17. React Router

Содержание урока

- Обзор;
- Что такое React Router?;
- Статический роутинг;
- Динамический роутинг;
- React Router environment packages;
- React Router API;
- `history`;
- Настройка React Router;
- Подведём итоги.

Обзор

Приветствую, дорогой друг! ✌🏻👨🏻‍🎓 Цель нашего урока – разузнать, как с помощью известного пакета `react-router` мы могли бы настроить маршрутизацию современного приложения. 🚦

Итак, к делу! 🥊

Что такое React Router?

React Router – это библиотека для маршрутизации React-приложений. Тебе понадобится роутер для менеджмента URL всякий раз, когда будет необходимо настроить роутинг на веб-сайте, представленном в виде React-приложения с несколькими `экранами`. React Router выполняет эту функцию, синхронизируя при этом UI-состояние приложения с текущим URL.

В двух словах, React Router – это коллекция компонентов навигации, декларативно соединяющихся с приложением.

🔎 Хозяйке на заметку:

> Термин `императивно` в программировании описывает твои инструкции компилятору к тому, что ты хочешь, чтобы было выполнено твоей программой при ее запуске, шаг за шагом. В `декларативном` подходе программирования ты пишешь код, описывающий то, что ты хочешь, но не обязательно то, как этого достичь

> (объявляешь желаемый результат, но не в пошаговом стиле).

Пример `императивного` кода:

```
const double = (arr) => {
  const result = [];

  for (let i = 1; i < arr.length; i++) {
    results.push(arr[i] * 2);
  }

  return result;
}
```

Пример `декларативного` кода:

```
const double = (arr) => arr.map((item) => item * 2);
```

По мере постройки своего приложения ты, возможно, не всегда уверен, как обработать URL правильно. React Router в курсе, как это сделать так, чтобы тебе не нужно было принимать столь множественные важные и большие решения. Как только твой UI выглядит так, как тебе того хотелось бы – приложение готово, чтобы его `разроутить`. 🚦

🔎 Хозяйке на заметку:

> `Роутинг` – это процесс синхронизации URL браузера с соответствующим UI.

`Ментальная модель` React Router называется `динамический роутинг`. `Динамический роутинг` отличается от `статического роутинга`, с которым ты, вероятно, уже знаком из других роутинговых интрументов, используемых в `Rails`, `Express`, `Ember`, `Angular` и других фреймворках. 🐶🐱🐭🐰

Или предыдущих версиях React Router. 🐣

Статический роутинг

Следуя этой ментальной модели, объявление роутов является частью инициализации приложения перед этапом рендеринга. Например, вот как статический роутинг может выглядеть в фреймворке `express`:

```
app.get('/', handleIndex);
app.get('/users', handleUsers);
app.get('/users/:id', handleUsers);
app.get('/users/:id/edit', handleUsers);

app.listen();
```

Стоит обратить внимание, как в этом примере роутинг стал частью процесса инициализации приложения до того, как приложение начинает `слушать`: роуты объявлены на строках кода `1–4`, а `начальная фаза` приложения – на строке кода `6`, после интерпретации инструкций роутинга.

Старые версии React Router также следовали той самой статической модели. ⌚ Давным-давно разработчики React Router чувствовали себя ограниченными тем API, что имели. Фактически они переделывали части React (такие, как методы "жизненного цикла" и другие), и это просто не совпадало с декларативной природой React, данной нам для составления UI. 🖥

Так что сразу после этапа прототипирования разработчики вышли с совсем новым API, что `снаружи` уже не React. Новый API совпадает с React на природном уровне.

Динамический роутинг

`Динамический роутинг` означает, что роутинг является частью `фазы рендеринга`, а не частью фазы конфигурации. Это значит, что в React Router почти всё является компонентом.

Например, вот как можно реализовать вложенный роутинг в React Router v4:

```
const App => (
    <section>
        <Route path ='/cars' component = { Cars } />
    </section>
);

const Cars = () => (
    <section>
        <Route path ='/cars/trucks' component = { Trucks } />
    </section>
);
```

`Route` – это обычный компонент, ровно как и элемент `div` или `section`. Чтобы вложить `Route` в другой `Route` или `div`, нужно просто сделать по тому же принципу, который мог бы быть использован, чтобы описать UI в обычном сценарии, вкладывая `div` в `div` или `Route` в `Route`. 🧙

React Router environment packages

Главный девиз React Router – `learn once, route anywhere (выучи однажды, роути везде)`. 🔔

В каком-то смысле React Router был задуман как `лего-конструктор`. Основные механизмы являются частью низкоуровневых пакетов, которые используются высокоуровневыми драйверами для разных окружений (`environment packages`). 🌏

`Router` - это низкоуровневый интерфейс, общий для остальных высокоуровневых роутеров. А в реальном мире разработки приложения чаще всего используют один из следующих высокоуровневых роутеров:

- `BrowserRouter` - для современных браузерных окружений;
- `HashRouter` - для устаревших браузерных окружений;
- `MemoryRouter` - сохраняет URL-историю в памяти (не осуществляет операции записи-чтения из URL-строки в браузере). Может пригодиться в тестировании окружений, отличных от браузерного, например, React Native;
- `NativeRouter` - для iOS и Android приложений, скомпилированных с использованием окружения React Native;
- `StaticRouter` - никогда не меняет адрес. Может пригодиться при использовании приёма серверного рендеринга (`SSR`).

