Итоговое проектное задание для дисциплины

Структура проект

- **Итоговый проект:** Веб-приложение «Блог идей», где пользователи могут:
- **Главная страница** - приветствие и описание возможностей сайта
- **Галерея идей** - просмотр всех опубликованных идей
- **Добавление идеи** - форма для публикации новой идеи
- **Редактирование идеи** - форма для изменения существующей идеи
- **Авторизация** (будет добавлена позже)

Технологии: React + TypeScript + Vite (фронтенд), Express + TypeScript (бэкенд)

Часть 1: Настройка окружения разработчика

Лабораторная работа 1: Установка инструментов

Цель работы

Настроить среду разработки на Windows: установить Node.js, pnpm, Git и VS Code. Все эти инструменты необходимы для создания современного веб-приложения.

Ожидаемый результат

После выполнения всех шагов у вас будут установлены и проверены в работе: Node.js (среда выполнения JavaScript), pnpm (менеджер пакетов), Git (система контроля версий).

Шаг 1.1: Установка Node.js через fnm (менеджер версий)

Создайте папку проекта project_ivanov (ваша фамилия) и откройте ее (Open folder) в Visual Code.

Зачем это нужно: Node.js позволяет запускать JavaScript вне браузера. Нам он нужен для:

- Запуска сервера разработки Vite
- Установки пакетов через npm/pnpm
- Работы с бэкендом на Express

Что такое fnm: Fast Node Manager - программа, которая позволяет переключаться между разными версиями Node.js. Это важно, так как разные проекты могут требовать разные версии.

Действия: --- если возникает ошибка, то есть альтернатива ниже ---

1. Откройте **PowerShell** или терминал в VS Code (сочетание клавиш `Ctrl + ``)
2. Установите fnm через winget (встроенный магазин приложений Windows):

```
powershell
```

```
winget install Schniz.fnm
```

Результат:

```
Скачивание https://github.com/Schniz/fnm/releases/download/v1.39.0/fnm-windows.zip
                                        3.26 MB / 3.26 MB
Хэш установщика успешно проверен
Извлечение архива...
Архив извлечен
Запуск установки пакета...
Переменная среды пути изменена; перезапустите оболочку, чтобы использовать новое значение.
Добавлен псевдоним командной строки: "fnm"
Успешно установлено
```

Объяснение команды:

- `winget` - менеджер пакетов Windows
- `install` - команда для установки
- `Schniz.fnm` - идентификатор пакета (автор.название)

3. **ВАЖНО:** Перезапустите терминал полностью (закройте и откройте заново) и выполните:

```
powershell
```

```
fnm env --use-on-cd | Out-String | Invoke-Expression
```

Объяснение: Эта команда настраивает fnm для автоматической активации нужной версии Node.js при переходе в папку проекта.

4. Установите LTS версию Node.js:

```
powershell
```

```
fnm install --lts
```

Результат:

```
PS D:\Yandex.Disk\Документы\для института текущее\web\5 full stack\project> fnm install --
lts
Installing Node v24.15.0 (x64)
00:00:06                                        34.78 MiB/34.78 MiB (5.30 MiB/s, 0s)
```

Что такое LTS: Long-Term Support - версия, которая получает обновления безопасности в течение длительного времени. Для новичков и серьезных проектов это лучший выбор.

5. Сделайте эту версию версией по умолчанию:

```
powershell
```

```
fnm default lts-latest
```

6. **ПРОВЕРКА:** Выполните команды. Должны увидеть номера версий:

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

Ожидаемый результат: `v24.15.0 и 11.12.1` или новее.

--- Действия: альтернатива (если fnm не устанавливается) ---

Установка инструментов с использованием стандартных средств Windows:

- Вместо `fnm` → используем официальный установщик Node.js с сайта
- Вместо `pnpm` → используем стандартный npm (идет в комплекте с Node.js)

Необходимо настроить среду разработки на Windows: установить Node.js, Git и VS Code для создания современного веб-приложения.

Ожидаемый результат

После выполнения всех шагов у вас будут установлены и проверены в работе: Node.js (среда выполнения JavaScript), npm (менеджер пакетов) и Git (система контроля версий).

Шаг 1.1 (альтернатива): Установка Node.js через официальный установщик

- **Зачем это нужно:** Node.js позволяет запускать JavaScript вне браузера. Нам он нужен для:

- Запуска сервера разработки Vite
- Установки пакетов через npm
- Работы с бэкендом на Express

Действия:

1. Откройте браузер и перейдите на официальный сайт Node.js:

   ```
   https://nodejs.org/
   ```
2. Скачайте **LTS версию** (Long-Term Support):
   - Нажмите на зеленую кнопку "LTS"
   - Обычно это версия типа 20.x.x или 22.x.x
   - Файл будет называться `node-v20.xx.x-x64.msi`
3. Запустите скачанный установщик (`.msi` файл)
4. **Важно:** В процессе установки:
   - Нажимайте "Next" (Далее)
   - Примите лицензионное соглашение
   - Оставьте путь по умолчанию: `C:\Program Files\nodejs\`
   - Убедитесь, что отмечена опция "Add to PATH" (добавить в переменные среды)
   - Нажмите "Install" (Установить)
5. После завершения установки нажмите "Finish"
6. **ПРОВЕРКА:** Откройте **новое окно терминала** (обязательно новое!) и выполните:

   ```
   powershell

   node --version
   npm --version
   ```

Ожидаемый результат:

```
v20.18.0   (или другая LTS версия)
10.8.2     (или соответствующая версия npm)
```

Что делать, если команды не работают:

1. Перезагрузите компьютер
2. Проверьте переменные среды: Панель управления → Система → Дополнительные параметры → Переменные среды → PATH должна содержать `C:\Program Files\nodejs\`

Шаг 1.2: Установка pnpm (современный менеджер пакетов)

Зачем это нужно: pnpm устанавливает пакеты (библиотеки) для вашего проекта. Он быстрее и экономит место на диске по сравнению с npm.

Краткий синтаксис TypeScript/JavaScript: Пакеты - это код, написанный другими разработчиками, который мы используем в своем проекте. Например, React - это пакет.

Действия: --- если возникает ошибка, то есть альтернатива ниже ---

1. Установите pnpm глобально через npm:

```
powershell
npm install -g pnpm
```

Объяснение команды:

- `npm install` - команда для установки пакета
- `-g` (global) - установить глобально, чтобы было доступно в любом месте системы
- `pnpm` - название пакета

2. **ПРОВЕРКА:**

```
powershell
pnpm --version
```

Ожидаемый результат: Номер версии, например `10.33.2 или выше`

Шаг 1.2 (альтернатива): Проверка npm (менеджер пакетов)

- `npm` устанавливается автоматически вместе с Node.js. Дополнительных действий не требуется.

```
powershell
npm --version
```

- **Ожидаемый результат:** Номер версии (например, `10.8.2`)

- **Что такое npm:** Node Package Manager — менеджер пакетов для установки библиотек. Он будет использоваться вместо pnpm.

Шаг 1.3: Установка и настройка Git

Зачем это нужно: Git отслеживает изменения в коде, позволяет вернуться к любой предыдущей версии и синхронизироваться с GitHub.

Действия: --- если возникает ошибка, то есть альтернатива ниже ---

1. Установите Git через winget:

```
powershell

winget install --id Git.Git -e --source winget
```

Объяснение: `--id Git.Git` - точный идентификатор пакета Git

2. **ВАЖНО:** Перезапустите терминал (чтобы Git добавился в переменные PATH)
3. **ПРОВЕРКА:**

```
powershell

git --version
```

Ожидаемый результат: `git version 2.54.0.windows.1`

4. Настройте имя пользователя и email, если они не были настроены на Вашем компьютере ранее (они будут подписывать ваши коммиты):

```
powershell

git config --global user.name "Иван Иванов"
git config --global user.email "ivan@example.com"
```

Объяснение: Эти данные сохранятся в глобальном конфиге Git и будут автоматически добавляться к каждому вашему коммиту.

Шаг 1.3 (альтертатива): Установка Git (Система контроля версий)

Действия:

1. Скачайте Git с официального сайта:

   `https://git-scm.com/download/win`
2. Запустите установщик (файл `.exe`)
3. **Важные настройки в процессе установки:**
   - Выбирайте "Git from the command line and also from 3rd-party software"
   - Выберите "Use Windows default console window"
   - Настройки по умолчанию подходят для новичков

ПРОВЕРКА: Перезапустите терминал и выполните:

```
powershell


git --version
```

Ожидаемый результат: `git version 2.4x.x`

5. **Настройка:** Представьтесь Git (один раз):

```powershell
git config --global user.name "Ваше Имя"
git config --global user.email ваш-email@example.com
```

Шаг 1.4: Установка расширений VS Code

Действия:

- Откройте VS Code (`Ctrl+Shift+X` для панели расширений)
- Установите:
  - "Prettier - Code formatter" (автор: Esben Petersen)
  - "ES7+ React/Redux/React-Native snippets"

РЕЗУЛЬТАТ ЧАСТИ 1:

- Node.js установлен и работает
- pnpm установлен
- Git настроен

Часть 2: Создание фронтенда (React + TypeScript + Vite)

Лабораторная работа 2: Создание проекта и настройка маршрутизации

Цель работы

Создать React-приложение с TypeScript, настроить маршрутизацию (переходы между страницами) и разработать основные компоненты.

Структура страниц, которую мы создадим:

Страница	URL	Описание
Главная	`/`	Приветствие, описание сайта, призыв к действию
Галерея идей	`/gallery`	Список всех идей, каждая идея отображается в карточке
Добавление идеи	`/add`	Форма с полями: никнейм, название, описание
Редактирование идеи	`/edit/:id`	Форма, предварительно заполненная данными выбранной идеи

Шаг 2.1: Создание проекта Vite + React + TypeScript

Зачем это нужно: Vite - это быстрый инструмент сборки, который создает каркас проекта с готовой конфигурацией.

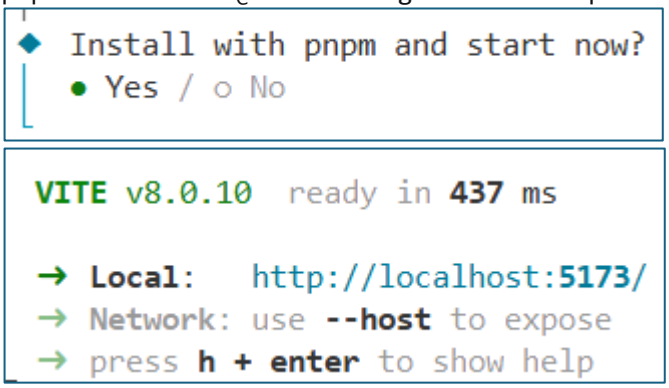
Действия: --- если возникает ошибка или не установлен pnpm, то есть альтернатива ниже ---

1. В терминале перейдите в папку проекта, если она итак не является текущей папкой:

```
powershell
cd project_ivanov
```

2. Создайте проект:

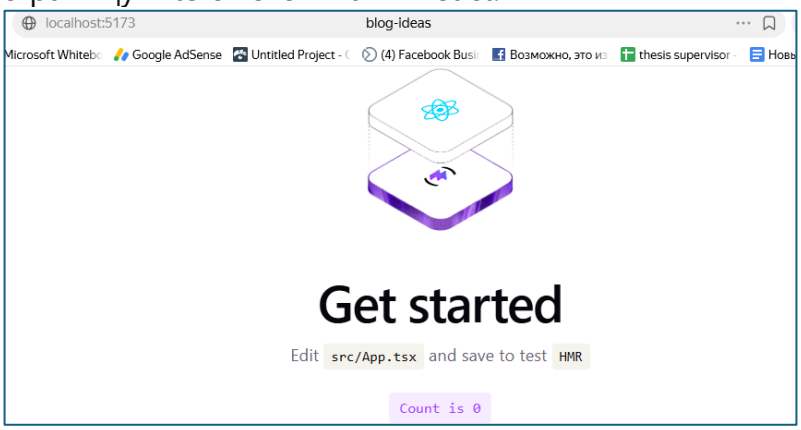
```
powershell
pnpm create vite@latest blog-ideas --template react-ts
```



Объяснение команды:

- `pnpm create` - создает новый проект через шаблон
- `vite@latest` - используем последнюю версию Vite
- `blog-ideas` - название папки проекта
- `--template react-ts` - шаблон с React и TypeScript

ПРОВЕРКА: Откройте браузер по адресу `http://localhost:5173`. Вы увидите стандартную страницу Vite с логотипами React.



