

React Router DOM

Роутеры

Маршрутизация

Навигация

layout

Вложенные роуты

Динамические роуты

Перенаправление

Кнопка назад

Обработка ошибок

Загрузка данных

Ленивая загрузка

Авторизация и защищенные роуты

RouterProvider

Роутеры

В React Router DOM 6 есть несколько типов роутеров, каждый из которых используется в зависимости от особенностей проекта.

- **BrowserRouter:** работает с обычными URL и требует настройки на сервере для поддержки маршрутизации.
- **HashRouter:** использует хэши в URL и не требует серверной настройки.
- **MemoryRouter:** хранит маршруты в памяти, не изменяя URL, что полезно для тестирования.
- **StaticRouter:** предназначен для серверного рендеринга, не взаимодействуя с историей браузера напрямую.

Каждый роутер имеет своё применение в зависимости от специфики приложения.

BrowserRouter

Описание: Это основной роутер для использования в веб-приложениях, работающих через стандартные URL. Он использует HTML5 API `pushState`, `replaceState` и обработку событий навигации.

Когда использовать: Когда ваше приложение работает с современными браузерами и вам нужно поддерживать URL без хэшей (например, `/about`, `/contact`).

Особенности:

- Поддерживает чистые URL.
- Требуется правильная настройка сервера для работы с маршрутами на стороне клиента (сервер должен возвращать `index.html` для всех запросов).

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

function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </BrowserRouter>
  );
}
```

HashRouter

Описание: Этот роутер использует символ # в URL для маршрутизации. Он работает без необходимости настройки сервера, так как всё управление URL осуществляется через часть после # (например, /#/about).

Когда использовать: Когда серверная часть не настроена для поддержки маршрутов на стороне клиента или в ситуациях, где необходимо поддерживать старые браузеры, которые не поддерживают HTML5 API для работы с историей.

Особенности:

- Упрощенная настройка, не требует изменений на стороне сервера.
- URL содержит хэш, что может быть нежелательно с точки зрения SEO.

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

function App() {
  return (
    <HashRouter>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </HashRouter>
  );
}
```

MemoryRouter

Описание: Этот роутер хранит состояние истории в памяти, а не в URL. Он не изменяет строку адреса в браузере.

Когда использовать: В тестах, когда нужно изолировать роутинг или в приложениях, где не требуется управлять URL (например, в мобильных приложениях или специальных компонентах).

Особенности:

- Не работает с URL, что делает его идеальным для тестирования компонентов или специфических сценариев.
- Не интегрируется с кнопками "Назад" и "Вперед" в браузере.

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

function App() {
  return (
    <MemoryRouter>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </MemoryRouter>
  );
}
```

StaticRouter

Описание: Этот роутер предназначен для серверного рендеринга (SSR) и не изменяет состояние истории. Вместо этого он получает историю как пропсы.

Когда использовать: В проектах с серверным рендерингом (например, с использованием Next.js или других SSR решений), где маршрутизация обрабатывается на стороне сервера.

Особенности:

- Используется для статических сайтов или для приложений с серверной маршрутизацией.
- Позволяет контролировать рендеринг в зависимости от маршрутов на сервере.

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

function App({ location }) {
  return (
    <StaticRouter location={location}>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </StaticRouter>
  );
}
```

Маршрутизация

Маршрутизация — это процесс, посредством которого приложение определяет, какой компонент или ресурс должен быть загружен и отображен на основе URL-адреса, который запрашивает пользователь.

- **Компонентный способ (<Routes> и <Route>)** — это более декларативный и читаемый подход. Он часто используется, когда количество маршрутов заранее известно и не меняется.
- **Хук `useRoutes`** — более программный подход, который можно использовать для динамической генерации маршрутов. Этот способ удобен, если маршруты могут изменяться в зависимости от данных или условий.

Оба способа могут быть полезны в зависимости от контекста и задач проекта.

Компонентный способ

Это наиболее распространенный способ, где маршруты описываются с использованием компонентов **<Routes>** и **<Route>**.

