

React

сетевые запросы

Сетевые запросы

Fetch | AXIOS | ky js

Сетевые запросы в React

React-Router-DOM: loader

SWR (vercel)

React Query

Связь back-end-а и front-end-а (API)

REST API — универсальный стандарт, хорошо подходит для большинства веб-приложений, особенно если вам важна простота и легкость в использовании.

GraphQL — используйте, если необходимо гибко запрашивать данные и минимизировать количество запросов.

gRPC — подходит для высокопроизводительных микросервисов и распределённых систем, где важны производительность и минимизация трафика.

SOAP — применим в крупных корпоративных системах с особыми требованиями к безопасности и формализованным стандартам.

WebSockets — лучший выбор для приложений в реальном времени, таких как чаты или многопользовательские игры.

Библиотеки для реализации связи с back-end-ом

Axios — это одна из самых популярных библиотек для выполнения HTTP-запросов, которая легко интегрируется с React.

SWR — хук для React, который управляет фетчингом данных и кэшированием, делает запросы повторяемыми и обновляет данные в реальном времени.

Apollo Client — наиболее популярная библиотека для работы с GraphQL в React. Она предоставляет хук `useQuery` для выполнения запросов и работы с данными.

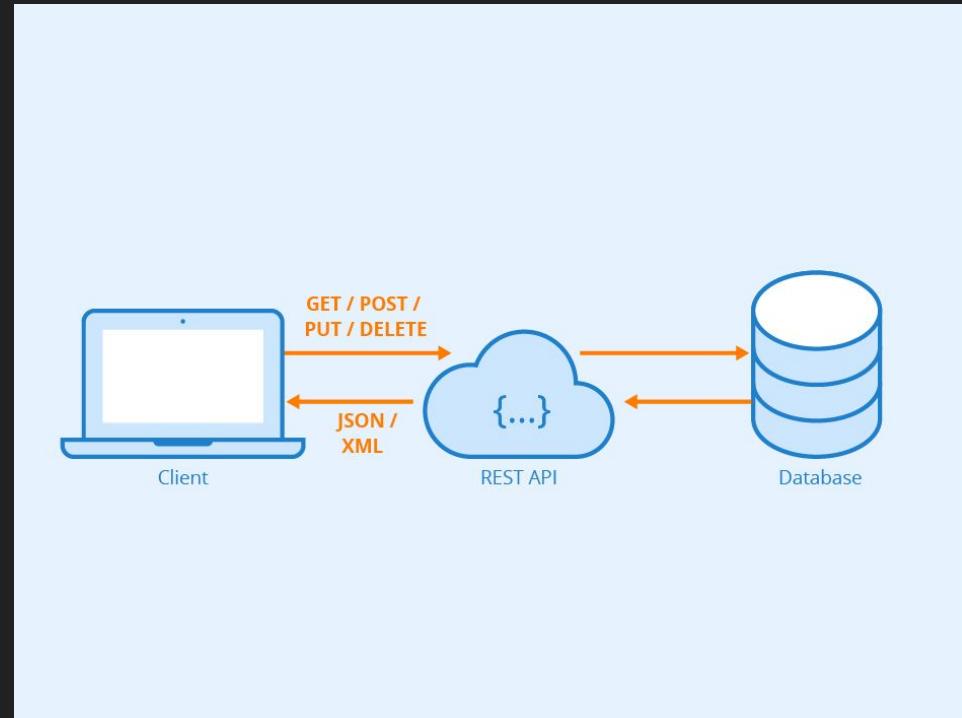
Socket.IO — библиотека для работы с WebSockets, которая легко интегрируется с React. Поддерживает real-time соединения, что идеально для чатов или игр.

strong-soap — хотя SOAP чаще используется на сервере, эту библиотеку можно применять и в React-приложениях для работы с SOAP API.

grpc-web — библиотека для работы с gRPC в браузере. Подходит для использования с React при работе с gRPC-сервисами через HTTP/1.1.

Сетевые запросы во Front-end-е (Rest API)

REST API (Representational State Transfer) — это архитектурный стиль для построения веб-сервисов. На фронтенде REST API используется для взаимодействия с сервером через HTTP-запросы. Обычно это означает отправку запросов на сервер и получение данных, которые используются для отображения на странице или обновления UI.



