

REACT ROUTER



МИХАИЛ ЛАРЧЕНКО



МИХАИЛ ЛАРЧЕНКО

Frontend разработчик



horrorgr@gmail.com



ПЛАН ЗАНЯТИЯ

1. [Router, Route](#)
2. [Вложенность Route](#)
3. [Link](#)
4. [Состояние ссылок](#)
5. [Switch](#)
6. [Route с параметрами](#)
7. [Redirect](#)

React Router

React - это библиотека для создания пользовательского интерфейса. Это не фреймворк, поэтому, для получения возможностей создания современных приложений, нам нужно использовать дополнительные инструменты (либо написать их самим).

Один из них таких инструментов — библиотека управления роутингом `React Router`.



ROUTER, ROUTE

Router, Route

Рассмотрим основные компоненты библиотеки - `Router` и `Route`. Это обычные React-компоненты с одной лишь разницей - они не имеют визуального представления (не создают DOM-элементы).

Приведем пример их использования. Мы хотим сделать приложение, которое показывает список кулинарных рецептов и позволяет посмотреть детальную информацию о каждом из них. Так же мы хотим иметь возможность кидать ссылку на рецепт.

Ссылки на страницы нашего приложения:

<http://localhost:3000/recipes> <http://localhost:3000/new>

ПОДКЛЮЧАЕМ React-Router

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

```
yarn add react-router-dom
```

```
1 import {BrowserRouter as Router, Route} from 'react-router-dom';  
2 const App = () => (  
3   <Router>  
4     <div>  
5       <Route path="recipes" component={RecipesList} />  
6       <Route path="new" component={NewRecipe} />  
7     </div>  
8   </Router>  
9 )
```



BROWSERROUTE

BrowserRouter - это Router (компонент, отвечающий за навигацию) использующий HTML5 History API для синхронизации дерева компонентов с URL.

Есть ещё HashRouter, который манипулирует `location.hash`. Но поскольку поддержка History API на данный момент вполне достаточна, мы будем использовать BrowserRouter.

КОНФИГУРАЦИЯ СЕРВЕРА

Использование *History API* потребует некоторых изменений на стороне сервера. Потому что при запросе `/recipes` сервер будет искать не `index.html` (в котором скомпилировано React-приложение), а путь `/recipes`.

Поэтому сервер настраивают так, при отсутствии файла по определённом пути отдаётся `index.html`, который загружает React и React-Router и исходя из URL грузит компоненты.

КОНФИГУРАЦИЯ СЕРВЕРА

Например, для nginx:

```
1 location / {  
2     try_files $uri /index.html =404;  
3 }
```

Для apache:

```
1 RewriteEngine On  
2 RewriteCond %{DOCUMENT_ROOT}%{REQUEST_URI} -f [OR]  
3 RewriteCond %{DOCUMENT_ROOT}%{REQUEST_URI} -d  
4 RewriteRule ^ - [L]  
5  
6 RewriteRule ^ /index.html
```



ROUTE

Route - это ключевой компонент, который отвечает за сопоставление пути с текущим путём в URL. Исходя из того, проходит сопоставление или нет - "активируется" тот или иной роут, и, соответственно, отрисовывается дочерний компонент или нет.

СОЗДАДИМ СТРАНИЦЫ ПРИЛОЖЕНИЯ

Для начала создадим страницы с отображением списка рецептов и созданием нового. Здесь, у приложения есть два роута, `recipes` и `new`. Если текущий путь совпадает с одним из них, то рендирится тот компонент, с которым данный роут связан.

МЫ НА `/recipes`

Допустим путь имеет вид `/recipes`, тогда будет отображен компонент `Recipes`, а виртуальный DOM будет иметь следующий вид:

```
1  <App>
2    <Router>
3      <div>
4        <Route path="/recipes">
5          <RecipesList />
6        </Route>
7        <Route path="/new">
8          null
9        </Route>
10     </div>
11   </Router>
12 </App>
```

Как видно, компонент `NewRecipe` внутри роута с `new` не был добавлен в дерево.

ЛОГИКА РАБОТЫ Router

`Router` отслеживает изменение `path` в URL браузера и перерисовывает приложение, если тот изменился.

`Router` создается в единственном экземпляре и если он не является предком остальных компонент `React Router`, то они работать не будут!

ЛОГИКА РАБОТЫ `Route`

`Route` связывает атрибут `path` с React-компонентом (атрибут `component`, `render`, `children`), который отобразится при нахождении `path` в URL.

Это означает, что если `path=/recipes`, то `Route` сработает в случае `/recipes`, но также в случае `/recipes/about`.

