

top

КОМПЬЮТЕРНАЯ
АКАДЕМИЯ

Создание веб-приложений с использованием Angular и React

Урок № 8

React:
расширенные
приемы

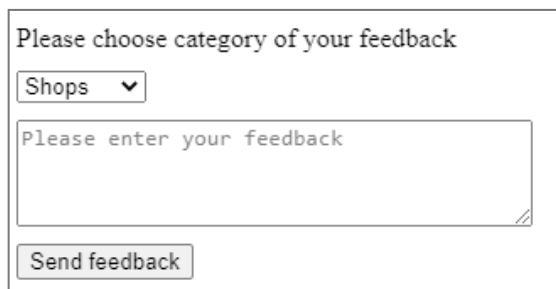
Содержание

И ещё немного о работе с формами	3
Маршруты	11
Атрибут children	20
Дочерние пути.....	22
Создание меню навигации.....	25
Передача параметров маршрута.....	30
Необязательные параметры	36
Маршруты, ссылки, массивы	39
Домашнее задание.....	46

И ещё немного о работе с формами

В любом деле есть два очень сложных момента. Всегда очень сложно начинать. И всегда очень сложно продолжать. Продолжать гораздо сложнее, чем начинать. Если вы читаете этот текст значит вы продолжаете изучать React. И мы поздравляем вас с этим! Вы молодцы!

Сейчас мы ещё раз погрузимся в специфику форм и поговорим о нескольких ранее не изученных аспектах использования форм. Начнем с примера, у которого будет такой UI.



Please choose category of your feedback

Shops ▼

Please enter your feedback

Send feedback

Рисунок 1

Пользователь выбирает в списке категорию фидбека. Пишет текст и отправляет его при нажатии на кнопку «[Send feedback](#)». В коде мы отобразим информационное сообщение с его данными, без отсылки данных на сервер.

В этом примере мы будем использовать список и большое текстовое поле. Код файла *UserForm.js* (мы переименовали *App.js*):

```

import React from "react";
import { useState } from "react";
import "./styles.css";

export default function UserForm() {
  const [content, setContent] = useState("");
  const [selectedItem, setSelectedItem] =
    useState("Shops");

  const handlerTextAreaChanged = event => {
    setContent(event.target.value);
  };
  const handlerSelectChanged = event => {
    setSelectedItem(event.target.value);
  };
  const handlerSubmit = event => {
    event.preventDefault();
    const msg = 'Your feedback about ${selectedItem}:
      \n${content}';
    alert(msg);
  };

  return (
    <div>
      <form className="userForm"
        onSubmit={handlerSubmit}>
        <label>
          Please choose category of your feedback
          <select value={selectedItem}
            onChange={handlerSelectChanged}>
            <option>Service</option>
            <option>Products</option>
            <option>Shops</option>
          </select>
        </label>
        <textarea
          value={content}

```

```

        onChange={handlerTextAreaChanged}
        placeholder="Please enter your feedback"
        required
      />

      <input type="submit" value="Send feedback" />
    </form>
  </div>
);
}

```

В коде примера мы переименовали компонент `App` в `UserForm`. Кроме того, мы внесли новое имя, где это было необходимо. Для создания текстового поля и списка используются теги `textarea` и `select`. Начнем с кода по созданию `textarea`.

```

<textarea
  value={content}
  onChange={handlerTextAreaChanged}
  placeholder="Please enter your feedback"
  required
/>

```

Для связывания переменной состояния и элемента управления мы используем атрибут `value`. Для отображения изменений в текстовом поле мы создали обработчик `onChange`.