Самый распространённый случай использования низкоуровневого `Router` – это синхронизация состояния роутинга с библиотекой для управления состоянием приложения `Redux`. Хотя тут стоит заметить, что использование библиотеки для управления состоянием − вовсе необязательный шаг, но допустимая опция глубокой интеграции.

React Router API

Нам доступно два типа роутеров при постройке приложения для браузера:

- `BrowserRouter`;
- `HashRouter`.

Разница между ними заключается в том, каким образом отображена URL, создаваемая каждым из них:

```
import { BrowserRouter } from 'react-router-dom';
http://example.com/about

import { HashRouter } from 'react-router-dom';
http://example.com/#/about
```

✋ Совет бывалых:

> В данной конфигурации более предпочтительным вариантом является `BrowserRouter`, за счёт использования им более современного `HTML5 History API` для работы с историей роутинга. С другой стороны, `HashRouter` использует хеш для того, чтобы запоминать состояние роутинга. `HashRouter` является верным выбором всякий раз, когда необходимо поддерживать устаревшее браузерное

> окружение.

`Route` – это компонент, рендерящий указанный UI, если текущий адрес в браузере совпадает с адресом данного `Route`. У компонента `Route` есть специальный пропс `path`, описывающий его адрес, и каждый `Route` рендерится всякий раз при совпадении строкового значения этого пропса с URL браузера. 📡

```
<Route exact path = '/' component = { Home } />
<Route path = '/login' component = { Login } />
```

Адрес `/` совпадает и с самим `/`, и с адресом `/login`. Таким образом, в обычном сценарии оба компонента `Route` совпадут с адресом браузера `/login` и будут отображены на UI. Во избежание этого, булевой пропс `exact` может быть передан компоненту `Route` с пропсом `path = '/'` в случае, если данный эффект нежелателен.

В ответ на совпадения значения пропса `path` компонента `Route` с текущим адресом в браузере – создастся специальный объект `match`. Объект 'match' несет в себе дополнительную информацию о состоянии навигации. Информация доступна посредством обращения к соответствующим свойствам объекта `match`:

- `match.url` - строковое значение, возвращающее совпавшую часть URL;
- `match.path` - строковое значение, возвращающее значение пропса `path` компонента `Route`;
- `match.isExact` - булевое значение, возвращающее true, если совпадение с текущим URL осуществлено посредством использования пропса `exact` компонента `Route`;
- `match.params` - объект, содержащий пары ключ/значение параметров из URL.

🔎 Хозяйке на заметку:

> Объекты `match` и `history`, или любые другие методы и свойства React Router доступны посредством обращения к `this.props`. Чуть позже в этом уроке мы детально изучим объект `history`. 🔮

Всякий раз, когда несколько компонентов `Route` использованы вместе, все роуты, совпавшие с URL хотя бы частично, будут отрендерены.

```
<Route exact path = '/' component = { Home } />
<Route path = '/products' component = { Products } />
<Route path = '/category' component = { Category } />
<Route path = '/:id' render = { () => (
        <p>I want this text to show up for all routes other than '/', '/products' and '/category' </p>
) } />
```

🔎 Хозяйке на заметку:

> Пропс `render` компонента `Route` позволяет осуществлять рендеринг JSX напрямую, посредством возвращения его из функции. Функция, переданная пропсу `render`, будет вызвана всякий раз, когда адрес браузера совпадает со значением пропса `path` компонента `Route`.

Если URL браузера примет значение `/products`, все роуты, совпавшие с данным адресом, будут отрендерены. Поэтому `Route`, имеющий пропс `path` со значением `:id`, будет отрендерен вместе с компонентом `Products`.

Но если это поведение нежелательно, компонент `Switch` может помочь исправить ситуацию. `Switch` рендерит только `первый` компонент 'Route', совпавший с текущим адресом в браузере.

```
<Switch>
    <Route exact path = '/' component = { Home } />
    <Route path = '/products' component = { Products } />
    <Route path = '/category' component = { Category } />
    <Route path = '/:id' render = { () => (
            <p>I want this text to show up for all routes other than '/', '/products' and '/category' </p>
    ) } />
</Switch>
```

Компонент `Redirect` замещает текущее значения адреса браузера в стеке истории навигации точно так же, как и серверный редирект. Новый адрес возможно указать посредством использования пропса `to`.

Вот небольшой пример использования компонента `Redirect`:

```
render () {
    const { authenticated } = this.props;

    return authenticated ? <App/> : <Redirect to = '/login' />;
}
```

Таким образом, если кто-нибудь попытается обратиться к защищённому роуту, не будучи аутентифицированным, он будет перенаправлен к роуту `/login`.