3. Откройте новое окно powershell. Перейдите в папку проекта и установите зависимости. Они актуальны. Но запомните эту команду для актуализации зависимостей:

```
powershell
cd blog-ideas
pnpm install
```

```
Lockfile is up to date, resolution step is skipped
Already up to date
Done in 776ms using pnpm v10.32.1
```

Объяснение: `pnpm install` читает файл `package.json` и скачивает все необходимые библиотеки в папку `node_modules`.

4. Для запуска сервера разработки далее используйте команду (если сервер не запустился – запустите командой):

```
powershell
pnpm run dev
```

5. Перейдите в окно powershell с запущенным сервером. Остановите сервер: `Ctrl + C` (затем `Y` и `Enter` – если потребуется).

Шаг 2.1 (альтернатива): Создание проекта Vite + React + TypeScript

Действия:

1. В терминале перейдите в папку для проектов:

   ```
   powershell
   ```

```
cd project_ivanov
```

2. Создайте проект через Vite:

```
powershell

npm create vite@latest blog-ideas -- --template react-ts
```

Обратите внимание на `-- --` — это важно для передачи параметров в npm.

Альтернативный способ (если команда не работает):

```
powershell

npx create-vite@latest blog-ideas --template react-ts
```

3. Перейдите в папку проекта:

```
powershell

cd blog-ideas
```

4. Установите зависимости:

```
powershell

npm install
```

5. Запустите сервер разработки:

```
powershell

npm run dev
```

6. **Объяснение отличий от pnpm:**

pnpm	npm	Что делает
`pnpm create vite`	`npm create vite`	Создание проекта
`pnpm install`	`npm install`	Установка зависимостей
`pnpm run dev`	`npm run dev`	Запуск сервера

6. **ПРОВЕРКА:** Откройте `http://localhost:5173`

Шаг 2.2: Установка React Router

Теория:

React Router - это библиотека для навигации в React-приложениях. Обычные сайты при переходе по ссылкам перезагружают страницу полностью. React Router позволяет менять содержимое страницы **без перезагрузки**, создавая ощущение работы нативного приложения.

Проблема, которую решает React Router:

Без React Router у вас есть два плохих варианта:

1. **Одностраничное приложение без маршрутизации** - всё на одной странице, нельзя отправить ссылку на конкретный раздел
2. **Многостраничное приложение** - каждый переход перезагружает страницу, теряется состояние приложения

Как работает React Router:

```
Пользователь нажимает ссылку "Галерея идей"
         ↓
React Router перехватывает клик (не дает браузеру перезагрузить страницу)
         ↓
Изменяет URL в адресной строке на /gallery
         ↓
Находит в списке маршрутов компонент для URL /gallery
         ↓
Показывает компонент GalleryPage, скрывая предыдущий
```

Основные компоненты React Router:

Компонент	Назначение	Пример
`<BrowserRouter>`	Обертка для всего приложения, включает механику роутинга	`<BrowserRouter><App /></BrowserRouter>`
`<Routes>`	Контейнер для всех маршрутов	`<Routes>...</Routes>`
`<Route>`	Определяет связь "URL → компонент"	`<Route path="/gallery" element={<GalleryPage />} />`
`<Link>`	Компонент-ссылка (вместо `<a href>`)	`<Link to="/gallery">Галерея</Link>`
`useNavigate()`	Хук для программной навигации	`navigate('/gallery')`
`useParams()`	Хук для получения параметров из URL	`const { id } = useParams()`

Что происходит при установке:

```
powershell
pnpm add react-router-dom
```

Эта команда:

1. Скачивает код библиотеки из реестра npm
2. Сохраняет ее в `node_modules/react-router-dom`
3. Добавляет запись в `package.json` в секцию `dependencies`

Почему `react-router-dom`, а не просто `react-router`?

- `react-router` - ядро библиотеки

- `react-router-dom` - адаптация для браузерного окружения (есть также `react-router-native` для мобильных приложений)

Действия: --- если возникает ошибка или не установлен pnpm, то есть альтернатива ниже ---

Убедитесь, что вы находитесь в папке blog-ideas. Установите библиотеку маршрутизации:

```
powershell
```

```
pnpm add react-router-dom
```

```
Progress: resolved 188, reused 163, downloaded 0, added 4, done

dependencies:
+ react-router-dom 7.14.2

Done in 1.9s using pnpm v10.32.1
```

Краткий синтаксис TypeScript: `react-router-dom` - библиотека для навигации в React-приложениях. Она предоставляет компоненты:

- `BrowserRouter` - обертка для всего приложения
- `Routes` - контейнер для маршрутов
- `Route` - отдельный маршрут (URL → компонент)
- `Link` - компонент для перехода без перезагрузки страницы

Шаг 2.2 (альтернатива): Установка React Router

```
powershell
```

```
npm install react-router-dom
```

Шаг 2.3: Очистка проекта и создание структуры папок

Что нужно удалить: Стандартные файлы Vite, которые нам не нужны.

Действия:

1. В папке `src` удалите файлы:
   - `App.css`
   - `index.css`
   - `vite-env.d.ts` (если есть)
   - папку assets
2. Создайте следующую структуру папок. Перейдите в `blog-ideas`, затем:

```
powershell
```

```
cd src
mkdir components
```

```
mkdir pages
```

Итоговая структура:

```
blog-ideas/
├── src/
│   ├── components/     # Переиспользуемые компоненты (Header)
│   ├── pages/          # Компоненты-страницы
│   ├── App.tsx         # Главный компонент с маршрутами
│   ├── main.tsx        # Точка входа
│   └── types.ts        # Описания типов TypeScript (будет создан на следующем шаге)
├── index.html
├── package.json
└── tsconfig.json
└── и другие
```

Шаг 2.4: Создание типов TypeScript

Зачем это нужно: TypeScript позволяет описывать структуру данных. Это помогает избежать ошибок и делает код самодокументированным.

Краткий синтаксис TypeScript:

- `interface` - описание структуры объекта
- `export` - делает интерфейс доступным для импорта в других файлах
- `string`, `number` - базовые типы

Действия:

Создайте файл `src/types.ts`:

```
typescript
// Интерфейс - описание структуры объекта "Идея"
export interface Idea {
  id: number;         // уникальный идентификатор (число)
  nick: string;       // никнейм автора (строка)
  name: string;       // название идеи (строка)
  description: string; // описание идеи (строка)
}
```

Что мы здесь сделали: Объявили, что каждая идея обязательно должна иметь эти 4 поля с указанными типами.

Шаг 2.5: Создание компонента Header (шапка сайта)

Зачем это нужно: Header будет отображаться на всех страницах и содержать навигационные ссылки.

Краткий синтаксис JSX/TypeScript:

- `export function` - экспорт компонента для использования в других файлах
- `return (...)` - JSX разметка, которую компонент отрисовывает
- `style={{ }}` - инлайн-стили в React (двойные фигурные скобки: внешние для JS-выражения, внутренние для объекта)
- `<Link to="/">` - компонент `React Router` для навигации

Действия:

Создайте файл `src/components/Header.tsx` (обязательно с заглавной буквы!):

```
tsx
import { Link } from 'react-router-dom';

export function Header() {
  return (
    <header style={{
      backgroundColor: '#282c34',
      padding: '10px 20px',
      marginBottom: '20px'
    }}>
      <nav>
        <Link to="/" style={{
          color: 'white',
          marginRight: '20px',
          textDecoration: 'none'
        }}>
          Главная
        </Link>
        <Link to="/gallery" style={{
          color: 'white',
          marginRight: '20px',
          textDecoration: 'none'
        }}>
          Галерея идей
        </Link>
        <Link to="/add" style={{
          color: 'white',
          textDecoration: 'none'
        }}>
          Добавить идею
        </Link>
      </nav>
    </header>
  );
```

```
}
```

Шаг 2.6: Создание страниц

Зачем это нужно: Каждая страница - отдельный компонент, который будет отображаться при переходе на соответствующий URL.

Действия:

1. **Главная страница** - `src/pages/HomePage.tsx`:

```
tsx
function HomePage() {
  return (
    <div style={{ padding: '20px', textAlign: 'center' }}>
      <h1>Добро пожаловать в Блог Идей! ✨</h1>
      <p style={{ fontSize: '18px', maxWidth: '600px', margin: '20px auto' }}>
        Это место, где вы можете делиться своими мыслями,
        находить вдохновение от других и воплощать идеи в жизнь!
      </p>
      <div style={{ marginTop: '40px' }}>
        <h3>Что вы можете делать на сайте?</h3>
        <ul style={{ textAlign: 'left', maxWidth: '400px', margin: '0 auto' }}>
          <li>Публиковать свои идеи</li>
          <li>Смотреть идеи других пользователей</li>
          <li>Редактировать свои идеи</li>
          <li>Удалять идеи</li>
        </ul>
      </div>
    </div>
  );
}

export default HomePage;
```

2. **Галерея идей** - `src/pages/GalleryPage.tsx` (создаем заглушку, позже подключим API, т.е. создадим реальный запрос):

```
tsx
import { useState, useEffect } from 'react';
import { Link } from 'react-router-dom';
import type { Idea } from '../types';

function GalleryPage() {
  const [ideas, setIdeas] = useState<Idea[]>([]);
  const [loading, setLoading] = useState(true);

  // useEffect - хук для выполнения действий при загрузке компонента
  useEffect(() => {
    // TODO: Позже заменим на реальный запрос к бэкенду
    console.log('Загрузка идей...');
    setLoading(false);
  }, []);

  if (loading) return <div>Загрузка идей...</div>;

  return (
```

```
    <div style={{ padding: '20px' }}>
      <h1>Галерея идей</h1>
      {ideas.length === 0 ? (
        <p>Пока нет идей. Будьте первым, кто добавит идею!</p>
      ) : (
        <div style={{ display: 'flex', flexWrap: 'wrap', gap: '24px', justifyContent: 'center'
}}>
          {ideas.map((idea) => (
            <div key={idea.id} style={{
              border: '1px solid #ddd',
              borderRadius: '8px',
              padding: '16px',
              width: '280px',
              boxShadow: '0 2px 4px rgba(0,0,0,0.1)'
            }}>
              <h3>{idea.name}</h3>
              <p><strong>Автор:</strong> {idea.nick}</p>
              <p>{idea.description}</p>
              <Link to={`/edit/${idea.id}`}>
                <button>Редактировать</button>
              </Link>
            </div>
          ))}
        </div>
      )}
    </div>
  );
}

export default GalleryPage;
```

