

Приложение на React + Node.js

Функциональные требования

Добавление поста:

- Создание формы для ввода данных нового поста (например, заголовков и содержание).
- Валидация введенных данных на стороне клиента.
- Отправка данных нового поста на сервер для сохранения.
- Получение постов:
- Запрос к серверу для получения списка всех постов.
- Отображение полученных постов на странице.

Требования к frontend (React.js)

Установка зависимостей:

- React.js
- Axios (для отправки HTTP запросов на сервер)
- Создание компонентов:
- Форма для добавления поста
- Компонент для отображения списка постов

Логика работы:

- Реализация функционала добавления поста с использованием Axios для отправки POST запроса на сервер.
- Получение списка постов с помощью Axios при загрузке страницы.

Требования к backend (Node.js с PostgreSQL и Sequelize)

Установка зависимостей:

- Node.js
- Express.js (фреймворк для создания сервера)
- Sequelize (ORM для работы с базой данных PostgreSQL)
- PostgreSQL (реляционная база данных)

Создание модели для постов:

- Определение структуры таблицы в базе данных для хранения постов (например, заголовок, содержание, дата создания).

Создание API:

- Реализация эндпоинтов для добавления и получения постов.
- Обработка POST запроса для добавления нового поста.
- Обработка GET запроса для получения списка всех постов.

Подключение к базе данных:

- Настройка подключения к PostgreSQL с помощью Sequelize.

Реализация Backend

Создадим бэкенд на Node.js с использованием PostgreSQL и Sequelize.

Шаг 1. Создание нового проекта и установка зависимостей.

Для этого открываем VScode, или любой другой текстовый редактор и создаем новый проект как показано на рисунках 1.1, 1.2, 1.3, 1.4, 1.5.

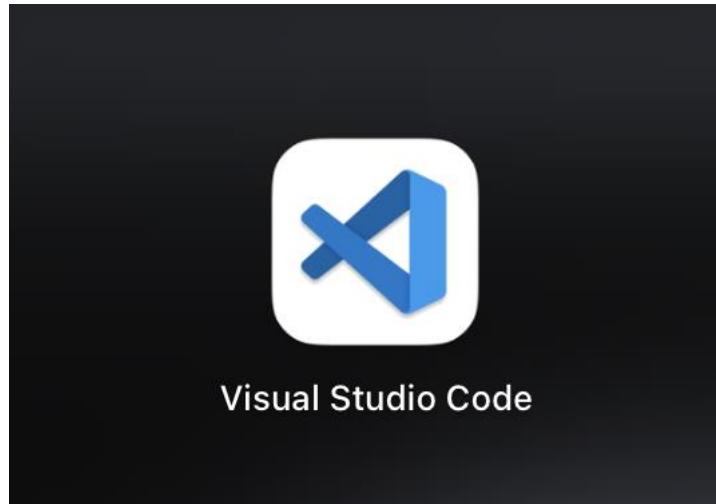


Рисунок 1.1 – Открытие VScode

На рабочем столе создайте папку с осмысленным названием вашего проекта. Откройте созданную папку в VScode.

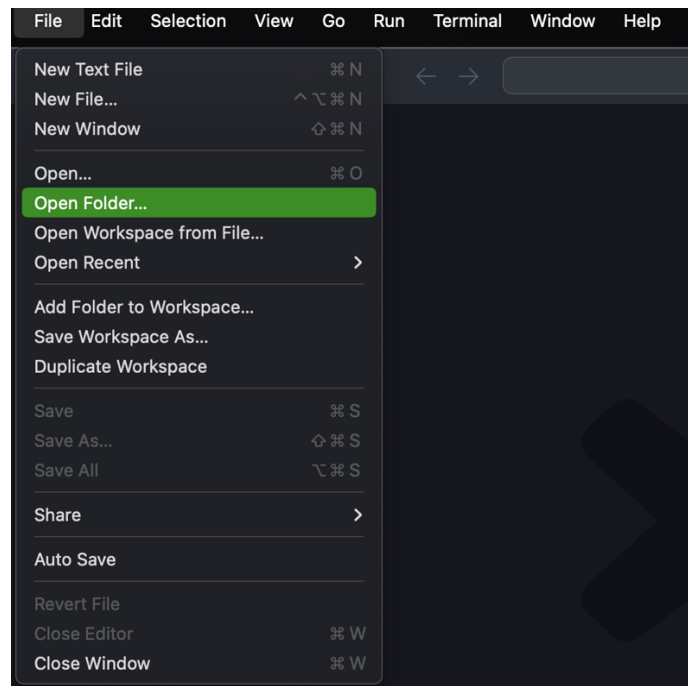


Рисунок 1.2 – Открытие созданной папки

В открывшемся меню вам нужно создать две папки, которые будут называться **backend** и **frontend**.