Rest API - CRUD

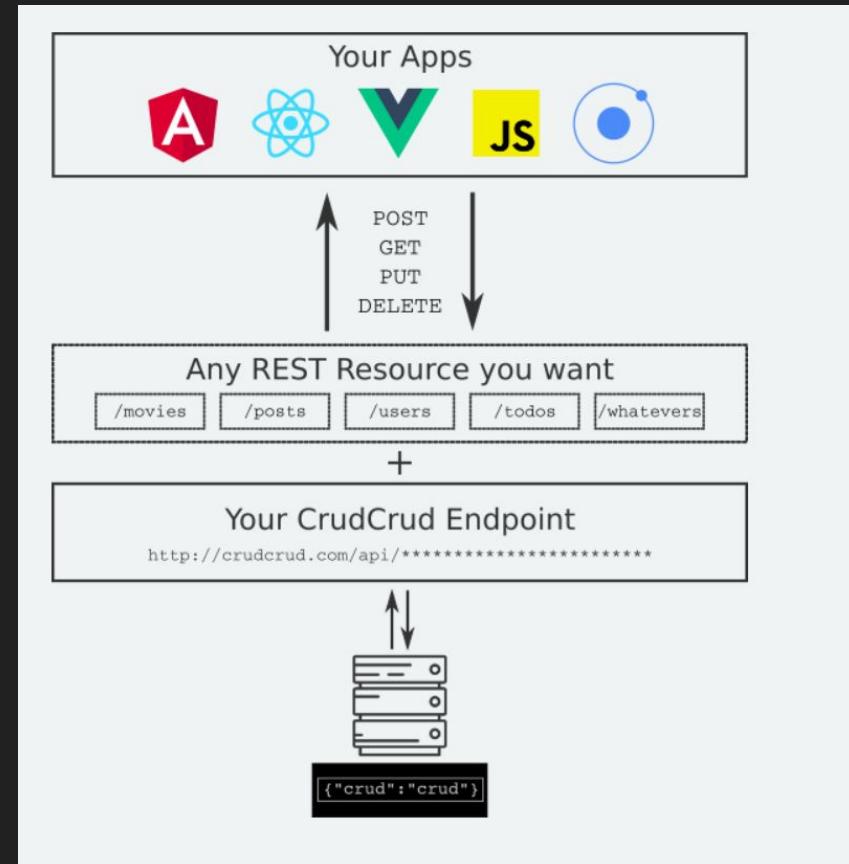
Основные типы HTTP-запросов:

(Create) POST: отправить данные на сервер (например, добавить новый пост или пользователя).

(Read) GET: получить данные с сервера (например, список пользователей, посты и т.д.).

(Update) PUT/PATCH: обновить данные на сервере (например, изменить информацию о пользователе).

(Delete) DELETE: удалить данные с сервера (например, удалить пост).



Сетевые запросы в JavaScript-е (REST API)

В JavaScript на фронтенде для выполнения сетевых запросов к REST API обычно используются встроенные функции и библиотеки.

- **AJAX** (Asynchronous JavaScript and XML) (встроенный функционал)
- **Fetch API** (встроенный функционал)
- **AXIOS** (библиотека)
- **ky js** (библиотека)

AJAX (XMLHttpRequest)

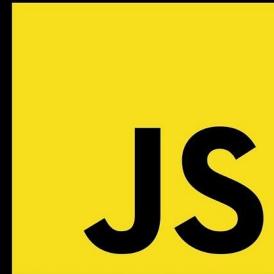
AJAX — это подход для выполнения асинхронных запросов к серверу и обновления части веб-страницы без необходимости перезагружать всю страницу. Основной инструмент AJAX — это объект XMLHttpRequest.

Основные характеристики:

Использует XMLHttpRequest: Объект для выполнения HTTP-запросов.

Асинхронность: Позволяет делать запросы в фоновом режиме.

Кросс-браузерность: Поддерживается во всех современных браузерах.



AJAX

Web API, предоставляемое браузером, и не является частью языка JavaScript. Он используется для выполнения асинхронных HTTP-запросов и работает через механизм AJAX.

Пример AJAX (XMLHttpRequest)

Старый способ выполнения запросов, требует работы с состоянием запроса и обработкой ошибок.

