {
    color: #333;
    margin-bottom: 20px;
}

.section {
    background: white;
    border-radius: 8px;
    padding: 20px;
    margin-bottom: 20px;
    box-shadow: 0 2px 4px rgba(0,0,0,0.1);
}

.section h2 {
    margin-bottom: 15px;
    color: #555;
}

form {
    display: flex;
    flex-direction: column;
```

```
    gap: 10px;
}

input, textarea, select {
    padding: 10px;
    border: 1px solid #ddd;
    border-radius: 4px;
    font-size: 14px;
}

button {
    padding: 10px 20px;
    background: #007bff;
    color: white;
    border: none;
    border-radius: 4px;
    cursor: pointer;
    font-size: 14px;
}

button:hover {
    background: #0056b3;
}

button.delete {
    background: #dc3545;
}

button.delete:hover {
```

```
    background: #a71d2a;
}

table {
    width: 100%;
    border-collapse: collapse;
}

th, td {
    padding: 12px;
    text-align: left;
    border-bottom: 1px solid #ddd;
}

th {
    background: #007bff;
    color: white;
}

.error {
    color: #dc3545;
    padding: 10px;
    background: #ffebee;
    border-radius: 4px;
    margin-bottom: 10px;
}

.success {
    color: #28a745;
```

```
            padding: 10px;
            background: #d4edda;
            border-radius: 4px;
            margin-bottom: 10px;
        }

        .loading {
            text-align: center;
            padding: 20px;
            color: #666;
        }
    </style>
</head>
<body>
    <div class="container">
        <h1>📚 Управление библиотекой</h1>

        <div id="message"></div>

        <!-- Форма создания -->
        <div class="section">
            <h2>➕ Добавить книгу</h2>
            <form id="create-form">
                <input type="text" name="title" placeholder="Название" required>
                <input type="text" name="author" placeholder="Автор" required>
                <input type="number" name="year" placeholder="Год издания">
                <input type="number" name="price" step="0.01" placeholder="Цена">
```

```
            <button type="submit">Добавить книгу</button>
        </form>
    </div>

    <!-- Таблица книг -->
    <div class="section">
        <h2>📖 Список книг</h2>
        <div id="books-container">
            <div class="loading">Загрузка...</div>
        </div>
    </div>
</div>

<script src="app.js"></script>
</body>
</html>
```

Пример 2. JavaScript для работы с API

javascript

```
// app.js

// API URL (настроить под свой сервер)
const API_URL = 'http://localhost:3000/api/books';

// Показать сообщение пользователю
function showMessage(text  type = 'error') {
    const messageDiv = document getElementById('message');
    messageDiv innerHTML = `<div class="${type}">${text}</div>`;
    setTimeout(() => {
        messageDiv.innerHTML = '';
```

```
    }, 3000);
}

// Загрузка всех книг
async function loadBooks() {
    const container = document.getElementById('books-container');
    container.innerHTML = '<div class="loading">Загрузка...</div>';

    try {
        const response = await fetch(API_URL);

        if (!response.ok) {
            throw new Error(`Ошибка HTTP: ${response.status}`);
        }

        const books = await response.json();
        displayBooks(books);
    } catch (error) {
        container.innerHTML = `<div class="error">Ошибка: ${error.message}</div>`;
        console.error('Ошибка загрузки:', error);
    }
}

// Отображение книг в таблице
function displayBooks(books) {
    const container = document.getElementById('books-container');

    if (!books || books.length === 0) {
```

```
            container.innerHTML = '<div class="loading">Нет книг. Добавьте первую!</div>';
            return;
        }

        const table = document.createElement('table');
        table.innerHTML = `
            <thead>
                <tr>
                    <th>ID</th><th>Название</th><th>Автор</th><th>Год</th><th>Цена</th><th>Действия</th>
                </tr>
            </thead>
            <tbody id="books-table-body"></tbody>
        `;

        const tbody = table.querySelector('#books-table-body');

        books.forEach(book => {
            const row = document.createElement('tr');
            row.innerHTML = `
                <td>${book.id}</td>
                <td><span id="title-${book.id}">${escapeHtml(book.title)}</span></td>
                <td><span id="author-${book.id}">${escapeHtml(book.author)}</span></td>
                <td><span id="year-${book.id}">${book.year || '—'}</span></td>
```

```
                <td><span id="price-${book.id}">${book.price ? book.price + ' ₽' : '—'}</span></td>
                <td>
                    <button onclick="editBook(${book.id})" style="margin-right:5px">✏</button>
                    <button onclick="deleteBook(${book.id})" class="delete">🗑</button>
                </td>
            `;
            tbody.appendChild(row);
        });

        container.innerHTML = '';
        container.appendChild(table);
    }

    // Создание книги
    async function createBook(event) {
        event.preventDefault();

        const form = event.target;
        const formData = new FormData(form);
        const data = {
            title: formData.get('title'),
            author: formData.get('author'),
            year: parseInt(formData.get('year')) || null,
            price: parseFloat(formData.get('price')) || 0
        };
```

```
    // Валидация
    if (!data.title || !data.author) {
        showMessage('Название и автор обязательны!', 'error');
        return;
    }

    try {
        const response = await fetch(API_URL, {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json'
            },
            body: JSON.stringify(data)
        });

        if (!response.ok) {
            const error = await response.json();
            throw new Error(error.error || 'Ошибка создания');
        }

        const newBook = await response.json();
        showMessage(`Книга "${newBook.title}" добавлена!`, 'success');
        form.reset();
        loadBooks();  // Перезагружаем список
    } catch (error) {
        showMessage(error.message, 'error');
    }
}
```

```
        // Редактирование книги
        async function editBook(id) {
            // Получаем текущие значения
            const titleSpan = document getElementById(`title-${id}`);
            const authorSpan = document getElementById(`author-${id}`);
            const yearSpan = document getElementById(`year-${id}`);
            const priceSpan = document getElementById(`price-${id}`);

            // Создаём форму редактирования
            const newTitle = prompt('Введите новое название:', titleSpan innerText);
            if (newTitle === null) return;

            const newAuthor = prompt('Введите нового автора:', authorSpan innerText);
            if (newAuthor === null) return;

            const newYear = prompt('Введите новый год:', yearSpan innerText === '—' ? '' : yearSpan innerText);
            const newPrice = prompt('Введите новую цену:', priceSpan innerText === '—' ? '' : priceSpan innerText replace(' ₽', ''));

            const data = {
                title: newTitle,
                author: newAuthor,
                year: newYear ? parseInt(newYear) : null,
                price: newPrice ? parseFloat(newPrice) : 0
            };

            try {
```

```
        const response = await fetch(`${API_URL}/${id}`, {
            method: 'PUT',
            headers: {
                'Content-Type': 'application/json'
            },
            body: JSON.stringify(data)
        });

        if (!response.ok) {
            const error = await response.json();
            throw new Error(error.error || 'Ошибка обновления');
        }

        showMessage('Книга обновлена!', 'success');
        loadBooks();
    } catch (error) {
        showMessage(error.message, 'error');
    }
}

// Удаление книги
async function deleteBook(id) {
    if (!confirm('Вы уверены, что хотите удалить эту книгу?')) return;

    try {
        const response = await fetch(`${API_URL}/${id}`, {
            method: 'DELETE'
        });
```

```
            if (!response.ok && response.status !== 204) {
                const error = await response.json();
                throw new Error(error.error || 'Ошибка удаления');
            }

            showMessage('Книга удалена', 'success');
            loadBooks();
        } catch (error) {
            showMessage(error.message, 'error');
        }
    }

    // Поиск книг
    async function searchBooks(query) {
        if (!query) {
            loadBooks();
            return;
        }

        const container = document.getElementById('books-container');
        container.innerHTML = '<div class="loading">Поиск...</div>';

        try {
            const response = await fetch(`${API_URL}/search?q=${encodeURIComponent(query)}`);
            if (!response.ok) throw new Error('Ошибка поиска');

            const result = await response.json();
            displayBooks(result.results || result);
```

```
        } catch (error) {
            container.innerHTML = `<div class="error">Ошибка: ${error.message}</div>`;
        }
    }

    // Фильтрация по автору
    async function filterByAuthor(author) {
        const container = document.getElementById('books-container');
        container.innerHTML = '<div class="loading">Фильтрация...</div>';

        try {
            const url = author ? `${API_URL}?author=${encodeURIComponent(author)}` : API_URL;
            const response = await fetch(url);
            if (!response.ok) throw new Error('Ошибка фильтрации');

            const books = await response.json();
            displayBooks(books);
        } catch (error) {
            container.innerHTML = `<div class="error">Ошибка: ${error.message}</div>`;
        }
    }

    // Экранирование HTML (защита от XSS)
    function escapeHtml(str) {
        if (!str) return '';
        return str
```

```
            .replace(/&/g, '&amp;')
            .replace(/</g, '&lt;')
            .replace(/>/g, '&gt;')
            .replace(/"/g, '&quot;')
            .replace(/'/g, '&#39;');
    }

    // Инициализация при загрузке страницы
    document.addEventListener('DOMContentLoaded', () => {
        // Загрузка книг
        loadBooks();

        // Подключение обработчика формы
        const createForm = document.getElementById('create-form');
        if (createForm) {
            createForm.addEventListener('submit', createBook);
        }

        // Добавление поля поиска (опционально)
        const searchContainer = document.querySelector('.section:first-of-type');
        if (searchContainer) {
            const searchHtml = `
                <div class="section">
                    <h2>🔍 Поиск книг</h2>
                    <input type="text" id="search-input" placeholder="Поиск по названию..." style="width: 100%">
                </div>
            `;
            searchContainer.insertAdjacentHTML('afterend', searchHtml);
```

```
        const searchInput = document getElementById('search-input');
        let searchTimeout
        searchInput addEventListener('input', (e) => {
            clearTimeout(searchTimeout);
            searchTimeout = setTimeout(() => {
                searchBooks(e target value);
            }, 300);
        });
    }
});
```

Пример 3. Настройка CORS на бэкенде

```
javascript
// server.js (Express)
const express = require('express');
const cors = require('cors');
const sqlite3 = require('sqlite3').verbose();

const app = express();
const PORT = 3000;

// Настройка CORS
app use(cors({
    origin: ['http://localhost:3001', 'http://127.0.0.1:3001', 'http://localhost:5500'],
    methods: ['GET', 'POST', 'PUT', 'DELETE', 'OPTIONS'],
    allowedHeaders: ['Content-Type', 'Authorization']
}));
```

```
app.use(express.json());

// Остальной код сервера...
```

Пример 4. Полный пример сервера с поддержкой CORS

javascript

```
// server.js
const express = require('express');
const cors = require('cors');
const sqlite3 = require('sqlite3').verbose();
const path = require('path');

const app = express();
const PORT = process.env.PORT || 3000;

// Настройка CORS
app.use(cors({
    origin: ['http://localhost:3001', 'http://localhost:5500', 'http://127.0.0.1:5500'],
    credentials: true
}));

app.use(express.json());
app.use(express.static('public'));  // Для статических файлов (HTML, CSS, JS)

// Подключение к БД
const db = new sqlite3.Database('./books.db');

// Создание таблицы
```

```
db.run(`
    CREATE TABLE IF NOT EXISTS books (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        title TEXT NOT NULL,
        author TEXT NOT NULL,
        year INTEGER,
        price REAL,
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    )
`);

// Заполнение тестовыми данными
db.get('SELECT COUNT(*) as count FROM books', [], (err, row) => {
    if (row.count === 0) {
        const books = [
            ['Война и мир', 'Лев Толстой', 1869, 599],
            ['Преступление и наказание', 'Фёдор Достоевский', 1866, 499],
            ['Мастер и Маргарита', 'Михаил Булгаков', 1967, 699]
        ];
        const stmt = db.prepare('INSERT INTO books (title, author, year, price)
VALUES (?, ?, ?, ?)');
        books.forEach(book => stmt.run(book));
        stmt.finalize();
    }
});