- **<Routes>**: Это контейнер для всех маршрутов, который выбирает первый подходящий маршрут.
- **<Route>**: Определяет путь и компонент, который должен отображаться при совпадении пути. В атрибуте `element` передается компонент, который будет рендериться.

```
import { BrowserRouter, Routes, Route } from 'react-router-dom';
import Home from './Home';
import About from './About';

function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </BrowserRouter>
  );
}

export default App;
```

Хук useRoutes

С помощью хука **useRoutes** можно динамически создавать маршруты в функциональных компонентах. Этот способ может быть полезен, когда нужно гибко управлять списком маршрутов, например, загружать их асинхронно.

- **useRoutes:** Этот хук принимает массив объектов маршрутов и возвращает компоненты, соответствующие текущему пути.
- Здесь маршруты определяются как массив объектов с ключами **path** и **element**, что позволяет гибко управлять маршрутами внутри компонентов.

```
import { BrowserRouter, useRoutes } from 'react-router-dom';
import Home from './Home';
import About from './About';

function AppRoutes() {
  let routes = useRoutes([
    { path: "/", element: <Home /> },
    { path: "/about", element: <About /> }
  ]);

  return routes;
}

function App() {
  return (
    <BrowserRouter>
      <AppRoutes />
    </BrowserRouter>
  );
}

export default App;
```

Способы навигации в React Router DOM 6:

Навигация в контексте веб-разработки — это процесс перемещения пользователя между различными страницами или разделами веб-приложения. Она включает в себя изменение URL-адреса и загрузку соответствующего контента без необходимости перезагрузки всей страницы.

- **Компонент Link**
- **Компонент NavLink**
- **Хук useNavigate**
- **Хук useLocation**
- **Хук useParams**
- **Хук useSearchParams**

Link

Это основной способ навигации между страницами. Он рендерит обычный HTML-элемент `<a>`, но вместо перезагрузки страницы выполняет навигацию через клиентский роутинг.

to: путь, на который мы хотим перейти.

Поддерживаются как абсолютные, так и относительные пути.

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

function Navigation() {
  return (
    <nav>
      <Link to="/">Home</Link>
      <Link to="/about">About</Link>
      <Link to="/contact">Contact</Link>
    </nav>
  );
}
```

NavLink

Это расширенная версия Link, которая позволяет добавлять активный стиль к ссылкам, когда они соответствуют текущему маршруту. Это полезно для отображения, какая ссылка активна.

activeClassName: класс, который добавляется, если ссылка активна (актуален для v5). В v6 для этого используется объект стиля или функция.

```
import { NavLink } from 'react-router-dom';

function Navigation() {
  return (
    <nav>
      <NavLink to="/" activeClassName="active">Home</NavLink>
      <NavLink to="/about" activeClassName="active">About</NavLink>
      <NavLink to="/contact" activeClassName="active">Contact</NavLink>
    </nav>
  );
}
```

```
<NavLink
  to="/about"
  style={({ isActive }) => ({ color: isActive ? 'red' : 'blue' })>
  About
</NavLink>
```

useNavigate

Хук `useNavigate` предоставляет программный способ навигации. Это аналог хука `useHistory` из предыдущих версий, но с обновленным API.

- **`navigate(path, options)`**: позволяет перейти на указанный путь программно.
- **`options.replace`**: если указать как `true`, заменит текущую запись в истории, вместо добавления новой.
- **`options.state`**: передача состояния при навигации, например, данных между страницами.

Пример с заменой истории и передачей состояния:

```
navigate('/about', { replace: true, state: { from: 'home' } });
```

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

function MyComponent() {
  const navigate = useNavigate();

  const goToAbout = () => {
    navigate('/about');
  };

  return (
    <button onClick={goToAbout}>Go to About</button>
  );
}
```

useLocation

Хук useLocation возвращает объект, содержащий текущий URL, pathname, search (query параметры) и state.

```
import { useLocation } from 'react-router-dom';

function MyComponent() {
  const location = useLocation();

  console.log(location.pathname); // текущий путь
  console.log(location.search); // строка query параметров
  console.log(location.state); // состояние переданное при навигации

  return <div>Current path: {location.pathname}</div>;
}
```

Хук useParams

Хук useParams используется для получения параметров маршрута, таких как динамические сегменты URL.