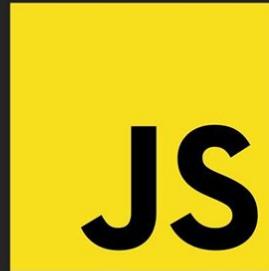
```
const xhr = new XMLHttpRequest();
xhr.open('GET', 'https://jsonplaceholder.typicode.com/users', true);
xhr.onload = function () {
  if (xhr.status >= 200 && xhr.status < 300) {
    const users = JSON.parse(xhr.responseText);
    console.log(users);
  } else {
    console.error('Error:', xhr.statusText);
  }
};
xhr.onerror = function () {
  console.error('Request failed');
};
xhr.send();
```

Fetch API

`fetch` — это современный API для выполнения HTTP-запросов, который поддерживает промисы и предоставляет удобный интерфейс для работы с запросами и ответами.

Основные характеристики:

- Промисы:** Возвращает промис, который упрощает работу с асинхронными запросами.
- Простота использования:** Интуитивно понятный синтаксис.
- Поддержка потоков:** Позволяет обрабатывать большие объемы данных.



Fetch API

Современный Web API предоставляемый браузером, и интегрированный в спецификацию JavaScript как часть современных стандартов для выполнения сетевых запросов.

Пример Fetch API

Современный способ работы с сетевыми запросами, возвращает промисы и поддерживает удобный синтаксис.

```
fetch('https://jsonplaceholder.typicode.com/users')
  .then(response => {
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    return response.json();
  })
  .then(users => console.log(users))
  .catch(error => console.error('Error:', error));
```

AXIOS

Axios — это популярная библиотека для выполнения HTTP-запросов, которая построена на основе промисов и предоставляет богатый функционал для работы с запросами и ответами.

Основные характеристики:

- **Удобство:** Простота в использовании, поддержка запросов и ответов в JSON.
- **Промисы:** Поддержка асинхронного кода.
- **Расширенные возможности:** Поддержка отмены запросов, перехватчиков запросов и ответов.

A X I O S

Установка через npm:
npm install axios

Пример AXIOS

Популярная библиотека с богатым функционалом для выполнения запросов, также возвращает промисы и предоставляет удобные методы для работы с запросами и ответами.

```
// Убедитесь, что Axios установлен, например, через npm: npm install axios
import axios from 'axios';

axios.get('https://jsonplaceholder.typicode.com/users')
  .then(response => console.log(response.data))
  .catch(error => console.error('Error:', error));
```

ky

Ку — это легковесная библиотека для выполнения HTTP-запросов, построенная на основе `fetch`. Она предлагает упрощенный API и удобные функции для работы с запросами.

Основные характеристики:

- **Минимализм:** Легковесная и простая в использовании.
- **Поддержка промисов:** Работает с промисами, как `fetch`.
- **Конфигурация:** Простое управление заголовками и параметрами запросов.



極楽要求(しなさい)

Delightful HTTP Requests

Установка через npm:
npm install ky

Пример ky

Легковесная библиотека для выполнения HTTP-запросов, построенная на fetch, с упрощенным API.

```
// Убедитесь, что Ку установлен, например, через npm: npm install ky
import ky from 'ky';

ky.get('https://jsonplaceholder.typicode.com/users')
  .json()
  .then(users => console.log(users))
  .catch(error => console.error('Error:', error));
```

Сетевые запросы в React (REST API)

Сетевые запросы в React, особенно при взаимодействии с REST API, — это процесс получения данных с сервера и их отображения в компоненте. REST API (Representational State Transfer) — это архитектурный стиль для построения веб-сервисов, который использует HTTP запросы для доступа и манипуляции ресурсами, представленными в виде JSON или XML.

В React для выполнения сетевых запросов обычно используются следующие подходы:

- базовый синтаксис React
- Пользовательский хук (hook)
- Пользовательский компонент высшего порядка (HOC)
- Используя loader в React Router DOM
- подключая к проекту SWR (vercel)
- React Query - библиотека работы с сетевыми запросами

Пример

Приложение отображения списка пользователей

На каждый метод реализации связи запросов и UI через react реализована отдельная страница, также сам HTTP клиент на стороне фронта реализован через fetch API, и библиотеки: axios, ky js. Можно быстро сменить HTTP клиент.