Код по созданию списка:

```

<select value={selectedItem}
  onChange={handlerSelectChanged}>
  <option>Service</option>

```

```

    <option>Products</option>
    <option>Shops</option>
  </select>

```

Для создания списка мы используем тег `select`. Для заполнения списка строками мы применяем `option`. И опять же мы используем `value` и `onChange`. Элемент списка, чьё значение указано в `value`, будет выделен. При старте это строка с надписью `Shops`. Обратите внимание, что в обычном HTML для выделения строки используется атрибут `selected`:

```

const [content, setContent] = useState("");
const [selectedItem, setSelectedItem] =
  useState("Shops");

```

Код для работы с состоянием вам уже известен. Мы используем хук состояния дважды. Переменная `content` будет отвечать за текстовое поле. Переменная `selectedItem` за список.

```

const handlerTextAreaChanged = event => {
  setContent(event.target.value);
};
const handlerSelectChanged = event => {
  setSelectedItem(event.target.value);
};
const handlerSubmit = event => {
  event.preventDefault();
  const msg = 'Your feedback about ${selectedItem}:
    \n${content}';
  alert(msg);
};

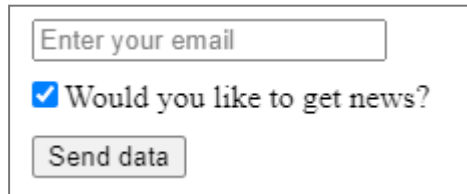
```

Код обработчиков `onChanged` для элементов управления весьма схож. Разница только в том какая функция вызывается. Для текстового поля это `setContent`. Для списка это `setSelectedItem`.

Код обработчика `onSubmit` не должен вызывать у вас сложностей.

Ссылка на проект: <https://codesandbox.io/s/formex-zb3eb>.

В последнем примере код обработчика события `onChange` разных элементов управления был весьма схож. В новом примере мы сделаем один обработчик на два элемента управления. Также мы используем элемент управления `checkbox`. Внешний вид приложения:



The image shows a simple web form. At the top is a text input field with the placeholder text "Enter your email". Below the input field is a checkbox that is checked, followed by the text "Would you like to get news?". At the bottom of the form is a button labeled "Send data".

Рисунок 2

Пользователь вводит свой почтовый адрес. Указывает нужна ли ему подписка на новости. После чего нажимает на кнопку с надписью «`Send data`». В коде мы отобразим информационное сообщение с его данными, без отсылки данных на сервер. Код `UserForm.js`:

```
import React from "react";
import { useState } from "react";
import "./styles.css";

export default function UserForm() {
  const [news, setNews] = useState(true);
```

```

const [email, setEmail] = useState("");
const handlerSubmit = event => {
  event.preventDefault();
  let msg = "";
  if (news === true) {
    msg = "Thank you for subscription!\n";
  }
  msg += "Your email:" + email;
  alert(msg);
};
/*
  Один обработчик события на два input
*/
const handlerChanged = event => {
  const target = event.target;
  /*
    Проверяем для какого элемента возникло событие
  */
  target.name === "aboutNews"
    ? setNews(event.target.checked)
    : setEmail(target.value);
};

return (
  <div>
    <form className="userForm"
      onSubmit={handlerSubmit}>
      <input
        name="userEmail"
        type="email"
        required
        placeholder="Enter your email"
        value={email}
        onChange={handlerChanged}
      />
      <input
        type="checkbox"

```



```

        name="aboutNews"
        checked={news}
        onChange={handlerChanged}
      />
      <label>Would you like to get news?</label>
      <input type="submit" value="Send data" />
    </form>
  </div>
);
}

```

Для отображения `checkbox` мы используем тег `input`:

```

<input
  type="checkbox"
  name="aboutNews"
  checked={news}
  onChange={handlerChanged}
/>

```

Свойство `checked` определяет выбран ли чекбокс. За этим свойством мы закрепляем нашу переменную состояния. На `onChange` мы закрепляем функцию-обработчик `handlerChanged`. Мы указали, что у чекбокса атрибут `name` равен `aboutNews`. Это значение нам понадобится, когда мы будем проверить для какого элемента управления сработал обработчик `onChange`.