Если в маршруте указано что-то вроде /user/:userId, то useParams вернет объект с параметром userId.

```
import { useParams } from 'react-router-dom';

function UserProfile() {
  const { userId } = useParams();

  return <div>User ID: {userId}</div>;
}
```

Хук useSearchParams

Хук для работы с query-параметрами строки запроса.

- **searchParams.get('param'):** получение значения query-параметра.
- **setSearchParams({ param: value }):** обновление query-параметров.

```
import { useSearchParams } from 'react-router-dom';

function SearchPage() {
  const [searchParams, setSearchParams] = useSearchParams();

  const updateSearch = () => {
    setSearchParams({ filter: 'new' });
  };

  return (
    <div>
      <p>Current Filter: {searchParams.get('filter')}</p>
      <button onClick={updateSearch}>Set New Filter</button>
    </div>
  );
}
```

Вложенные роуты

В React Router DOM v6 вложенные маршруты (роуты) реализуются немного иначе, чем в предыдущих версиях. В v6 маршруты определяются как элементы, и вложенные маршруты указываются внутри родительского маршрута с помощью компонентов `Route`.

Основная идея вложенных маршрутов это шаблонные компоненты `Layout`

В React Router DOM v6 параметр `index` используется для определения маршрута, который должен отображаться по умолчанию при совпадении родительского маршрута. Это особенно полезно для вложенных маршрутов, где вы хотите указать, какой маршрут должен быть активен по умолчанию, если пользователь переходит к родительскому маршруту без указания конкретного вложенного маршрута.

Layout

В react-router-dom 6 концепция layout (макет) позволяет вам организовать и управлять структурой страниц и компонентов, которые повторяются на разных маршрутах, например, шапку, боковую панель или футер. Макеты помогают разделить код и сделать его более организованным, обеспечивая единый стиль и функционал для нескольких страниц.

Основные принципы использования Layout в React Router DOM 6

- **Создание Layout компонента:** Layout компонент — это базовый компонент, который содержит общие элементы интерфейса, такие как шапка, меню и футер. Он будет рендерить вложенные маршруты с помощью `<Outlet>`.
- **Вложенные маршруты:** Вложенные маршруты позволяют Layout компоненту определять общую структуру, а внутренние маршруты определяют контент, который изменяется в зависимости от URL.

Пример использования Layout

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

function MainLayout() {
  return (
    <div>
      <header>
        <nav>
          <Link to="/">Home</Link>
          <Link to="/profile">Profile</Link>
        </nav>
      </header>
      <main>
        <Outlet />
      </main>
      <footer>
        <p>Footer content here</p>
      </footer>
    </div>
  );
}

export default MainLayout;
```

```
import { BrowserRouter, Routes, Route } from 'react-router-dom';
import MainLayout from './MainLayout';
import Home from './Home';
import Profile from './Profile';

function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<MainLayout />}>
          <Route index element={<Home />} />
          <Route path="profile" element={<Profile />} />
        </Route>
      </Routes>
    </BrowserRouter>
  );
}

export default App;
```

```
// Home.js
function Home() {
  return <h1>Home Page</h1>;
}

export default Home;
```

```
// Profile.js
function Profile() {
  return <h1>Profile Page</h1>;
}

export default Profile;
```

Динамические роуты

Динамические маршруты в React Router DOM позволяют создавать маршруты, которые могут изменяться в зависимости от параметров URL. Это особенно полезно, когда вы хотите отображать данные, связанные с конкретными параметрами, такими как идентификаторы, имена пользователей или любые другие переменные значения.

Пример использования динамических маршрутов

Рассмотрим пример приложения, в котором у нас есть список пользователей, и мы хотим отображать информацию о каждом пользователе на основе их идентификатора.

Объяснение

- **path="/users/:userId"**: Это определение динамического маршрута, где `:userId` — это параметр маршрута, который будет заменяться конкретным значением из URL. Например, если URL — `/users/1`, то `userId` будет `1`.
- **useParams**: Хук из `react-router-dom`, который позволяет извлекать параметры из URL. В компоненте `UserDetail` мы используем `useParams` для получения значения `userId`, которое затем можно использовать для отображения информации о пользователе.