Часто это не совсем то поведение, которое мы ожидаем.

СТРОГОЕ СООТВЕТСТВИЕ

Чтобы установить строгое соответствие `path`, необходимо добавить в `Route` атрибут `exact`. Тогда `Route` с `path=/recipes` сработает только когда `/recipes`.



ВЛОЖЕННОСТЬ ROUTE

Route МОЖЕТ БЫТЬ ВЛОЖЕННЫМ

Route можно вкладывать в друг друга. Это полезно, если у нас имеется компонент, который должен присутствовать на разных URL.

Например мы хотим, чтобы при показе компонента создания рецепта, также отображался и список:

- <http://localhost:3000/recipes>
- <http://localhost:3000/recipes/new>

РЕАЛИЗУЕМ ВЛОЖЕННЫЙ Route

```
1  const App = () => (  
2    <Router>  
3    <div>  
4      <Route path="/recipes" component={RecipesList} />  
5    </div>  
6    </Router>  
7  );  
8  
9  const RecipesList = ({match}) => (  
10    <div>  
11      <h1>  
12        Recipes List!  
13      </h1>  
14      <Route path={`/${match.url}/new`} component={NewRecipe} />  
15    </div>  
16  );
```

ДЕРЕВО КОМПОНЕНТОВ ПРИ `path=/recipes/new`

Теперь при заходе в приложение с `path` равным `/recipes/new`, дерево компонентов будет выглядеть следующим образом:

```
1  <App>
2    <Router>
3      <div>
4        <Route path="/recipes">
5          <RecipesList>
6            <Route path="/recipes/new">
7              <NewRecipe />
8            </Route>
9          </RecipesList>
10         </Route>
11       </div>
12     </Router>
13   </App>
```

ОБЪЕКТ `match`

Обратите внимание, что мы используем `match.url` для составления атрибута `path` вложенному `Route`.

Каждый раз, когда рендерится компонент, привязанный к определенному `Route`, в него передается атрибут `match`. `match` это объект, который содержит много свойств, но нас пока интересует только `url`.

В самом простом случае `match.url` равен атрибуту `path` у `Route`, а значит `match.url` имеет значение `/recipes`.

ОБЪЕКТ `history`

Помимо объекта `match`, передается также объект `history`, позволяющий программно управлять навигацией с помощью методов:

- `push(path)` - перейти по определённому пути (переход попадает в историю, т.е. доступен с помощью кнопок назад и вперед)
- `replace(path)` - перейти по определённому пути (переход не попадает в историю, т.е. не доступен с помощью кнопок назад и вперед)
- `go(n)` - передвигает указатель истории на n позиций (аналог кнопки назад, нажатой несколько раз, если $n < 0$, и кнопке вперед, если $n > 0$)
- `goBack()` - эквивалент `go(-1)`
- `goForward()` - эквивалент `go(1)`

ОБЪЕКТ `location`

Предоставляет информацию о текущем состоянии роутера (либо о том, где мы находились, либо будем находится - в `componentDidUpdate`):

- `pathname` - путь в URL
- `search` - query string (то, что после ?)
- и т.д.

props

Здесь стоит сделать важное замечание: все эти props будут доступны только в трёх случаях:

1. `<Route component={MyComponent} />`
2. `<Route render={props => <MyComponent {...props} />} />`
3. `<Route>{props => <MyComponent {...props} />}</Route>*`

Примечание*: случае даже если совпадения не произойдёт, содержимое `props.children` всё равно будет отрисовано, но `match` будет равно `null`

АЛЬТЕРНАТИВЫ

Q: Но что делать, если мы хотим получить доступ в других случаях?

A: Конечно же, есть альтернативы:

— Хуки:

- useHistory
- useLocation
- useParams
- useRouterMatch

(все они возвращают объекты: `const history = useHistory();`)

— HOC `withRouter`, который прокидывает эти props (а как пользоваться HOC вы уже знаете).



LINK

ПЕРЕКЛЮЧЕНИЕ МЕЖДУ СТРАНИЦАМИ SPA

Сейчас в нашем приложении мы можем переключаться между страницами только напрямую вбивая URL в браузере. Это очень неудобно.

В HTML для навигации по сайту используется тэг `<a>`. Он появился в то время, когда интернет представлял из себя набор статичных HTML страниц. Поэтому при использовании его в SPA, возникают трудности.