Для создания поля для ввода почтового адреса мы также используем тег `input`:

```

<input
  name="userEmail"
  type="email"

```

```

    required
    placeholder="Enter your email"
    value={email}
    onChange={handlerChanged}
  />

```

Мы указали, что у поля для ввода `email` атрибут `name` равен `userEmail`. Это значение нам понадобится, когда мы будем проверить для какого элемента управления сработал обработчик `onChange`. На `onChange` мы закрепляем тот же обработчик `handlerChanged`. Вот его код:

```

const handlerChanged = event => {
  const target = event.target;
  /*
  Проверяем для какого элемента возникло событие
  */
  target.name === "aboutNews"
    ? setNews(event.target.checked)
    : setEmail(target.value);
};

```

Мы должны проверить для какого элемента управления вызвался обработчик. Для этого мы используем свойство `target.name`. Если оно равно `aboutNews`, значит обработчик сработал для чекбокса. И нам нужно обновить его состояние, иначе обработчик был вызван для текстового поля. И тогда обновление состояния нужно ему. Проанализируйте внимательно ещё раз весь код примера, чтобы лучше понять его.

Ссылка на проект: <https://codesandbox.io/s/formex2-24bx4>.

Маршруты

Понятие маршрута известно вам из реальной жизни. Мы уверены, что у вас есть любимые маршруты для прогулок по городу — наличие маршрутов жизненно необходимо нам. Ведь без чёткого маршрута нельзя добраться из точки **A** в точку **B**. При разработке приложений с помощью **React** нам также понадобится механизм маршрутов. Он позволит нам встроить возможности навигации внутрь наших проектов.

Пока во всех наших проектах мы используем только один адрес, но любое веб-приложение обычно использует разные адреса. Например, один для отображения информации о компании, другой для отображения информации о сотрудниках и т.д.

Для подключения механизма маршрутизации (роутинга) в проект нужно использовать модуль **react-router-dom**:

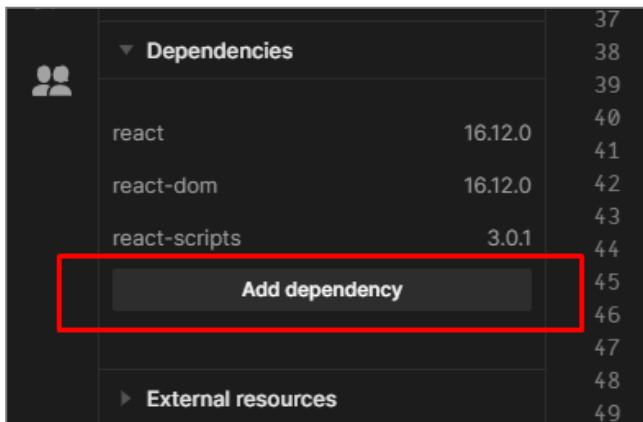


Рисунок 3

Он изначально не подключен к шаблону проекта на CodeSandbox. Для его включения нужно кликнуть на кнопку «[Add dependency](#)» в левом, нижнем углу (рис. 3).

В открывшемся окне для подключения модулей, напишите `react-router-dom`.

