Работа с fetch в React с использованием TypeScript предполагает добавление типизации для запросов и ответов. Это позволяет легче обрабатывать данные и предотвратить ошибки в раннем этапе разработки. Ниже рассмотрим пошагово, как получать и отправлять запросы с fetch в React + TypeScript, с примерами.

**I. Получение данных (GET запрос)**

Чаще всего выполняется GET-запрос для получения данных. Для этого важно:

Заранее определить структуру данных, которые вы ожидаете получить.

Типизировать результат запроса, чтобы получать подсказки от TypeScript при работе с ответом.

Пример: Получение списка пользователей

Определяем структуру данных

Вначале создайте интерфейс или тип для описания структуры данных, которые возвращает сервер. Например, API возвращает список пользователей:

interface User {

    id: number;

    name: string;

    email: string;

}

**Функция для запроса**

Напишем обёртку над fetch, которая типизирует запрос и данные ответа:

async function fetchUsers(): Promise<User[]> {

    const response = await fetch('https://api.example.com/users');

    if (!response.ok) {

        throw new Error('Ошибка при загрузке пользователей');

    }

    return response.json(); // TS будет знать, что возвращается User[]

}

**Использование в компоненте**

Создадим React-компонент и добавим логику запроса:

import React, { useEffect, useState } from 'react';

interface User {

    id: number;

    name: string;

    email: string;

}

const UserList: React.FC = () => {

    const [users, setUsers] = useState<User[]>([]);

    const [error, setError] = useState<string | null>(null);

    const [loading, setLoading] = useState<boolean>(true);

    useEffect(() => {

        fetch('https://api.example.com/users')

            .then(res => {

                if (!res.ok) throw new Error('Ошибка загрузки данных');

                return res.json(); // возвращает данные, что типизируются как `any`

            })

            .then((data: User[]) => {

                setUsers(data); // TS понимает структуру данных

                setLoading(false);

            })

            .catch((err) => {

                setError(err.message);

                setLoading(false);

            });

    }, []);

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

    if (error) return <p>{error}</p>;

    return (

        <ul>

            {users.map(user => (

                <li key={user.id}>

                    {user.name} - {user.email}

                </li>

            ))}

        </ul>

    );

};

export default UserList;

**Разъяснения:**

useState задает тип для состояния, которое хранит список пользователей.

Метод then типизируется: data: User[], если структура ответа заранее определена.

Ошибки обрабатываются через catch и сохраняются в состоянии error.

**II. Отправка данных (POST запрос)**

При отправке данных через POST-запрос:

Типизируем отправляемое тело (body).

Определяем тип ожидаемого ответа от сервера.

Пример: Создание нового пользователя

Определяем типы

Задаем тип для данных, которые отправляем, и тип ответа:

interface CreateUserRequest {

    name: string;

    email: string;

}

interface CreateUserResponse {

    id: number;

    name: string;

    email: string;

}

**Функция для отправки запроса**

Создадим функцию для POST-запроса:

async function createUser(user: CreateUserRequest): Promise< CreateUserResponse> {

    const response = await fetch('https://api.example.com/users', {

        method: 'POST',

        headers: {

            'Content-Type': 'application/json',

        },

        body: JSON.stringify(user),

    });

    if (!response.ok) {

        throw new Error('Ошибка при создании пользователя');

    }

    return response.json(); // TS знает, что возвращается CreateUserResponse

}

**Использование в компоненте**

Теперь интегрируем POST-запрос в React:

import React, { useState } from 'react';

interface CreateUserRequest {

    name: string;

    email: string;

}

interface CreateUserResponse {

    id: number;

    name: string;

    email: string;

}

const CreateUserForm: React.FC = () => {

    const [name, setName] = useState<string>('');

    const [email, setEmail] = useState<string>('');

    const [status, setStatus] = useState<string | null>(null);

    const handleSubmit = async (event: React.FormEvent) => {

        event.preventDefault();

        const newUser: CreateUserRequest = { name, email };

        try {

            const createdUser = await createUser(newUser);

            setStatus(`Пользователь ${createdUser.name} создан!`);

        } catch (error) {

            if (error instanceof Error) setStatus(error.message);

            else setStatus('Неизвестная ошибка');

        }

    };

    return (

        <form onSubmit={handleSubmit}>

            <div>

                <label>Имя:</label>

                <input

                    type="text"

                    value={name}

                    onChange={(e) => setName(e.target.value)}

                />

            </div>

            <div>

                <label>Email:</label>

                <input

                    type="email"

                    value={email}

                    onChange={(e) => setEmail(e.target.value)}

                />

            </div>

            <button type="submit">Создать</button>

            {status && <p>{status}</p>}

        </form>

    );

};

export default CreateUserForm;

Разъяснение:

Создаем объект CreateUserRequest и передаем его в createUser.

Состояние обновляется после успешного создания пользователя.

Обрабатываются ошибки запроса через try/catch.

**III. Совмещение с хуками React**

Для лучшего управления состоянием запросов можно использовать пользовательский хук.

**Пример: пользовательский хук useFetch**

import { useState, useEffect } from 'react';

function useFetch(url: string) {

    const [data, setData] = useState(null);

    const [loading, setLoading] = useState(true);

    const [error, setError] = useState(null);

    useEffect(() => {

        fetch(url)

            .then((res) => {

                if (!res.ok) throw new Error('Ошибка при загрузке');

                return res.json() as Promise;

            })

            .then(setData)

            .catch((err) => setError(err.message))

            .finally(() => setLoading(false));

    }, [url]);

    return { data, loading, error };

}

**Использование:**

const { data, loading, error } = useFetch('https://api.example.com/users');

**Заключение**

Для работы с fetch в React + TypeScript важно правильно типизировать:

отправляемые данные (например, через интерфейсы),

получаемые ответы (например, через Promise).

Вы можете выносить логику API-запросов в отдельные модули, использовать пользовательские хуки или сторонние библиотеки, такие как axios, для удобства. Типизация и грамотное использование состояния React помогают избежать ошибок и сделать код удобным для работы.