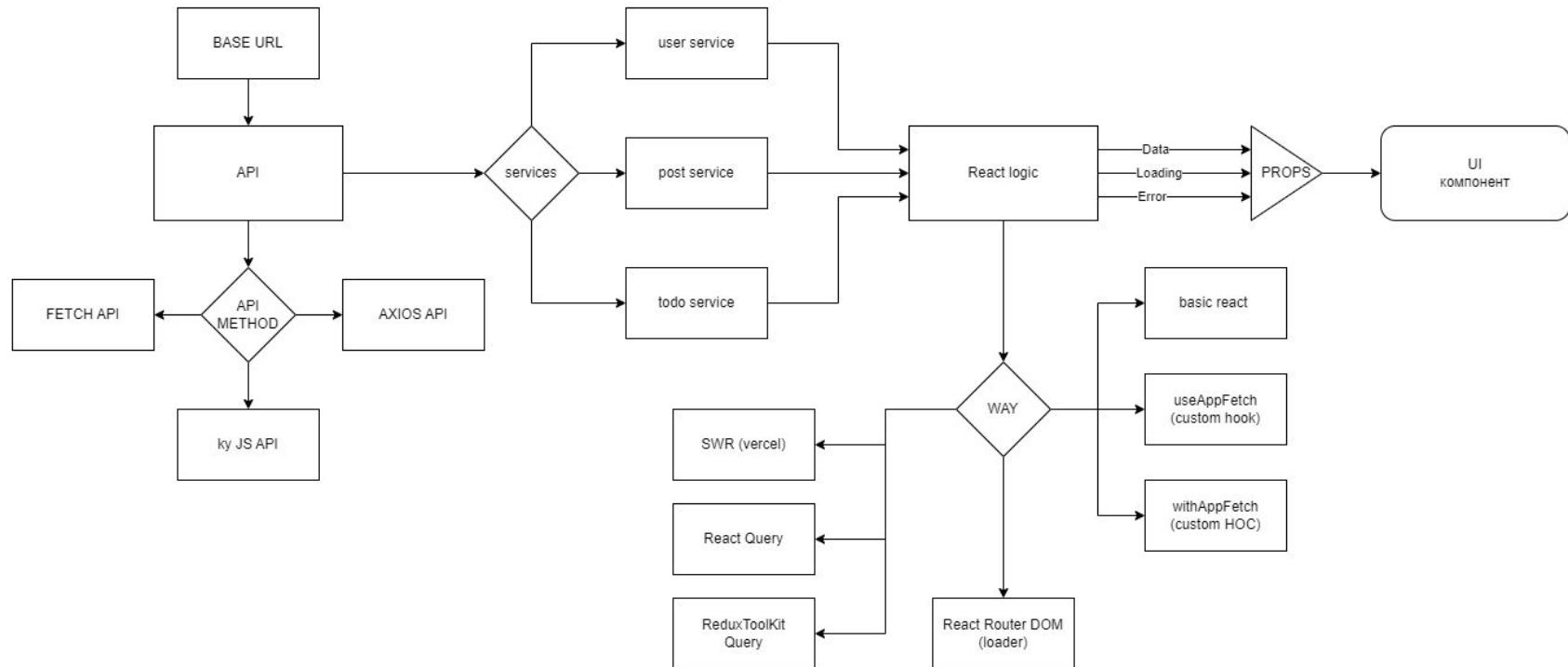
- [Main page](#)
- [Basic hook page](#)
- [Custom hook page](#)
- [Custom HOC page](#)
- [React-router-dom page](#)

BasicHookPage

[refresh request](#)

- user with id:1
user with name:Leanne Graham
- user with id:2
user with name:Ervin Howell
- user with id:3
user with name:Clementine Bauch
- user with id:4
user with name:Patricia Lebsack
- user with id:5
user with name:Chelsey Dietrich
- user with id:6
user with name:Mrs. Dennis Schulist
- user with id:7
user with name:Kurtis Weissnat
- user with id:8
user with name:Nicholas Runolfsdottir V
- user with id:9
user with name:Glenna Reichert
- user with id:10
user with name:Clementina DuBuque

Схема реализации сетевых запросов



API

Получение данных с мокового сервера JSONPlaceholder

API - <https://jsonplaceholder.typicode.com/users>

Запросы реализованы 3 способами:

- fetch API
- axios API
- ky API

Также запрос обернут 2 декораторами:

- декоратором логирования в консоль запроса
- декоратором искусственной задержки запроса

```
import fetchAPI from "./fetch-api";
import kyAPI from "./ky-api";
import axiosAPI from "./axios-api";

import { USER_API_ENDPOINT } from "@/const";
import { delayDecorator, withAsyncLogs } from "@/tools";

const API_METHODS = {
  fetchAPI,
  kyAPI,
  axiosAPI,
};

const requestDelayImitation = 1000;

const userServiceCore = API_METHODS.axiosAPI;

const userServiceWithLogs = withAsyncLogs(userServiceCore);

const userServiceDelayed = delayDecorator(
  userServiceWithLogs,
  requestDelayImitation
);

export const userService = () => userServiceDelayed(USER_API_ENDPOINT);
```