А вот ещё пример того, как можно использовать компонент `Redirect` вместе с компонентом `Switch` для обработки 404-редиректа:

```
<Switch>
    <Route exact path = '/' component = { Home } />
    <Route path = '/products' component = { Products } />
    <Route path = '/category' component = { Category } />
    <Redirect to = '/404' component = { NotFound } />
</Switch>
```

Компонент `Link` может быть использован для навигации между страницами. Его возможно сравнить с HTML-элементом `<a>`. Однако использование подобного HTML-элемента приведёт к полному обновлению страницы браузера, что вовсе не то, чего мы хотим. Поэтому вместо него мы можем использовать компонент `Link` для навигации к определённому URL и перерендеривать UI умным способом, без полной перезагрузки страницы.

```
<Link to = '/about'>About</Link>
```

Компонент `Link` также имеет булевой пропс `replace`, который, будучи заданным, даёт инструкцию `Link` замещать текущее значение адреса в стеке истории навигации вместо того, чтобы добавлять новое значение поверх него.

Компонент `NavLink` работает так же, как и `Link`, однако дает некоторые дополнительные возможности стилизации, в зависимости от того, совпадает ли текущее значение URL браузера со значением компонента `NavLink` или нет.

```
<NavLink
    to = '/faq'
    activeClassName = 'selected'
>FAQs</NavLink>
```

Всякий раз, когда адрес браузера будет принимать значение `/faq`, данный `NavLink` будет получать CSS-класс `.selected`.

Таким образом, использование компонента `NavLink` позволяет задавать определённые CSS-классы на случай, когда `NavLink` будет совпадать с текущим адресом браузера.

`history`

Пакет `history` — это библиотека, позволяющая нам удобно управлять состоянием истории навигации браузера с помощью JavaScript. `history` предоставляет минималистичный API, позволяющий управлять стеком истории, осуществлять действия навигации, подтверждать навигацию, отменять её, а также сохранять состояние навигации между юзер-сессиями. 📲

Каждый компонент `Router` создает объект `history`, содержащий в себе информацию о текущем адресе, а также о предыдущих адресах истории навигации в стеке. Всякий раз при смене адреса браузера происходит ре-рендер с соответствующим UI и, таким образом, осуществляется ощущение навигации.

Но как именно управлять историей? Объект `history` имеет набор методов для данной цели, таких как `history.push()`, а также `history.replace()`. Метод `history.push()` вызывается всякий раз, когда осуществлён клик по компоненту `<Link />`, а `history.replace()` вызывается при срабатывании компонента `<Redirect />`. 🤺

Другие методы, такие как `history.goBack()` и `history.goForward()`, могут быть использованы для путешествия стеком истории вперед и назад соответственно. 📃

`history` – одна из главных зависимостей React Router, кроме самого React, конечно. К данному объекту можно получить доступ посредством обращения к `this.props.history`. 🗺️

Настройка React Router

Первым делом давайте возьмем компонент `Router` для интересующего нас окружения. В текущей конфигурации мы предпочтём `BrowserRouter` для роутинга в окружении браузера:

```
// src/index.js

import React from 'react';
import ReactDOM from 'react-dom';
import { BrowserRouter } from 'react-router-dom';
import App from './containers/App';

ReactDOM.render(
    <BrowserRouter>
        <App/>
    </BrowserRouter>,
    document.getElementById('root')
);
```

Далее возьмём компонент `Link` для подготовки возможности навигации к другим роутам. А также отрендерим непосредственно сам компонент `Route` для отображения определённого UI при посещении соответствующего адреса URL:

```
// src/containers/App

import React, { Component } from 'react';
import { Link } from 'react-router-dom';
import { Route } from 'react-router';
import Dashboard from 'components/Dashboard';

export default class App extends Component {
    render () {
        return (
            <section>
                <nav>
                    <Link to = '/dashboard'>Dashboard</Link>
                </nav>
                <div>
```

```
16                      <Route path = '/dashboard' component = { Dashboard
}/>
17                  </div>
18              </section>
19          )
20      }
21  }
```

Этот `Route` отрендерит `<Dashboard { ...props } />`, где `props` – это пропсы, связанные с роутингом, например, `match`, `location` или `history`. Если юзер посетит адрес браузера, отличный от `/dashboard`, то `Route` отрендерит ровно ничего (null). Это в целом минимальная композиция роутинга React-приложения. 🤖

Подведём итоги

React Router преследует философию декларативного `динамического роутинга`. Каждый `Route` в `react-router` – это всего лишь компонент. Композиция роутов может выглядеть точно так же, как и композиция обычных компонентов в React-приложении.

Для того чтобы сопоставить нашу интуицию на один уровень с React Router, необходимо думать о компонентах, но не о статических роутах. Думайте о том, как решить встающие проблемы, используя декларативный потенциал React, так как каждый вопрос React Router, по сути, является вопросом React.

С `React Router v4` обращаться значительно проще, так как это Just Components™. 🤠

> Если у вас есть идеи, как улучшить этот урок, пожалуйста, пишите их нам на электропочту hello@lectrum.io. Ваш отзыв очень важен для нас.