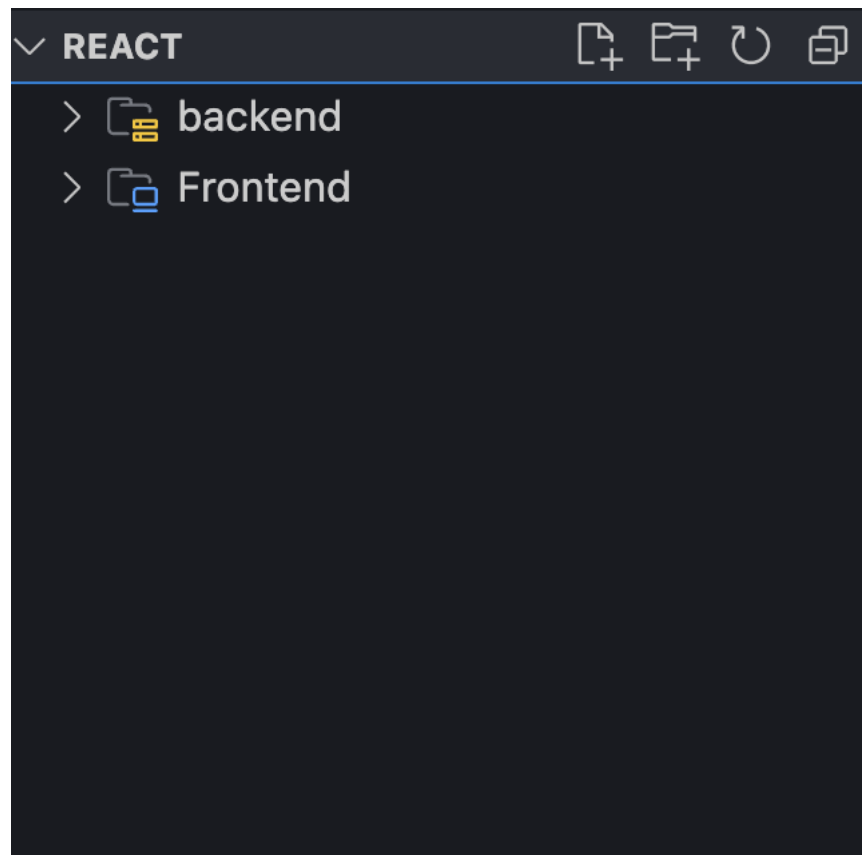


Рисунок 1.3 – создание структуры проекта

Установим нужные нам зависимости для реализации серверной части приложения. Открываем новый терминал в верхней части редактора.

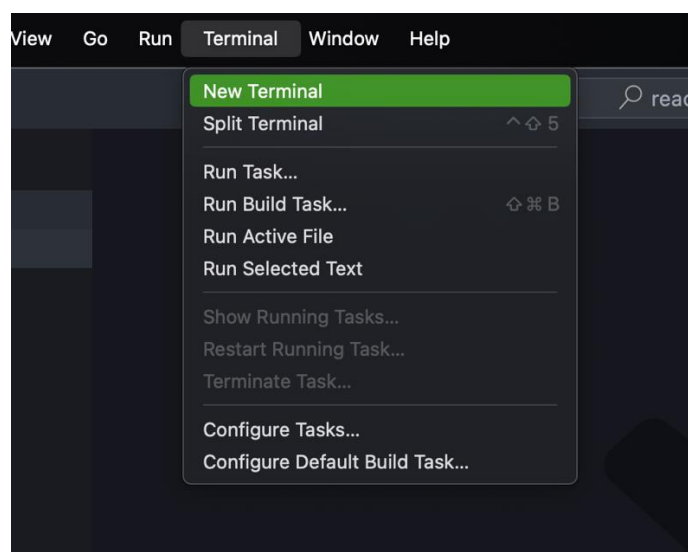
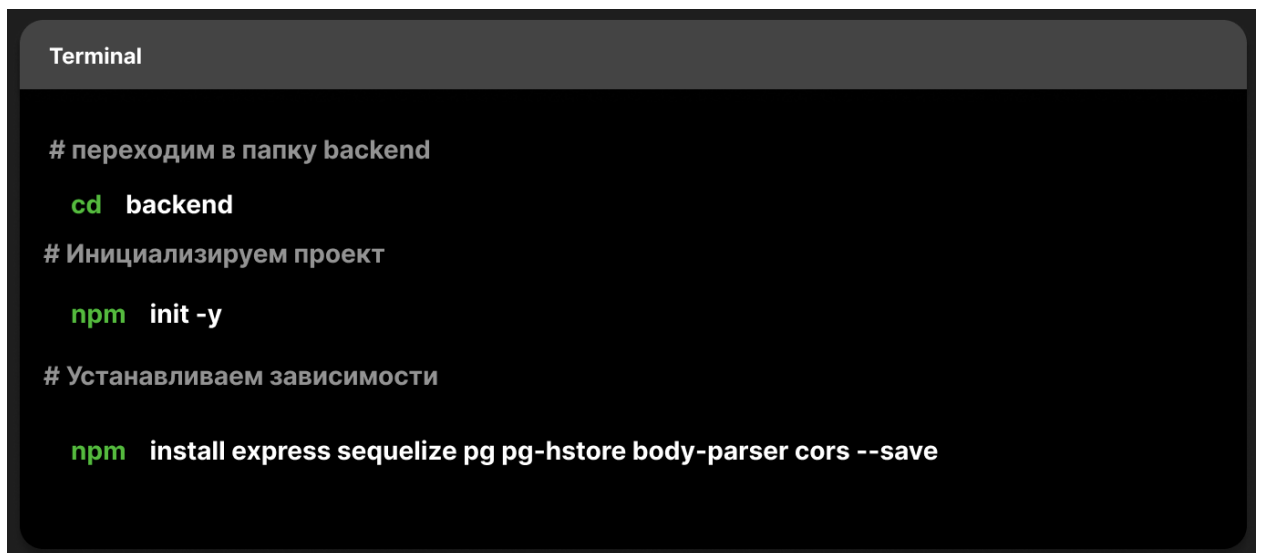


Рисунок 1.4 – Создание нового терминала

Для того чтобы проинициализировать проект и установить зависимости именно в папке backend, нужно прописать определенные команды в терминале.



```
Terminal

# переходим в папку backend
cd backend
# Инициализируем проект
npm init -y
# Устанавливаем зависимости
npm install express sequelize pg pg-hstore body-parser cors --save
```

Рисунок 1.5 – установка зависимостей

Шаг 2. Подключение к базе данных, настройка проекта.

Реализуем следующую структуру проекта. Создайте файлы и папки, указанные на скриншоте ниже, исключение (node_modules, package.json, package-lock.json) они создаются в результате команд, установленных выше.