// ========== API ЭНДПОИНТЫ ==========

// GET все книги
```

```
app.get('/api/books', (req, res) => {
  db.all('SELECT * FROM books ORDER BY id', [], (err, rows) => {
    if (err) return res.status(500).json({ error: err.message });
    res.json(rows);
  });
});

// GET книга по ID
app.get('/api/books/:id', (req, res) => {
  const id = req.params.id;
  db.get('SELECT * FROM books WHERE id = ?', [id], (err, row) => {
    if (err) return res.status(500).json({ error: err.message });
    if (!row) return res.status(404).json({ error: 'Книга не найдена' });
    res.json(row);
  });
});

// GET поиск книг
app.get('/api/books/search', (req, res) => {
  const query = req.query.q;
  if (!query) return res.json([]);

  const searchPattern = `%${query}%`;
  db.all(
    'SELECT * FROM books WHERE title LIKE ? OR author LIKE ?',
    [searchPattern, searchPattern],
    (err, rows) => {
      if (err) return res.status(500).json({ error: err.message });
      res.json(rows);
```

```
        }
      );
    });

    // POST создать книгу
    app.post('/api/books', (req, res) => {
      const { title, author, year, price } = req.body;

      if (!title || !author) {
        return res.status(400).json({ error: 'Название и автор обязательны' });
      }

      db.run(
        'INSERT INTO books (title, author, year, price) VALUES (?, ?, ?, ?)',
        [title, author, year || null, price || 0],
        function(err) {
          if (err) return res.status(500).json({ error: err.message });

          db.get('SELECT * FROM books WHERE id = ?', [this.lastID], (err, newBook) => {
            if (err) return res.status(500).json({ error: err.message });
            res.status(201).json(newBook);
          });
        }
      );
    });

    // PUT обновить книгу
    app.put('/api/books/:id', (req, res) => {
```

```
    const id = req.params.id;
    const { title, author, year, price } = req.body;

    if (!title || !author) {
      return res.status(400).json({ error: 'Название и автор обязательны' });
    }

    db.get('SELECT * FROM books WHERE id = ?', [id], (err, book) => {
      if (err) return res.status(500).json({ error: err.message });
      if (!book) return res.status(404).json({ error: 'Книга не найдена' });

      db.run(
        'UPDATE books SET title = ?, author = ?, year = ?, price = ? WHERE id = ?',
        [title, author, year || null, price || 0, id],
        function(err) {
          if (err) return res.status(500).json({ error: err.message });

          db.get('SELECT * FROM books WHERE id = ?', [id], (err, updatedBook) => {
            if (err) return res.status(500).json({ error: err.message });
            res.json(updatedBook);
          });
        }
      );
    });
  });

  // DELETE удалить книгу
```

```
app.delete('/api/books/:id', (req, res) => {
    const id = req.params.id

    db.get('SELECT * FROM books WHERE id = ?', [id], (err, book) => {
        if (err) return res.status(500).json({ error: err.message });
        if (!book) return res.status(404).json({ error: 'Книга не найдена' });

        db.run('DELETE FROM books WHERE id = ?', [id], function(err) {
            if (err) return res.status(500).json({ error: err.message });
            res.status(204).send();
        });
    });
});

// Запуск сервера
app.listen(PORT, () => {
    console.log(`Сервер запущен на http://localhost:${PORT}`);
    console.log(`API доступно по адресу http://localhost:${PORT}/api/books`);
});
```

2.2. Индивидуальные задания

2.2.1. Базовый уровень (15 вариантов)

Общее задание для базового уровня:

Разработать простой интерфейс для взаимодействия с API из предыдущих лабораторных работ. Требования:

1. HTML-страница с формой для создания записи.
2. Таблица/список для отображения всех записей.
3. Реализация GET (чтение) и POST (создание) записей через fetch.

4. Кнопка для обновления списка (перезагрузка).
5. Базовая обработка ошибок и уведомления.

Варианты предметных областей (базовый уровень):

	Сущность	Поля для отображения	Поля формы
	Книги	id, title, author	title, author, year, price
	Товары	id, name, price	name, price, category, stock
	Студенты	id, full_name, group	full_name, group, age, email
	Фильмы	id, title, director	title, director, year, rating
	Сотрудники	id, name, position	name, position, salary
	Рецепты	id, title, cuisine	title, cuisine, cooking_time
	Автомобили	id, model, brand	model, brand, year, price
	Песни	id, title, artist	title, artist, duration
	Отели	id, name, city	name, city, stars

	Сущность	Поля для отображения	Поля формы
0	Курсы	id, title, instructor	title, instructor, duration
1	Лекарства	id, name, manufacturer	name, manufacturer, price
2	Мероприятия	id, name, location	name, location, date
3	Заказы	id, customer, amount	customer, product, amount
4	Статьи	id, title, author	title, author, content
5	Спортсмены	id, name, sport	name, sport, country

2.2.2. Средний уровень (15 вариантов)

Общее задание для среднего уровня:

1. Выполнить требования базового уровня, а также дополнительно реализовать:
2. Редактирование записи (PUT/PATCH) через модальное окно или inline-редактирование.
3. Удаление записи (DELETE) с подтверждением.
4. Поиск/фильтрацию данных на стороне сервера (query-параметры).

5. Стилизацию с помощью CSS (минимум 50 строк).
6. Обработку состояния загрузки (spinner или текст "Загрузка...").

Варианты дополнительных функций (средний уровень):

№	Дополнительная функция
1	Сортировка по клику на заголовок таблицы
2	Пагинация (кнопки "Предыдущая"/"Следующая")
3	Экспорт данных в CSV/JSON
4	Валидация формы перед отправкой
5	Тёмная/светлая тема (переключатель)
6	Адаптивный дизайн (мобильная версия)
7	Подтверждение удаления (modal dialog)
8	Уведомления (toast) вместо alert
9	Панель фильтров по полям
10	Автоматическое обновление списка (polling)
11	Отмена последнего удаления (undo)
12	Выделение новой/изменённой записи
13	Статистика (количество записей)
14	Копирование данных в буфер обмена

№	Дополнительная функция
15	Печать таблицы

2.2.3. Продвинутый уровень (15 вариантов)

Общее задание для продвинутого уровня:

1. Выполнить требования среднего уровня, а также дополнительно реализовать:
2. Интеграцию с аутентификацией (JWT или сессии) — добавление токена в заголовки fetch.
3. Регистрацию/вход пользователя через отдельную форму.
4. WebSocket для real-time обновлений (добавление/удаление без перезагрузки).
5. Пагинацию на сервере с использованием limit и offset.
6. Ленивую загрузку (бесконечный скроллинг).
7. Кэширование данных в localStorage/IndexedDB.
8. Индикатор прогресса при загрузке/отправке.

Варианты продвинутых функций:

№	Дополнительная функция
1	JWT аутентификация (login/register, токен в localStorage)
2	Real-time обновления через WebSocket
3	Бесконечный скроллинг (infinite scroll)
4	Drag-and-drop для изменения порядка элементов

№	Дополнительная функция
5	Bulk-операции (массовое удаление/экспорт)
6	История действий (undo/redo)
7	Офлайн-режим (Service Worker + IndexedDB)
8	Голосовое управление (Web Speech API)
9	Экспорт в PDF (jsPDF)
10	Графики/диаграммы на основе данных (Chart.js)
11	Интеграция с календарём (для мероприятий)
12	Загрузка изображений (multipart/form-data)
13	Интеграция с картой (для локаций)
14	Импорт данных из Excel/CSV
15	PWA (Progressive Web App) возможности

Пример с аутентификацией (JWT):

javascript

```
// auth.js
let token = localStorage getItem('token');

async function login(username, password) {
    const response = await fetch('/api/login', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
```

```
        body: JSON.stringify({ username, password })
    });

    if (response.ok) {
        const data = await response.json();
        token = data.token;
        localStorage.setItem('token', token);
        return true;
    }
    return false;
}

// Добавление токена к fetch-запросам
async function authenticatedFetch(url, options = {}) {
    options.headers = {
        ...options.headers,
        'Authorization': `Bearer ${token}`
    };
    return fetch(url, options);
}
```

Пример WebSocket для real-time обновлений:

```
javascript
// ws-client.js
let ws = new WebSocket('ws://localhost:3000');

ws.onmessage = (event) => {
    const data = JSON.parse(event.data);

    switch (data.type) {
```

```
        case 'BOOK_CREATED':
            addBookToList(data.book);
            break;
        case 'BOOK_UPDATED':
            updateBookInList(data.book);
            break;
        case 'BOOK_DELETED':
            removeBookFromList(data.bookId);
            break;
    }
};

// При создании книги через API
async function createBook(bookData) {
    const response = await fetch('/api/books', {
        method: 'POST',
        body: JSON.stringify(bookData)
    });
    // WebSocket отправит уведомление всем клиентам
}
```

3. Контрольные вопросы

1. Что такое fetch и для чего он используется?
2. В чём разница между fetch и XMLHttpRequest?
3. Как обработать ошибки при выполнении fetch-запроса?
4. Как отправить данные в формате JSON через fetch?
5. Как получить данные из JSON-ответа?
6. Что такое CORS и почему возникает ошибка CORS?
7. Как настроить CORS на сервере Express?

8. Как отменить перезагрузку страницы при отправке формы?
9. Как динамически обновить таблицу без перезагрузки страницы?
10. Как реализовать поиск с задержкой (debounce)?
11. Как защититься от XSS-атак при отображении пользовательских данных?
12. Как сохранить токен аутентификации в localStorage?
13. Как реализовать бесконечный скроллинг?
14. Что такое WebSocket и чем он отличается от HTTP?
15. Как организовать real-time обновления данных?

4. Требования к отчёту

Отчёт должен быть оформлен в электронном виде и содержат:

Титульный лист с названием работы, ФИО студента, номером группы, вариантом.

Цель и задачи работы (переписать из методических указаний).

Теоретическое обоснование (краткий конспект по теме).

Ход выполнения работы:

Листинг HTML/CSS кода.

Листинг JavaScript кода.

Скриншоты страницы в браузере:

Отображение списка записей

Форма создания

Успешное создание записи

Редактирование записи

Удаление записи

Поиск/фильтрация

(Средний уровень) Скриншоты дополнительных функций.

(Продвинутый уровень) Скриншоты аутентификации, WebSocket.

Ответы на контрольные вопросы (письменно, развёрнуто).

Заключение (выводы).

Список использованных источников.

5. Критерии оценивания

Уровень выполнения	Требования	Оценка
Базовый	HTML-страница с формой и таблицей, реализованы GET (чтение) и POST (создание), обработка ошибок, уведомления.	Удовлетворительно
Средний	Выполнены требования базового уровня + редактирование, удаление, поиск/фильтрация, стилизация, индикатор загрузки.	Хорошо
Продвинутый	Выполнены требования среднего уровня + аутентификация (JWT/сессии), WebSocket для real-time обновлений, пагинация/бесконечный скроллинг, кэширование.	Отлично

Дополнительные условия:

1. Приложение должно корректно работать без ошибок в консоли.
2. Интерфейс должен быть удобным и интуитивно понятным.
3. При защите студент должен объяснить код и принципы взаимодействия с API.

6. Список рекомендуемой литературы и интернет-ресурсов

Литература (доступна в РФ)

1. Браун, И. (2021). *Express.js: Разработка веб-приложений на Node.js*. ДМК Пресс.
2. Кантелон, М., Хартер, М., & Холстейн, С. (2020). *Node.js в действии*. Питер.

Интернет-ресурсы

Название	Ссылки	Описание
Fetch API	https://developer.mozilla.org/ru/docs/Web/API/Fetch_API	MDN документация
CORS	https://developer.mozilla.org/ru/docs/Web/HTTP/CORS	MDN о CORS
cors npm	https://www.npmjs.com/package/cors	Документация CORS middleware
WebSocket	https://developer.mozilla.org/ru/docs/Web/API/WebSocket	MDN о WebSocket
async/await	https://javascript.info/async-await	Современный JavaScript
HTML Living Standard	https://html.spec.whatwg.org/	Спецификация HTML

Все ресурсы доступны и легальны на территории Российской Федерации.

Лабораторная работа № 17

Тема: Деплой бэкенд-приложения на облачную платформу

Цель работы:

Освоить процесс развёртывания (деплоя) простого бэкенд-приложения на облачную платформу для обеспечения доступа к API из интернета и демонстрации работоспособности.

Задачи работы:

1. Зарегистрироваться на платформе для хостинга (Render.com / Railway.app / Glitch.com).
2. Подготовить бэкенд-приложение к деплою (настройка порта, скриптов, зависимостей).
3. Подключить репозиторий GitHub к платформе.
4. Настроить переменные окружения на платформе.
5. Выполнить деплой приложения и получить публичный URL.
6. Протестировать работоспособность API через публичный URL.
7. Настроить автоматический деплой при изменении кода (CI/CD).

1. Теоретическое обоснование

1.1. Что такое деплой?

Деплой (развёртывание) - это процесс размещения приложения на сервере, доступном из интернета. В результате пользователи могут получить доступ к приложению через публичный IP-адрес или доменное имя.

1.2. Варианты хостинга для бэкенда

Платформа	Бесплатный тариф	Особенности	Сложность
Render.com	Да (750 часов/мес)	Поддержка Node.js, Python, Go, Docker; авто-деплой из GitHub	★★☆☆☆
Railway.app	Да (500 часов/мес + 5$ кредитов)	Простой интерфейс, поддержка многих языков	★★☆☆☆
Glitch.com	Да (ограничения по ресурсам)	Простой редактор кода в браузере, подходит для прототипов	★☆☆☆☆
Cyclic.sh	Да (ограниченно)	Специализируется на Node.js, простой деплой	★★☆☆☆
Heroku	Нет (платный с 2022)	(не рекомендуется для новых проектов)	—
VPS (Yandex Cloud, AWS, DigitalOcean)	Нет (бесплатный период есть у некоторых)	Полный контроль, но сложнее в настройке	★★★★☆

В данной лабораторной работе рекомендуется использовать Render.com или Railway.app как наиболее подходящие для начинающих.

1.3. Требования к приложению для деплоя

Для успешного деплоя Node.js-приложения необходимо:

Файл package.json с корректными скриптами:

json