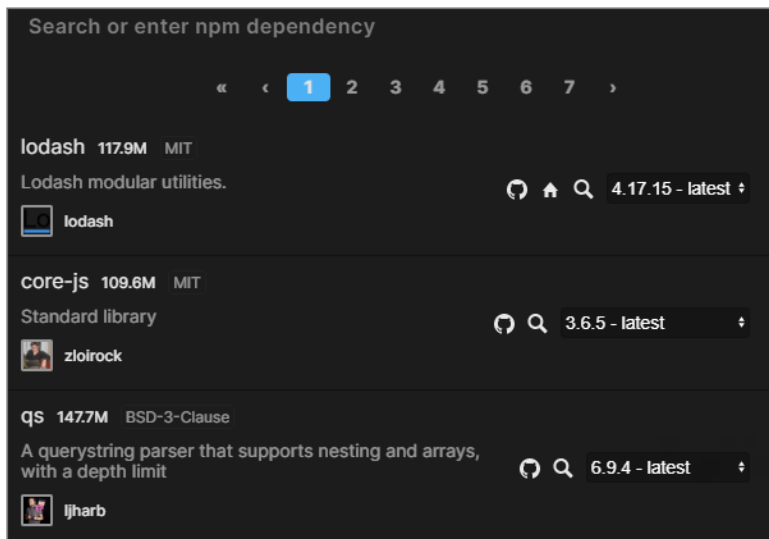


Рисунок 4

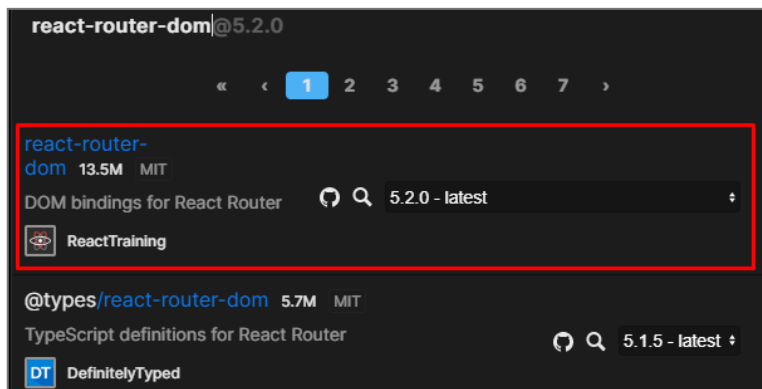


Рисунок 5

Кликните на первую ссылку для установки этого модуля в наш проект. Если всё прошло успешно `react-router-dom` появится в окне зависимостей нашего проекта.

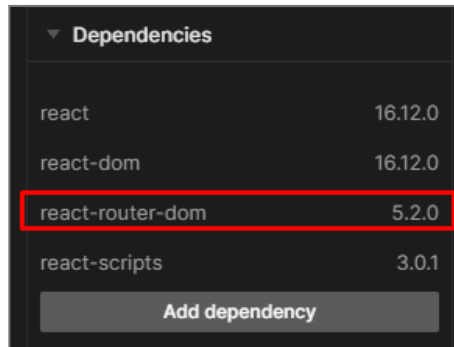


Рисунок 6

Первые шаги предварительной подготовки сделаны. Сейчас мы начнем анализировать, как встроить механизм маршрутизации в наши проекты.

В нашем первом примере мы настроим несколько маршрутов для посещения пользователями веб-приложения. У нас будет три маршрута. Первый будет вести на главную страницу, второй на страницу с информацией о компании, третий на страницу новостей. Если пользователь попытается зайти на неизвестный маршрут, мы покажем сообщение о том, что страница не найдена.

Посещение главной страницы:

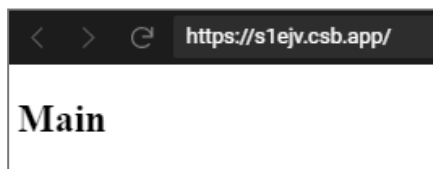


Рисунок 7

Обратите внимание, что главная страница открывается по корневому адресу сайта. Мы не указывали никакого дополнительного пути.

Посещение страницы «О компании»:

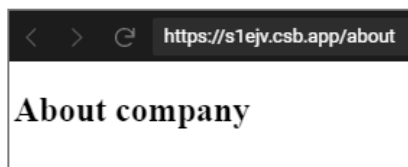


Рисунок 8

Для того, чтобы попасть на эту страницу мы добавили к пути [/about](#).

Посещение страницы новостей:

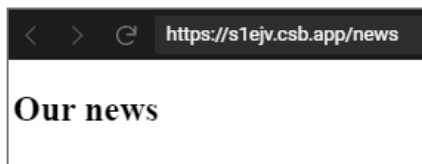


Рисунок 9

Для того, чтобы попасть на эту страницу мы добавили к пути [/news](#).