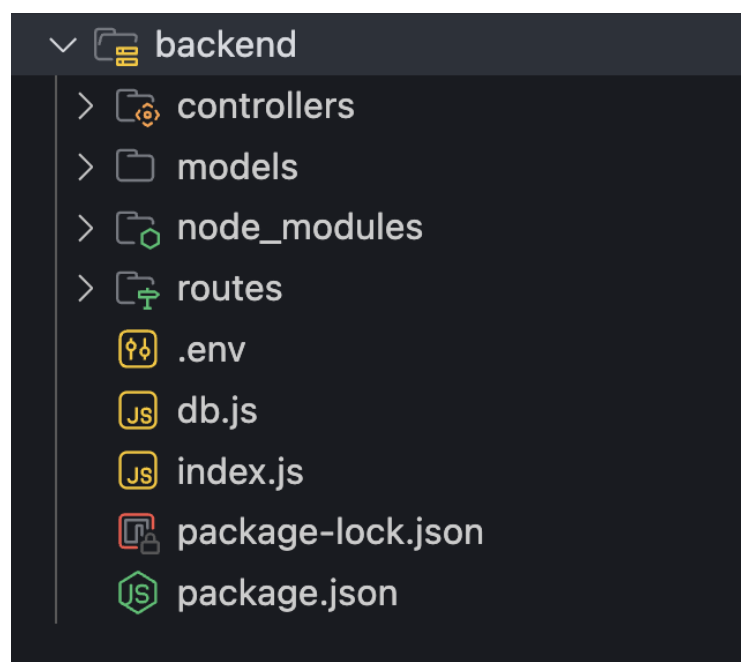
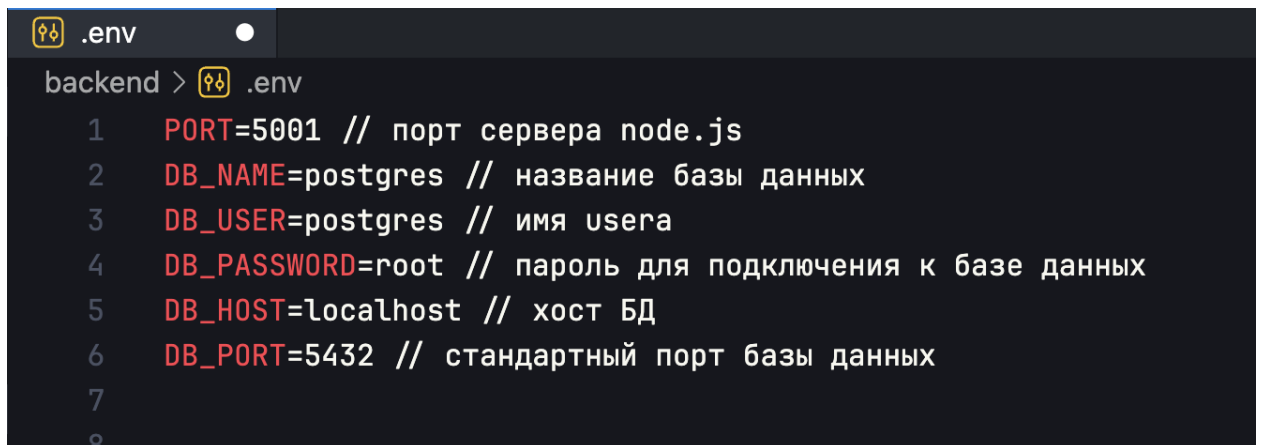


Рисунок 2.1 – Настройка структуры проекта

Настроим конфигурационный файл `.env` туда заносим данные для подключения базы данных.



```
.env
backend > .env
1  PORT=5001 // порт сервера node.js
2  DB_NAME=postgres // название базы данных
3  DB_USER=postgres // имя usera
4  DB_PASSWORD=root // пароль для подключения к базе данных
5  DB_HOST=localhost // хост БД
6  DB_PORT=5432 // стандартный порт базы данных
7
8
```

Рисунок 2.3 – Настройка файла `.env`

Для того чтобы `.env` работал и мы могли к нему обращаться и получать данные из переменных, установим данный пакет через терминал.




```
Terminal

# Установка пакета dotenv

npm install dotenv
```

Рисунок 2.4 – Установка пакета `dotenv`

Настроим файл `db.js` в который передадим наши переменные для подключения к базе данных.



```
db.js

const {Sequelize} = require('sequelize')

module.exports = new Sequelize(
  process.env.DB_NAME,
  process.env.DB_USER,
  process.env.DB_PASSWORD,
  {
    dialect: 'postgres',
    host: process.env.DB_HOST,
    port: process.env.DB_PORT
  }
)
```

Рисунок 2.5 – Подключение к базе данных PostgreSQL

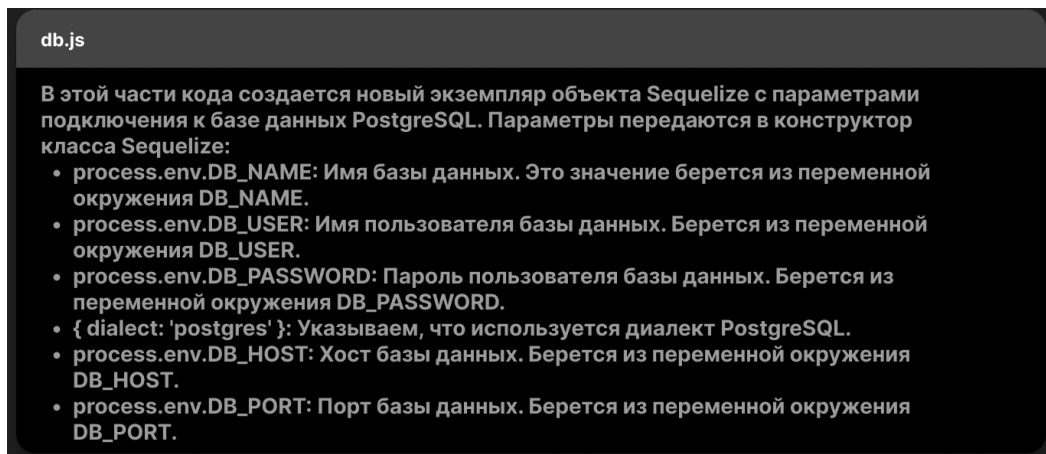


Рисунок 2.6 – Описание db.js

Шаг 3. Создание модели

Развернем папку models и создадим в ней файл post.js как на скриншоте ниже.