```
{
  "scripts": {
    "start": "node app.js",
    "dev": "nodemon app.js"
  }
}
```

Переменная окружения PORT - платформа сама назначает порт:

javascript

```
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

Файл .gitignore исключающий node_modules/ и .env

Все зависимости в dependencies (не в devDependencies, если они нужны для работы)

1.4. Процесс деплоя на Render.com

Создать аккаунт на render.com

Подключить GitHub репозиторий

Создать Web Service

Указать:

Name: my-api

Environment: Node

Build Command: npm install

Start Command: npm start

Добавить переменные окружения

Нажать "Create Web Service"

Дождаться деплоя (первые минуты)

Получить URL вида: https://my-api.onrender.com

1.5. Переменные окружения на платформе

Настройки, которые были в файле .env локально, нужно добавить в панели управления платформы:

Переменная	Значение	Описание
PORT	(назначается автоматически)	Порт сервера
DB_PATH	./database.db	Путь к SQLite (если используется файл)
JWT_SECRET	(секретный ключ)	Секрет для JWT
NODE_ENV	production	Режим продакшена

1.6. Проблема с SQLite на облачных платформах

SQLite хранит данные в файле. На облачных платформах файловая система может быть эфемерной (очищается при перезапуске). Для production-приложений лучше использовать облачные базы данных:

Render.com предоставляет PostgreSQL (есть бесплатный уровень)

Railway.app предоставляет PostgreSQL и MySQL

Для целей лабораторной работы допустимо использовать SQLite, но данные будут теряться при перезапуске сервера.

1.7. Настройка PostgreSQL на Render.com

javascript

```
// Замена SQLite на PostgreSQL в коде
const { Pool } = require('pg');

const pool = new Pool({
  connectionString: process.env.DATABASE_URL,
  ssl: { rejectUnauthorized: false }
});

// Выполнение запросов
pool.query('SELECT * FROM users', (err, result) => {
  console.log(result.rows);
});
```

1.8. CI/CD - автоматический деплой

При подключении GitHub репозитория к Render.com автоматически настраивается автоматический деплой:

При каждом git push в основную ветку

При создании Pull Request (создаётся тестовое окружение)

2. Методика выполнения работы

2.1. Примеры и их подробный разбор

Пример 1. Подготовка приложения к деплою

package.json:

json

```
{
  "name": "my-api",
  "version": "1.0.0",
  "description": "Simple REST API",
  "main": "app.js",
  "scripts": {
    "start": "node app.js",
    "dev": "nodemon app.js"
  },
  "dependencies": {
    "express": "^4.18.2",
    "sqlite3": "^5.1.6",
    "cors": "^2.8.5",
    "dotenv": "^16.3.1",
    "bcrypt": "^5.1.1"
  },
  "devDependencies": {
    "nodemon": "^3.0.1"
  }
}
```

app.js (фрагмент с правильным портом):

javascript

```
require('dotenv').config();
const express = require('express');
const cors = require('cors');

const app = express();
```

```
// Настройка CORS для продакшена
app.use(cors({
    origin: process.env.CORS_ORIGIN || '*'
}));

app.use(express.json());

// ... остальной код ...

// Порт из переменной окружения
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
    console.log(`Server running on port ${PORT}`);
    console.log(`Environment: ${process.env.NODE_ENV || 'development'}`);
});
```

.gitignore:

```
node_modules/
.env
*.log
.DS_Store
```

Пример 2. Регистрация на Render.com

Шаг 1. Создание аккаунта

Перейти на render.com

Нажать "Sign Up"

Выбрать "Sign up with GitHub" (рекомендуется)

Авторизоваться в GitHub и разрешить доступ

Шаг 2. Подключение репозитория

На дашборде нажать "New +" → "Web Service"

Выбрать "Connect GitHub account"

Выбрать нужный репозиторий

Шаг 3. Настройка Web Service

Поле	Значение
Name	my-backend-api
Region	Frankfurt (EU) (ближе к России)
Branch	main
Root Directory	(оставить пустым)
Runtime	Node
Build Command	npm install
Start Command	npm start

Шаг 4. Добавление переменных окружения

Нажать "Advanced" → "Add Environment Variable":

Key	Value
NODE_ENV	production
DB_PATH	./database.db
JWT_SECRET	(сгенерировать случайную строку)

Шаг 5. Деплой

Нажать "Create Web Service". Дождаться завершения сборки (лог в интерфейсе).

Пример 3. Настройка PostgreSQL на Render.com

Шаг 1. Создание базы данных

На дашборде Render → "New +" → "PostgreSQL"

Заполнить поля:

Name: my-db

Database: mydb

User: myuser

Нажать "Create Database"

Шаг 2. Получение connection string

После создания будет доступна строка подключения вида:

text

postgresql://myuser:password@host:5432/mydb

Шаг 3. Использование в коде

javascript

```
const { Pool } = require('pg');

const pool = new Pool({
  connectionString: process.env.DATABASE_URL,
  ssl: { rejectUnauthorized: false }
});

// Создание таблиц
pool.query(`
  CREATE TABLE IF NOT EXISTS users (
    id SERIAL PRIMARY KEY,
    username TEXT UNIQUE NOT NULL,
    password_hash TEXT NOT NULL
```

```
  )
`);
```

Шаг 4. Добавление DATABASE_URL в переменные окружения

В настройках Web Service добавить:

```
DATABASE_URL = postgresql://myuser:password@host:5432/mydb
```

Пример 4. Деплой на Railway.app

Шаг 1. Регистрация

Перейти на railway.app

Нажать "Login" → "Login with GitHub"

Шаг 2. Создание проекта

Нажать "New Project"

Выбрать "Deploy from GitHub repo"

Выбрать репозиторий

Шаг 3. Настройка

Railway автоматически определяет Node.js приложение и устанавливает зависимости. Переменные окружения добавляются через вкладку "Variables".

Шаг 4. Получение URL

После деплоя Railway выдаёт URL вида: https://my-api.up.railway.app

Пример 5. Тестирование после деплоя

bash

```
# Проверка корневого маршрута
curl https://my-api.onrender.com/

# Проверка GET /api/books
curl https://my-api.onrender.com/api/books

# Проверка POST /api/books
curl -X POST https://my-api.onrender.com/api/books \
```

```
  -H "Content-Type: application/json" \
  -d '{"title":"Новая книга","author":"Автор"}'
```

Пример 6. Автоматический деплой через GitHub Actions (альтернатива)

yaml

```yaml
# .github/workflows/deploy.yml
name: Deploy to Render

on:
  push:
    branches: [ main ]

jobs:
  deploy:
    runs-on: ubuntu latest
    steps:
      - uses: actions/checkout@v3

      - name: Deploy to Render
        env:
          RENDER_API_KEY: ${{ secrets.RENDER_API_KEY }}
        run: |
          curl -X POST "https://api.render.com/deploy/srv-xxxxx" \
            -H "Authorization: Bearer $RENDER_API_KEY"
```

Пример 7. Полный пример сервера, готового к деплою

javascript

```javascript
// app.js
require('dotenv').config();
const express = require('express');
const cors = require('cors');
```

```
const path = require('path');
const fs = require('fs');

const app = express();

// Настройка CORS
app.use(cors({
    origin: process.env.CORS_ORIGIN || '*',
    methods: ['GET', 'POST', 'PUT', 'DELETE', 'OPTIONS'],
    allowedHeaders: ['Content-Type', 'Authorization']
}));

app.use(express.json());
app.use(express.static('public'));  // для статического фронтенда

// Проверка здоровья (health check)
app.get('/health', (req, res) => {
    res.json({
        status: 'OK',
        timestamp: new Date().toISOString(),
        environment: process.env.NODE_ENV || 'development'
    });
});

// Корневой маршрут
app.get('/', (req, res) => {
    res.json({
        message: 'API работает!',
        version: '1.0.0',
```

```
        endpoints: {
            health: 'GET /health',
            books: 'GET /api/books',
            createBook: 'POST /api/books'
        }
    });
});

// Подключение к SQLite (с созданием директории)
const sqlite3 = require('sqlite3').verbose();
const dbDir = path.join(__dirname, 'data');
if (!fs.existsSync(dbDir)) {
    fs.mkdirSync(dbDir, { recursive: true });
}

const dbPath = process.env.DB_PATH || path.join(dbDir, 'database.db');
const db = new sqlite3.Database(dbPath, (err) => {
    if (err) {
        console.error('Ошибка подключения к БД:', err.message);
    } else {
        console.log('Подключено к БД:', dbPath);

        // Создание таблиц
        db.run(`
            CREATE TABLE IF NOT EXISTS books (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                title TEXT NOT NULL,
                author TEXT NOT NULL,
                year INTEGER,
```

```
            price REAL,
            created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
        )
    `);
  }
});

// API эндпоинты
app.get('/api/books', (req, res) => {
  db.all('SELECT * FROM books ORDER BY id', [], (err, rows) => {
    if (err) return res.status(500).json({ error: err.message });
    res.json(rows);
  });
});

app.post('/api/books', (req, res) => {
  const { title, author, year, price } = req.body;

  if (!title || !author) {
    return res.status(400).json({ error: 'Название и автор обязательны' });
  }

  db.run(
    'INSERT INTO books (title, author, year, price) VALUES (?, ?, ?, ?)',
    [title, author, year || null, price || 0],
    function(err) {
      if (err) return res.status(500).json({ error: err.message });
      res.status(201).json({ id: this.lastID, title, author, year, price });
    }
```

```
    );
  });

  app.get('/api/books/:id', (req, res) => {
    const id = req.params.id;
    db.get('SELECT * FROM books WHERE id = ?', [id], (err, row) => {
      if (err) return res.status(500).json({ error: err.message });
      if (!row) return res.status(404).json({ error: 'Книга не найдена' });
      res.json(row);
    });
  });

  app.put('/api/books/:id', (req, res) => {
    const id = req.params.id;
    const { title, author, year, price } = req.body;

    if (!title || !author) {
      return res.status(400).json({ error: 'Название и автор обязательны' });
    }

    db.run(
      'UPDATE books SET title = ?, author = ?, year = ?, price = ? WHERE id = ?',
      [title, author, year || null, price || 0, id],
      function(err) {
        if (err) return res.status(500).json({ error: err.message });
        if (this.changes === 0) return res.status(404).json({ error: 'Книга не найдена' });
        res.json({ id, title, author, year, price });
```

```
        }
      );
    });

    app.delete('/api/books/:id', (req, res) => {
      const id = req.params.id;
      db.run('DELETE FROM books WHERE id = ?', [id], function(err) {
        if (err) return res.status(500).json({ error: err.message });
        if (this.changes === 0) return res.status(404).json({ error: 'Книга не найдена' });
        res.status(204).send();
      });
    });

    // Обработчик 404
    app.use((req, res) => {
      res.status(404).json({ error: 'Маршрут не найден' });
    });

    // Глобальный обработчик ошибок
    app.use((err, req, res, next) => {
      console.error('Ошибка:', err.stack);
      res.status(500).json({ error: 'Внутренняя ошибка сервера' });
    });

    // Запуск сервера
    const PORT = process.env.PORT || 3000;
    app.listen(PORT, () => {
      console.log(`
```

```
╔═══════════════════════════════════════════════════════════╗
║  🚀 Сервер успешно запущен!                               ║
╠═══════════════════════════════════════════════════════════╣
║  Порт: ${PORT.toString().padEnd(41)}║
║  Режим: ${process.env.NODE_ENV || 'development'}.padEnd(41)}║
║  URL: http://localhost:${PORT}${' '.repeat(41 - (PORT.toString().length + 21))}}║
╚═══════════════════════════════════════════════════════════╝
    `);
});

// Корректное завершение
process.on('SIGINT', () => {
    console.log('\n👋 Сервер остановлен');
    db.close(() => process.exit(0));
});
```