ДОБАВИМ НАВИГАЦИЮ

Чтобы реализовать подобную навигацию:

<http://localhost:3000>

нам придется отслеживать клик по каждой ссылке, отменять действие по умолчанию (чтобы не было перезагрузки и самим вызывать метод

`history.pushState`)

РЕАЛИЗУЕМ НАВИГАЦИЮ ЧЕРЕЗ `<a>`

```
1 function AppWithoutLinks() {
2   const onLinkClick = e => {
3     const path = e.target.pathname + e.target.search || ''
4     e.preventDefault()
5     history.push(path)
6   }
7   return (
8     <Router history={history}>
9       <div>
10        <div>
11          <a href='/recipes?foo=bar'
12            onClick={onLinkClick}>Рецепты</a>
13        </div>
14        <div>
15          <a href='/recipes/new'
16            onClick={onLinkClick}>Новый рецепт</a>
17        </div>
18        <Route exact path='/recipes' component={RecipesList}/>
19        <Route exact path='/recipes/new' component={NewRecipe}/>
20      </div>
21    </Router>
22  )
23 }
```

Link

Неплохо было бы иметь компонент, который как тэг `<a>`, решает проблему навигации, но при этом создан специально для одностраничных приложений.

И такой компонент в `React Router` есть, это - `Link`.

ЛОГИКА РАБОТЫ `Link`

`Link` принимает атрибут `to`, который можно назвать аналогом `href`.

При клике по `Link` не происходит перезагрузки страницы. Вместо этого значение в атрибуте `to` ищется в `path` множества всех `Route`, и при совпадении найденный роут рендерит свой компонент, а URL браузера меняется на значение из `to`.

СВОЙСТВА АТТРИБУТА `to`

`to` может принимать объект со следующими свойствами:

- `pathname` : путь роута, например `/recipes` ;
- `search` : строка запроса, например `order=decent` ;
- `hash` : якорь, например `#the-hash` ;
- `state` : объект состояния, например `{fromDashboard: true}` .

Первые три значения аналогичны значениям в `window.location` .

`state` - произвольный объект, и имеет тоже самое назначение, что и первый аргумент в `history.pushState` .

ДОБАВИМ ССЫЛКИ

Добавим ссылки на страницы списка и создания рецепта:

```
1  const App = () => (  
2    <Router>  
3    <div>  
4      <nav>  
5        <Link to="/recipes">Рецепты</Link>  
6        <Link to="/recipes/new">Новый рецепт</Link>  
7      </nav>  
8      <Route exact path="/recipes" component={RecipesList} />  
9      <Route exact path="/recipes/new" component={NewRecipe} />  
10   </div>  
11   </Router>  
12 )
```



СОСТОЯНИЕ ССЫЛОК



NavLink

На многих сайтах в шапке есть меню. При переключении элементов меню обычно выделяют текущий пункт меню, чтобы пользователю было проще понимать где он сейчас находится.

Специально для таких случаев в `React Router` есть компонент `NavLink`.

АТТРИБУТЫ `NavLink`

`NavLink` идентичен `Link`, но имеет дополнительные атрибуты:

- `activeClassName`;
- `activeStyle`;
- `exact`;
- `strict`.

ОСОБЕННОСТИ ИСПОЛЬЗОВАНИЯ NavLink

- Если атрибут `to` содержится в `location.path`, то в `NavLink` устанавливаются значения, `className=activeClassName` и `style=activeStyle`.
- При установке `exact`, `activeClassName`, `activeStyle` будет применен, только если `location.path` полностью совпадает с `to` (по аналогии с `Route.exact`).
- С атрибутом `strict` учитывается `/` в конце `location.path`.

ВЫДЕЛИМ АКТИВНЫЕ ССЫЛКИ

Реализуем выделение активных ссылок

Объявим `activeStyle`:

```
1  const activeStyle = {  
2    color: 'red'  
3  }
```

КОД НАШЕГО МЕНЮ

```
1  const App = () => (  
2    <Router>  
3    <div>  
4      <nav>  
5        <NavLink to="/recipes" activeStyle={activeStyle}>  
6          Рецепты  
7        </NavLink>  
8        <NavLink strict to="/recipes/" activeStyle={activeStyle}>  
9          Рецепты со slash  
10       </NavLink>  
11       <NavLink to="/recipes/new" activeStyle={activeStyle}>  
12         Новый рецепт  
13       </NavLink>  
14       <NavLink to="/recipes/new?foo=bar" activeStyle={activeStyle}>  
15         Новый рецепт с параметрами  
16       </NavLink>  
17     </nav>  
18     <Route exact path="/recipes" component={RecipesList} />  
19     <Route exact path="/recipes/new" component={NewRecipe} />  
20   </div>  
21 </Router>  
22 );
```



SWITCH

КОМПОНЕНТ `Switch`

Почти во всех примерах выше мы ставили атрибут `exact` у компонентов `Route`. Делали мы этого для того, чтобы несколько `Route` не рендерились одновременно.