Компонент отображения пользователей

Виджет списка пользователей
состоит из 3 компонентов

- Список пользователей - отображает список пользователей
- Компонент визуального оповещения о загрузке
- Компонент визуального оповещения о ошибке

```
import { AppUserList, AppError, AppLoading } from "@/components";
import PropTypes from "prop-types";
import { memo } from "react";
const AppUsersCore = ({ userListLoading, userListData, userListError }) => {
  if (userListLoading) {
    return <AppLoading />;
  }
  if (userListError) {
    return <AppError message="user list error" />;
  }
  return <AppUserList userList={userListData} />;
};

AppUsersCore.propTypes = {
  userListLoading: PropTypes.bool.isRequired,
  userListData: PropTypes.array.isRequired,
  userListError: PropTypes.bool.isRequired,
};

export const AppUsers = memo(AppUsersCore);
```

Базовый функционал React

userList — хранит список пользователей, который загружается из API.

userError — флаг, указывающий на наличие ошибки при запросе данных.

userLoading — флаг загрузки, который указывает, идет ли процесс запроса данных.

Функция **getUserListHandler** — для получения списка пользователей из сервиса **userService**.

useEffect вызывает функцию **getUserListHandler** при первом рендеринге компонента.

```
import { userService } from "@/api";
import { AppUsers } from "@/widjets";
import { useCallback, useEffect, useState } from "react";

export const BasicHookPage = () => [
  const [userList, setUserList] = useState([]);
  const [userError, setUserError] = useState(false);
  const [userLoading, setUserLoading] = useState(false);

  const getUserListHandler = useCallback(() => {
    const getData = async () => {
      try {
        setUserError(false);
        setUserLoading(true);
        const data = await userService();
        setUserList(data);
      } catch (error) {
        setUserError(true);
      } finally {
        setUserLoading(false);
      }
    };

    getData();
  }, []);

  useEffect(() => {
    getUserListHandler();
  }, [getUserListHandler]);

  return (
    <>
      <h1>BasicHookPage</h1>
      <div>
        <button disabled={userLoading} onClick={getUserListHandler}>
          refresh request
        </button>
      </div>
      <AppUsers
        userListData={userList}
        userListError={userError}
        userListLoading={userLoading}
      />
    </>
  );
};
```

Проблема сетевых запросов в React

Основной минус данного подхода — необходимость повторно описывать логику для каждого сетевого запроса. Это приводит к дублированию кода, что может затруднить поддержку и масштабирование приложения. В каждом компоненте приходится реализовывать одно и то же: управление состояниями загрузки, ошибок и сам запрос. Вот несколько основных проблем такого подхода:

- **Дублирование кода:** Для каждого запроса необходимо отдельно описывать логику для загрузки данных, обработки ошибок и состояния загрузки. В крупных приложениях это приведет к повторению одного и того же кода во многих компонентах.
- **Трудности в поддержке:** При изменении логики обработки запросов (например, нужно добавить кэширование, авторизацию или обработку токенов) придется изменять код в нескольких местах, что увеличивает риск ошибок и усложняет поддержку.
- **Негибкость:** Такой подход не масштабируется хорошо. Если нужно изменять поведение для одного компонента, возможно, придется переписывать части логики для других компонентов, если они используют ту же структуру для сетевых запросов.

Пользовательский хук (hook)