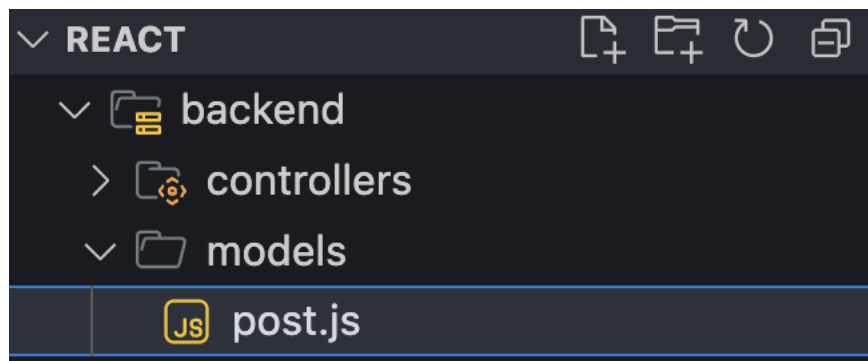


Рисунок 3.1 – создание post.js

Напишем код для создания нашей модели, указываем название модели (таблицы), дальше указываем нужные поля.

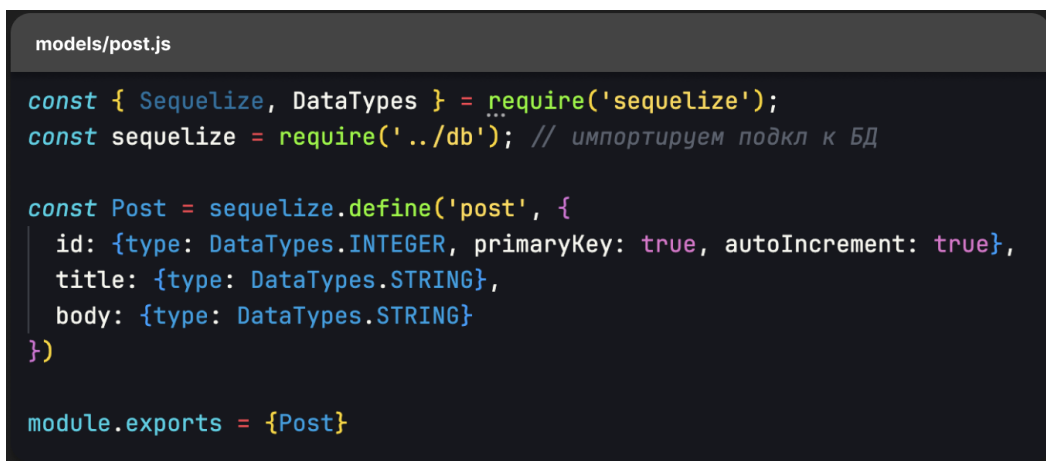


Рисунок 3.2 – Описание модели

Шаг 4. Разработка контроллера.

Создадим новый контроллер, `postController.js` и опишем в нем логику для добавления поста и получения поста.

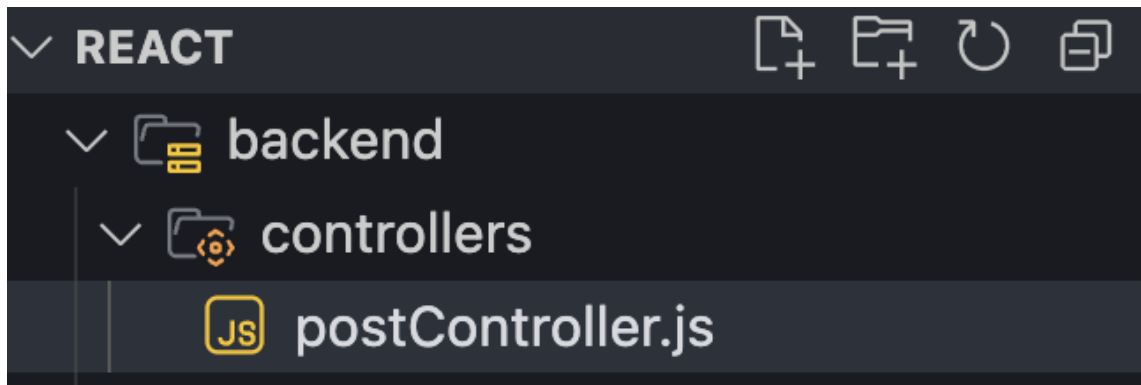


Рисунок 4.1 – новый контроллер

```
controllers/postController.js

const { Post } = require('../models/post');

// Контроллер для управления постами
class PostController {
  // Метод для добавления нового поста
  async addPost(req, res) {
    // Извлечение заголовка и содержания поста из запроса
    const { title, body } = req.body;
    // Создание нового поста в базе данных с помощью модели Post
    const post = await Post.create({ title, body });
    // Отправка созданного поста в формате JSON в ответ на запрос
    return res.json(post);
  }

  // Метод для получения всех постов
  async getPost(req, res) {
    // Получение всех постов из базы данных с помощью модели Post
    const post = await Post.findAll();
    // Отправка списка постов в формате JSON в ответ на запрос
    return res.json(post);
  }
}

// Экспорт экземпляра объекта контроллера
module.exports = new PostController();
```

Рисунок 4.2 – Описание контроллера

Шаг 5. Разработка роутера.

В паке routes создадим два файла index.js и postRouter.js в данных файлах напишем маршруты для нашего контроллера.

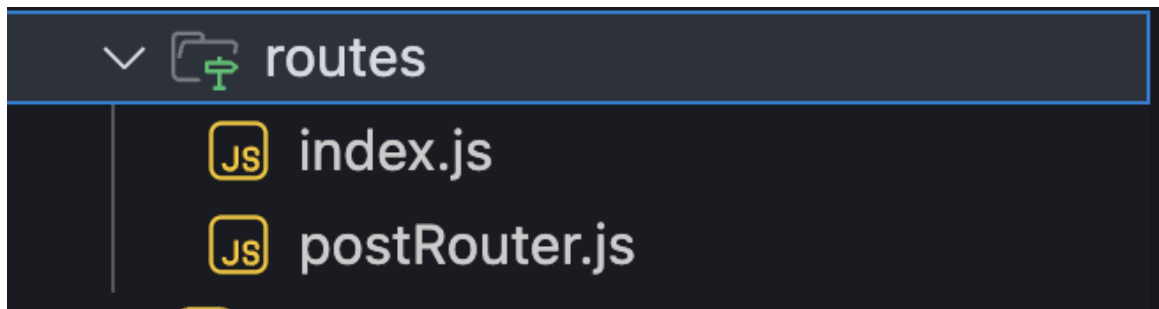


Рисунок 5.1 – файлы роутеров

Напишем логику для файла postRouter.