2.2. Индивидуальные задания

2.2.1. Базовый уровень (15 вариантов)

Общее задание для базового уровня:

Развернуть существующее бэкенд-приложение (из предыдущих лабораторных работ) на облачной платформе. Требования:

Зарегистрироваться на платформе (Render.com или Railway.app).

Подготовить приложение к деплою (порт, скрипты, .gitignore).

Загрузить код в GitHub репозиторий.

Выполнить деплой.

Получить публичный URL.

Протестировать минимум 3 эндпоинта через публичный URL.

Варианты платформ (базовый уровень):

№	Платформа	Тип базы данных
1	Render.com	SQLite (файл)
2	Railway.app	SQLite (файл)
3	Glitch.com	SQLite (файл)
4	Cyclic.sh	SQLite (файл)
5	Render.com	PostgreSQL
6	Railway.app	PostgreSQL
7	Render.com	SQLite
8	Cyclic.sh	SQLite
9	Railway.app	PostgreSQL
10	Render.com	SQLite
11	Glitch.com	SQLite
12	Cyclic.sh	PostgreSQL
13	Railway.app	SQLite
14	Render.com	PostgreSQL
15	Glitch.com	PostgreSQL

2.2.2. Средний уровень (15 вариантов)

Общее задание для среднего уровня:

Выполнить требования базового уровня, а также дополнительно реализовать:

Подключение облачной базы данных (PostgreSQL на Render/Railway вместо SQLite).

Настройку переменных окружения на платформе (минимум 3 переменные).

Health check endpoint (GET /health).

Документацию API в README репозитория.

Автоматический деплой при push в main ветку (настроить).

Варианты дополнительных требований (средний уровень):

№	Дополнительное требование
1	Настроить CORS для конкретного домена
2	Добавить rate limiting (защита от DDoS)
3	Настроить логирование через morgan
4	Добавить Swagger/OpenAPI документацию
5	Настроить HTTPS (автоматически на Render)
6	Добавить кэширование (Redis на Render)
7	Настроить backup базы данных
8	Добавить мониторинг (Uptime Robot)
9	Настроить автодеплой из ветки dev

№	Дополнительное требование
10	Добавить тесты перед деплоем
11	Настроить email-уведомления об ошибках
12	Добавить админ-панель
13	Настроить переменные окружения для разных окружений
14	Добавить файл docker-compose для локальной разработки
15	Настроить сервер статики (фронтенд) на том же домене

2.2.3. Продвинутый уровень (15 вариантов)

Общее задание для продвинутого уровня:

1. Выполнить требования среднего уровня, а также дополнительно реализовать:
2. Собственный домен (бесплатный .tk или дешёвый .ru) и его привязка.
3. Настройку CI/CD через GitHub Actions с автоматическим тестированием перед деплоем.
4. Использование Docker (контейнеризация приложения и деплой через Docker).
5. Настройку CDN (CloudFlare) для ускорения доставки.
6. Мониторинг и алертинг (UptimeRobot, Sentry или аналог).
7. Бэкап базы данных на облачное хранилище.

Варианты продвинутых функций:

№	Дополнительная функция
1	Docker-контейнер + деплой на Render
2	GitHub Actions + тесты + деплой
3	Собственный домен + SSL (Let's Encrypt)
4	CloudFlare CDN + DDoS защита
5	Prometheus + Grafana мониторинг
6	Sentry для отслеживания ошибок
7	UptimeRobot + Telegram алерты
8	Автоматический бэкап БД в S3 (или аналог)
9	Горизонтальное масштабирование (несколько экземпляров)
10	Redis для кэширования + деплой
11	WebSocket поддержка (на Render/Railway)
12	Фоновые задачи (Bull/BullMQ)
13	GraphQL вместо REST API
14	gRPC сервер
15	Serverless деплой (AWS Lambda + API Gateway)

Пример Dockerfile для продвинутого уровня:

dockerfile

```
# Dockerfile
FROM node:18-alpine

WORKDIR /app

# Копирование package.json и package-lock.json
COPY package*.json ./

# Установка зависимостей
RUN npm ci --only=production

# Копирование исходного кода
COPY . .

# Создание директории для БД
RUN mkdir -p /app/data

# Порт приложения
EXPOSE 3000

# Запуск приложения
CMD ["npm", "start"]
```

Пример GitHub Actions workflow (продвинутый уровень):

yaml

```
# .github/workflows/deploy.yml
name: Deploy to Render

on:
```

```yaml
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: '18'
      - run: npm ci
      - run: npm test
        env:
          NODE_ENV: test
          DATABASE_URL: ${{ secrets.TEST_DATABASE_URL }}

  deploy:
    needs: test
    runs-on: ubuntu-latest
    if: github.event_name == 'push' && github.ref == 'refs/heads/main'
    steps:
      - uses: actions/checkout@v3

      - name: Deploy to Render
        env:
          RENDER_API_KEY: ${{ secrets.RENDER_API_KEY }}
```

```
run: |
  curl -X POST "https://api.render.com/deploy/srv-${{ secrets.RENDER_SERVICE_ID }}" \
    -H "Authorization: Bearer $RENDER_API_KEY" \
    -H "Content-Type: application/json"
```

3. Контрольные вопросы

1. Что такое деплой и зачем он нужен?
2. Какие существуют платформы для деплоя Node.js-приложений?
3. Какие требования предъявляются к приложению для деплоя на облачную платформу?
4. Как получить переменную порта на облачной платформе?
5. Как настроить автоматический деплой при push в GitHub?
6. В чём проблема использования SQLite на облачных платформах?
7. Какую базу данных лучше использовать для production?
8. Как добавить переменные окружения на Render.com?
9. Как проверить, что приложение успешно развернуто?
10. Что такое health check и зачем он нужен?
11. Как настроить CORS для продакшн-окружения?
12. Как обновить приложение после деплоя?
13. Что такое CI/CD и как он связан с деплоем?
14. Как настроить собственный домен для приложения?
15. Как отслеживать ошибки в production-приложении?

4. Требования к отчёту

Отчёт должен быть оформлен в электронном виде и содержать:

Титульный лист с названием работы, ФИО студента, номером группы, вариантом.

Цель и задачи работы (переписать из методических указаний).

Теоретическое обоснование (краткий конспект по теме).

Ход выполнения работы:

Ссылка на GitHub репозиторий с кодом

Скриншоты настроек на платформе (переменные окружения, команды)

Скриншот лога деплоя

Публичный URL развёрнутого приложения

Скриншоты запросов к API через браузер/Postman

(Средний уровень) Скриншоты health check и документации

(Продвинутый уровень) Скриншоты мониторинга, Docker, CI/CD

Ответы на контрольные вопросы (письменно, развёрнуто).

Заключение (выводы).

Список использованных источников.

5. Критерии оценивания

Уровень выполнения	Требования	Оценка
Базовый	Приложение развёрнуто на облачной платформе, доступно по публичному URL, работают базовые эндпоинты.	Удовлетворительно
Средний	Выполнены требования базового уровня + облачная БД (PostgreSQL), health check, документация,	Хорошо

Уровень выполнения	Требования	Оценка
	автоматический деплой.	
Продвинутый	Выполнены требования среднего уровня + Docker, CI/CD (GitHub Actions), собственный домен, мониторинг/алертинг.	Отлично

Дополнительные условия:

1. Приложение должно корректно работать после перезапуска.
2. При защите студент должен продемонстрировать работу API через публичный URL.

6. Список рекомендуемой литературы и интернет-ресурсов

Литература (доступна в РФ)

1. Браун, И. (2021). *Express.js: Разработка веб-приложений на Node.js*. ДМК Пресс.
2. Кантелон, М., Хартер, М., & Холстейн, С. (2020). *Node.js в действии*. Питер.

Интернет-ресурсы

Название	Ссылка	Описание
Render.com Docs	https://render.com/docs	Официальная

Название	Ссылка	Описание
		документация
Railway.app Docs	https://docs.railway.app/	Документация Railway
Glitch.com Docs	https://glitch.com/help/	Документация Glitch
GitHub Actions	https://docs.github.com/ru/actions	Документация CI/CD
Docker Documentation	https://docs.docker.com/	Документация Docker
PostgreSQL	https://www.postgresql.org/docs/	Документация PostgreSQL
UptimeRobot	https://uptimerobot.com/	Мониторинг доступности
Sentry	https://sentry.io/	Отслеживание ошибок

Все ресурсы доступны и легальны на территории Российской Федерации.

Лабораторная работа № 18

Тема: Итоговая комплексная работа

Цель работы:

Разработать законченное веб-приложение с бэкендом на Express (или Flask), фронтендом на HTML/CSS/JS, базой данных SQLite, полным CRUD API и базовой аутентификацией, интегрируя все знания, полученные в течение курса.

Задачи работы:

1. Самостоятельно выбрать тему предметной области.
2. Спроектировать схему базы данных (минимум 3 связанные таблицы).
3. Разработать REST API с полным CRUD для основных сущностей.
4. Реализовать систему аутентификации (регистрация, вход, защита маршрутов).
5. Разработать клиентскую часть (HTML/CSS/JS) для взаимодействия с API.
6. Реализовать валидацию входных данных на сервере.
7. Настроить логирование запросов и ошибок.
8. Подготовить документацию и презентацию проекта.
9. Защитить проект преподавателю.