3. **Добавление идеи** - `src/pages/AddIdeaPage.tsx`:

```
tsx
import { useState } from 'react';
import { useNavigate } from 'react-router-dom';

function AddIdeaPage() {
  // useState - хук для создания состояния компонента
  // [значение, функция_для_изменения] = useState(начальное_значение)
  const [nick, setNick] = useState('');
  const [name, setName] = useState('');
  const [description, setDescription] = useState('');
  const navigate = useNavigate(); // для перенаправления

  const handleSubmit = async (e: React.FormEvent) => {
    e.preventDefault(); // предотвращаем перезагрузку страницы
    // TODO: Отправить данные на сервер
    console.log({ nick, name, description });
    alert('Идея будет отправлена после настройки бэкенда!');
  };

  return (
    <div style={{ maxWidth: '600px', margin: '40px auto', padding: '20px' }}>
      <h1>Добавить новую идею</h1>
      <form onSubmit={handleSubmit}>
        <div style={{ marginBottom: '16px' }}>
          <label>Ваш никнейм:</label>
          <input
            type="text"
            value={nick}
            onChange={(e) => setNick(e.target.value)}
            required
            style={{ width: '100%', padding: '10px', marginTop: '8px' }}
          />
```

```
        </div>
        <div style={{ marginBottom: '16px' }}>
          <label>Название идеи:</label>
          <input
            type="text"
            value={name}
            onChange={(e) => setName(e.target.value)}
            required
            style={{ width: '100%', padding: '10px', marginTop: '8px' }}
          />
        </div>
        <div style={{ marginBottom: '16px' }}>
          <label>Описание:</label>
          <textarea
            value={description}
            onChange={(e) => setDescription(e.target.value)}
            required
            rows={4}
            style={{ width: '100%', padding: '10px', marginTop: '8px' }}
          />
        </div>
        <button type="submit" style={{ padding: '10px 20px', cursor: 'pointer' }}>
          Опубликовать идею
        </button>
      </form>
    </div>
  );
}

export default AddIdeaPage;
```

4. **Редактирование идеи** - `src/pages/EditIdeaPage.tsx`:

```
tsx
import { useState, useEffect } from 'react';
import { useParams, useNavigate } from 'react-router-dom';

function EditIdeaPage() {
  const { id } = useParams(); // получаем id из URL
  const navigate = useNavigate();
  const [nick, setNick] = useState('');
  const [name, setName] = useState('');
  const [description, setDescription] = useState('');
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    // TODO: Загрузить данные идеи по id из URL
    console.log(`Загрузка идеи с id: ${id}`);
    setLoading(false);
  }, [id]);

  const handleSubmit = async (e: React.FormEvent) => {
    e.preventDefault();
    // TODO: Отправить обновленные данные на сервер
    console.log(`Обновление идеи ${id}:`, { nick, name, description });
    alert('Сохранение будет доступно после настройки бэкенда!');
  };

  if (loading) return <div>Загрузка...</div>;

  return (
    <div style={{ maxWidth: '600px', margin: '40px auto', padding: '20px' }}>
      <h1>Редактирование идеи #{id}</h1>
      <form onSubmit={handleSubmit}>
        <div style={{ marginBottom: '16px' }}>
```

```
          <label>Никнейм:</label>
          <input
            type="text"
            value={nick}
            onChange={(e) => setNick(e.target.value)}
            required
            style={{ width: '100%', padding: '10px', marginTop: '8px' }}
          />
        </div>
        <div style={{ marginBottom: '16px' }}>
          <label>Название:</label>
          <input
            type="text"
            value={name}
            onChange={(e) => setName(e.target.value)}
            required
            style={{ width: '100%', padding: '10px', marginTop: '8px' }}
          />
        </div>
        <div style={{ marginBottom: '16px' }}>
          <label>Описание:</label>
          <textarea
            value={description}
            onChange={(e) => setDescription(e.target.value)}
            required
            rows={4}
            style={{ width: '100%', padding: '10px', marginTop: '8px' }}
          />
        </div>
        <button type="submit" style={{ padding: '10px 20px', cursor: 'pointer' }}>
          Сохранить изменения
        </button>
      </form>
    </div>
  );
}

export default EditIdeaPage;
```

Шаг 2.7: Настройка маршрутов в App.tsx

Теория:

Что такое маршрутизация и как она работает?

Маршрутизация (routing) - это механизм, который определяет, какой компонент показывать при переходе на определенный URL.

Разбор будущего кода в `src/App.tsx`:

```
import { BrowserRouter, Routes, Route } from 'react-router-dom';
```

Объяснение импортов:

- `BrowserRouter` - компонент, который синхронизирует UI с URL. Он слушает изменения в адресной строке и обновляет отображаемый компонент
- `Routes` - компонент-контейнер, который ищет первый подходящий маршрут среди вложенных `Route`
- `Route` - компонент, связывающий путь (path) с компонентом (element)

Продолжение разбора будущего кода в `src/App.tsx`:

```
function App() {
  return (
    <BrowserRouter>      {/* 1. Включаем роутинг */}
      <Header />          {/* 2. Шапка отображается всегда */}
      <Routes>            {/* 3. Контейнер для маршрутов */}
        <Route path="/" element={<HomePage />} />
        <Route path="/gallery" element={<GalleryPage />} />
        <Route path="/add" element={<AddIdeaPage />} />
        <Route path="/edit/:id" element={<EditIdeaPage />} />
      </Routes>
    </BrowserRouter>
  );
}
```

Детальное объяснение каждой строки:

1. `<BrowserRouter>` - создает "историю" (history), отслеживает кнопки "Назад"/"Вперед" в браузере. Без него ссылки не будут работать.
2. `<Header />` - находится **вне** `<Routes>`, поэтому шапка будет видна на всех страницах. Это типичный паттерн: общие элементы (шапка, подвал, боковое меню) располагаются вне `<Routes>`.
3. `<Routes>` - при изменении URL пробегает по всем дочерним `<Route>` и находит совпадение. Показывает ТОЛЬКО первый подходящий маршрут (в отличие от старого `<Switch>`).
4. `path="/edit/:id"` - динамический маршрут. `:id` - это параметр, который может быть любым значением. Например:
    - `/edit/1` → параметр `id = "1"`
    - `/edit/abc` → параметр `id = "abc"`
    - `/edit/123/extra` → не совпадет

Что происходит при переходе на `/gallery`:

```
1. Пользователь нажимает <Link to="/gallery">
2. React Router изменяет URL (без перезагрузки)
3. BrowserRouter замечает изменение
4. Routes проверяет все Route:
   - path="/" → не совпадает (нужен точный /)
   - path="/gallery" → СОВПАДАЕТ!
   - path="/add" → не проверяется (уже нашли совпадение)
   - path="/edit/:id" → не проверяется
5. Routes рендерит <GalleryPage />
6. Остальные Route игнорируются
```

Порядок маршрутов важен:

```
<Routes>
  {/* ⚠️ Плохо: этот Route перехватит ВСЕ запросы, включая /gallery */}
  <Route path="/" element={<HomePage />} />
```

```
  {/* ✅ Хорошо: специфичные маршруты выше общих */}
  <Route path="/gallery" element={<GalleryPage />} />
  <Route path="/add" element={<AddIdeaPage />} />
  <Route path="/" element={<HomePage />} />
</Routes>
```

Действия:

Откройте `src/App.tsx`, удалите текущий код и вставьте следующий код:

```
tsx
import { BrowserRouter, Routes, Route } from 'react-router-dom';
import { Header } from './components/Header';
import HomePage from './pages/HomePage';
import GalleryPage from './pages/GalleryPage';
import AddIdeaPage from './pages/AddIdeaPage';
import EditIdeaPage from './pages/EditIdeaPage';

function App() {
  return (
    <BrowserRouter>      {/* Оборачиваем все приложение для работы роутинга */}
      <Header />          {/* Шапка будет на всех страницах */}
      <Routes>            {/* Контейнер для маршрутов */}
        <Route path="/" element={<HomePage />} />
        <Route path="/gallery" element={<GalleryPage />} />
        <Route path="/add" element={<AddIdeaPage />} />
        <Route path="/edit/:id" element={<EditIdeaPage />} />
      </Routes>
    </BrowserRouter>
  );
}

export default App;
```

Шаг 2.8: Настройка точки входа main.tsx

Действия:

Откройте `src/main.tsx`, удалите текущий код и вставьте следующий код:

```
tsx
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';

// Находим элемент с id="root" в index.html и рендерим приложение
ReactDOM.createRoot(document.getElementById('root')!).render(
  <React.StrictMode>   {/* StrictMode - помощник для поиска проблем */}
    <App />
  </React.StrictMode>,
);
```

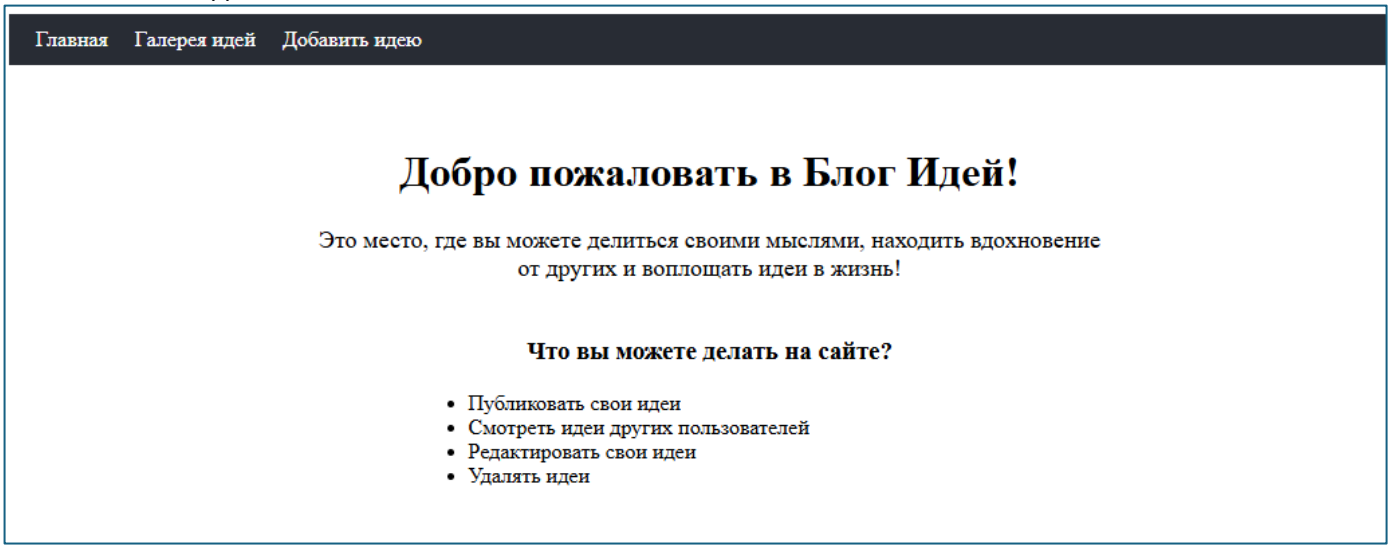
Шаг 2.9: Запуск и проверка

Действия:

1. Запустите сервер:

```
powershell
pnpm run dev
```

2. **ПРОВЕРКА в браузере:**
   - Перейдите по адресу `http://localhost:5173` → Главная страница
   - Нажмите "Галерея идей" → URL `/gallery`, видим сообщение "Пока нет идей"
   - Нажмите "Добавить идею" → URL `/add`, видим форму
   - Вручную введите `http://localhost:5173/edit/123` → страница редактирования идеи №123



РЕЗУЛЬТАТ ЧАСТИ 2:

- Работает навигация между страницами
- Формы для добавления и редактирования готовы (пока без отправки на сервер)

Часть 3: Git и GitHub (Первый коммит и публикация)

Лабораторная работа 3: Система контроля версий

Цель работы

Научиться использовать Git для отслеживания изменений в коде и публиковать проект на GitHub.