```
- /  
  - /users  
    - /:userId (должен отображать информацию о пользователе по ID)
```

```
// App.jsx
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';
import UserList from './components/UserList';
import UserDetails from './components/UserDetail';

function App() {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<h1>Home Page</h1>} />
        <Route path="/users" element={<UserList />} />
        <Route path="/users/:userId" element={<UserDetail />} />
      </Routes>
    </Router>
  );
}

export default App;
```

```
// components/UserDetail.jsx
import { useParams } from 'react-router-dom';

function UserDetails() {
  const { userId } = useParams();
  // Здесь вы можете сделать запрос к API или использовать данные
  // Например, используя userId для получения информации о пользователе
  return <h2>Details for user ID: {userId}</h2>;
}

export default UserDetails;
```

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

function UserList() {
  const users = [
    { id: 1, name: 'Alice' },
    { id: 2, name: 'Bob' },
    { id: 3, name: 'Charlie' },
  ];

  return (
    <div>
      <h1>User List</h1>
      <ul>
        {users.map(user => (
          <li key={user.id}>
            <Link to={`/${users}/${user.id}`}>>{user.name}</Link>
          </li>
        ))}
      </ul>
    </div>
  );
}

export default UserList;
```

Пример с множественными параметрами

Рассмотрим приложение для блогов, в котором у нас есть пользователи и посты. Мы хотим, чтобы URL содержал и идентификатор пользователя, и идентификатор поста, чтобы отображать конкретный пост конкретного пользователя.

Объяснение

- **path="users/:userId/posts/:postId"**: В этом маршруте используются два параметра — `userId` и `postId`. Эти параметры могут быть любыми строками, но обычно они используются для идентификации записей.
- **useParams**: Хук `useParams` позволяет извлекать значения параметров маршрута. В компоненте `PostDetail` мы используем этот хук для получения значений `userId` и `postId` из URL.
- **Создание ссылок**: В компоненте `UserPostsList` мы используем компонент `<Link>` для создания ссылок, которые содержат значения параметров в URL. Это позволяет пользователям перейти к конкретному посту определенного пользователя.

```
- /  
  - /users/:userId/posts/:postId  
    - отображает пост конкретного пользователя
```

```
// components/PostDetail.jsx
import { useParams } from 'react-router-dom';

function PostDetail() {
  const { userId, postId } = useParams();

  // Здесь вы можете сделать запрос к API или использовать данные
  // Например, используя userId и postId для получения информации о посте
  return (
    <div>
      <h2>Post Detail</h2>
      <p>User ID: {userId}</p>
      <p>Post ID: {postId}</p>
      {/* Здесь можно добавить дополнительную логику для отображения данных поста */}
    </div>
  );
}
```

```
export default PostDetail;
```

```
// App.jsx
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';
import PostDetail from './components/PostDetail';

function App() {
  return (
    <Router>
      <Routes>
        <Route path="/" element=<h1>Home Page</h1> />
        <Route path="users/:userId/posts/:postId" element=<PostDetail /> />
      </Routes>
    </Router>
  );
}
```

```
export default App;
```

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

```
function UserPostsList() {
  const posts = [
    { userId: 1, postId: 101, title: 'Post 1' },
    { userId: 1, postId: 102, title: 'Post 2' },
    { userId: 2, postId: 201, title: 'Post 3' },
  ];

  return (
    <div>
      <h1>User Posts</h1>
      <ul>
        {posts.map(post => (
          <li key={post.postId}>
            <Link to={`~/users/${post.userId}/posts/${post.postId}`}>
              {post.title}
            </Link>
          </li>
        ))}
      </ul>
    </div>
  );
}
```

```
export default UserPostsList;
```

Перенаправление

Перенаправление в React Router DOM позволяет автоматически перенаправлять пользователей с одного маршрута на другой. Это может быть полезно в различных сценариях, таких как:

- Переход с устаревшего маршрута на новый
- Направление неавторизованных пользователей на страницу входа
- Направление пользователей на страницу с ошибкой (например, 404)

Как реализовать перенаправление в React Router DOM

Перенаправление реализуется с помощью компонента и хука:

- компонент **<Navigate>**
- хук **useNavigate**

```
import { Navigate } from 'react-router-dom';

function RedirectExample() {
  return <Navigate to="/new-route" />;
}
```

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

function Login() {
  const navigate = useNavigate();

  const handleLogin = () => {
    // Выполните действия по входу
    // ...