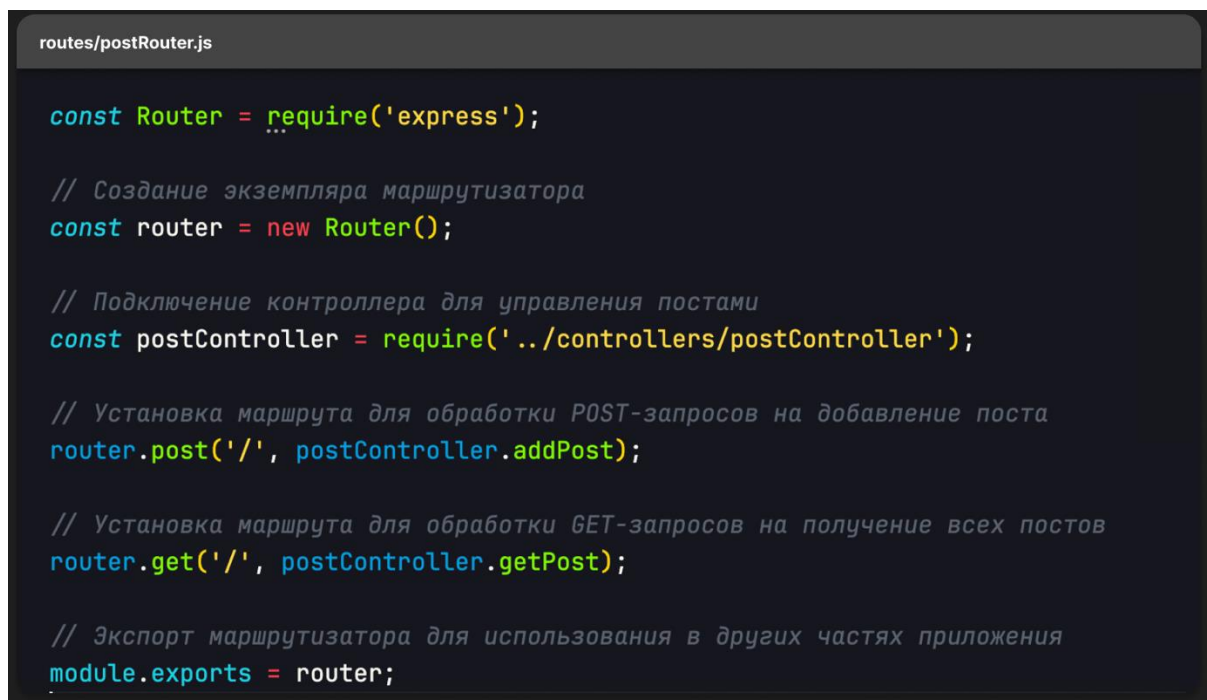


Рисунок 5.2 – логика роутера

Передадим наш роутер в главный файл index.js для дальнейшего использования в других местах кода.

```
routes/index.js

const Router = require('express');

// Создание экземпляра маршрутизатора
const router = new Router();

// Подключение маршрутов для работы с постами из отдельного маршрутизатора
const postRouter = require('./postRouter');

// Установка префикса URL '/posts' для всех маршрутов, определенных в postRouter
router.use('/posts', postRouter);

// Экспорт маршрутизатора для использования в других частях приложения
module.exports = router;
```

Рисунок 5.3 – логика index.js

Код использует Express для создания маршрутизатора, который будет обрабатывать запросы к различным URL-адресам. После создания маршрутизатора подключается отдельный маршрутизатор, который содержит маршруты для работы с постами (postRouter). Маршруты, определенные в postRouter, будут доступны по URL, начинающемуся с '/posts'. Например, если в postRouter определен маршрут для добавления поста по адресу '/add', то для доступа к этому маршруту из основного приложения будет использоваться URL '/posts/add'.

Шаг 6. Запуск сервера.

Напишем логику для файла index.js и запустим сервер.

```
index.js

// Подключение пакета dotenv для загрузки переменных среды из файла .env
require('dotenv').config();
// Подключение фреймворка Express для создания веб-приложений
const express = require('express');
// Подключение модуля, представляющего собой экземпляр Sequelize для взаимодействия с базой данных
const sequelize = require('./db');
// Подключение middleware для обработки данных, отправленных в теле HTTP-запроса
const bodyParser = require('body-parser');
// Подключение middleware для обработки запросов CORS (Cross-Origin Resource Sharing)
const cors = require('cors');
// Подключение модуля, содержащего определения маршрутов для обработки запросов
const router = require('./routes/index');
// Определение порта, на котором будет работать сервер,
// с использованием переменной среды PORT или порта по умолчанию 5002
const PORT = process.env.PORT || 5002;

const app = express();
// Использование middleware body-parser для обработки данных в форматах JSON и URL-кодированных данных.
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
// Использование middleware cors для разрешения запросов с других доменов
app.use(cors());
// Использование встроенного middleware express.json() для обработки данных в формате JSON.
app.use(express.json());
// Установка маршрута '/api' для обработки запросов с использованием определенного маршрутизатора router.
app.use('/api', router);

// Запуск сервера на определенном порту, проверка подключения к базе данных и синхронизация моделей с базой данных.
const start = async () => {
  try {
    await sequelize.authenticate();
    await sequelize.sync();
    app.listen(PORT, () => console.log(`SERVER STARTED ON PORT ${PORT}`));
  } catch (e) {
    console.log(e);
  }
};
start();
```