Кастомные хуки
Создание кастомных хуков для работы с API поможет переиспользовать логику запросов и избавиться от дублирования. Например, можно создать хук `useAppFetch`, который будет универсальным для всех типов запросов.

```
/* useAppFetch.js */
import { useCallback, useState } from "react";

export function useAppFetch(initialValue, fetchAPI) {
  const [data, setData] = useState(initialValue);

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

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

  const getData = useCallback(() => {
    (async () => {
      try {
        setError(false);
        setLoading(true);
        const data = await fetchAPI();
        setData(data);
      } catch (error) {
        setError(true);
      } finally {
        setLoading(false);
      }
    })();
  }, [fetchAPI]);

  return [data, error, loading, getData];
}

import { userService } from "@/api";
import { useAppFetch } from "@/use";
import { AppUsers } from "@/widjets";
import { useEffect } from "react";

export const CustomHookPage = () => {
  const [userList, userError, userLoading, getUserListHandler] = useAppFetch(
    [],
    userService
  );
  useEffect(() => {
    getUserListHandler();
  }, [getUserListHandler]);

  return (
    <>
      <h1>CustomHookPage</h1>
      <div>
        <button disabled={userLoading} onClick={getUserListHandler}>
          refresh request
        </button>
      </div>
      <AppUsers
        userListData={userList}
        userListError={userError}
        userListLoading={userLoading}
      />
    </>
  );
};
```

Пользовательский компонент (НОС)

Компонент, который получает данные через НОС, остается "чистым" — он не содержит в себе логики для сетевых запросов. Это делает его проще и легче для тестирования.

```
import { useAppFetch } from "@/use";

export function withAppFetch(WrappedComponent, fetchAPI, initialValue = null) {
  return function WithAppFetchComponent(props) {
    const [data, error, loading, getData] = useAppFetch(initialValue, fetchAPI);

    return (
      <WrappedComponent
        {...props}
        data={data}
        error={error}
        loading={loading}
        action={getData}
      />
    );
  };
}
```

```
import { userService } from "@/api";
import { withAppFetch } from "@/hoc";
import { AppUsers } from "@/widjets";
import PropTypes from "prop-types";
import { memo, useEffect } from "react";

const CustomHocPageCore = ({ data, error, loading, action }) => {
  useEffect(() => {
    action();
  }, [action]);
  return (
    <>
      <h1>CustomHocPage</h1>
      <div>
        <button disabled={loading} onClick={action}>
          | refresh request
        </button>
      </div>
      <AppUsers
        userListData={data}
        userListError={error}
        userListLoading={loading}
      />
    </>
  );
};

CustomHocPageCore.propTypes = {
  data: PropTypes.array,
  error: PropTypes.bool,
  loading: PropTypes.bool,
  action: PropTypes.func,
};

export const CustomHocPage = withAppFetch(
  memo(CustomHocPageCore),
  userService,
  []
);
```

Сетевые запросы через React Router DOM

В React Router DOM v6 добавлена возможность использования loaders для асинхронной загрузки данных, необходимых для отображения компонента до рендеринга. Это делает загрузку данных более декларативной и позволяет упрощать логику, связанную с запросами внутри компонентов.

loader — это функция, которая вызывается перед тем, как компонент маршрута будет отрендерен.

Она загружает данные, необходимые для этого маршрута, и передает их в компонент с помощью объекта **useLoaderData**.



В React Router DOM v6 появился метод **defer**, который позволяет загружать данные асинхронно и управлять их частичной загрузкой. Это полезно, когда нужно загрузить часть данных сразу, а другие данные можно загрузить позже, не блокируя рендеринг страницы.

```
import { userService } from "@/api";
import { APP_PATH } from "@/const";
import { AppLayout } from "@/layout";
import {
  MainPage,
  BasicHookPage,
  CustomHookPage,
  CustomHocPage,
  ErrorPage,
  ReactRouterDOMPage,
} from "@/pages";

import { createBrowserRouter, Navigate, defer } from "react-router-dom";

export const router = createBrowserRouter([
  {
    element: <AppLayout />,
    children: [
      { index: true, element: <Navigate to={APP_PATH.MAIN} replace /> },
      { path: APP_PATH.MAIN, element: <MainPage /> },
      { path: APP_PATH.BASIC_HOOK, element: <BasicHookPage /> },
      { path: APP_PATH.CUSTOM_HOOK, element: <CustomHookPage /> },
      { path: APP_PATH.CUSTOM_HOC, element: <CustomHocPage /> },
      {
        path: APP_PATH.REACT_ROUTER_DOM,
        element: <ReactRouterDOMPage />,
        loader: () => defer({ data: userService(), random: Math.random() }),
      },
    ],
  },
  { path: APP_PATH.ERROR, element: <ErrorPage /> },
  { path: "*", element: <Navigate to={APP_PATH.ERROR} replace /> },
];
)
```

```
import { AppError, AppLoading } from "@/components";
import { AppUsers } from "@/widjets";
import { Suspense } from "react";
import { useLoaderData, useRevalidator, Await } from "react-router-dom";

export const ReactRouterDOMPage = () => {
  const { data } = useLoaderData();
  const { revalidate, state } = useRevalidator();

  return (
    <>
      <h1>React-Router-DOM page</h1>
      <div>
        <button disabled={state === "loading"} onClick={() => revalidate(0)}>
          | refresh request
        </button>
      </div>
      <Suspense fallback={<AppLoading />}>
        <Await
          resolve={data}
          errorElement={<AppError message="User List Error" />}
        >
          {((resolvedData) => (
            <AppUsers
              userListData={resolvedData}
              userListError={false}
              userListLoading={state === "loading"}
            />
          ))}
        </Await>
      </Suspense>
    </>
  );
};
```