1. Теоретическое обоснование

1.1. Требования к итоговому проекту

Итоговая комплексная работа должна представлять собой полноценное веб-приложение, объединяющее все темы курса:

Модуль	Что должно быть реализовано
Backend	Express-сервер, маршрутизация, middleware
База данных	SQLite, 3+ связанные таблицы, внешние ключи
API	REST API, CRUD операции, GET/POST/PUT/DELETE
Аутентификация	Регистрация, вход, защищённые маршруты (JWT или сессии)
Валидация	Проверка входных данных, обработка ошибок
Логирование	Morgan + Winston (или аналог)
Конфигурация	Переменные окружения (dotenv)
Frontend	HTML/CSS, fetch API, динамическое обновление
Документация	README, описание API, инструкция по запуску

1.2. Схема оценивания

Критерий	Макс. балл
Функциональность API	20

Критерий	Макс. балл
Работа с базой данных	15
Аутентификация	15
Фронтенд	20
Качество кода	10
Документация	10
Защита (устно)	10
Итого	100

1.3. Возможные темы проектов

	Тема	Сущности
	Библиотека	Книги, Авторы, Жанры, Читатели, Выдача книг
	Интернет-магазин	Товары, Категории, Покупатели, Корзина, Заказы
	Система задач (Task manager)	Задачи, Проекты, Пользователи, Комментарии
	Блог	Статьи, Авторы, Категории, Комментарии, Теги
	Кинотека	Фильмы, Режиссёры,

	Тема	Сущности
		Жанры, Оценки, Отзывы
	Студенческая система	Студенты, Группы, Дисциплины, Оценки, Преподаватели
	Курсы	Курсы, Студенты, Преподаватели, Записи на курсы
	Спортивный клуб	Тренировки, Участники, Тренеры, Записи
	Отели	Отели, Номера, Бронирования, Гости
0	Ресторан	Блюда, Заказы, Сотрудники, Ингредиенты
1	Аптека	Лекарства, Производители, Заказы, Покупатели
2	Автосалон	Автомобили, Бренды, Клиенты, Тест-драйвы
3	Платежи	Счета, Платежи, Пользователи, Категории расходов
	События	Мероприятия,

	Тема	Сущности
4		Организаторы, Участники, Билеты
5	Расписание	Занятия, Преподаватели, Аудитории, Студенты

2. Методика выполнения работы

2.1. Пример проекта (Библиотека)

Шаг 1. Проектирование базы данных

sql

```sql
-- users — таблица пользователей (для аутентификации)
CREATE TABLE users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    username TEXT NOT NULL UNIQUE,
    email TEXT NOT NULL UNIQUE,
    password_hash TEXT NOT NULL,
    role TEXT DEFAULT 'user',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);


-- authors — авторы
CREATE TABLE authors (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL UNIQUE,
    birth_year INTEGER,
    country TEXT
);
```

```
-- genres — жанры
CREATE TABLE genres (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL UNIQUE
);

-- books — книги
CREATE TABLE books (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    title TEXT NOT NULL,
    author_id INTEGER NOT NULL,
    year INTEGER,
    price REAL,
    in_stock INTEGER DEFAULT 1,
    created_by INTEGER,     -- кто добавил книгу
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (author_id) REFERENCES authors(id),
    FOREIGN KEY (created_by) REFERENCES users(id)
);

-- book_genres — связь книг и жанров (многие-ко-многим)
CREATE TABLE book_genres (
    book_id INTEGER NOT NULL,
    genre_id INTEGER NOT NULL,
    FOREIGN KEY (book_id) REFERENCES books(id) ON DELETE CASCADE,
    FOREIGN KEY (genre_id) REFERENCES genres(id) ON DELETE CASCADE,
```

```
    PRIMARY KEY (book_id, genre_id)
);

-- loans — выдача книг (связь читателей и книг)
CREATE TABLE loans (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER NOT NULL,
    book_id INTEGER NOT NULL,
    loan_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    return_date TIMESTAMP,
    is_returned INTEGER DEFAULT 0,
    FOREIGN KEY (user_id) REFERENCES users(id),
    FOREIGN KEY (book_id) REFERENCES books(id)
);
```

Шаг 2. API Эндпоинты

Метод	Эндпоинт	Описание	Доступ
POST	/api/register	Регистрация	public
POST	/api/login	Вход (возвращает JWT)	public
GET	/api/profile	Профиль пользователя	auth
GET	/api/books	Список книг (с автором и жанрами)	auth

Метод	Эндпоинт	Описание	Доступ
GET	/api/books/:id	Книга по ID	auth
POST	/api/books	Создание книги	admin
PUT	/api/books/:id	Обновление книги	admin
DELETE	/api/books/:id	Удаление книги	admin
GET	/api/books/search	Поиск книг	auth
GET	/api/authors	Список авторов	auth
POST	/api/authors	Добавление автора	admin
GET	/api/genres	Список жанров	auth
POST	/api/genres	Добавление жанра	admin
P	/api/loans	Выдать книгу	use

Метод	Эндпоинт	Описание	Доступ
OST		пользователю	r
PUT	/api/loans/:id/return	Вернуть книгу	user
GET	/api/loans/my	Мои книги на руках	user
GET	/api/admin/users	Список пользователей	admin

Шаг 3. Структура проекта

text

```
library-app/
├── .env
├── .gitignore
├── package.json
├── app.js
├── config.js
├── logger.js
├── database/
│   └── init.sql
├── middleware/
│   ├── auth.js
│   └── validation.js
├── routes/
│   ├── auth.js
│   ├── books.js
```

```
│   ├── authors.js
│   ├── genres.js
│   └── loans.js
├── public/
│   ├── index.html
│   ├── css/
│   │   └── style.css
│   └── js/
│       ├── app.js
│       ├── auth.js
│       └── api.js
├── logs/
└── README.md
```

Шаг 4. Код основных модулей

config.js — конфигурация

javascript

```
require('dotenv').config();

const requiredEnv = ['JWT_SECRET'];
const missing = requiredEnv.filter(env => !process.env[env]);

if (missing.length > 0) {
    console.error(`✕ Отсутствуют переменные: ${missing.join(', ')}`);
    process.exit(1);
}

module.exports = {
    port: process.env.PORT || 3000,
    jwtSecret: process.env.JWT_SECRET,
```

```
    jwtExpiresIn: process.env.JWT_EXPIRES_IN || '7d',
    bcryptRounds: parseInt(process.env.BCRYPT_ROUNDS) || 10,
    dbPath: process.env.DB_PATH || './database/library.db',
    nodeEnv: process.env.NODE_ENV || 'development',
    isDev: process.env.NODE_ENV === 'development',
    logLevel: process.env.LOG_LEVEL || 'info'
};
```

middleware/auth.js — аутентификация

javascript

```
const jwt = require('jsonwebtoken');
const config = require('../config');

function authenticate(req, res, next) {
    const authHeader = req.headers.authorization;

    if (!authHeader || !authHeader.startsWith('Bearer ')) {
        return res.status(401).json({ error: 'Требуется авторизация' });
    }

    const token = authHeader.split(' ')[1];

    try {
        const decoded = jwt.verify(token, config.jwtSecret);
        req.user = decoded;
        next();
    } catch (error) {
        if (error.name === 'TokenExpiredError') {
            return res.status(401).json({ error: 'Токен истёк' });
        }
```

```
        return res.status(401).json({ error: 'Неверный токен' });
    }
}

function requireAdmin(req, res, next) {
    if (req.user.role !== 'admin') {
        return res.status(403).json({ error: 'Доступ запрещён. Требуются права администратора' });
    }
    next();
}

module.exports = { authenticate, requireAdmin };
```

routes/books.js — маршруты для книг

javascript

```
const express = require('express');
const router = express.Router();
const { authenticate, requireAdmin } = require('../middleware/auth');
const { validateBook } = require('../middleware/validation');

function createBookRoutes(db) {
    // GET /api/books — список книг
    router.get('/', authenticate, (req, res) => {
        const sql = `
            SELECT b.*, a.name as author_name,
                   GROUP_CONCAT(g.name, ', ') as genres
            FROM books b
            LEFT JOIN authors a ON b.author_id = a.id
            LEFT JOIN book_genres bg ON b.id = bg.book_id
```

```
        LEFT JOIN genres g ON bg.genre_id = g.id
        GROUP BY b.id
        ORDER BY b.id DESC
    `;

    db.all(sql, [], (err, rows) => {
        if (err) return res.status(500).json({ error: err.message });
        res.json(rows);
    });
});

// GET /api/books/:id — книга по ID
router.get('/:id', authenticate, (req, res) => {
    const id = req.params.id;
    const sql = `
        SELECT b.*, a.name as author_name
        FROM books b
        LEFT JOIN authors a ON b.author_id = a.id
        WHERE b.id = ?
    `;

    db.get(sql, [id], (err, row) => {
        if (err) return res.status(500).json({ error: err.message });
        if (!row) return res.status(404).json({ error: 'Книга не найдена' });
        res.json(row);
    });
});

// POST /api/books — создание книги (только админ)
```

```
router.post('/', authenticate, requireAdmin, validateBook, (req, res) => {
  const { title, author_id, year, price, in_stock, genre_ids } = req.body;

  db.run(
    `INSERT INTO books (title, author_id, year, price, in_stock, created_by)
     VALUES (?, ?, ?, ?, ?, ?)`,
    [title, author_id, year || null, price || 0, in_stock !== undefined ? in_stock : 1, req.user.id],
    function(err) {
      if (err) return res.status(500).json({ error: err.message });

      const bookId = this.lastID;

      // Добавление жанров
      if (genre_ids && genre_ids.length) {
        const stmt = db.prepare('INSERT INTO book_genres (book_id, genre_id) VALUES (?, ?)');
        genre_ids.forEach(genre_id => stmt.run([bookId, genre_id]));
        stmt.finalize();
      }

      res.status(201).json({ id: bookId, message: 'Книга добавлена' });
    }
  );
});

// PUT /api/books/:id — обновление книги (только админ)
router.put('/:id', authenticate, requireAdmin, (req, res) => {
```

```
    const id = req.params.id;
    const { title, author_id, year, price, in_stock } = req.body;

    db.get('SELECT * FROM books WHERE id = ?', [id], (err, book) => {
      if (err) return res.status(500).json({ error: err.message });
      if (!book) return res.status(404).json({ error: 'Книга не найдена' });

      db.run(
        `UPDATE books SET title = ?, author_id = ?, year = ?, price = ?, in_stock = ?
         WHERE id = ?`,
        [title || book.title, author_id || book.author_id, year !== undefined ? year : book.year,
         price !== undefined ? price : book.price, in_stock !== undefined ? in_stock : book.in_stock, id],
        function(err) {
          if (err) return res.status(500).json({ error: err.message });
          res.json({ message: 'Книга обновлена' });
        }
      );
    });
  });

  // DELETE /api/books/:id — удаление книги (только админ)
  router.delete('/:id', authenticate, requireAdmin, (req, res) => {
    const id = req.params.id;

    db.get('SELECT * FROM books WHERE id = ?', [id], (err, book) => {
      if (err) return res.status(500).json({ error: err.message });
```

```
        if (!book) return res.status(404).json({ error: 'Книга не найдена' });

        db.run('DELETE FROM books WHERE id = ?', [id], function(err) {
            if (err) return res.status(500).json({ error: err.message });
            res.status(204).send();
        });
    });
});

// GET /api/books/search — поиск
router.get('/search', authenticate, (req, res) => {
    const query = req.query.q;
    if (!query) return res.json([]);

    const searchPattern = `%${query}%`;
    const sql = `
        SELECT DISTINCT b.*, a.name as author_name
        FROM books b
        LEFT JOIN authors a ON b.author_id = a.id
        WHERE b.title LIKE ? OR a.name LIKE ?
        LIMIT 50
    `;

    db.all(sql, [searchPattern, searchPattern], (err, rows) => {
        if (err) return res.status(500).json({ error: err.message });
        res.json(rows);
    });
});
```

```
    return router;
}

module exports = createBookRoutes;
```

app.js — главный файл приложения

```
javascript
const express = require('express');
const cors = require('cors');
const path = require('path');
const sqlite3 = require('sqlite3').verbose();
const fs = require('fs');

const config = require('./config');
const logger = require('./logger');

const app = express();

// Middleware
app.use(cors());
app.use(express.json());
app.use(express.static('public'));

// Логирование запросов
app.use((req, res, next) => {
    logger.info(`${req.method} ${req.url}`);
    next();
});

// Подключение к БД
```

```
const dbDir = path.dirname(config.dbPath);
if (!fs.existsSync(dbDir)) fs.mkdirSync(dbDir, { recursive: true });

const db = new sqlite3.Database(config.dbPath, (err) => {
  if (err) {
    logger.error('Ошибка подключения к БД:', err);
    process.exit(1);
  }
  logger.info('Подключено к БД');

  // Инициализация таблиц
  const schema = fs.readFileSync(path.join(__dirname, 'database', 'init.sql'),
'utf8');
  db.exec(schema, (err) => {
    if (err) logger.error('Ошибка создания таблиц:', err);
    else logger.info('Таблицы созданы/проверены');
  });
});

// Routes
app.use('/api/auth', require('./routes/auth')(db));
app.use('/api/books', require('./routes/books')(db));
app.use('/api/authors', require('./routes/authors')(db));
app.use('/api/genres', require('./routes/genres')(db));
app.use('/api/loans', require('./routes/loans')(db));

// Health check
app.get('/health', (req, res) => {
  res.json({ status: 'OK', timestamp: new Date().toISOString() });
```

```
});

// Обработчик 404
app.use((req, res) => {
    res.status(404).json({ error: 'Маршрут не найден' });
});

// Глобальный обработчик ошибок
app.use((err, req, res, next) => {
    logger.error(err.stack);
    res.status(500).json({ error: 'Внутренняя ошибка сервера' });
});

// Запуск
app.listen(config.port, () => {
    logger.info(`Сервер запущен на порту ${config.port}`);
    logger.info(`Режим: ${config.nodeEnv}`);
    logger.info(`URL: http://localhost:${config.port}`);
});

process.on('SIGINT', () => {
    logger.info('Завершение работы...');
    db.close(() => process.exit(0));
});
```