    // Перенаправление на страницу после успешного входа
    navigate('/dashboard');
  };

  return <button onClick={handleLogin}>Login</button>;
}
```

Кнопка назад

Браузерная кнопка "Назад" и другие навигационные кнопки, такие как "Вперед", по умолчанию работают с React Router DOM, так как он использует history API. Однако, если вам нужно выполнить дополнительные действия при навигации, вы можете использовать хук `useNavigate` или `useLocation`.

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

function MyComponent() {
  const navigate = useNavigate();

  const goBack = () => {
    navigate(-1); // Навигация на одну запись назад в истории
  };

  return (
    <button onClick={goBack}>Back</button>
  );
}
```

404 страница

Для обработки несуществующих маршрутов вы можете использовать маршрут с `path="*" в вашем компоненте <Routes> . Это будет действовать как "покрытие" для всех маршрутов, которые не соответствуют ранее определенным маршрутам.`

```
function NotFound() {  
  return (  
    <div>  
      <h1>404 - Страница не найдена</h1>  
      <p>Извините, мы не можем найти эту страницу.</p>  
    </div>  
  );  
}  
  
export default NotFound;
```

```
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';  
import Home from './components/Home';  
import About from './components/About';  
import NotFound from './components/NotFound';  
  
function App() {  
  return (  
    <Router>  
      <Routes>  
        <Route path="/" element={<Home />} />  
        <Route path="/about" element={<About />} />  
        <Route path="*" element={<NotFound />} /> { /* 404 обработка */}  
      </Routes>  
    </Router>  
  );  
}  
  
export default App;
```

Обработка ошибок

Подавляющее большинство ошибок вашего приложения обрабатывается React Router автоматически. Он перехватывает все ошибки, возникающие при:

- рендеринга
- загрузка данных
- обновлении данных

На практике это практически все ошибки в вашем приложении, за исключением тех, которые возникают в обработчиках событий (<button onClick>) или useEffect. В приложениях React Router их, как правило, очень мало.

При возникновении ошибки вместо отображения element маршрута отображается errorElement.

Если маршрут не имеет errorElement, то ошибка переходит на ближайший родительский маршрут с errorElement:

```
<Route
  path="/"
  loader={() => {
    something.that.throws.an.error();
  }}
  // this will not be rendered
  element={<HappyPath />}
  // but this will instead
  errorElement={<ErrorBoundary />}
/>
```

Загрузка данных

Поскольку сегменты URL обычно связаны с постоянными данными вашего приложения, React Router предоставляет обычные крючки загрузки данных для инициирования загрузки данных во время навигации. В сочетании с вложенными маршрутами все данные для нескольких макетов по определенному URL-адресу могут загружаться параллельно.

Для реализации требуются:

- атрибут/параметр **loader**
- хук **useLoaderData**

Как это работает

- **Конфигурация маршрутов:** В конфигурации маршрутов вы определяете маршрут с **loader**, который будет вызван при активации маршрута.
- **Вызов loader:** Когда пользователь переходит по маршруту, функция **loader** выполняется и загружает данные.
- **Получение данных:** В компоненте, связанный с этим маршрутом, вы используете хук **useLoaderData** для получения данных, загруженных функцией **loader**.

loader

Функция loader в react-router-dom 6 позволяет загружать данные до рендеринга компонента маршрута. Это важно для сценариев, когда вам нужно предварительно загрузить данные, чтобы они были доступны вашему компоненту до его рендеринга.