Ожидаемый результат

- Создан локальный Git-репозиторий
- Сделан первый коммит (сохранение состояния проекта)
- Проект загружен на GitHub

Шаг 3.1: Создание .gitignore

Зачем это нужно: Некоторые файлы не нужно хранить в Git (например, `node_modules` - там тысячи файлов, которые можно восстановить командой `pnpm install`).

Действия:

В корне проекта (папка `blog-ideas`) найдите файл (или создайте, если его нет) `.gitignore`. Добавьте код:

```
# Зависимости (не нужно хранить в Git)
node_modules/
dist/
dist-ssr/

# Логи
*.log
npm-debug.log*
pnpm-debug.log*

# Файлы окружения
.env
.env.local
.env.production

# IDE
.vscode/
.idea/
```

Шаг 3.2: Инициализация Git-репозитория

Действия:

1. Откройте терминал в корне проекта `blog-ideas`
2. Инициализируйте Git:

```
powershell
git init
```

Объяснение: Создает скрытую папку `.git`, где Git будет хранить историю изменений.

3. Проверьте статус:

```
powershell
git status
```

Ожидаемый результат: Список файлов, которые не отслеживаются Git (красным цветом).

Шаг 3.3: Первый коммит

Что такое коммит: Снимок состояния всех файлов в определенный момент времени. В будущем можно вернуться к любому коммиту.

Действия:

1. Добавьте все файлы в область подготовки (staging):

```
powershell
git add .
```

Объяснение: `.` означает "все файлы в текущей папке и подпапках".

2. Проверьте, что файлы добавлены:

```
powershell
git status
```

Ожидаемый результат: Файлы теперь зеленые (готовы к коммиту).

3. Сделайте коммит:

```
powershell
git commit -m "Initial commit: создан проект Blog Ideas"
```

Объяснение: `-m` - сообщение коммита (должно описывать, что изменилось).

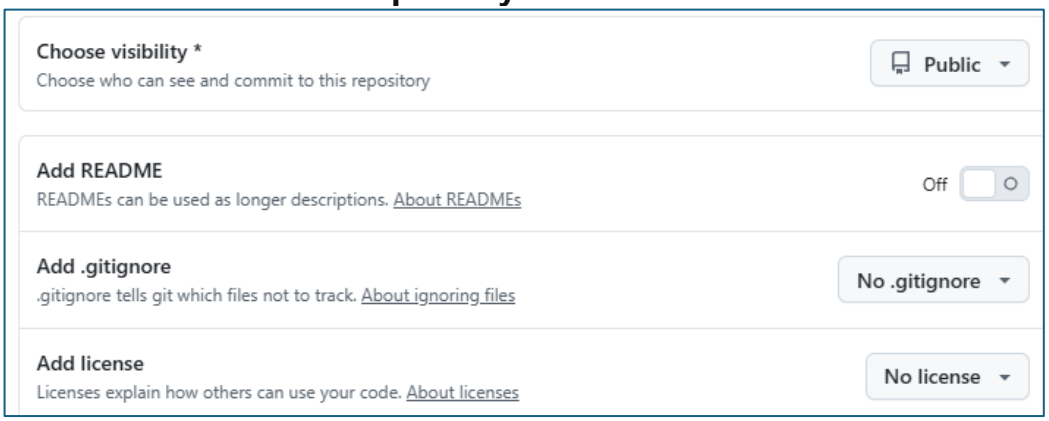
```
commit -m "Initial commit: создан проект Blog Ideas"
[master (root-commit) 3f47ee4] Initial commit: создан проект Blog Ideas
 23 files changed, 2296 insertions(+)
```

Шаг 3.4: Создание репозитория на GitHub

Зачем это нужно: GitHub - облачное хранилище для Git-репозиториев. Позволяет делиться кодом, работать в команде и иметь резервную копию.

Действия:

1. Откройте сайт GitHub и войдите в аккаунт
2. Нажмите зеленую кнопку **"New repository"**
3. Заполните форму:
   - **Repository name:** `blog-ideas`
   - **Description:** (необязательно) "Блог идей - веб-приложение на React + Express"
   - **Public/Private:** Public (или Private, если хотите скрытый репозиторий)
   - **Не создавайте README, .gitignore или лицензию** (мы уже создали локально)
4. Нажмите **"Create repository"**



Шаг 3.5: Связывание локального репозитория с GitHub

Действия:

GitHub покажет инструкции. Нам нужны команды для существующего репозитория.

Выполните в терминале (замените `ВАШ-АККАУНТ` на ваш логин в gitHub):

`powershell`

```
git remote add origin https://github.com/ВАШ-АККАУНТ/blog-ideas.git
```

Объяснение: `remote add` - добавляем удаленный репозиторий с именем `origin` (стандартное имя для основного удаленного репозитория).

Отправка кода на GitHub:

```
powershell
git push -u origin master
```

Результат:

```
Enumerating objects: 30, done.
Counting objects: 100% (30/30), done.
Delta compression using up to 6 threads
Compressing objects: 100% (29/29), done.
```

…

Объяснение:

- `git push` - отправить коммиты
- `-u origin master` - установить связь между локальной веткой `master` и удаленной `origin/master`
- В современных версиях ветка может называться `main`. Проверьте: `git branch`

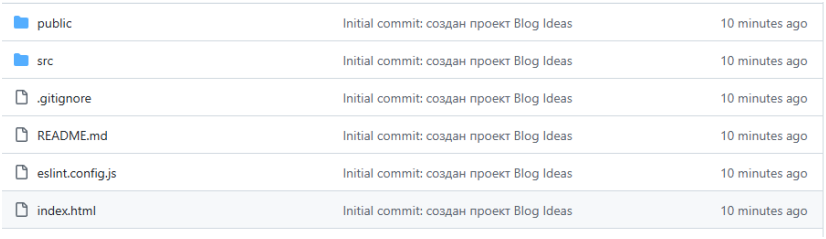
Если ветка называется `main`:

```
powershell
git push -u origin main
```

Шаг 3.6: Проверка на GitHub

Действия:

1. Обновите страницу репозитория на GitHub
2. **Ожидаемый результат:** Вы видите все файлы вашего проекта, загруженные на GitHub!



…

РЕЗУЛЬТАТ ЧАСТИ 3:

- Код сохранен в Git
- Проект опубликован на GitHub
- В будущем изменения можно публиковать командой `git push`

Часть 4: Настройка форматирования кода (Prettier)

Лабораторная работа 4: Prettier

Цель

Настроить автоматическое форматирование кода, чтобы весь проект выглядел единообразно.

Шаг 4.1: Установка Prettier

Действия: -- если возникает ошибка или не установлен pnpm, то есть альтернатива ниже --

```
powershell
pnpm add -D --save-exact prettier
```

Результат:

```
Packages: +1
+
Progress: resolved 189, reused 163, downloaded 1, added 1, done

devDependencies:
+ prettier 3.8.3

Done in 1.6s using pnpm v10.32.1
```

Объяснение:

- `-D` (devDependency) - нужен только во время разработки
- `--save-exact` - фиксирует точную версию (чтобы у всех разработчиков была одинаковая)

Действия (альтернатива):

```
powershell
npm install --save-dev --save-exact prettier
```

Добавьте скрипты в `package.json`:

```
json
```

```
"scripts": {
  "format": "prettier --write .",
  "format:check": "prettier --check ."
}
```

Далее перейти к шагу 4.5

Шаг 4.2: Создание конфигурации Prettier

Действия:

Создайте в корне файл `.prettierrc`:

```
json
{
  "semi": true,
  "singleQuote": true,
  "tabWidth": 2,
  "trailingComma": "es5",
  "printWidth": 100,
  "bracketSpacing": true,
  "arrowParens": "always",
  "endOfLine": "auto"
}
```

Объяснение настроек:

Настройка	Значение	Что делает
`semi`	`true`	Добавлять точку с запятой в конце
`singleQuote`	`true`	Использовать одинарные кавычки
`tabWidth`	`2`	Отступ - 2 пробела
`printWidth`	`100`	Переносить строки длиннее 100 символов

Шаг 4.3: Создание .prettierignore

Действия:

Создайте `.prettierignore`:

```
node_modules
dist
pnpm-lock.yaml
*.log
.env
```

Шаг 4.4: Добавление скриптов в package.json

Действия:

Откройте `package.json` и добавьте в секцию `"scripts"`:

```
json
"format": "prettier --write .",
"format:check": "prettier --check ."
```

Шаг 4.5: Проверка работы

Действия:

```
powershell
pnpm run format
```

Ожидаемый результат: Prettier прошелся по всем файлам и отформатировал их.

РЕЗУЛЬТАТ ЧАСТИ 4:

Код автоматически форматируется командой `pnpm format`.

Часть 5: Создание бэкенда (Express + TypeScript)

Лабораторная работа 5: Backend и API

Цель работы

Создать сервер на Express, который будет обрабатывать запросы от фронтенда и хранить данные.

Что такое API?

API (Application Programming Interface) - набор правил, по которым фронтенд общается с сервером. Мы создадим REST API - определенные URL, каждый из которых выполняет конкретное действие:

Метод	URL	Что делает
GET	`/ideas`	Получить все идеи

GET	`/ideas/:id`	Получить одну идею по ID
POST	`/ideas`	Создать новую идею
PUT	`/ideas/:id`	Обновить идею целиком
DELETE	`/ideas/:id`	Удалить идею

Шаг 5.1: Создание папки бэкенда

Действия:

1. Вернитесь в папку, где находится `blog-ideas` (фронтенд) – папку `project_ivanov`.
2. Создайте папку для бэкенда:

```
powershell
mkdir blog-ideas-backend
cd blog-ideas-backend
```

Шаг 5.2: Инициализация проекта (Инициализация package.json)

Теория:

Что такое package.json и зачем он нужен?

package.json - это "паспорт" вашего Node.js проекта. Этот файл содержит всю метаинформацию о проекте.