Рисунок 6.1 – логика главного index.js

Откроем терминал и установим пакет nodemon для удобного запуска сервера.

```
terminal/backend

npm i -D nodemon
```

Рисунок 6.2 – Установка nodemon

Откроем файл package.json и настроим запуск, в поле script напишем данный код.

```
package.json

"main": "index.js",
  Debug
"scripts": {
  "dev": "nodemon index.js"
},
```

Рисунок 6.3 – Настройка package.json

Вводим команду для запуска сервера.

```
terminal

npm run dev
```

Рисунок 6.4 – запуск сервера

```
terminal

> npm run dev
> backend@1.0.0 dev
> nodemon index.js

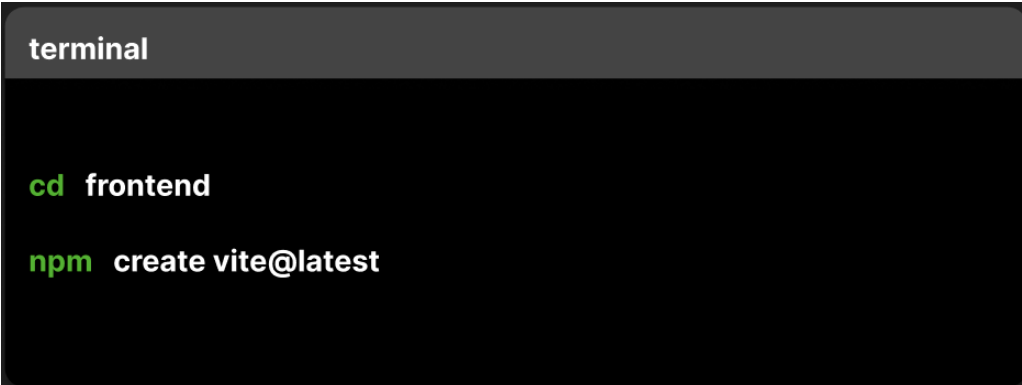
[nodemon] 3.0.3
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node index.js`
Executing (default): SELECT 1+1 AS result
Executing (default): SELECT table_name FROM information_schema.tables WHERE table_schema = 'public' AND table_name = 'posts'
Executing (default): CREATE TABLE IF NOT EXISTS "posts" ("id" SERIAL , "title" VARCHAR(255), "body" VARCHAR(255), "createdAt" TIMESTAMP WITH TIME ZONE NOT NULL, "updatedAt" TIMESTAMP WITH TIME ZONE NOT NULL, PRIMARY KEY ("id"));
Executing (default): SELECT i.relname AS name, ix.indisprimary AS primary, ix.indisunique AS unique, ix.indkey AS indkey, array_agg(a.attname) AS column_indexes, array_agg(a.attname) AS column_names, pg_get_indexdef(ix.indexrelid) AS definition FROM pg_class t, pg_class i, pg_index ix, pg_attribute a WHERE t.oid = ix.indrelid AND i.oid = ix.indexrelid AND a.attrelid = t.oid AND t.relkind = 'r' and t.relname = 'posts' GROUP BY i.relname, ix.indexrelid, ix.indisprimary, ix.indisunique, ix.indkey ORDER BY i.relname;
SERVER STARTED ON PORT 5001
```

Рисунок 6.5 Итог запуска

Реализация Frontend

Шаг 1. Установка React.

Создаем новый терминал и переходим в папку frontend, устанавливаем react с помощью команды ниже.

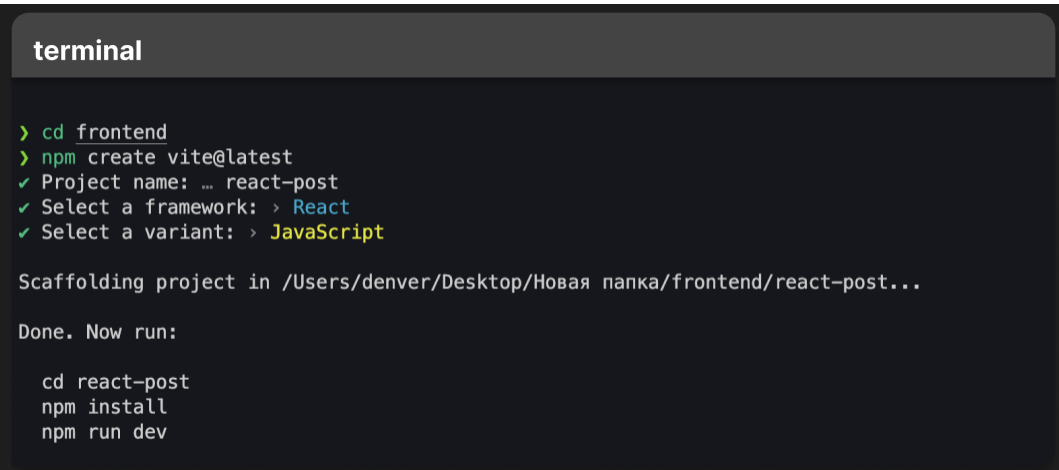


```
terminal

cd frontend

npm create vite@latest
```

Рисунок 1.1 – Установка React



```
terminal

> cd frontend
> npm create vite@latest
✓ Project name: ... react-post
✓ Select a framework: > React
✓ Select a variant: > JavaScript

Scaffolding project in /Users/denver/Desktop/Новая папка/frontend/react-post...

Done. Now run:

  cd react-post
  npm install
  npm run dev
```

Рисунок 1.2 – Результат установки

Шаг 2. Настройка проекта

Раскроем структуру проекта и оставим только два основных файла для работы. Main.jsx и App.jsx.