Анализ кода начнем с обзора кода в *App.js*:

```
import React from "react";
import { BrowserRouter as Router, Route, Switch }
from "react-router-dom";
import "./styles.css";

function Main() {
  return <h2>Main</h2>;
}
```

```

function AboutCompany() {
  return <h2>About company</h2>;
}
function News() {
  return <h2>Our news</h2>;
}
function NotFound() {
  return <h2>Not found</h2>;
}
export default function App() {
  return (
    <div>
      <Router>
        <Switch>
          <Route exact path="/" component={Main} />
          <Route path="/about"
            component={AboutCompany} />
          <Route path="/news" component={News} />
          <Route component={NotFound} />
        </Switch>
      </Router>
    </div>
  );
}

```

Для использования возможностей роутинга в коде, кроме добавления зависимости, нужно импортировать ряд объектов из [react-router-dom](#).

```

import { BrowserRouter as Router, Route, Switch }
  from "react-router-dom";

```

Мы указали, что импортируем **BrowserRouter** и дали ему псевдоним **Router**. Переименовывать было необязательно. Можно было использовать первоначальное имя.

BrowserRouter будет содержать в себе все маршруты. Его можно условно назвать агрегатором маршрутов. **Switch** помогает выбрать только один подходящий маршрут. **Route** — это конкретный маршрут. Рассмотрим код для создания маршрутов.

```
export default function App() {
  return (
    <div>
      <Router>
        <Switch>
          <Route exact path="/" component={Main} />
          <Route path="/about"
            component={AboutCompany} />
          <Route path="/news" component={News} />
          <Route component={NotFound} />
        </Switch>
      </Router>
    </div>
  );
}
```

Все маршруты заключены внутрь **Router**. Кроме того все маршруты заключены в **Switch**.

```
<Router>
  <Switch>
    Маршруты
  </Switch>
</Router>
```

У вас может возникнуть вопрос, можно ли убрать **Switch**? Можно. Мы покажем вам к чему это приведет чуть позже. Пока примите на веру, что так нужно в этом примере.

Конкретный маршрут описывается так:

```
<Route exact path="/" component={Main} />
```

Мы указали, что при обращении к корневому адресу нужно загрузить компонент `Main`. Для указания пути указывается атрибут `path`. Для указания имени компонента нужно использовать атрибут `component`. Об `exact` поговорим чуть позднее.

Код компонента `Main`:

```
function Main() {
  return <h2>Main</h2>;
}
```

В коде нет ничего непривычного для нас. Рассмотрим создание ещё одного маршрута:

```
<Route path="/about" component={AboutCompany} />
```

При обращении к адресу `корневой_адрес/about` нужно загрузить компонент `AboutCompany`.

Маршрут `NotFound` описывается схожим образом. Попробуем активизировать его:

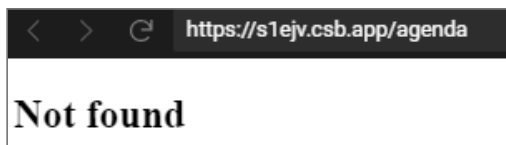


Рисунок 10

Мы попытались обратиться к неизвестному адресу. В ответ на неправильное обращение был активизирован компонент `NotFound`.

Зададимся вопросом, а что будет, если мы добавим к определенным нами маршрутам дополнительный адрес. Например, к пути `about` добавим название города: `корневой_адрес/about/london`:

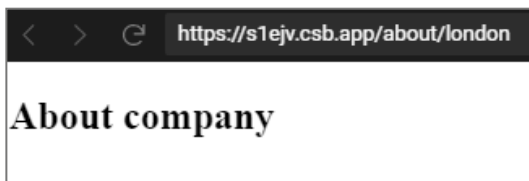


Рисунок 11

Несмотря на неточное совпадение маршрута происходит загрузка компонента `About`. Так происходит из-за того, что мы не указали `exact` при описании этого маршрута. Атрибут `exact` используется, когда нам нужно полное совпадение пути. Если мы добавим `exact` к описанию маршрута `About`, попытка обращения к неточному пути вызовет загрузку компоненты `NotFound` (путь не найден, мы просили точное совпадение):

```
<Route exact path="/about" component={AboutCompany} />
```