Шаг 5. Фронтенд (HTML + JS)

public/index.html

```
html
<!DOCTYPE html>
<html lang="ru">
```

```
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Библиотека | Управление книгами</title>
    <link rel="stylesheet" href="/css/style.css">
</head>
<body>
    <div class="container">
        <header>
            <h1>📚 Система управления библиотекой</h1>
            <div id="auth-buttons" class="auth-buttons">
                <button id="login-btn" class="btn">Вход</button>
                <button id="register-btn" class="btn">Регистрация</button>
            </div>
            <div id="user-info" class="user-info" style="display:none">
                <span id="username"></span>
                <span id="user-role" class="role-badge"></span>
                <button id="logout-btn" class="btn small">Выход</button>
            </div>
        </header>

        <!-- Формы аутентификации -->
        <div id="login-form" class="form-modal" style="display:none">
            <h2>Вход</h2>
            <input type="text" id="login-username" placeholder="Имя пользователя">
            <input type="password" id="login-password" placeholder="Пароль">
```

```
            <button id="login-submit">Войти</button>
            <button id="login-cancel">Отмена</button>
        </div>

        <div id="register-form" class="form-modal" style="display:none">
            <h2>Регистрация</h2>
            <input type="text" id="reg-username" placeholder="Имя пользователя">
            <input type="email" id="reg-email" placeholder="Email">
            <input type="password" id="reg-password" placeholder="Пароль (мин. 6 символов)">
            <button id="register-submit">Зарегистрироваться</button>
            <button id="register-cancel">Отмена</button>
        </div>

        <!-- Основное содержимое (видно только авторизованным) -->
        <div id="app-content" style="display:none">
            <!-- Поиск -->
            <div class="search-section">
                <input type="text" id="search-input" placeholder="Поиск книг по названию или автору...">
                <button id="search-btn">🔍 Найти</button>
                <button id="reset-search">Сброс</button>
            </div>

            <!-- Форма добавления книги (только для админа) -->
            <div id="admin-panel" class="admin-panel" style="display:none">
                <h3>+ Добавить книгу</h3>
                <form id="add-book-form">
```

```
                    <input type="text" id="book-title" placeholder="Название"
required>
                    <select id="book-author"></select>
                    <input type="number" id="book-year" placeholder="Год">
                    <input type="number" id="book-price" step="0.01"
placeholder="Цена">
                    <select id="book-genres" multiple></select>
                    <button type="submit">Добавить книгу</button>
                </form>
            </div>

            <!-- Таблица книг -->
            <div class="books-section">
                <h3>📖 Список книг</h3>
                <div id="books-list">Загрузка...</div>
            </div>
        </div>
    </div>

    <script src="/js/api.js"></script>
    <script src="/js/auth.js"></script>
    <script src="/js/app.js"></script>
</body>
</html>
public/js/api.js
javascript
const API_BASE = window.location.origin;

let authToken = localStorage.getItem('token');
```

```
function setAuthToken(token) {
    authToken = token;
    if (token) {
        localStorage setItem('token', token);
    } else {
        localStorage removeItem('token');
    }
}

async function apiRequest(endpoint, options = {}) {
    const headers = {
        'Content-Type': 'application/json',
        ...options headers
    };

    if (authToken) {
        headers['Authorization'] = `Bearer ${authToken}`;
    }

    const response = await fetch(`${API_BASE}${endpoint}`, {
        ...options,
        headers
    });

    if (response status === 401) {
        // Токен истёк или недействителен
        setAuthToken(null);
        window location reload();
```

```
        throw new Error('Не авторизован');
    }

    const data = await response.json().catch(() => ({}));

    if (!response.ok) {
        throw new Error(data.error || `Ошибка ${response.status}`);
    }

    return data;
}
```

public/js/auth.js

javascript

```
async function register(username, email, password) {
    return apiRequest('/api/auth/register', {
        method: 'POST',
        body: JSON.stringify({ username, email, password })
    });
}

async function login(username, password) {
    const data = await apiRequest('/api/auth/login', {
        method: 'POST',
        body: JSON.stringify({ username, password })
    });

    if (data.token) {
        setAuthToken(data.token);
        return data;
```

```
    }
    throw new Error('Токен не получен');
}

async function logout() {
    setAuthToken(null);
    window.location.reload();
}

async function getProfile() {
    return apiRequest('/api/auth/profile');
}
```

public/js/app.js

javascript

```
// Глобальные переменные
let currentUser = null;

// Инициализация
document.addEventListener('DOMContentLoaded', async () => {
    setupEventListeners();

    if (authToken) {
        try {
            currentUser = await getProfile();
            showApp();
        } catch (error) {
            console.error('Ошибка авторизации:', error);
            showAuth();
        }
```

```
        } else {
            showAuth();
        }
    });

    function setupEventListeners() {
        // Кнопки аутентификации
        document.getElementById('login-btn').onclick = () => toggleModal('login-form', true);
        document.getElementById('register-btn').onclick = () => toggleModal('register-form', true);
        document.getElementById('logout-btn').onclick = logout;

        // Формы аутентификации
        document.getElementById('login-submit').onclick = handleLogin;
        document.getElementById('register-submit').onclick = handleRegister;
        document.getElementById('login-cancel').onclick = () => toggleModal('login-form', false);
        document.getElementById('register-cancel').onclick = () => toggleModal('register-form', false);

        // Поиск
        document.getElementById('search-btn').onclick = handleSearch;
        document.getElementById('reset-search').onclick = loadBooks;
        document.getElementById('search-input').addEventListener('keypress', (e) => {
            if (e.key === 'Enter') handleSearch();
        });
```

```
    // Добавление книги
    const addForm = document.getElementById('add-book-form');
    if (addForm) addForm.onsubmit = handleAddBook;
}

function showAuth() {
    document.getElementById('auth-buttons').style.display = 'flex';
    document.getElementById('user-info').style.display = 'none';
    document.getElementById('app-content').style.display = 'none';
}

function showApp() {
    document.getElementById('auth-buttons').style.display = 'none';
    document.getElementById('user-info').style.display = 'flex';
    document.getElementById('app-content').style.display = 'block';
    document.getElementById('username').textContent =
currentUser.username;
    document.getElementById('user-role').textContent = currentUser.role;

    if (currentUser.role === 'admin') {
        document.getElementById('admin-panel').style.display = 'block';
        loadAuthors();
        loadGenres();
    }

    loadBooks();
}

function toggleModal(modalId, show) {
```

```
    document.getElementById(modalId).style.display = show ? 'flex' : 'none';
}

async function handleLogin() {
    const username = document.getElementById('login-username').value;
    const password = document.getElementById('login-password').value;

    try {
        const data = await login(username, password);
        currentUser = data.user;
        toggleModal('login-form', false);
        showApp();
        showNotification('Вход выполнен успешно!', 'success');
    } catch (error) {
        showNotification(error.message, 'error');
    }
}

async function handleRegister() {
    const username = document.getElementById('reg-username').value;
    const email = document.getElementById('reg-email').value;
    const password = document.getElementById('reg-password').value;

    if (password.length < 6) {
        showNotification('Пароль должен содержать минимум 6 символов',
'error');
        return;
    }
```

```
        try {
            await register(username, email, password);
            toggleModal('register-form', false);
            showNotification('Регистрация успешна! Теперь войдите.', 'success');
        } catch (error) {
            showNotification(error.message, 'error');
        }
    }

    async function loadBooks() {
        const container = document.getElementById('books-list');
        container.innerHTML = '<div class="loading">Загрузка...</div>';

        try {
            const books = await apiRequest('/api/books');
            displayBooks(books);
        } catch (error) {
            container.innerHTML = `<div class="error">Ошибка: ${error.message}</div>`;
        }
    }

    function displayBooks(books) {
        const container = document.getElementById('books-list');

        if (!books || books.length === 0) {
            container.innerHTML = '<div class="empty">Нет книг. Добавьте первую!</div>';
            return;
```

```
        }

        const table = document.createElement('table');
        table.innerHTML = `
            <thead>
<tr><th>ID</th><th>Название</th><th>Автор</th><th>Год</th><th>Цена</th><th>Статус</th><th>Действия</th></tr>
            </thead>
            <tbody id="books-table-body"></tbody>
        `;

        const tbody = table.querySelector('#books-table-body');

        books.forEach(book => {
            const row = document.createElement('tr');
            row.innerHTML = `
                <td>${book.id}</td>
                <td>${escapeHtml(book.title)}</td>
                <td>${escapeHtml(book.author_name || '—')}</td>
                <td>${book.year || '—'}</td>
                <td>${book.price ? book.price + ' ₽' : '—'}</td>
                <td class="${book.in_stock ? 'status-in-stock' : 'status-out-of-stock'}">
                    ${book.in_stock ? 'В наличии' : 'Нет в наличии'}
                </td>
                <td>
                    ${currentUser.role === 'admin' ? `
```

```
                    <button class="btn-edit" onclick="editBook(${book.id})">✏</button>
                    <button class="btn-delete" onclick="deleteBook(${book.id})">🗑</button>
                ` : ''}
            </td>
        `;
        tbody.appendChild(row);
    });

    container.innerHTML = '';
    container.appendChild(table);
}

async function handleSearch() {
    const query = document.getElementById('search-input').value.trim();

    if (!query) {
        loadBooks();
        return;
    }

    const container = document.getElementById('books-list');
    container.innerHTML = '<div class="loading">Поиск...</div>';

    try {
        const books = await apiRequest(`/api/books/search?q=${encodeURIComponent(query)}`);
        displayBooks(books);
```

```
        } catch (error) {
            container.innerHTML = `<div class="error">Ошибка: ${error.message}</div>`;
        }
    }

    async function handleAddBook(e) {
        e.preventDefault();

        const title = document.getElementById('book-title').value;
        const author_id = document.getElementById('book-author').value;
        const year = document.getElementById('book-year').value;
        const price = document.getElementById('book-price').value;
        const genreSelect = document.getElementById('book-genres');
        const genre_ids = Array.from(genreSelect.selectedOptions).map(opt => parseInt(opt.value));

        try {
            await apiRequest('/api/books', {
                method: 'POST',
                body: JSON.stringify({ title, author_id, year: year || null, price: price || 0, genre_ids })
            });

            showNotification('Книга добавлена!', 'success');
            document.getElementById('add-book-form').reset();
            loadBooks();
        } catch (error) {
            showNotification(error.message, 'error');
```

```
    }
  }

  async function loadAuthors() {
    try {
      const authors = await apiRequest('/api/authors');
      const select = document.getElementById('book-author');
      select.innerHTML = authors.map(a => `<option value="${a.id}">${escapeHtml(a.name)}</option>`).join('');
    } catch (error) {
      console.error('Ошибка загрузки авторов:', error);
    }
  }

  async function loadGenres() {
    try {
      const genres = await apiRequest('/api/genres');
      const select = document.getElementById('book-genres');
      select.innerHTML = genres.map(g => `<option value="${g.id}">${escapeHtml(g.name)}</option>`).join('');
    } catch (error) {
      console.error('Ошибка загрузки жанров:', error);
    }
  }

  async function editBook(id) {
    const newTitle = prompt('Введите новое название:');
    if (!newTitle) return;
```

```
    try {
        await apiRequest(`/api/books/${id}`, {
            method: 'PUT',
            body: JSON.stringify({ title: newTitle })
        });
        showNotification('Книга обновлена!', 'success');
        loadBooks();
    } catch (error) {
        showNotification(error.message, 'error');
    }
}

async function deleteBook(id) {
    if (!confirm('Вы уверены, что хотите удалить эту книгу?')) return;

    try {
        await apiRequest(`/api/books/${id}`, { method: 'DELETE' });
        showNotification('Книга удалена!', 'success');
        loadBooks();
    } catch (error) {
        showNotification(error.message, 'error');
    }
}

function escapeHtml(str) {
    if (!str) return '';
    return str
        .replace(/&/g, '&amp;')
        .replace(/</g, '&lt;')
```

```
        .replace(/>/g, '&gt;')
        .replace(/"/g, '&quot;')
        .replace(/'/g, '&#39;');
}

function showNotification(message, type = 'info') {
    const notification = document.createElement('div');
    notification.className = `notification ${type}`;
    notification.textContent = message;
    document.body.appendChild(notification);

    setTimeout(() => {
        notification.remove();
    }, 3000);
}
```