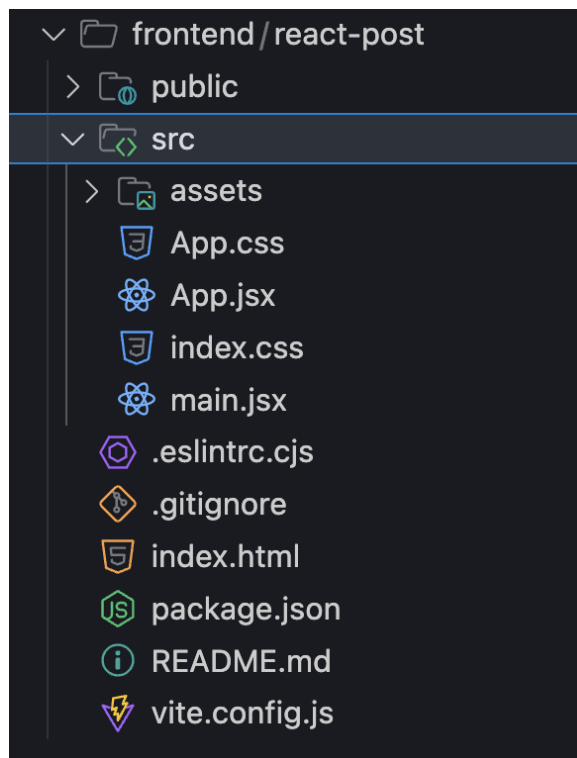


Рисунок 2.1 – Структура проекта

После удаления не нужных файлов, код в файле App.jsx и Main.jsx должен выглядеть как на картинке 2.2, 2.3.

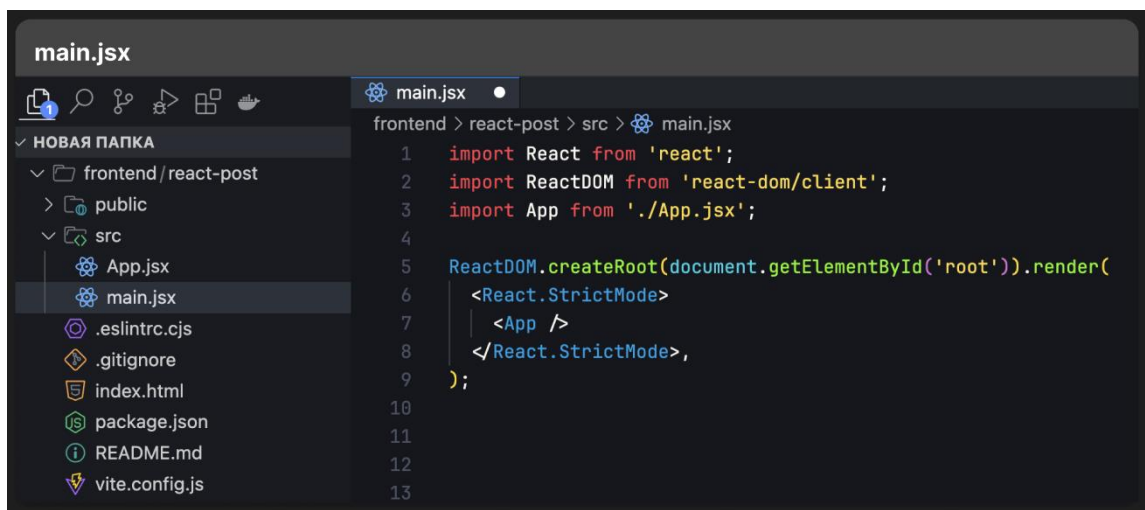


Рисунок 2.2 – Файл main.jsx

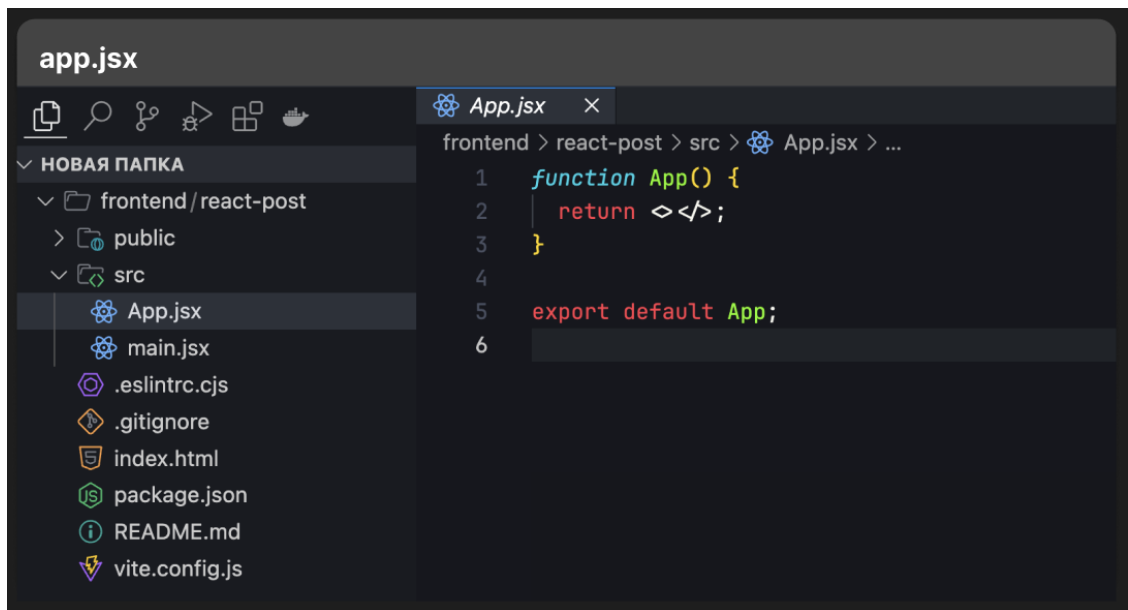


Рисунок 2.3 – Файл App.jsx

Шаг 3. Создание компонентов и установка Axios.

Установим Axios пакет через терминал с помощью команды.