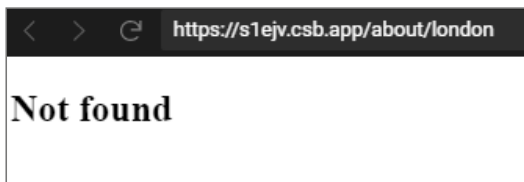


Рисунок 12

Попробуем убрать `exact` из описания корневого адреса:

```
<Route path="/" component={Main} />
```

Как вы думаете, что произойдет? Подумайте минуту. Правильный ответ: при попытке обращения к любому адресу будет загружен компонент **Main**, так как мы не требуем точного соответствия адреса, а путь к корневому адресу содержится в любом пути нашего приложения.

Остальные маршруты не будут вызываться никогда. Не забывайте указывать **exact** для вашего корневого адреса, чтобы не было таких удивительных последствий.

Теперь попробуем убрать **Switch** из нашего кода.

```
<Router>
  <Route exact path="/" component={Main} />
  <Route path="/about" component={AboutCompany} />
  <Route path="/news" component={News} />
  <Route component={NotFound} />
</Router>
```

Обратимся к корневому адресу:

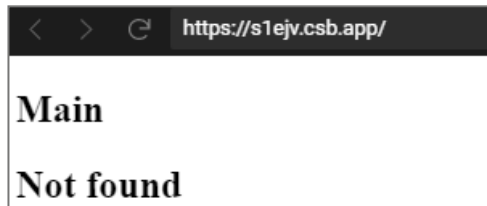


Рисунок 13

Обратимся к **About**:

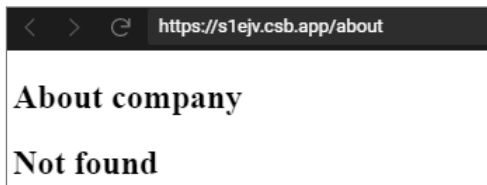


Рисунок 14

Почему, кроме правильной компоненты ещё подгружается компонента `NotFound`?

`Switch` отвечает за выбор первого подходящего маршрута, после его нахождения остальные маршруты не анализируются. `Switch` работает по принципу, схожему с конструкцией `switch` в любом языке программирования.

Сейчас в нашем коде нет `Switch`. Это значит, что отобразится не первый подходящий маршрут, а все, которые подходят под условие. Компонента `NotFound` не содержит атрибут `path`. Отсюда можно сделать вывод, что она подходит под любой путь. Именно поэтому у нас сначала отображается `About`, а потом подгружается `NotFound`.

Для того, чтобы избежать подобного эффекта, используйте `Switch`.

Ссылка на проект: <https://codesandbox.io/s/route1-s1ejv>.

Атрибут `children`

Мы можем описать компонент для конкретного маршрута при его определении. Для этого используется атрибут `children`. Немного изменим код нашего примера для его демонстрации. Код файла `App.js`:

```
import React from "react";
import { BrowserRouter as Router, Route, Switch }
    from "react-router-dom";

function Main() {
    return <h2>Main</h2>;
}

function AboutCompany() {
    return <h2>About company</h2>;
}
```

```
export default function App() {
  return (
    <div>
      <Router>
        <Switch>
          <Route exact path="/" component={Main} />
          <Route exact path="/about"
            component={AboutCompany} />
          <Route strict path="/news/"
            children={ () =>
              <h2>Our news</h2> } />
          <Route children={ () => <h2>Not found</h2>} />
        </Switch>
      </Router>
    </div>
  );
}
```

В коде описано несколько маршрутов. Однако, для маршрутов `news` и `NotFound` мы указали тело компонента в атрибуте `children`:

```
<Route children={ () => <h2>Not found</h2> } />
```

Также в коде появился новый атрибут `strict`. Он требует ещё более строгого совпадения пути. Это значит, что маршрут `news` должен быть `/`, так как мы указали `/` в `path`.