Также есть другой способ получить такое же поведение - использовать компонент `Switch`.

ЛОГИКА РАБОТЫ Switch

Его поведение похоже на `switch` в JavaScript. `Switch` идет по списку `Route`, которые находятся внутри него, находит первое совпадение и на этом останавливается. Все остальные `Route`, отрисованы не будут.

Перепишем предыдущий пример с использованием `Switch`.

ИСПОЛЬЗУЕМ Switch

```
1  const App = () => (  
2    <Router>  
3    <div>  
4      <nav>  
5        <NavLink to="/recipes" activeStyle={activeStyle}>  
6          Рецепты  
7        </NavLink>  
8        // ... остальные ссылки  
9      </nav>  
10     <Switch>  
11       <Route path="/recipes/new" component={NewRecipe} />  
12       <Route path="/recipes" component={RecipesList} />  
13     </Switch>  
14   </div>  
15   </Router>  
16 );
```

Нужно отсортировать `Route` от более специфичного к менее, иначе `Switch` всегда будет останавливаться на первом `Route`.



ROUTE С ПАРАМЕТРАМИ



НАСТРОЙКА ПОКАЗА РЕЦЕПТОВ

Итак, у нас есть два роута. Один показывает список рецептов, а второй позволяет создать новый. Теперь осталось самое главное, показывать сами рецепты.

ЛОГИКА ПЕРЕКЛЮЧЕНИЯ

Пусть каждому рецепту будет соответствовать уникальный `id`. Тогда существующие рецепты можно посмотреть по пути `/recipes/1`, `/recipes/2` и так далее.

Чтобы создать `Route`, который будет срабатывать на любой `path`, описанный выше, мы напишем в атрибут `path` строку вида `/recipes/:id`.

При такой записи `:id` приобретает иной смысл. Двоеточие указывает на то, что `id` теперь не строка, а параметр, принимающий произвольное значение.

СОЗДАДИМ КОНТЕКСТ ДЛЯ РЕЦЕПТОВ

```
1 | const RecipesContext = createContext([]);  
2 | export default RecipesContext;
```

СОЗДАДИМ PROVIDER ДЛЯ РЕЦЕПТОВ

```
1  export default function RecipesProvider(props) {
2    const [recipes, setRecipes] = useState([
3      {
4        id: 1,
5        name: 'Borsch'
6      },
7      {
8        id: 2,
9        name: 'PekinDuck'
10     },
11     {
12       id: 3,
13       name: 'FugaFish'
14     }
15   ]);
16
17   return (
18     <RecipesContext.Provider value={recipes}>
19       {props.children}
20     </RecipesContext.Provider>
21   )
22 }
```


ДОБАВИМ ПРЕДСТАВЛЕНИЕ РЕЦЕПТА

```
1  const Recipe = () => (  
2    <div>  
3    Recipe!  
4    </div>  
5  );
```

```
1  const App = () => (  
2    <RecipesProvider>  
3    <Router>  
4      <div>  
5        <nav>  
6          <Link to="/recipes/new">Новый рецепт</Link>  
7        </nav>  
8        <RecipesList>  
9        <Switch>  
10         <Route path="/recipes/new" component={NewRecipe} />  
11         <Route path="/recipes/:id" component={Recipe} />  
12        </Switch>  
13      </div>  
14    </Router>  
15  </RecipesProvider>  
16  );
```

RECIPESLIST

```
1  const RecipesList = () => {
2    const recipes = useContext(RecipesContext);
3    return (
4      <React.Fragment>
5        <h2>Рецепты</h2>
6        <ul>
7          {recipes.map(o =>
8            <li>
9              <Link to={`/recipes/${o.id}`}>{o.name}</Link>
10             </li>
11           )}
12        </ul>
13      </React.Fragment>
14    )
15  }
```

НАШ РЕЦЕПТ НЕ МЕНЯЕТСЯ

При переходе по ссылкам рецептов мы видим все время один и тот же текст рецепта. Исправим эту ситуацию с помощью `match.params`.

`match` передается в компонент, который был отрисован определенным `Route`. Если в атрибуте `path` роута были определены параметры, то в свойстве `params`, которое является объектом, они будут ключами.

В нашем случае `params` будет иметь единственный ключ `id`, и если URL имеет вид `/recipes/1`, то значение ключа будет `1`.