2.2. Индивидуальные задания

2.2.1. Базовый уровень (15 вариантов)

Требования для получения оценки «удовлетворительно»:

№	Тема	Минимальный функционал
1	Библиотека	CRUD для книг, список авторов
2	Магазин	CRUD для товаров, список категорий
3	Задачи	CRUD для задач, отметка о выполнении

№	Тема	Минимальный функционал
4	Блог	CRUD для статей, просмотр статей
5	Фильмы	CRUD для фильмов, список режиссёров
6	Студенты	CRUD для студентов, группы
7	Курсы	CRUD для курсов, инструкторы
8	Рецепты	CRUD для рецептов, кухни
9	Автомобили	CRUD для авто, бренды
10	Песни	CRUD для песен, исполнители
11	Отели	CRUD для отелей, города
12	Лекарства	CRUD для лекарств, производители
13	Мероприятия	CRUD для мероприятий, локации
14	Заказы	CRUD для заказов, клиенты
1	Спортсмены	CRUD для спортсменов, виды

№	Тема	Минимальный функционал
5		спорта

2.2.2. Средний уровень (15 вариантов)

Дополнительные требования для оценки «хорошо» (к базовому уровню):

1. Аутентификация (регистрация, вход, JWT)
2. Связь многие-ко-многим (дополнительная таблица)
3. Поиск и фильтрация
4. Ролевая модель (user/admin)
5. Стилизованный фронтенд (CSS)
6. Документация API (README)

2.2.3. Продвинутый уровень (15 вариантов)

Дополнительные требования для оценки «отлично» (к среднему уровню):

1. Полноценный фронтенд с динамическим обновлением
2. Валидация на сервере и клиенте
3. Пагинация или бесконечный скроллинг
4. Экспорт данных в CSV/JSON
5. Деплой на облачную платформу (Render/Railway)
6. Презентация проекта

3. Требования к защите

На защите студент должен:

Продемонстрировать работающее приложение (локально или на хостинге).

Показать код и объяснить архитектуру (бэкенд, БД, фронтенд).

Ответить на вопросы по реализованному функционалу.

Показать документацию (README, описание API).

Примерный план презентации (5-7 минут)

Слайд	Содержание
1	Титульный лист (тема, ФИО, группа)
2	Цель и задачи работы
3	Схема базы данных (ER-диаграмма)
4	Архитектура приложения (бэкенд, фронтенд, БД)
5	API эндпоинты (таблица)
6	Демонстрация работы (скриншоты / живая демонстрация)
7	Заключение (что сделано, какие трудности)

4. Критерии оценивания

Критерий	Базовый (удовл.)	Средний (хорошо)	Продвинутый (отлично)
Функциональность	CRUD для 1 сущности	CRUD для 3+ сущностей	Полноценное приложение
База данных	2	3+ таблицы,	Нормали

Критерий	Базовый (удовл.)	Средний (хорошо)	Продвинутый (отлично)
	таблицы	связи	зация, индексы
Аутентификация	Отсутствует	JWT, регистрация/вход	Роли, защита маршрутов
Фронтенд	Минимальный (терминал)	HTML + CSS + fetch	Полноценное SPA
Качество кода	Работает	Структурирован	Комментарии, модульность
Документация	Отсутствует	README	README + описание API
Дополнительно	—	Поиск/фильтрация	Деплой, презентация

5. Список рекомендуемой литературы и интернет-ресурсов

Литература (доступна в РФ)

1. Браун, И. (2021). *Express.js: Разработка веб-приложений на Node.js*. ДМК Пресс.

2. Кантелон, М., Хартер, М., & Холстейн, С. (2020). *Node.js в действии*. Питер.
3. Флэнаган, Д. (2021). *JavaScript. Подробное руководство*. Символ-Плюс.

Интернет-ресурсы

Название	Ссылка	Описание
Express.js Docs	https://expressjs.com/ru/	Официальная документация
SQLite Docs	https://www.sqlite.org/docs.html	Документация SQLite
JWT.io	https://jwt.io/	Информация о JWT
MDN Web Docs	https://developer.mozilla.org/ru/	Документация по веб-технологиям
Render.com	https://render.com/	Платформа для деплоя

Все ресурсы доступны и легальны на территории Российской Федерации.

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

по организации и проведению самостоятельной работы по дисциплине
«Введение в бэкенд разработку»
для студентов направления подготовки
09.03.04 Программная инженерия Направленность (профиль)
«Разработка и сопровждение программного обеспечения»

Ставрополь 2026

ВВЕДЕНИЕ

Методические указания предназначены для организации самостоятельной работы студентов при изучении дисциплины «Введение в бэкенд-разработку». Дисциплина изучается на 2 курсе в 3 семестре и является основой для последующего освоения курса «Разработка Web-приложений».

Самостоятельная работа студентов является неотъемлемой частью учебного процесса и направлена на:

1. углубление и закрепление теоретических знаний;
2. формирование практических навыков разработки серверных приложений;
3. развитие способности к самообучению и работе с технической документацией;
4. подготовку к выполнению лабораторных работ и промежуточной аттестации.

Данные методические указания содержат:

1. общую характеристику самостоятельной работы (виды, трудоёмкость, планирование);
2. методические рекомендации по изучению теоретического материала;
3. методические указания по выполнению практических работ (лабораторных);
4. вопросы для собеседования по самостоятельно изученной литературе;
5. критерии оценивания компетенций, формируемых в ходе самостоятельной работы.

1. ОБЩАЯ ХАРАКТЕРИСТИКА САМОСТОЯТЕЛЬНОЙ РАБОТЫ

1.1. Цели и задачи самостоятельной работы

Цель самостоятельной работы – формирование у студентов способности к самоорганизации, самообучению и практическому применению полученных знаний при решении профессиональных задач в области бэкенд-разработки.

Задачи самостоятельной работы:

систематизация и закрепление теоретических знаний, полученных на лекциях;

формирование навыков работы с научно-технической литературой и документацией;

подготовка к выполнению лабораторных работ и их защите;

развитие навыков программирования и отладки серверных приложений;

подготовка к промежуточной аттестации (зачёту).

1.2. Виды самостоятельной работы

№	Вид самостоятельной работы	Трудоёмкость (часов)	Форма контроля
1	Изучение лекционного материала	10	Собеседование, опрос
2	Изучение дополнительной литературы и интернет-ресурсов	8	Собеседование
3	Подготовка к лабораторным работам (предварительная)	12	Допуск к ЛР
4	Оформление отчётов по лабораторным работам	18	Проверка отчёта
5	Подготовка к защите лабораторных работ	10	Защита ЛР
6	Выполнение индивидуальных заданий	8	Проверка кода

№	Вид самостоятельной работы	Трудоёмкость (часов)	Форма контроля
	повышенной сложности		
7	Подготовка итогового проекта	14	Защита проекта
8	Подготовка к зачёту	10	Зачёт
Итого		90	

1.3. Планирование самостоятельной работы по темам

Тема лекции	Самостоятельная работа	Рекомендуемое время (часы)
Тема 1. Введение в бэкенд-разработку	Изучение архитектуры веб-приложений, обзор технологий	4
Тема 2. Протокол HTTP	Изучение структуры HTTP, кодов состояния, заголовков	6
Тема 3. Модели обработки запросов	Изучение Event Loop, асинхронности	6
Тема 4. Введение в базы данных	Изучение реляционной модели, нормализации	8
Тема 5. SQL	Изучение SQL-запросов (JOIN, GROUP BY, подзапросы)	10

Тема лекции	Самостоятельная работа	Рекомендуемое время (часы)
Тема 6. Подключение БД к серверу	Изучение работы с SQLite из кода	8
Тема 7. REST API	Изучение принципов REST, сравнение с SOAP/GraphQL	8
Тема 8. Аутентификация и безопасность	Изучение JWT, хеширования, OWASP	10
Тема 9. Введение в масштабирование	Изучение балансировки, кэширования, деплоя	6
Итого		66

1.4. Рекомендации по организации времени

Составьте график работы на семестр с учётом сроков сдачи лабораторных работ.

Разделите крупные задачи (подготовка к зачёту, итоговый проект) на небольшие подзадачи.

Выделяйте ежедневно 1-2 часа на самостоятельную работу.

Чередуйте теоретическое изучение материала с практическими заданиями.

Не откладывайте выполнение лабораторных работ на последнюю неделю.

2. МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ ПО ИЗУЧЕНИЮ ТЕОРЕТИЧЕСКОГО МАТЕРИАЛА

2.1. Алгоритм изучения лекционного материала

Предварительный просмотр – бегло ознакомьтесь с материалом предстоящей лекции (5-10 минут).

Конспектирование на лекции – фиксируйте основные понятия, определения, схемы.

Постобработка – в течение 24 часов после лекции перечитайте конспект, дополните его.

Углублённое изучение – используйте рекомендуемую литературу для проработки сложных тем.

Самопроверка – ответьте на контрольные вопросы в конце каждой темы.

2.2. Работа с литературой и интернет-ресурсами

Методика работы с книгой:

1. Ознакомьтесь с оглавлением, найдите нужный раздел.
2. Прочитайте текст, выделяя ключевые идеи.
3. Составьте краткий конспект или ментальную карту.
4. Выполните практические примеры из книги (если они есть).

Рекомендуемые источники для самостоятельного изучения:

Тема	Литература / Ресурс	Разделы
HTTP/ HTTPS	MDN Web Docs	HTTP overview, Methods, Status codes
Express .js	Официальная документация	Routing, Middleware, Error handling
SQL	SQLite Documentation	SELECT, JOIN, GROUP BY
REST API	restfulapi.net	Principles, Best practices

Тема	Литература / Ресурс	Разделы
JWT	jwt.io	Introduction, Structure
Безопасность	OWASP Top 10	A1-A10

2.3. Подготовка к собеседованию

Для успешного прохождения собеседования по самостоятельному изучению литературы рекомендуется:

Составить список ключевых терминов по каждой теме.