Библиотеки SWR, React Query, и RTK Query

Библиотеки SWR, React Query, и RTK Query предоставляют более мощные и гибкие механизмы для работы с асинхронными запросами и кэшированием данных в React. Основная "фишка" этих библиотек заключается в том, что они упрощают управление состоянием данных, кэшированием, синхронизацией и повторным запросом данных, освобождая разработчиков от ручного управления асинхронными запросами.

Общие преимущества этих библиотек

Кэширование данных: Эти библиотеки автоматически кэшируют результаты запросов, что позволяет избежать лишних повторных запросов и ускоряет рендеринг.

Повторное использование данных: Кэшированные данные можно повторно использовать в разных компонентах без необходимости повторного запроса, пока данные актуальны.

Автоматическое обновление данных: Если данные изменились на сервере, библиотеки могут автоматически обновлять кэш и состояние в компоненте, обеспечивая актуальность данных.

Функции для повторного запроса: Все библиотеки поддерживают автоматические и ручные повторные запросы данных, например, при изменении сети, потере соединения или активации страницы.

Управление состоянием загрузки и ошибок: Легкое управление состояниями, такими как загрузка, ошибки и успешное получение данных.

SWR vercel

SWR (от "stale-while-revalidate") — это библиотека от разработчиков Next.js, которая фокусируется на простом и эффективном механизме работы с данными.

- **Простота:** Минимальная настройка и простой API, который интегрируется с любым фетчингом данных (например, с `fetch` или `axios`).
- **Stale-While-Revalidate:** Данные сначала возвращаются из кэша (даже если они устарели), а затем библиотека отправляет запрос для обновления данных в фоновом режиме.
- **Интеграция с React Hooks:** Простой в использовании интерфейс с помощью хуков.

The logo for SWR, consisting of the letters 'SWR' in a bold, white, sans-serif font.

React хуки для выборки данных

Установка:
npm install swr

Пример SWR

useSWR(SWR_USER_LIST_KEY, userService, {...}) — этот хук делает запрос к серверу через `userService` и сохраняет данные под ключом `SWR_USER_LIST_KEY`. SWR управляет кэшированием данных и автоматической проверкой обновлений. Опции:

revalidateIfStale — если данные устарели, происходит повторный запрос.

revalidateOnMount — данные будут запрошены заново при монтировании компонента.

keepPreviousData — сохраняет предыдущие данные до получения новых (чтобы избежать "мигания" интерфейса).

data: userList — данные (список пользователей), полученные от сервера. Если данных нет, будет `undefined`.

error: userError — ошибка, если произошел сбой при запросе.

isValidating: userValidating — указывает, происходит ли сейчас валидация (запрос данных).

isLoading: userLoading — показывает, загружаются ли данные.