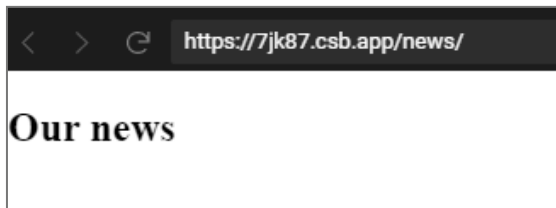


Рисунок 15

Так подходит. У нас точное соответствие `news/`:

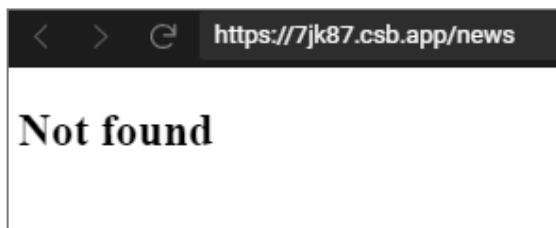


Рисунок 16

А так не подходит. В пути не указан «/».

Если бы на месте `strict` был `exact`, оба адреса вызывали загрузку компоненты `News`.

Ссылка на проект: <https://codesandbox.io/s/route2-7jk87>.

Дочерние пути

Модифицируем наш пример. Оставим главную компоненту, `AboutCompany`, `News`, `NotFound`. Внутри компоненты `News` настроим дочерние маршруты к новостям филиала в конкретном городе.

Внешний вид приложения при обращении к маршруту новостей:

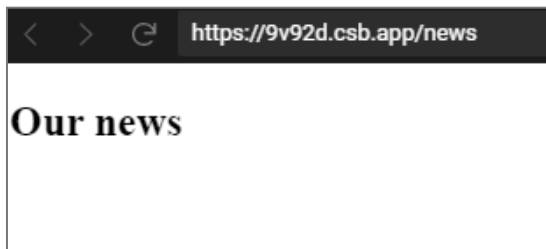


Рисунок 17

Внешний вид маршрута новостей конкретного города:

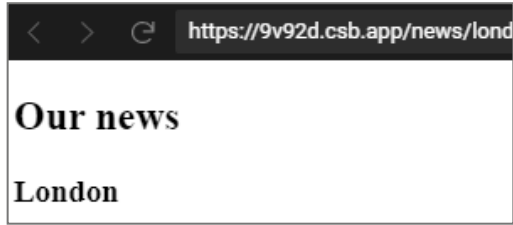


Рисунок 18

Рассмотрим код *App.js*:

```
import React from "react";
import { BrowserRouter as Router, Route, Switch }
    from "react-router-dom";
import "./styles.css";

function Main() {
    return <h2>Main</h2>;
}

function AboutCompany() {
    return <h2>About company</h2>;
}

function News() {
    return (
        <div>
            <h2>Our news</h2>
            <Switch>
                <Route path="/news/london"
                    component={London}/>;
                <Route path="/news/berlin"
                    component={Berlin}/>;
                <Route path="/news/paris"
                    component={Paris}/>;
            </Switch>
        </div>
    );
}
```

```

function NotFound() {
  return <h2>Not found</h2>;
}

function London() {
  return <h3>London</h3>;
}

function Paris() {
  return <h3>Paris</h3>;
}

function Berlin() {
  return <h3>Berlin</h3>;
}

export default function App() {
  return (
    <div>
      <Router>
        <Switch>
          <Route exact path="/" component={Main} />
          <Route path="/about"
            component={AboutCompany} />
          <Route path="/news" component={News} />
          <Route component={NotFound} />
        </Switch>
      </Router>
    </div>
  );
}

```

Маршрут новостей настроен уже знакомым нам образом:

```
<Route path="/news" component={News} />
```


А вот тело компоненты `News` реализовано немного по-другому:

```
function News() {  
  return (  
    <div>  
      <h2>Our news</h2>  
      <Switch>  
        <Route path="/news/london"  
              component={London} />;  
        <Route path="/news/berlin"  
              component={Berlin} />;  
        <Route path="/news/paris"  
              component={Paris} />;  
      </Switch>  
    </div>  
  );  
}
```

Мы указываем маршруты на новости конкретного города внутри компоненты `News`. Для каждого маршрута определена своя компонента. Например, для Лондона это компонента `London`.