Команда:

```
powershell
pnpm init
-- или если не установлен pnpm, то есть альтернатива ниже –
powershell
   npm init -y
```

Что происходит при выполнении `pnpm init`:

```
1. Пакетный менеджер задает вопросы:
   - package name: (название папки) → можно нажать Enter
   - version: (1.0.0) → можно нажать Enter
   - description: (пусто) → можно написать "Бэкенд для блога идей"
   - entry point: (index.js) → изменим позже на dist/index.js
   - test command: (пусто) → пока не нужно
   - git repository: (пусто) → можно пропустить
```

- keywords: (пусто) → можно пропустить
- author: (пусто) → можно написать свое имя
- license: (ISC) → можно нажать Enter

2. После ответов создается файл package.json

Структура package.json (после инициализации):

```
json
{
  "name": "blog-ideas-backend",    // Название проекта
  "version": "1.0.0",              // Версия (мажорная.минорная.патч)
  "description": "Бэкенд для блога идей",
  "main": "index.js",              // Точка входа (будем менять)
  "scripts": {                     // Команды для запуска
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],                  // Ключевые слова для поиска
  "author": "Ваше имя",
  "license": "ISC"                 // Тип лицензии
}
```

Важные поля в package.json:

Поле	Назначение	Пример
`name`	Уникальное имя проекта	`blog-ideas-backend`
`version`	Семантическое версионирование (major.minor.patch)	`1.0.0`
`scripts`	Псевдонимы для команд	`"dev": "ts-node-dev src/index.ts"`
`dependencies`	Пакеты, нужные для работы приложения	`express: ^4.18.2`
`devDependencies`	Пакеты, нужные только для разработки	`typescript: ^5.0.0`
`main`	Точка входа при запуске через `node`	`dist/index.js`

Почему мы пропускаем вопросы?

- Большинство полей можно отредактировать позже вручную
- `entry point` мы изменим, когда настроим сборку TypeScript
- Сейчас важно создать сам файл

Действия:

```
powershell
pnpm init
```

Результат: добавлены записи в package.json:

```
{
  "name": "blog-ideas-backend",
  "version": "1.0.0",
```

```
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "packageManager": "pnpm@10.32.1"
}
```

Шаг 5.3: Установка зависимостей

Действия:

powershell

```
pnpm add express
```

-- или если не установлен pnpm, то есть альтернатива ниже –

Результат:

```
Packages: +65
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Progress: resolved 65, reused 2, downloaded 63, added 65, done

dependencies:
+ express 5.2.1

Done in 1.9s using pnpm v10.32.1
```

powershell

```
pnpm add -D typescript @types/node @types/express ts-node-dev
```

Результат:

```
 WARN  3 deprecated subdependencies found: glob@7.2.3, inflight@1.0.6, rimraf@2.7.1
Packages: +70
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Progress: resolved 136, reused 71, downloaded 64, added 70, done

devDependencies:
+ @types/express 5.0.6
+ @types/node 25.6.0
+ ts-node-dev 2.0.0
+ typescript 6.0.3

Done in 2.8s using pnpm v10.32.1
```

Действия (альтернатива):

powershell

- `npm install express`
- `npm install --save-dev typescript @types/node @types/express ts-node-dev`

Что мы установили:

Пакет	Для чего
`express`	Веб-фреймворк для создания сервера

`typescript`	Компилятор TypeScript в JavaScript
`@types/node`	Типы для Node.js (fs, path и др.)
`@types/express`	Типы для Express
`ts-node-dev`	Автоматический перезапуск сервера при изменениях

Шаг 5.4: Настройка TypeScript (Настройка TypeScript (tsconfig.json))

Теория:

Что такое tsconfig.json?

tsconfig.json - это конфигурационный файл TypeScript. Он указывает компилятору `tsc`, как преобразовывать TypeScript код в JavaScript.

Команда:

```
powershell
npx tsc --init
```

Что делает эта команда:

- Создает файл `tsconfig.json` с настройками по умолчанию
- Закомментирует большинство опций (начинаются с `//`)
- Нам нужно раскомментировать и настроить нужные опции

Разбор содержимого будущего файла tsconfig.json:

```
json
{
  "compilerOptions": {
    "target": "ES2022",
    "module": "commonjs",
    "outDir": "./dist",
    "rootDir": "./src",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules", "dist"]
}
```

Детальное объяснение каждой настройки:

Настройка	Значение	Что делает
`target`	`ES2022`	В какой версии JavaScript компилировать код. ES2022 - современная версия, поддерживается Node.js 18+

`module`	`commonjs`	Система модулей. CommonJS - стандарт Node.js (`require()` и `module.exports`)
`outDir`	`./dist`	Папка, куда компилятор складывает готовые JavaScript файлы
`rootDir`	`./src`	Папка с исходным TypeScript кодом
`strict`	`true`	Включает все строгие проверки типов. **Очень рекомендуется для новичков**
`esModuleInterop`	`true`	Позволяет импортировать CommonJS модули как ES-модули
`skipLibCheck`	`true`	Не проверять типы в библиотеках (ускоряет компиляцию)
`forceConsistentCasingInFileNames`	`true`	Требует одинакового регистра в именах файлов (важно для Windows/Linux)

Что означают `"include"` и `"exclude"`:

```
json
"include": ["src/**/*"]
```

- `**` - любая папка на любой глубине
- `*` - любой файл
- Значит: "включи все файлы в папке src и всех её подпапках"

```
json
"exclude": ["node_modules", "dist"]
```

- Исключает папки с зависимостями и результат компиляции

Процесс компиляции TypeScript:

```
Исходные файлы (src/index.ts)
        ↓
tsc читает tsconfig.json
        ↓
TypeScript → JavaScript (ES2022)
        ↓
Результат (dist/index.js)
```

Действия:

```
powershell
npx tsc -init
```

Результат: создан `tsconfig.json`

```
-- или если не установлен pnpm, то есть альтернатива ниже -
powershell
```

- `npx tsc --init`

Действия:

Откройте `tsconfig.json` и замените содержимое:

```
json
{
  "compilerOptions": {
    "target": "ES2022",
    "module": "commonjs",
    "outDir": "./dist",
    "rootDir": "./src",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules", "dist"]
}
```

Шаг 5.5: Создание сервера (базовый код index.ts, Создание эндпоинтов API)

Теория:

Разбор будущего кода сервера в index.ts

```
typescript
import express, { Request, Response } from 'express';
```

Что это значит:

- `import express` - импортируем библиотеку Express
- `{ Request, Response }` - импортируем типы TypeScript (нужны для типизации)

```
typescript
const app = express();
const PORT = 3001;
```

Объяснение:

- `app` - экземпляр Express приложения. Это главный объект, через который мы настраиваем сервер
- `PORT` - порт, на котором сервер будет слушать запросы. 3001 - стандартный порт для бэкенда (фронтенд на 5173)

```
typescript
let ideas = [
  { id: 1, nick: "ДжонДоу", name: "Умный будильник", description: "..." },
  // ...
];
let nextId = 4;
```

Почему мы используем `let`, а не `const`:

- `let` - переменная может меняться (будем добавлять/удалять идеи)
- `nextId` - увеличивается при каждом добавлении новой идеи

typescript
```
app.use(express.json());
```

Middlewares в Express:

- `app.use()` - подключает middleware (функцию, которая обрабатывает запрос до основного обработчика)
- `express.json()` - парсит JSON из тела запроса и добавляет его в `req.body`

Без этой строки:

typescript
```
app.post('/ideas', (req, res) => {
  console.log(req.body); // undefined!
});
```

С этой строкой:

typescript
```
app.post('/ideas', (req, res) => {
  console.log(req.body); // { nick: "...", name: "...", description: "..." }
});
```
typescript
```
app.use((req, res, next) => {
  res.header('Access-Control-Allow-Origin', 'http://localhost:5173');
  // ...
  next();
});
```

Что такое CORS (Cross-Origin Resource Sharing):

Обычно браузер запрещает сайту с одного домена делать запросы на другой домен. Это защита от атак.

- **Фронтенд:** `http://localhost:5173`
- **Бэкенд:** `http://localhost:3001`

Это **разные порты** → разные "источники" (origins).

Наш middleware добавляет специальные заголовки, которые разрешают браузеру делать такие запросы.

typescript
```
app.listen(PORT, () => {
  console.log(`Server running on http://localhost:${PORT}`);
});
```

`app.listen()` - запускает сервер. Функция внутри выполняется, когда сервер успешно запустился.

Действия:

1. В `blog-ideas-backend` создайте папку `src`:

powershell

```
mkdir src
```

2. Создайте файл `src/index.ts` и добавьте код:

typescript

```
import express, { Request, Response } from 'express';

const app = express();
const PORT = 3001;

// Express ДАННЫЕ (храним в памяти, пока без базы данных)
let ideas = [
  { id: 1, nick: "ДжонДоу", name: "Умный будильник", description: "Будильник, который
анализирует фазы сна" },
  { id: 2, nick: "Креативщик", name: "Платформа для обмена навыками", description: "Меняю
английский на программирование" },
  { id: 3, nick: "Эко-Френдли", name: "Карта переработки", description: "Точки приема
вторсырья с бонусами" }
];
let nextId = 4;

// MIDDLEWARE
app.use(express.json()); // Позволяет читать JSON из тела запроса

// Настройка CORS (разрешаем запросы с фронтенда)
app.use((req, res, next) => {
  res.header('Access-Control-Allow-Origin', 'http://localhost:5173');
  res.header('Access-Control-Allow-Headers', 'Origin, X-Requested-With, Content-Type,
Accept');
  res.header('Access-Control-Allow-Methods', 'GET, POST, PUT, DELETE, OPTIONS');
  if (req.method === 'OPTIONS') {
    return res.sendStatus(200);
  }
  next();
});

// ============ ЭНДПОИНТЫ API ============

/**
 * GET /ideas - получить все идеи
 */
app.get('/ideas', (req: Request, res: Response) => {
  res.json(ideas);
});

/**
 * GET /ideas/:id - получить одну идею по ID
 */
app.get('/ideas/:id', (req: Request, res: Response) => {
```

```
  const id = parseInt(req.params.id as string);
  const idea = ideas.find(i => i.id === id);

  if (!idea) {
    return res.status(404).json({ message: 'Идея не найдена' });
  }

  res.json(idea);
});

/**
 * POST /ideas - создать новую идею
 */
app.post('/ideas', (req: Request, res: Response) => {
  const { nick, name, description } = req.body;

  // Валидация - проверяем, что все поля заполнены
  if (!nick || !name || !description) {
    return res.status(400).json({ message: 'Все поля обязательны' });
  }

  const newIdea = {
    id: nextId++,
    nick,
    name,
    description
  };

  ideas.push(newIdea);
  res.status(201).json(newIdea);
});

/**
 * PUT /ideas/:id - полностью обновить идею
 */
app.put('/ideas/:id', (req: Request, res: Response) => {
  const id = parseInt(req.params.id as string);
  const { nick, name, description } = req.body;

  const index = ideas.findIndex(i => i.id === id);

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

  ideas[index] = { id, nick, name, description };
  res.json(ideas[index]);
});

/**
 * DELETE /ideas/:id - удалить идею
 */
```

```
app.delete('/ideas/:id', (req: Request, res: Response) => {
  const id = parseInt(req.params.id as string);
  const index = ideas.findIndex(i => i.id === id);

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

  ideas.splice(index, 1);
  res.status(204).send();
});

// ЗАПУСК СЕРВЕРА
app.listen(PORT, () => {
  console.log(`Server running on http://localhost:${PORT}`);
});
```

Объяснения:

Что такое эндпоинт (endpoint)?