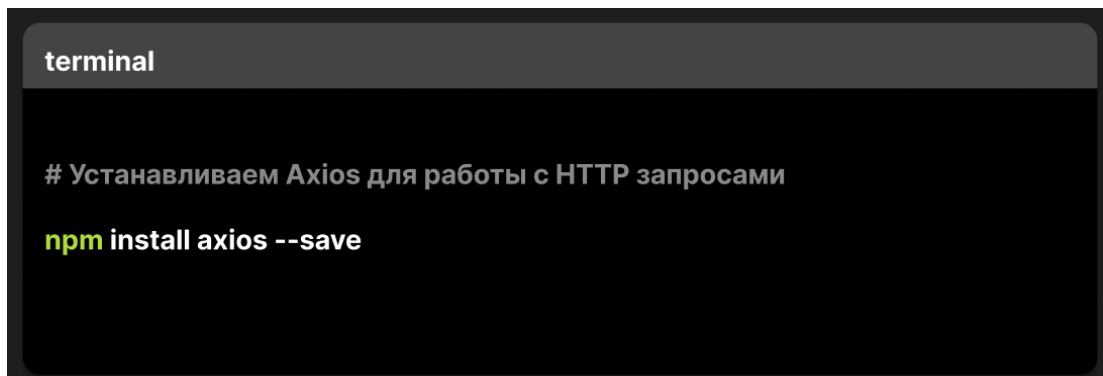


Рисунок 3.1 – Установка Axios

Создадим два компонента React: один для формы добавления поста, а второй для отображения списка постов.

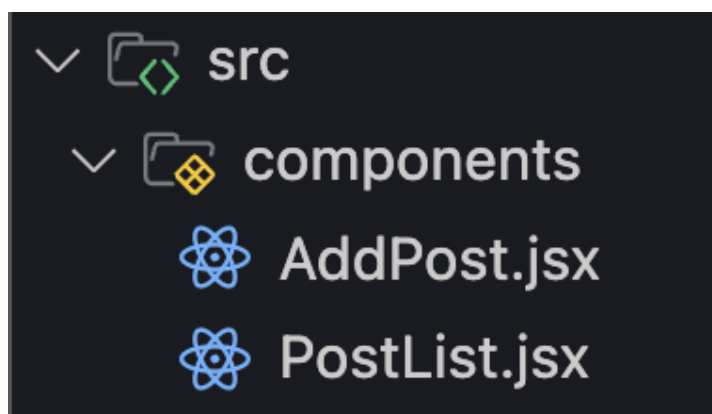


Рисунок 3.2 – Создание компонентов

Компонент AddPost.js

```
src/components/AddPost.jsx

// eslint-disable-next-line no-unused-vars
import React, { useState } from 'react';
import axios from 'axios';

const AddPost = () => {
  // Определение состояния для заголовка и тела поста с помощью хуков useState
  const [title, setTitle] = useState('');
  const [body, setBody] = useState('');

  // Функция для отправки POST-запроса на сервер с данными нового поста
  const addPost = async () => {
    try {
      // Отправка POST-запроса с использованием библиотеки Axios
      await axios.post('http://localhost:5001/api/posts', { title, body });
      // После успешной отправки запроса очищаем поля ввода заголовка и тела поста
      setTitle('');
      setBody('');
    } catch (error) {
      // В случае возникновения ошибки выводим ее в консоль
      console.error(error);
    }
  };

  // Возвращаем JSX - форму для добавления нового поста
  return (
    <div className='container'>
      <h2>Add Post</h2>
      <form onSubmit={addPost}>
        { /* Поле ввода для заголовка поста */ }
        <input
          type="text"
          value={title}
          onChange={(e) => setTitle(e.target.value)}
          placeholder="Title"
          required
        />
        { /* Поле ввода для тела поста */ }
        <textarea
          value={body}
          onChange={(e) => setBody(e.target.value)}
          placeholder="Body"
          required
        ></textarea>
        { /* Кнопка для отправки формы */ }
        <button type="submit">Add Post</button>
      </form>
    </div>
  );
};

// Экспорт компонента AddPost для его использования в других частях приложения
export default AddPost;
```

Рисунок 3.2 – Компонент AddPost.js

Компонент PostList.js

src/components/PostList.jsx

```
// eslint-disable-next-line no-unused-vars
import React, { useState, useEffect } from 'react';
import axios from 'axios';

const PostList = () => {
  // Используем хук useState для создания состояния, которое будет хранить список постов
  const [posts, setPosts] = useState([]);

  // Используем хук useEffect для выполнения побочных эффектов в функциональном компоненте
  useEffect(() => {
    // Определяем асинхронную функцию fetchPosts, которая будет получать данные о постах
    const fetchPosts = async () => {
      try {
        // Выполняем GET-запрос к серверу, который вернет список постов
        const response = await axios.get('http://localhost:5001/api/posts');
        // Обновляем состояние posts с помощью полученных данных
        setPosts(response.data);
      } catch (error) {
        // В случае возникновения ошибки выводим ее в консоль
        console.error(error);
      }
    };

    // Вызываем функцию fetchPosts при монтировании компонента
    fetchPosts();
    // Пустой массив зависимостей [] указывает, что эффект будет запущен только один раз после монтирования компонента
  }, []);

  // Возвращаем JSX - компонент списка постов
  return (
    <div>
      <h2 className='text'>Posts</h2>
      <ul>
        {/* Маппим каждый пост в массиве posts на элемент списка */}
        {posts.map((post) => (
          <li key={post.id}>
            {/* Отображаем заголовок и содержимое каждого поста */}
            <h3>{post.title}</h3>
            <p>{post.body}</p>
          </li>
        ))}
      </ul>
    </div>
  );
};

// Экспортируем компонент PostList, чтобы он мог быть использован в других частях приложения
export default PostList;
```

Рисунок 3.3 – Компонент PostList.js

Шаг 4: Интеграция компонентов в главный файл React

Отредактируем файл App.js, чтобы интегрировать созданные компоненты:

```
src/App.jsx

import AddPost from './components/AddPost';
import PostList from './components/PostList';
import './style/index.css';

const App = () => {
  return (
    <div>
      <AddPost />
      <PostList />
    </div>
  );
};

export default App;
```

Рисунок 4.1 – Интеграция в главный файл

Запускаем сервер backend и одновременно сервер frontend, пробуем добавить первый пост.

Задание

- Добавить стили к проекту
- Добавить удаление поста
- Добавить изменение поста