ИЗМЕНИМ Recipe

Переделаем `Recipe`, чтобы он отображал сам рецепт:

```
1  const Recipe = ({match}) => (  
2    const recipes = useContext(RecipesContext);  
3    const recipe = recipes.find(o => o.id === match.params.id);  
4    return (  
5      <div>  
6        <h1>{`Рецепт №${match.params.id}`}</h1>  
7        <p>  
8          {recipe ? recipe.name : 'Не найден'}  
9        </p>  
10     </div>  
11   );  
12 );
```

ПОЧЕМУ ИСПОЛЬЗОВАЛИ `Switch` ?

В нашем случае `Switch` необходим, потому что при переходе на `/recipes/new`, также сработает `/recipes/:id`, что совершенно нам не нужно.

СОЗДАНИЕ ПАРАМЕТРОВ `path`

`path` позволяет создавать параметры не только с помощью синтаксиса `:id`, в него также можно передавать регулярные выражения с некоторыми ограничениями.

Например, мы хотим получать рецепт не только по его `id`, но и по имени. Тогда `path` будет иметь вид:

```
/recipes/:id([0-9]+)?:name([a-zA-Z]+)?
```

Далее, в самом компоненте `Recipe`, мы проверяем либо параметр `id`, либо `name` и, в зависимости от проверки, ищем либо по `id` рецепта, либо по его названию.

ОБНОВИМ КОМПОНЕНТ **Recipe**

```
1  const Recipe = function({match}) {
2    const recipes = useContext(RecipesContext);
3    function findRecipe () {
4      if (match.params.id) {
5        const id = Number(match.params.id);
6        return recipes.find(o => o.id === id).name
7      }
8      if (match.params.name) {
9        const name = match.params.name.toLowerCase();
10       return recipes.find(o => o.name.toLowerCase() === name).name
11     }
12   }
13   return (
14     <p>
15       {findRecipe()}
16     </p>
17   )
18 }
```


СТРАНИЦА 404

Если пользователь ошибся в URL, необходимо показать страничку с ошибкой 404 и подписью `Страница не найдена`.

Решить данную ситуацию в `React Router` очень просто с помощью символа `*` в `path`:

```
1  function Page404() {  
2    return (  
3      <div>  
4        Страница не найдена  
5      </div>  
6    )  
7  }  
8  
9  <Route path='*' component={Page404}/>
```

* ИСПОЛЬЗУЮТ СО Switch

Символ `*` или астерикс матчится на любой URL, если представить его в виде регулярного выражения, то он будет иметь вид `(.*)`.

Очевидно, что такой `Router` будет срабатывать на любой URL. Чтобы такого недопустить, нам нужно использовать компонент `Switch`.

```
1 <Switch>
2   <Route path="/recipes/new" component={NewRecipe} />
3   <Route path="/recipes/:id" component={Recipe} />
4   <Route path='*' component={Page404} />
5 </Switch>
```

Подробнее про использования регулярных выражений в path можно почитать тут: <https://github.com/pillarjs/path-to-regexp>

УПРОЩЕНИЕ

Хотя для `Switch` достаточно:

```
1  <Switch>
2    <Route path="/recipes/new" component={NewRecipe} />
3    <Route path="/recipes/:id" component={Recipe} />
4    <Route component={Page404} />
5  </Switch>
```

Подробнее про использования регулярных выражений в path можно почитать тут: <https://github.com/pillarjs/path-to-regexp>



REDIRECT

Redirect

Помимо программного управления историей, у нас есть возможность перенаправлять пользователя, отрисовывая компонент `Redirect`.

`props` у него достаточно много и они позволяют очень гибко настраивать перенаправление:

- `to= '/path '` - перенаправляет на определённый путь
- `to={object}` - перенаправляет на определённый путь, передавая путь в виде объекта
- `push` - true (push), false (replace)



ИТОГИ



ИТОГИ

React Router - достаточно мощная и самая популярная на сегодняшний день библиотека, позволяющая организовать постраничную навигацию (по факту - скрывание/показ компонентов) и ей нужно уметь пользоваться.



HTTP

Стоит лишь отметить, что умея обрабатывать URL: извлекать параметры из него, мы можем в наших компонентах делать HTTP-запросы для получения нужных данных.

Как это делать - вы уже знаете: можно использовать встроенный хук `useEffect`, можно написать кастомный хук для получения данных, можно выполнять запросы из методов жизненного цикла либо использовать НОС.



Задавайте вопросы и напишите отзыв о лекции!

МИХАИЛ ЛАРЧЕНКО



horrorgr@gmail.com