- **Эндпоинт** - это конкретный URL и HTTP метод, по которому клиент может обратиться к серверу.
- **Структура эндпоинта:**

HTTP-метод + путь = эндпоинт
GET /ideas → получить идеи
POST /ideas → создать идею
PUT /ideas/1 → обновить идею 1
DELETE /ideas/1 → удалить идею 1

GET эндпоинт с параметром:

```
typescript
app.get('/ideas/:id', (req: Request, res: Response) => {
  const id = getSafeId(req.params.id);

  if (id === null) {
    return res.status(400).json({ message: 'Некорректный ID' });
  }

  const idea = ideas.find(i => i.id === id);

  if (!idea) {
    return res.status(404).json({ message: 'Идея не найдена' });
  }

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

Шаг 5.6: Настройка скриптов в package.json

Действия:

Откройте `package.json` бэкенда и обновите `"scripts"`:

```
json
"scripts": {
  "dev": "ts-node-dev --respawn --transpile-only src/index.ts",
  "build": "tsc",
  "start": "node dist/index.js"
}

-- или если не установлен pnpm, то есть альтернатива:
json
"scripts": {
  "dev": "ts-node-dev --respawn --transpile-only src/index.ts",
  "build": "tsc",
  "start": "node dist/index.js",
  "clean": "rimraf dist"
}
```

Шаг 5.7: Запуск сервера

Действия:

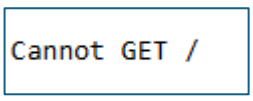
```
powershell
pnpm run dev

-- или если не установлен pnpm, то есть альтернатива:
powershell
```

- npm run dev

Ожидаемый результат: `Server running on http://localhost:3001`

Перейдите по указанному адресу:



Шаг 5.8: Проверьте работу API (частично)

Действия:

Откройте новое окно терминала.

GET запрос (получить все идеи):

powershell

```
curl http://localhost:3001/ideas
```

```
StatusCode        : 200
StatusDescription : OK
Content           : [{"id":1,"nick":"ДжонДоу","name":"Умный будильник","description":"Буд
                    ильник, который анализирует фазы сна"},{"id":2,"nick":"Креативщик","n
                    ame":"Платформа для обмена навыками","description":"Меняю англ...
RawContent        : HTTP/1.1 200 OK
```

...

GET запрос (получить одну идею):

powershell

```
curl http://localhost:3001/ideas/1
```

```
StatusCode        : 200
StatusDescription : OK
Content           : {"id":1,"nick":"ДжонДоу","name":"Умный будильник","description":"Буди
                    льник, который анализирует фазы сна"}
RawContent        : HTTP/1.1 200 OK
```

...

Объяснение:

HTTP статус-коды в API:

Код	Значение	Когда используется
200	OK	Успешный GET, PUT
201	Created	Успешный POST (создание)
204	No Content	Успешный DELETE
400	Bad Request	Некорректный ID или данные
404	Not Found	Ресурс не найден

РЕЗУЛЬТАТ ЧАСТИ 5:

Бэкенд запущен и ожидает запросов.

Часть 6: Работа с API - GET запросы

Лабораторная работа 6: Получение данных с сервера

Примечание: при выполнении работы без pnpm нужно учесть далее при выполнении:

Единственное, что нужно изменить в файлах GalleryPage.tsx, AddIdeaPage.tsx, EditIdeaPage.tsx — это **установка CORS заголовков** в бэкенде уже настроена правильно.

Цель

Научиться отправлять GET-запросы с фронтенда на бэкенд для получения списка идей и отдельной идеи.

Шаг 6.1: Обновление GalleryPage - получение всех идей (GET запрос в GalleryPage)

Теория:

Зачем: При загрузке страницы галереи нужно показать все идеи с сервера.

Краткий синтаксис:

- `fetch()` - встроенная функция для HTTP-запросов
- `await` / `async` - работа с асинхронным кодом (ждем ответ от сервера)
- `response.json()` - преобразует ответ сервера в JavaScript объект

Что такое fetch и как он работает?

fetch - встроенная функция браузера для HTTP-запросов.

Базовый синтаксис:

typescript

```typescript
const response = await fetch(url, options);
const data = await response.json();
```

Асинхронность (async/await):

typescript

```typescript
// Без async/await (через промисы)
fetch('/api/data')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error(error));

// С async/await (чище и понятнее)
async function loadData() {
  try {
    const response = await fetch('/api/data');  // Ждем ответ
    const data = await response.json();         // Ждем парсинг JSON
    console.log(data);
  } catch (error) {
    console.error(error);
  }
}
```

Что происходит при `await fetch(...)`:

text

1. Браузер отправляет HTTP-запрос к серверу
2. Функция "засыпает" (не блокирует интерфейс)
3. Сервер обрабатывает запрос
4. Браузер получает ответ
5. Функция "просыпается" и продолжает работу

Разбор будущего кода в GalleryPage.tsx:

typescript

```typescript
const loadIdeas = async () => {
  try {
    setLoading(true);  // Показываем индикатор загрузки

    // 1. Отправляем GET-запрос
    const response = await fetch('http://localhost:3001/ideas');

    // 2. Проверяем статус ответа
    if (!response.ok) {
      throw new Error(`Ошибка HTTP: ${response.status}`);
    }

    // 3. Парсим JSON в JavaScript объект
    const data = await response.json();

    // 4. Обновляем состояние компонента
    setIdeas(data);
  } catch (err) {
    setError('Не удалось загрузить идеи');
  } finally {
    setLoading(false);  // Убираем индикатор загрузки
  }
};
```

useEffect - когда загружать данные:

typescript

```typescript
useEffect(() => {
  loadIdeas();  // Загружаем данные при монтировании компонента
}, []);  // Пустой массив = выполнить только один раз
```

Действия:

Перейдите на фронтенд. Обновите `src/pages/GalleryPage.tsx`:

tsx

```tsx
import { useState, useEffect } from 'react';
import { Link } from 'react-router-dom';
import type { Idea } from '../types';

function GalleryPage() {
  const [ideas, setIdeas] = useState<Idea[]>([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState('');

  // Функция загрузки идей с сервера
  const loadIdeas = async () => {
    try {
      setLoading(true);
      // Отправляем GET-запрос на бэкенд
```

```
      const response = await fetch('http://localhost:3001/ideas');

      if (!response.ok) {
        throw new Error(`Ошибка HTTP: ${response.status}`);
      }

      const data = await response.json();
      setIdeas(data);
    } catch (err) {
      setError('Не удалось загрузить идеи. Убедитесь, что сервер запущен.');
      console.error(err);
    } finally {
      setLoading(false);
    }
  };

  // Загружаем идеи при монтировании компонента
  useEffect(() => {
    loadIdeas();
  }, []);

  if (loading) return <div>Загрузка идей...</div>;
  if (error) return <div style={{ color: 'red' }}>{error}</div>;

  return (
    <div style={{ padding: '20px' }}>
      <h1>Галерея идей</h1>
      <div style={{ display: 'flex', flexWrap: 'wrap', gap: '24px', justifyContent: 'center' }
}>
        {ideas.map((idea) => (
          <div key={idea.id} style={{
            border: '1px solid #ddd',
            borderRadius: '8px',
            padding: '16px',
            width: '280px',
            boxShadow: '0 2px 4px rgba(0,0,0,0.1)'
          }}>
            <h3>{idea.name}</h3>
            <p><strong>Автор:</strong> {idea.nick}</p>
            <p>{idea.description}</p>
            <div style={{ display: 'flex', gap: '8px', marginTop: '16px' }}>
              <Link to={`/edit/${idea.id}`}>
                <button style={{ padding: '6px 12px', cursor: 'pointer' }}>Редактировать</butt
on>
              </Link>
              <button
                onClick={() => {/* TODO: DELETE запрос */}}
                style={{ padding: '6px 12px', cursor: 'pointer', backgroundColor: '#dc3545', c
olor: 'white', border: 'none' }}
              >
                Удалить
              </button>
            </div>
          </div>
        ))}
      </div>
    </div>
  );
}

export default GalleryPage;
```

Шаг 6.2: Проверка GET запроса

Действия:

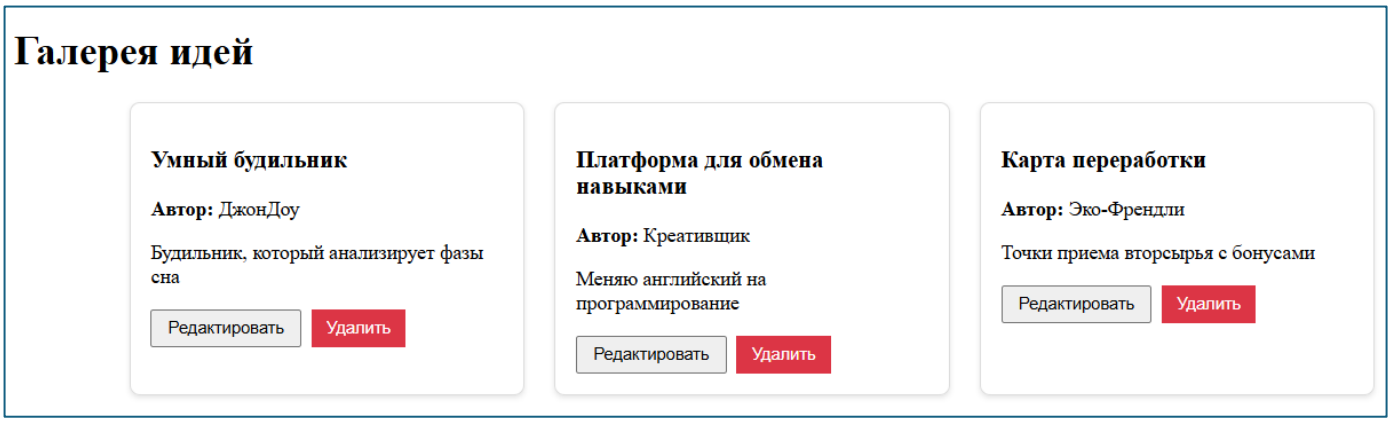
1. Убедитесь, что бэкенд запущен (терминал с `pnpm run dev` в папке `blog-ideas-backend`)
2. Запустите фронтенд (новый терминал в папке `blog-ideas`):

```
powershell

pnpm run dev
```

3. Откройте `http://localhost:5173/gallery`

Ожидаемый результат: Вы видите 3 карточки с идеями, загруженные с сервера!



4. Откройте инструменты разработчика (F12), вкладка Network → виден запрос `http://localhost:3001/ideas`

РЕЗУЛЬТАТ:

GET-запрос работает, данные отображаются.

Часть 7: Работа с API - POST запрос (Добавление идеи)

Лабораторная работа 7: Создание новых данных

Цель

Научиться отправлять POST-запросы для создания новых идей на сервере.

Теория:

Что такое POST запрос и форма?

Отличие GET от POST:

Характеристика	GET	POST
Назначение	Получение данных	Отправка/создание данных
Данные в URL	Да (query string)	Нет (в теле запроса)
Лимит данных	~2048 символов	Ограничен сервером
Кеширование	Да	Нет
Идемпотентность	Да (повтор дает тот же результат)	Нет

Почему форма не отправляется стандартным способом:

Обычная HTML-форма:

```
html
<form action="/submit" method="POST">
  <input name="nick" />
  <button type="submit">Отправить</button>
</form>
```

При отправке страница перезагружается.

В React мы делаем иначе:

```
tsx
<form onSubmit={handleSubmit}>
  {/* поля формы */}
</form>
```

1. Отменяем стандартное поведение: `e.preventDefault()`
2. Сами собираем данные из состояния
3. Сами отправляем fetch-запрос
4. Сами решаем, что делать после успеха

Структура POST-запроса:

```
typescript
const response = await fetch('http://localhost:3001/ideas', {
  method: 'POST',                    // 1. Тип запроса
  headers: {
    'Content-Type': 'application/json', // 2. Тип данных
  },
  body: JSON.stringify({             // 3. Тело запроса
    nick,
    name,
    description,
  }),
});
```

Почему `JSON.stringify()`:

- Сервер ожидает текст в формате JSON
- Объект JavaScript нужно преобразовать в строку

```
javascript
const obj = { nick: "Анна", name: "Идея" };
JSON.stringify(obj);  // '{"nick":"Анна","name":"Идея"}'
```

Обработка ответа:

```typescript
if (!response.ok) {
  const errorData = await response.json();
  throw new Error(errorData.message);
}
```

`response.ok` = true для статусов 200-299, false для 400-599

Кратко итог:

В обычном JavaScript форма отправляется и перезагружает страницу. В React мы:

1. Отменяем стандартную отправку (`e.preventDefault()`)
2. Собираем данные из состояния (`nick`, `name`, `description`)
3. Отправляем их на сервер через `fetch`
4. После успеха перенаправляем пользователя

Почему используются фигурные скобки `{}` в JSX?

- `{nick}` - вставка значения переменной
- `{` и `}` - границы JavaScript-выражения внутри JSX
- `onChange={(e) => setNick(e.target.value)}` - функция-обработчик в фигурных скобках

Шаг 7.1: Обновление AddIdeaPage (POST запрос в AddIdeaPage)

Действия:

Обновите `src/pages/AddIdeaPage.tsx`:

```tsx
import { useState } from 'react';
import { useNavigate } from 'react-router-dom';

function AddIdeaPage() {
  const [nick, setNick] = useState('');
  const [name, setName] = useState('');
  const [description, setDescription] = useState('');
  const [isSubmitting, setIsSubmitting] = useState(false);
  const [error, setError] = useState('');
  const navigate = useNavigate();

  const handleSubmit = async (e: React.FormEvent) => {
    e.preventDefault(); // ОТМЕНЯЕМ ПЕРЕЗАГРУЗКУ СТРАНИЦЫ
```

```
    setIsSubmitting(true);
    setError('');

    try {
      // ОТПРАВЛЯЕМ POST-ЗАПРОС НА СЕРВЕР
      const response = await fetch('http://localhost:3001/ideas', {
        method: 'POST',           // Тип запроса
        headers: {
          'Content-Type': 'application/json',  // Говорим, что отправляем JSON
        },
        body: JSON.stringify({    // Превращаем объект в JSON-строку
          nick,
          name,
          description,
        }),
      });

      if (!response.ok) {
        const errorData = await response.json();
        throw new Error(errorData.message || 'Ошибка при добавлении идеи');
      }

      // УСПЕХ - перенаправляем на галерею
      navigate('/gallery');
    } catch (err) {
      setError(err instanceof Error ? err.message : 'Произошла ошибка');
    } finally {
      setIsSubmitting(false);
    }
  };

  return (
    <div style={{ maxWidth: '600px', margin: '40px auto', padding: '20px', border: '1px solid
#ccc', borderRadius: '8px' }}>
      <h1>Добавить новую идею</h1>

      {error && (
        <div style={{ backgroundColor: '#f8d7da', color: '#721c24', padding: '10px', borderRad
ius: '4px', marginBottom: '20px' }}>
          {error}
        </div>
      )}

      <form onSubmit={handleSubmit}>
        <div style={{ marginBottom: '16px' }}>
          <label style={{ display: 'block', marginBottom: '8px', fontWeight: 'bold' }}>Ваш ник
нейм:</label>
          <input
            type="text"
            value={nick}
            onChange={(e) => setNick(e.target.value)}
            required
            disabled={isSubmitting}
            style={{ width: '100%', padding: '10px', border: '1px solid #ccc', borderRadius: '
4px' }}
            placeholder="Введите ваш никнейм"
          />
        </div>

        <div style={{ marginBottom: '16px' }}>
          <label style={{ display: 'block', marginBottom: '8px', fontWeight: 'bold' }}>Названи
е идеи:</label>
          <input
            type="text"
```

```
            value={name}
            onChange={(e) => setName(e.target.value)}
            required
            disabled={isSubmitting}
            style={{ width: '100%', padding: '10px', border: '1px solid #ccc', borderRadius: '
4px' }}
            placeholder="Краткое название вашей идеи"
          />
        </div>

        <div style={{ marginBottom: '16px' }}>
          <label style={{ display: 'block', marginBottom: '8px', fontWeight: 'bold' }}>Описани
е идеи:</label>
          <textarea
            value={description}
            onChange={(e) => setDescription(e.target.value)}
            required
            rows={4}
            disabled={isSubmitting}
            style={{ width: '100%', padding: '10px', border: '1px solid #ccc', borderRadius: '
4px' }}
            placeholder="Подробно опишите вашу идею..."
          />
        </div>

        <button
          type="submit"
          disabled={isSubmitting}
          style={{
            backgroundColor: '#28a745',
            color: 'white',
            border: 'none',
            padding: '10px 20px',
            borderRadius: '4px',
            cursor: isSubmitting ? 'not-allowed' : 'pointer',
            opacity: isSubmitting ? 0.7 : 1
          }}
        >
          {isSubmitting ? 'Отправка...' : 'Опубликовать идею'}
        </button>
      </form>
    </div>
  );
}

export default AddIdeaPage;
```

Шаг 7.2: Проверка POST запроса

Действия:

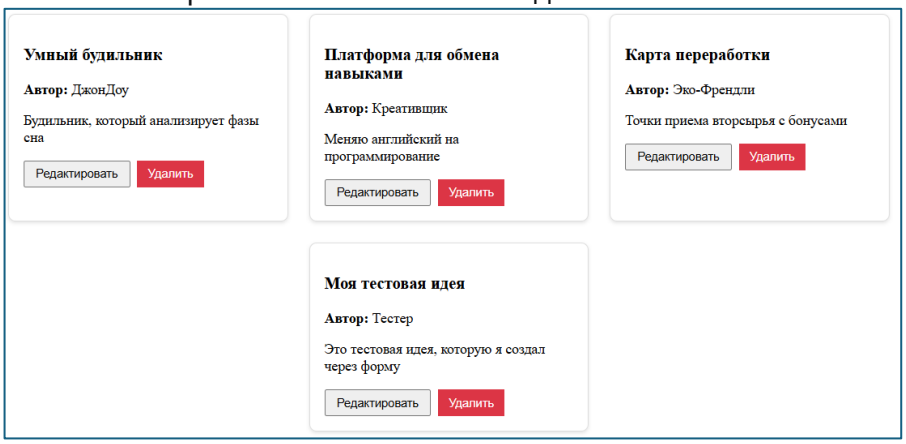
1. Убедитесь, что бэкенд и фронтенд запущены
2. Перейдите на страницу `http://localhost:5173/add`
3. Заполните форму:

- Никнейм: `Тестер`
- Название: `Моя тестовая идея`
- Описание: `Это тестовая идея, которую я создал через форму`

4. Нажмите "Опубликовать идею"

Ожидаемый результат:

- После отправки происходит перенаправление на `/gallery`
- В галерее появляется новая идея



5. **ПРОВЕРКА через curl (в отдельном терминале):**

```
powershell
curl http://localhost:3001/ideas
```

Вы увидите, что массив идей теперь содержит 4 элемента, включая новую, если она вмещается.

РЕЗУЛЬТАТ:

POST-запрос работает, новые идеи добавляются на сервер.

Часть 8: Работа с API - DELETE запрос (Удаление идеи)

Лабораторная работа 8: Удаление данных

Цель

Научиться удалять идеи через DELETE-запрос.

Теория:

Что такое DELETE запрос?

DELETE - HTTP метод для удаления ресурсов.

Особенности DELETE:

- Обычно не имеет тела запроса
- ID ресурса передается в URL
- Успешный ответ часто возвращает статус 204 (No Content) - без тела

Реализация удаления:

```typescript
const deleteIdea = async (id: number) => {
  // 1. Подтверждение действия (безопасность)
  if (!confirm('Вы уверены, что хотите удалить эту идею?')) {
    return;
  }

  try {
    // 2. Отправляем DELETE запрос
    const response = await fetch(`http://localhost:3001/ideas/${id}`, {
      method: 'DELETE',
    });

    // 3. Проверяем успешность
    if (!response.ok) {
      throw new Error('Ошибка при удалении');
    }

    // 4. Обновляем UI (оптимистичное обновление)
    setIdeas(prevIdeas => prevIdeas.filter(idea => idea.id !== id));
  } catch (err) {
    alert('Не удалось удалить идею');
  }
};
```

Оптимистичное обновление (optimistic update):

```typescript
// ❌ Плохо: ждем подтверждения от сервера
const response = await fetch(...);
if (response.ok) {
  setIdeas(prev => prev.filter(i => i.id !== id));
}

// ✅ Хорошо: обновляем сразу, если ошибка - показываем
setIdeas(prev => prev.filter(i => i.id !== id));  // Сразу убираем из UI
try {
  await fetch(...);
} catch (error) {
  // Если ошибка - возвращаем удаленный элемент обратно
  loadIdeas();  // Перезагружаем список с сервера
  alert('Ошибка при удалении');
}
```

filter() метод массива:

```javascript
const ideas = [{ id: 1, name: 'A' }, { id: 2, name: 'B' }, { id: 3, name: 'C' }];
const filtered = ideas.filter(idea => idea.id !== 2);
// filtered = [{ id: 1, name: 'A' }, { id: 3, name: 'C' }]
```

Шаг 8.1: Добавление удаления в GalleryPage (DELETE запрос в GalleryPage)

Действия:

Обновите `src/pages/GalleryPage.tsx`, добавив функцию `deleteIdea`:

```
tsx
// Добавьте эту функцию внутрь компонента GalleryPage, после loadIdeas (30 строка)
const deleteIdea = async (id: number) => {
  if (!confirm('Вы уверены, что хотите удалить эту идею?')) {
    return;
  }

  try {
    const response = await fetch(`http://localhost:3001/ideas/${id}`, {
      method: 'DELETE',
    });

    if (!response.ok) {
      throw new Error('Ошибка при удалении');
    }

    // Обновляем список - удаляем идею из состояния
    setIdeas(prevIdeas => prevIdeas.filter(idea => idea.id !== id));
  } catch (err) {
    alert('Не удалось удалить идею');
    console.error(err);
  }
};
```

А в кнопке удаления замените `onClick`:

```
tsx
<button
  onClick={() => deleteIdea(idea.id)}
  style={{ padding: '6px 12px', cursor: 'pointer', backgroundColor: '#dc3545', color: 'white',
border: 'none', borderRadius: '4px' }}
>
  Удалить
</button>
```

Действия:

Убедитесь, что вы находитесь в терминале в папке blog-ideas и запустите команду форматирования кода, чтобы поправить код во всех измененных файлах:

```
pnpm run format
```

Шаг 8.2: Проверка DELETE запроса

Действия:

1. Обновите страницу галереи
2. Нажмите "Удалить" на любой идее
3. Подтвердите удаление

Ожидаемый результат: Идея исчезает из галереи без перезагрузки страницы.

ПРОВЕРКА через curl:

Затем проверьте, что идеи с id=1 больше нет (если удалена первая идея):

```
powershell
curl http://localhost:3001/ideas
```

РЕЗУЛЬТАТ:

DELETE-запрос работает.

Часть 9: Работа с API - GET запрос для одной идеи (Редактирование)

Лабораторная работа 9: Получение данных для редактирования

Цель

При открытии страницы редактирования загрузить существующие данные идеи и отобразить их в форме.

Теория:

Что такое динамический маршрут?

Динамический маршрут - это URL, который содержит переменную часть.