mutate: getUserListHandler — функция для ручного обновления данных (можно вызывать для повторного запроса).

```
import { userService } from "@api";
import { USER_API_ENDPOINT } from "@/const";
import { AppUsers } from "@/widjets";
import { useEffect } from "react";
import useSWR from "swr";

export const SWRLibPage = () => {
  const {
    data: userList,
    error: userError,
    isValidating: userLoading,
    mutate: getUserListHandler,
  } = useSWR(USER_API_ENDPOINT, userService, { revalidateIfStale: true });

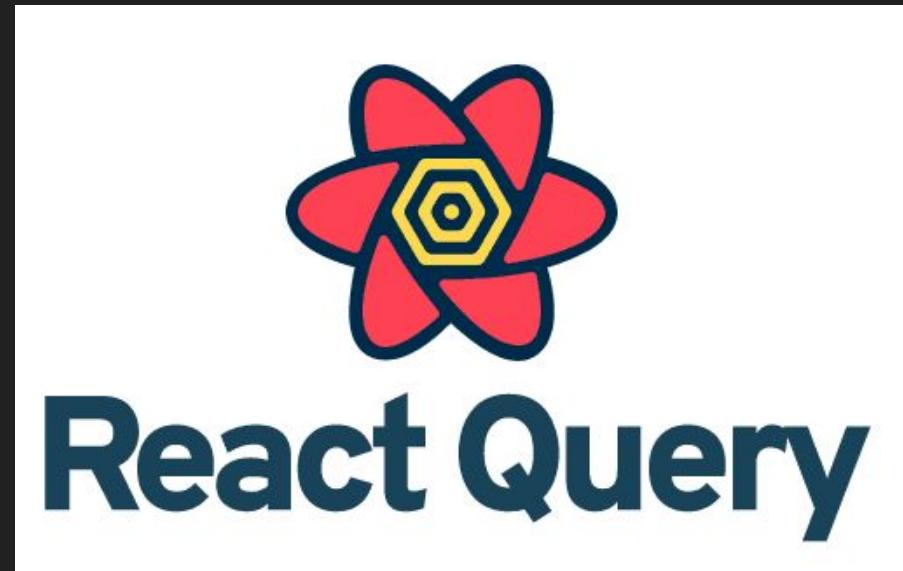
  useEffect(() => {
    getUserListHandler();
  }, [getUserListHandler]);

  return (
    <>
      <h1>SWR lib page</h1>
      <div>
        <button disabled={userLoading} onClick={getUserListHandler}>
          | refresh request
        </button>
      </div>
      <AppUsers
        userListData={userList}
        userListError={userError}
        userListLoading={userLoading}
      />
    </>
  );
};
```

React query (TanStack)

React Query — это популярная библиотека для управления серверным состоянием в React-приложениях. Она помогает эффективно извлекать, кэшировать, синхронизировать и обновлять данные, получаемые с сервера, упрощая работу с асинхронными запросами.

Библиотека React Query значительно упрощает управление серверными данными в приложении, устранивая необходимость ручного написания логики для запросов и кэширования.



Пример React Query

data: Полученные данные о пользователях.

error: Ошибка, если запрос завершился неудачей.

isLoading: Флаг, указывающий, что запрос находится в процессе загрузки.

isFetching: Флаг, указывающий, что запрос находится в процессе получения данных (в том числе при фоновом обновлении).

refetch: Функция для повторного запроса данных.

queryKey: Уникальный ключ для кэширования данных. В этом случае используется константа **REACT_QUERY_USER_LIST_KEY**.

queryFn: Функция, которая выполняет запрос. В данном случае это **userService**.

refetchOnMount: Параметр, который указывает, что данные должны повторно запрашиваться при монтировании компонента.

initialData: Начальные данные для запроса, если данные еще не загружены. Здесь это пустой массив.

```
import { userService } from "@api";
import { REACT_QUERY_USER_LIST_KEY } from "@const";
import { AppUsers } from "@widjets";
import { useQuery } from "tanstack/react-query";
import { withReactQuery } from "@hoc";

const ReactQueryLibPageCore = () => {
  const {
    data: userList,
    error: userError,
    isLoading: userLoading,
    isFetching: userFetching,
    refetch: getUserListHandler,
  } = useQuery({
    queryKey: [REACT_QUERY_USER_LIST_KEY],
    queryFn: userService,
    refetchOnMount: "always",
    initialData: [],
  });

  return (
    <>
      <h1>React lib page</h1>
      <div>
        <button
          disabled={userLoading || userFetching}
          onClick={getUserListHandler}
        >
          refresh request
        </button>
      </div>
      <AppUsers
        userListData={userList}
        userListError={userError}
        userListLoading={userLoading || userFetching}
      />
    </>
  );
};

export const ReactQueryLibPage = withReactQuery(ReactQueryLibPageCore);
```

Ресурсы

Код с урока (исходник git репозиторий) - [ТЫК](#)

Код с урока (деплой vercel) - [ТЫК](#)

Сетевые запросы (JavaScript Learn онлайн учебник, ru) - [ТЫК](#)

AXIOS документация (ru) - [ТЫК](#)

ky документация (en) - [ТЫК](#)

SWR (vercel) документация (ru) - [ТЫК](#)

React Query (TanStack - оф. документация, en) - [ТЫК](#)

React Query (учебник, ru) - [ТЫК](#)