Основные Концепции

- **Определение loader:** Функция loader задается на уровне маршрута в конфигурации маршрутизации. Она может быть асинхронной и должна возвращать данные, которые будут переданы компоненту маршрута.
- **Использование loader:** Когда маршрут с loader активируется, loader вызывается и загружает необходимые данные. Эти данные затем передаются в компонент маршрута через хук `useLoaderData`.
- **Асинхронные операции:** Функция loader может выполнять асинхронные операции, такие как запросы к API или другие операции, требующие времени.
- **Обработка ошибок:** Если функция loader вызывает ошибку, это можно обработать через механизм ошибок роутера или кастомные обработки ошибок.

```
<Route
  path="/"
  loader={async ({ request }) => {
    // loaders can be async functions
    const res = await fetch('/api/user.json', {
      signal: request.signal,
    });
    const user = await res.json();
    return user;
  }}
  element={<Root />}
>

<Route
  path=":teamId"
  // loaders understand Fetch Responses and will automatically
  // unwrap the res.json(), so you can simply return a fetch
  loader={({ params }) => {
    return fetch(`/api/teams/${params.teamId}`);
  }}
  element={<Team />}
>

<Route
  path=":gameId"
  loader={({ params }) => {
    // of course you can use any data store
    return fakeSdk.getTeam(params.gameId);
  }}
  element={<Game />}
/>
</Route>
</Route>
```

useLoaderData

Хук `useLoaderData` в `react-router-dom 6` используется для доступа к данным, которые были загружены функцией `loader` перед рендерингом компонента маршрута. Это обеспечивает удобный способ получения и использования данных, предварительно загруженных для данного маршрута.

Основные моменты использования `useLoaderData`

Как это работает:

- Когда маршрут с `loader` активируется, функция `loader` выполняется и загружает данные.
- Эти данные становятся доступными в компоненте через хук `useLoaderData`.

Получение данных:

- Хук `useLoaderData` возвращает данные, которые были загружены функцией `loader`. Вы можете использовать эти данные для рендеринга вашего компонента.

```
function Root() {  
  const user = useLoaderData();  
  // data from <Route path="/">  
}  
  
function Team() {  
  const team = useLoaderData();  
  // data from <Route path=":teamId">  
}  
  
function Game() {  
  const game = useLoaderData();  
  // data from <Route path=":gameId">  
}
```

Ленивая загрузка (lazy + suspense)

Ленивая загрузка и Suspense являются мощными инструментами в React для управления загрузкой компонентов по мере их необходимости, что может значительно улучшить производительность вашего приложения. Когда они используются в связке с react-router-dom 6, вы можете эффективно загружать и отображать компоненты маршрутов по мере их активации.

Как это работает

- **Ленивая загрузка:** Компоненты Home, About и NotFound загружаются лениво с помощью React.lazy. Это означает, что они будут загружены только когда пользователь перейдет на соответствующий маршрут.
- **Suspense:** Компонент Suspense оборачивает маршрутизатор и предоставляет запасной UI (<Loading />), который будет отображаться, пока загружаются ленивые компоненты.
- **Конфигурация маршрутов:** Конфигурация маршрутов определяется с помощью createBrowserRouter, где каждый маршрут связан с лениво загружаемым компонентом.

Пример

```
// Ленивая загрузка компонентов
const Home = lazy(() => import('./Home'));
const About = lazy(() => import('./About'));
const NotFound = lazy(() => import('./NotFound'));

// Компонент для отображения индикатора загрузки
const Loading = () => <div>Loading...</div>;
```

```
// Конфигурация маршрутов
const router = createBrowserRouter([
  {
    path: '/',
    element: <Home />,
  },
  {
    path: '/about',
    element: <About />,
  },
  {
    path: '*',
    element: <NotFound />,
  },
]);
```

```
// Основной компонент приложения
function App() {
  return (
    <Suspense fallback=<Loading />>
      <RouterProvider router={router} />
    </Suspense>
  );
}

export default App;
```

Авторизация и защищенные роуты

Авторизация и защищенные роуты в приложениях на React — важные аспекты для управления доступом к различным частям вашего приложения. В связке с `react-router-dom` 6, вы можете настроить защищенные роуты, которые требуют аутентификации пользователя для доступа.

Основные шаги для реализации авторизации и защищенных роутов

- **Создание контекста для аутентификации:** Используйте контекст для хранения состояния аутентификации и предоставления информации о текущем пользователе.
- **Создание защищенных маршрутов:** Реализуйте компонент, который проверяет, аутентифицирован ли пользователь, и перенаправляет его на страницу входа, если нет.
- **Настройка маршрутов:** Настройте маршруты так, чтобы защищенные маршруты использовали компонент проверки аутентификации.