```
tsx
<Route path="/edit/:id" element={<EditIdeaPage />} />
```

- `:id` - параметр, может быть любым значением
- При переходе на `/edit/5`, параметр `id = "5"`

- При переходе на `/edit/abc`, параметр `id = "abc"`

Получение параметра в компоненте:

```typescript
const { id } = useParams<{ id: string }>();
```

useParams() - хук React Router, возвращает объект со всеми параметрами URL.

Почему нужно указывать тип `<{ id: string }>`:

- По умолчанию useParams() возвращает `Record<string, string | undefined>`
- Мы явно говорим TypeScript, что параметр `id` существует и это строка

Загрузка данных для редактирования:

```typescript
useEffect(() => {
  const loadIdea = async () => {
    try {
      setLoading(true);

      // GET запрос для получения одной идеи
      const response = await fetch(`http://localhost:3001/ideas/${id}`);

      if (!response.ok) {
        throw new Error('Идея не найдена');
      }

      const data = await response.json();

      // Заполняем поля формы данными с сервера
      setNick(data.nick);
      setName(data.name);
      setDescription(data.description);
    } catch (err) {
      setError(err instanceof Error ? err.message : 'Ошибка загрузки');
    } finally {
      setLoading(false);
    }
  };

  if (id) {
    loadIdea();
  }
}, [id]);  // Зависимость от id - если id изменится, загрузим снова
```

Почему id в массиве зависимостей?

- Если пользователь перейдет с `/edit/1` на `/edit/2`, эффект выполнится снова
- Без этого старые данные останутся в форме

PUT запрос для обновления:

```typescript
```

```
const response = await fetch(`http://localhost:3001/ideas/${id}`, {
  method: 'PUT',                        // Обновление ресурса
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({
    nick,
    name,
    description,
  }),
});
```

Отличие PUT от PATCH:

Метод	Действие	Пример
PUT	Полная замена ресурса	Отправить все поля (даже если не менялись)
PATCH	Частичное обновление	Отправить только измененные поля

- Мы используем PUT, так как отправляем все поля формы.

Теперь необходимо добавить к проекту лабораторную работу по отделению стилей: то есть созданию для страниц файлов со стилем и переносу туда правил стилей из основных страниц

Шаг 9.1: Обновление EditIdeaPage (GET запрос для одной идеи)

Действия:

Обновите `src/pages/EditIdeaPage.tsx`:

```
tsx
import { useState, useEffect } from 'react';
import { useParams, useNavigate } from 'react-router-dom';

function EditIdeaPage() {
  const { id } = useParams<{ id: string }>();
  const navigate = useNavigate();
  const [nick, setNick] = useState('');
  const [name, setName] = useState('');
  const [description, setDescription] = useState('');
  const [loading, setLoading] = useState(true);
  const [isSubmitting, setIsSubmitting] = useState(false);
  const [error, setError] = useState('');

  // ЗАГРУЖАЕМ ДАННЫЕ ИДЕИ ПРИ ЗАГРУЗКЕ СТРАНИЦЫ
  useEffect(() => {
    const loadIdea = async () => {
      try {
        setLoading(true);
        // GET-запрос для получения одной идеи
        const response = await fetch(`http://localhost:3001/ideas/${id}`);
```

```
        if (!response.ok) {
          throw new Error('Идея не найдена');
        }

        const data = await response.json();
        setNick(data.nick);
        setName(data.name);
        setDescription(data.description);
      } catch (err) {
        setError(err instanceof Error ? err.message : 'Ошибка загрузки');
      } finally {
        setLoading(false);
      }
    };

    if (id) {
      loadIdea();
    }
  }, [id]);

  // ОТПРАВЛЯЕМ ОБНОВЛЕННЫЕ ДАННЫЕ НА СЕРВЕР
  const handleSubmit = async (e: React.FormEvent) => {
    e.preventDefault();
    setIsSubmitting(true);
    setError('');

    try {
      const response = await fetch(`http://localhost:3001/ideas/${id}`, {
        method: 'PUT',           // PUT - полное обновление
        headers: {
          'Content-Type': 'application/json',
        },
        body: JSON.stringify({
          nick,
          name,
          description,
        }),
      });

      if (!response.ok) {
        const errorData = await response.json();
        throw new Error(errorData.message || 'Ошибка при обновлении');
      }

      navigate('/gallery');
    } catch (err) {
      setError(err instanceof Error ? err.message : 'Произошла ошибка');
    } finally {
      setIsSubmitting(false);
    }
  };

  if (loading) return <div style={{ textAlign: 'center', padding: '50px' }}>Загрузка данных ид
еи...</div>;
  if (error) return <div style={{ color: 'red', textAlign: 'center', padding: '50px' }}>{error
}</div>;

  return (
    <div style={{ maxWidth: '600px', margin: '40px auto', padding: '20px', border: '1px solid
#ccc', borderRadius: '8px' }}>
      <h1>Редактирование идеи #{id}</h1>

      <form onSubmit={handleSubmit}>
```

```
        <div style={{ marginBottom: '16px' }}>
          <label style={{ display: 'block', marginBottom: '8px', fontWeight: 'bold' }}>Никнейм
:</label>
          <input
            type="text"
            value={nick}
            onChange={(e) => setNick(e.target.value)}
            required
            disabled={isSubmitting}
            style={{ width: '100%', padding: '10px', border: '1px solid #ccc', borderRadius: '
4px' }}
          />
        </div>

        <div style={{ marginBottom: '16px' }}>
          <label style={{ display: 'block', marginBottom: '8px', fontWeight: 'bold' }}>Названи
e:</label>
          <input
            type="text"
            value={name}
            onChange={(e) => setName(e.target.value)}
            required
            disabled={isSubmitting}
            style={{ width: '100%', padding: '10px', border: '1px solid #ccc', borderRadius: '
4px' }}
          />
        </div>

        <div style={{ marginBottom: '16px' }}>
          <label style={{ display: 'block', marginBottom: '8px', fontWeight: 'bold' }}>Описани
e:</label>
          <textarea
            value={description}
            onChange={(e) => setDescription(e.target.value)}
            required
            rows={4}
            disabled={isSubmitting}
            style={{ width: '100%', padding: '10px', border: '1px solid #ccc', borderRadius: '
4px' }}
          />
        </div>

        <div style={{ display: 'flex', gap: '10px' }}>
          <button
            type="submit"
            disabled={isSubmitting}
            style={{
              backgroundColor: '#ffc107',
              color: 'black',
              border: 'none',
              padding: '10px 20px',
              borderRadius: '4px',
              cursor: isSubmitting ? 'not-allowed' : 'pointer'
            }}
          >
            {isSubmitting ? 'Сохранение...' : 'Сохранить изменения'}
          </button>

          <button
            type="button"
            onClick={() => navigate('/gallery')}
            style={{
              backgroundColor: '#6c757d',
              color: 'white',
```

```
              border: 'none',
              padding: '10px 20px',
              borderRadius: '4px',
              cursor: 'pointer'
            }}
          >
            Отмена
          </button>
        </div>
      </form>
    </div>
  );
}

export default EditIdeaPage;
```

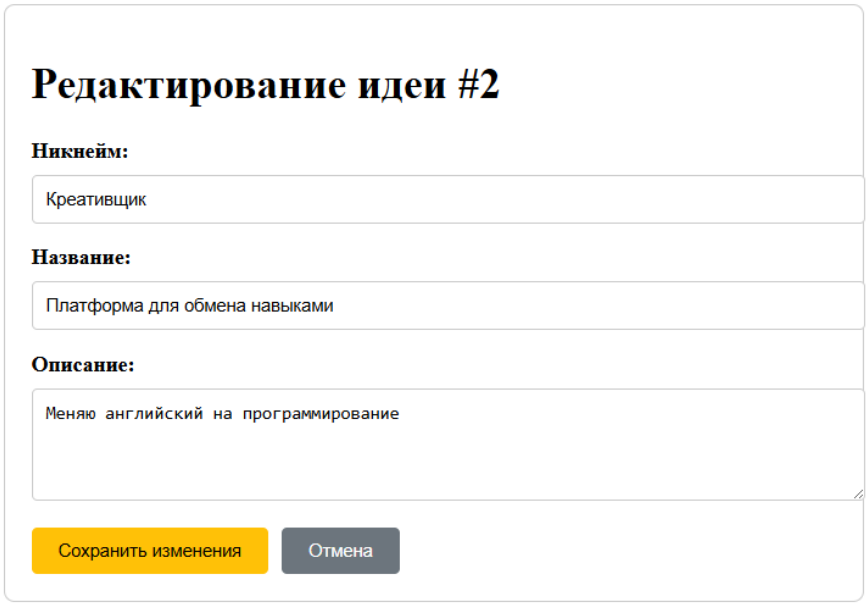
Шаг 9.2: Проверка GET и PUT запросов

Действия:

1. Перейдите в галерею `http://localhost:5173/gallery`
2. Нажмите "Редактировать" на любой идее

Ожидаемый результат:

- Страница загружается и показывает форму, заполненную данными идеи



3. Измените данные и нажмите "Сохранить изменения"

Ожидаемый результат:

- Перенаправление в галерею

- Обновленные данные отображаются в карточке

4. **ПРОВЕРКА через curl:**

```
powershell
# Получить идею с id=1
curl http://localhost:3001/ideas/1
```

РЕЗУЛЬТАТ:

Все CRUD операции работают:

- **GET /ideas** → список идей
- **GET /ideas/:id** → одна идея для редактирования
- **POST /ideas** → создание новой идеи
- **PUT /ideas/:id** → обновление идеи
- **DELETE /ideas/:id** → удаление идеи

Итоговая структура проекта

Фронтенд (`blog-ideas/`)

```
text
blog-ideas/
├── src/
│   ├── components/
│   │   └── Header.tsx
│   ├── pages/
│   │   ├── HomePage.tsx
│   │   ├── GalleryPage.tsx
│   │   ├── AddIdeaPage.tsx
│   │   └── EditIdeaPage.tsx
│   ├── App.tsx
│   ├── main.tsx
│   └── types.ts
├── .gitignore
├── .prettierrc
├── index.html
├── package.json
└── tsconfig.json
```

Бэкенд (`blog-ideas-backend/`)

```
text
blog-ideas-backend/
├── src/
│   └── index.ts
├── .gitignore
├── package.json
└── tsconfig.json
```

Запуск приложения

Терминал 1 (Бэкенд):

```
powershell
cd blog-ideas-backend
pnpm run dev
```

Терминал 2 (Фронтенд):

```
powershell
cd blog-ideas
pnpm run dev
```

Открыть в браузере: `http://localhost:5173`

Чему Вы научились:

- ✅ Настраивать окружение для веб-разработки на Windows
- ✅ Создавать React-приложение с TypeScript и Vite
- ✅ Настраивать маршрутизацию между страницами
- ✅ Работать с формами и состоянием в React
- ✅ Использовать Git и публиковать код на GitHub
- ✅ Настраивать бэкенд на Express + TypeScript
- ✅ Создавать REST API с эндпоинтами GET, POST, PUT, DELETE
- ✅ Связывать фронтенд и бэкден через fetch-запросы
- ✅ Выполнять все CRUD операции с данными