Подготовить устные ответы на контрольные вопросы.

Привести примеры из реальной практики.

Использовать схемы и диаграммы для визуализации сложных концепций.

2.4. Рекомендации по конспектированию

Методы конспектирования:

Метод Корнелла – разделение страницы на три части (заметки, ключевые слова, вывод).

Ментальные карты – визуальное представление связей между понятиями.

Метод вопросов-ответов – формулирование вопросов к каждому абзацу.

3. МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ВЫПОЛНЕНИЮ ЛАБОРАТОРНЫХ РАБОТ

3.1. Подготовка к выполнению лабораторной работы

Предварительная подготовка (за 2-3 дня до выполнения):

Действие	Описание	Время
1. Изучение цели и задач	Прочитайте цель и задачи работы	5 мин
2. Повторение теории	Изучите теоретическое обоснование	20-30 мин
3. Анализ примеров	Разберите примеры из методических указаний	20 мин
4. Планирование	Составьте план выполнения работы	10 мин
5. Настройка среды	Подготовьте необходимые инструменты (Node.js, БД, Postman)	10 мин

3.2. Выполнение лабораторной работы

Пошаговый алгоритм выполнения:

Выбор варианта – получите номер варианта у преподавателя.

Создание структуры проекта:

```
labXX/
├── app.js
├── package.json
├── database.db (или /data)
├── public/ (для ЛР16)
└── README.md
```

Реализация базового функционала (базовый уровень):

1. Напишите код согласно требованиям.
2. Запустите сервер.
3. Протестируйте через Postman.
4. Добавление функционала среднего уровня (при необходимости):

5. Расширьте код дополнительными функциями.
6. Протестируйте новые эндпоинты.
7. Добавление функционала продвинутого уровня (при необходимости):
8. Реализуйте задания повышенной сложности.
9. Проведите комплексное тестирование.

Отладка и исправление ошибок:

1. Проверьте работу всех эндпоинтов.
2. Обработайте граничные случаи.
3. Оформление отчёта:
4. Подготовьте отчёт по установленной структуре.
5. Добавьте скриншоты и листинги кода.

3.3. Рекомендации по отладке кода

Типичные ошибки и способы их решения:

Ошибка	**Возможная причина**	**Решение**
Cannot GET /api/items	Маршрут не зарегистрирован	Проверьте app.get()
SQLITE_ERROR: no such table	Таблица не создана	Выполните CREATE TABLE
Error: listen EADDRINUSE	Порт уже занят	Смените порт или завершите процесс
Unexpected token	Синтаксическая ошибка в JSON	Проверьте формат JSON
Cannot read property of undefined	Обращение к несуществующему полю	Проверьте наличие данных

Инструменты для отладки:

console.log() – вывод промежуточных значений.

node --inspect – отладка в Chrome DevTools.

nodemon – автоматический перезапуск при изменениях.

Postman Console – просмотр запросов и ответов.

3.4. Порядок защиты лабораторной работы

Демонстрация работы (5-7 минут):

Запуск сервера.

Демонстрация работы эндпоинтов через Postman.

(Для ЛР16-18) демонстрация фронтенда.

Объяснение кода (5 минут):

Структура проекта.

Ключевые функции и их назначение.

Особенности реализации.

Ответы на контрольные вопросы (3-5 минут).

Обсуждение результатов (2-3 минуты).

3.5. Требования к оформлению отчёта по лабораторной работе

Отчёт должен быть оформлен в электронном виде и содержать следующие разделы:

Титульный лист (название работы, ФИО, группа, вариант).

Цель и задачи работы.

Теоретическое обоснование (краткий конспект по теме работы).

Ход выполнения работы:

Листинг кода с комментариями.

Скриншоты выполнения запросов (Postman или браузер).

Скриншоты результатов (ответы сервера, логи).

Ответы на контрольные вопросы (письменно, развёрнуто).

Заключение (выводы о проделанной работе).

Список использованных источников.

Требования к форматированию:

Шрифт: Times New Roman, 14 пт.

Межстрочный интервал: 1,5.

Поля: 2 см со всех сторон.

Код: моноширинный шрифт (Courier New, 12 пт).

4. ВОПРОСЫ ДЛЯ СОБЕСЕДОВАНИЯ ПО САМОСТОЯТЕЛЬНОМУ ИЗУЧЕНИЮ ЛИТЕРАТУРЫ

4.1. Вопросы по курсу в целом

1. Что такое бэкенд-разработка и чем она отличается от фронтенда?
2. Опишите архитектуру клиент-серверного взаимодействия.
3. Какие существуют HTTP-методы и для чего каждый используется?
4. Какие коды состояния HTTP вы знаете? Опишите группы кодов.
5. Что такое REST API? Назовите основные принципы REST.
6. Чем отличается SQLite от MySQL/PostgreSQL?
7. Что такое первичный и внешний ключи в базе данных?
8. Как выполнить объединение двух таблиц (JOIN)?
9. Что такое JWT и как он используется для аутентификации?
10. В чём разница между аутентификацией и авторизацией?
11. Что такое SQL-инъекция и как от неё защититься?
12. Какие существуют уровни логирования?
13. Для чего используются переменные окружения?
14. Что такое CORS и как его настроить?
15. Какие платформы можно использовать для деплоя Node.js-приложений?

4.2. Вопросы по разделам для углублённого изучения

Раздел 1. HTTP и сети

1. В чём разница между HTTP/1.1 и HTTP/2?
2. Что такое WebSocket и чем он отличается от HTTP?
3. Как работает HTTPS и SSL/TLS?

Раздел 2. Базы данных

1. Какие существуют нормальные формы в реляционных БД?
2. Что такое транзакция и ACID?
3. В чём разница между SQL и NoSQL базами данных?

Раздел 3. Безопасность

1. Что такое XSS (межсайтовый скриптинг) и как от него защититься?
2. Что такое CSRF и как от него защититься?
3. Что такое OWASP Top 10?

Раздел 4. Архитектура

1. Что такое MVC (Model-View-Controller)?
2. Что такое микросервисная архитектура?
3. Что такое Docker и контейнеризация?

4.3. Практические задачи для собеседования

1. Как спроектировать базу данных для интернет-магазина?
2. Какие эндпоинты вы предложите для API блога?
3. Как реализовать систему ролей (user/admin) в REST API?
4. Как организовать пагинацию в API?
5. Как защитить API от перебора паролей (брутфорса)?

4.4. Перечень источников для самостоятельного изучения

Основная литература:

1. Браун, И. (2021). Express.js: Разработка веб-приложений на Node.js. ДМК Пресс.
2. Кантелон, М., Хартер, М., & Холстейн, С. (2020). Node.js в действии. Питер.

Дополнительная литература:

3. Флэнаган, Д. (2021). JavaScript. Подробное руководство. Символ-Плюс.
4. Мартин, Р. (2020). Чистый код. Питер.

Интернет-ресурсы:

5. Официальная документация Node.js: https://nodejs.org/ru/docs/
6. Официальная документация Express: https://expressjs.com/ru/
7. Официальная документация SQLite: https://www.sqlite.org/docs.html
8. MDN Web Docs: https://developer.mozilla.org/ru/
9. Metanit (Node.js): https://metanit.com/web/nodejs/

5. КРИТЕРИИ ОЦЕНИВАНИЯ КОМПЕТЕНЦИЙ

5.1. Уровни освоения компетенций

Уровень	Характеристика	Оценка
Пороговый	Студент демонстрирует базовое понимание материала, может выполнить типовые задания при	Удовлетворительно

Уровень	Характеристика	Оценка
	поддержке преподавателя	
Базовый	Студент самостоятельно выполняет типовые задания, понимает основные концепции, может ответить на контрольные вопросы	Хорошо
Продвинутый	Студент демонстрирует глубокое понимание материала, может решать нестандартные задачи, аргументирует свои решения	Отлично

5.2. Критерии оценивания самостоятельной работы

Вид самостоятельной работы	Пороговый уровень	Базовый уровень	Продвинутый уровень
Изучение лекционного материала	Конспект лекций с записями	Конспект + дополнительные источники	Конспект + анализ + собственные примеры
Работа с литературой	Краткий пересказ прочитанного	Развёрнутый ответ с примерами	Анализ, сравнение разных источников
Подготовка к ЛР	Ознакомление с	Изучение примеров,	Самостоятельная реализация

Вид самостоятельной работы	Пороговый уровень	Базовый уровень	Продвинутый уровень
	теоретическим обоснованием	написание шаблона кода	без обращения к примерам
Оформление отчёта	Отчёт содержит все разделы, но с недочётами	Отчёт оформлен аккуратно, код с комментариями	Отчёт с анализом, сравнительными таблицами, диаграммами
Защита ЛР	Код работает, но студент не может объяснить детали	Код работает, студент объясняет основные моменты	Код оптимален, студент глубоко понимает все аспекты
Итоговый проект	Базовый функционал, есть недочёты	Весь функционал базового и среднего уровня	Полный функционал + дополнительные возможности

5.3. Критерии оценивания компетенций (УК, ОПК, ПК)

Компетенция	Индикатор	Пороговый уровень	Продвинутый уровень
УК-1	ИУК-1.1	Находит информацию в	Самостоятельно находит и критически

Компетенция	Индикатор	Пороговый уровень	Продвинутый уровень
		рекомендованных источниках	оценивает источники
УК-2	ИУК-2.1	Декомпозирует задачу с помощью преподавателя	Самостоятельно декомпозирует и планирует работу
ОПК-5	ИОПК-5.1	Устанавливает ПО по инструкции	Самостоятельно настраивает и разрешает конфликты
ОПК-6	ИОПК-6.2	Пишет работающий код	Пишет структурированный, документированный код
ПК-9	ИПК-9.1	Оформляет README на русском	Оформляет документацию на русском и английском
ПК-10	ИПК-10.1	Знает об угрозах, но не всегда применяет защиту	Последовательно применяет меры безопасности

5.4. Шкала перевода баллов в оценку

Для оценки самостоятельной работы используется накопительная система баллов:

Вид работы	Максимальный балл
Изучение лекционного материала	10
Работа с литературой	10
Подготовка к ЛР (18 работ × 2 балла)	36
Оформление отчётов (18 работ × 2 балла)	36
Защита ЛР (18 работ × 2 балла)	36
Итоговый проект	30
Активность на занятиях	10
Итого	168

Шкала перевода:

Сумма баллов (в % от максимума)	Оценка
85% – 100%	Отлично
70% – 84%	Хорошо
55% – 69%	Удовлетворительно
< 55%	Неудовлетворительно

5.5. Критерии выставления зачёта

Зачёт по дисциплине выставляется при выполнении следующих условий:

Выполнено и защищено не менее 16 лабораторных работ.

Сдан итоговый проект (Лабораторная работа № 18).

Успешно пройдено собеседование по самостоятельно изученной литературе (не менее 70% правильных ответов).

Накоплено не менее 55% от максимального количества баллов.

Дифференцированный зачёт (с оценкой):

Условия	Оценка
Выполнены все работы на базовом уровне, защищены без замечаний, накоплено 55-69% баллов	Удовлетвори тельно
Выполнены работы на базовом и среднем уровнях, защищены с хорошим объяснением, накоплено 70-84% баллов	Хорошо
Выполнены работы на продвинутом уровне, защищены с глубоким пониманием, активность на занятиях, накоплено 85-100% баллов	Отлично

Самостоятельная работа является важнейшим компонентом успешного освоения дисциплины «Введение в бэкенд-разработку». Систематическое выполнение всех видов самостоятельной работы в соответствии с данными методическими указаниями позволит вам:

глубоко усвоить теоретический материал;

приобрести практические навыки бэкенд-разработки;

сформировать профессиональные компетенции, необходимые для дальнейшего обучения;

успешно пройти промежуточную аттестацию.

Помните: регулярность и планомерность в освоении материала – залог успешного результата. Не откладывайте выполнение лабораторных работ на последний момент, своевременно обращайтесь за консультацией к преподавателю, активно используйте рекомендуемые источники информации.