Создание контекста для аутентификации

AuthContext и AuthProvider управляют состоянием аутентификации. В данном случае, проверка состояния аутентификации может быть основана на наличии токена в localStorage.

```
import React, { createContext, useState, useContext, useEffect } from 'react';

// Создание контекста аутентификации
const AuthContext = createContext();

export const useAuth = () => useContext(AuthContext);

// Провайдер аутентификации
export const AuthProvider = ({ children }) => {
  const [auth, setAuth] = useState(false);

  useEffect(() => {
    // Логика для проверки текущего состояния аутентификации (например, проверка токена)
    const checkAuth = async () => {
      // Пример проверки аутентификации
      const token = localStorage.getItem('token');
      setAuth(!!token);
    };
    checkAuth();
  }, []);

  return (
    <AuthContext.Provider value={{ auth, setAuth }}>
      {children}
    </AuthContext.Provider>
  );
};
```

Создание компонента защищенного маршрута

Компонент `ProtectedRoute` проверяет состояние аутентификации. Если пользователь не аутентифицирован, он перенаправляется на страницу входа.

```
import React from 'react';
import { Navigate, useLocation } from 'react-router-dom';
import { useAuth } from '../AuthContext';

const ProtectedRoute = ({ element }) => {
  const { auth } = useAuth();
  const location = useLocation();

  if (!auth) {
    // Перенаправление на страницу входа или другую защищенную страницу
    return <Navigate to="/login" state={{ from: location }} />;
  }

  return element;
};

export default ProtectedRoute;
```

Настройка маршрутов

```
// Компоненты маршрутов
const Home = () => <div>Home Page</div>;
const Login = () => <div>Login Page</div>;
const Dashboard = () => <div>Dashboard Page</div>;
```

```
// Основной компонент приложения
function App() {
  return (
    <AuthProvider>
      <RouterProvider router={router} />
    </AuthProvider>
  );
}

export default App;
```

```
// Конфигурация маршрутов
const router = createBrowserRouter([
  {
    path: '/',
    element: <Home />,
  },
  {
    path: '/login',
    element: <Login />,
  },
  {
    path: '/dashboard',
    element: <ProtectedRoute element={<Dashboard />} />,
  },
]);
```

RouterProvider - нововведение в ReactRouterDOM 6

В **React Router DOM 6** появился новый компонент — **RouterProvider**, который упрощает настройку маршрутизации и делает её более гибкой. Он используется для передачи объекта маршрутизатора (router) в ваше приложение и позволяет гибко конфигурировать роуты, поддержку данных и даже асинхронную маршрутизацию.

Основные особенности компонента RouterProvider:

- **Управление маршрутизатором через объект:** Вместо стандартного использования **BrowserRouter** или **HashRouter**, компонент **RouterProvider** позволяет вам передавать объект маршрутизатора, что дает возможность более гибкой и централизованной настройки.
- **Асинхронная маршрутизация:** Теперь можно загружать данные, компоненты или даже целые маршруты асинхронно, и **RouterProvider** будет управлять этим процессом.
- **Поддержка различных типов маршрутизаторов:** Можно использовать не только стандартные маршрутизаторы, такие как **BrowserRouter**, но и кастомные маршрутизаторы или те, которые работают с другими протоколами (например, для серверного рендеринга).

Как работает RouterProvider:

createBrowserRouter: Эта функция создаёт браузерный маршрутизатор, который можно передать в RouterProvider.

RouterProvider: Передаёт созданный маршрутизатор вашему приложению.

```
import { createBrowserRouter, RouterProvider } from 'react-router-dom';
import Home from './Home';
import About from './About';
import ErrorPage from './ErrorPage';

const router = createBrowserRouter([
  {
    path: "/",
    element: <Home />,
    errorElement: <ErrorPage />, // компонент для отображения ошибок маршрутизации
  },
  {
    path: "about",
    element: <About />,
  },
]);

function App() {
  return (
    <RouterProvider router={router} />
  );
}

export default App;
```

Асинхронная маршрутизация с RouterProvider

Одним из сильных моментов React Router 6 является возможность загружать данные или компоненты асинхронно. Вы можете использовать специальные loaders для загрузки данных до того, как компонент будет отображен.

```
import { createBrowserRouter, RouterProvider } from 'react-router-dom';

// Функция загрузки данных
async function fetchUserData() {
  const response = await fetch('/api/user');
  return response.json();
}

const router = createBrowserRouter([
  {
    path: "/",
    element: <Home />,
    loader: fetchUserData, // Загружаем данные до рендеринга компонента Home
  },
]);

function App() {
  return (
    <RouterProvider router={router} /> // передаем маршрутизатор в RouterProvider
  );
}

export default App;
```

fallbackElement

`fallbackElement` — это элемент, который отображается во время загрузки данных или компонента для маршрута, если используются асинхронные загрузчики. Он обеспечивает более плавный пользовательский опыт, показывая пользователю что-то, пока основное содержимое загружается.

Объяснение:

- **fallbackElement:** Компонент `Loading`, который будет отображен, пока данные загружаются. Это особенно полезно, если загрузка может занять некоторое время.
- **Асинхронные загрузчики:** Пока данные загружаются через `loader`, `React Router` автоматически покажет `fallbackElement`.

```
import { createBrowserRouter, RouterProvider } from 'react-router-dom';

// Функция для загрузки данных
async function fetchData() {
  const response = await fetch('/api/data');
  return response.json();
}

// Компонент для отображения во время загрузки
const Loading = () => <div>Loading...</div>;

const router = createBrowserRouter([
  {
    path: "/",
    element: <Home />,
    loader: fetchData,      // Загрузка данных
    fallbackElement: <Loading /> // Заглушка во время загрузки данных
  },
]);

function App() {
  return (
    <RouterProvider router={router} />
  );
}

export default App;
```

Обработка ошибок маршрутизации

Если пользователь попытается перейти на несуществующую страницу, можно настроить компонент для обработки таких ошибок с помощью атрибута `errorElement`.

```
const router = createBrowserRouter([
  {
    path: "/",
    element: <Home />,
    errorElement: <ErrorPage />, // компонент, который будет отображаться при ошибке
  },
  {
    path: "about",
    element: <About />,
  },
]);
```

Продвинутая маршрутизация

Вы можете строить более сложные маршруты, используя вложенные маршруты и динамическую маршрутизацию.

```
const router = createBrowserRouter([
  {
    path: "/",
    element: <Home />,
    children: [
      {
        path: "about",
        element: <About />,
      },
      {
        path: "profile/:userId", // динамический сегмент маршрута
        element: <UserProfile />,
        loader: async ({ params }) => {
          const response = await fetch(`/api/user/${params.userId}`);
          return response.json();
        },
      },
    ],
  },
]);
```

Какие типы маршрутизаторов поддерживает RouterProvider?

В зависимости от того, как ваше приложение управляет навигацией, можно использовать разные маршрутизаторы:

- **createBrowserRouter** — используется для стандартной клиентской маршрутизации с историей браузера (включая поддержку `pushState`).
- **createHashRouter** — используется для маршрутизации, где используются хеши (#) в URL (например, для старых приложений).
- **createMemoryRouter** — используется для тестирования или серверного рендеринга, где история браузера не доступна.

Плюсы использования RouterProvider:

- Централизованное управление маршрутами: Все маршруты, обработчики ошибок и загрузки данных находятся в одном месте.
- Асинхронная маршрутизация: Поддержка loaders и actions для загрузки данных до рендеринга компонента.
- Гибкость и масштабируемость: Вы можете легко управлять вложенными маршрутами и динамическими сегментами URL.

RouterProvider — это мощный инструмент в React Router 6, который предоставляет разработчикам гибкость в создании маршрутов и управления навигацией. Он особенно полезен для приложений с асинхронной загрузкой данных, где важно централизованное управление маршрутизацией и эффективная обработка ошибок.

Ресурсы React-Router-DOM v6

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

Код урока деплой (vercel) - [ТЫК](#)

Официальная документация - [ТЫК](#)

Обзор функций (на русском) - [ТЫК](#)

Учебное пособие (разбор примера) - [ТЫК](#)

GIT репозиторий с примерами - [ТЫК](#)