Надеемся, что механизм дочерних маршрутов не вызовет у вас сложностей.

Ссылка на проект: <https://codesandbox.io/s/route3-9v92d>.

Создание меню навигации

Пока мы прокладывали маршруты исключительно в коде наших проектов. В новом примере мы добавим навигационное меню. С его помощью пользователь сможет активировать нужный ему маршрут.

Внешне наше приложение будет выглядеть так:



Рисунок 19

При клике на [About](#) откроется компонента [About-Company](#):



Рисунок 20

Хватит уже смотреть картинки, погружаемся в код:

```
import React from "react";
import {BrowserRouter as Router, Route, Switch, Link}
  from "react-router-dom";
import "./styles.css";

function Main() {
  return <h2>Main</h2>;
}

function AboutCompany() {
  return <h2>About company</h2>;
}
```

```

function News() {
  return <h2>Our news</h2>;
}

function NotFound() {
  return <h2>Not found</h2>;
}

function NavMenu() {
  return (
    <>
      <Link to="/" className="links">
        Main
      </Link>
      <Link to="/about" className="links">
        About
      </Link>
      <Link to="/news" className="links">
        News
      </Link>
    </>
  );
}

export default function App() {
  return (
    <div>
      <Router>
        <div>
          <NavMenu />
          <Switch>
            <Route exact path="/" component={Main} />
            <Route path="/about"
              component={AboutCompany} />
            <Route path="/news" component={News} />
            <Route component={NotFound} />
          </Switch>
        </div>
      </Router>
    </div>
  );
}

```

```

        </div>
      </Router>
    </div>
  );
}

```

Что же мы сделали для добавления меню навигации?

```

import {BrowserRouter as Router, Route, Switch, Link}
  from "react-router-dom";

```

Мы импортировали [Link](#). Он отвечает за создание ссылок. Для [Link](#) нужно указать атрибут `to`. Он используется для задания пути для ссылки. Для отображения меню ссылок мы создали компонент [NavMenu](#).

```

function NavMenu() {
  return (
    <>
      <Link to="/" className="links">
        Main
      </Link>
      <Link to="/about" className="links">
        About
      </Link>
      <Link to="/news" className="links">
        News
      </Link>
    </>
  );
}

```

В его теле мы описали набор ссылок.

```
<Link to="/" className="links">
  Main
</Link>
```

Описание конкретной ссылки. В данном случае ссылка ведет на корневую страницу. Для оформления мы указали название нашего пользовательского класса CSS. Его описание есть в [style.css](#) проекта.

```
.links {
  margin: 10px;
  text-decoration: none;
}
.links:hover {
  color: red;
}
```

Когда мы кликаем по конкретной ссылке происходит активизация маршрута по адресу в ссылке. Для отображения нашего функционального компонента [NavMenu](#) мы используем следующий код:

```
export default function App() {
  return (
    <div>
      <Router>
        <div>
          <NavMenu />
          <Switch>
            <Route exact path="/" component={Main} />
            <Route path="/about"
              component={AboutCompany} />
          </Switch>
        </div>
      </Router>
    </div>
  )
}
```

```

        <Route path="/news" component={News} />
        <Route component={NotFound} />
      </Switch>
    </div>
  </Router>
</div>
);
}

```

Мы добавили создание `NavMenu` в блок `Router` до `Switch`.

Ссылка на код проекта: <https://codesandbox.io/s/router4-p4we2>.

Передача параметров маршрута

В наших примерах мы ещё не использовали параметры. Когда вам могут понадобиться параметры в пути?

Например, если мы создаём веб-магазин, параметры в адресной строке помогут нам отображать информацию о конкретном товаре. Рассмотрим применение параметров на примере.

В приложении мы будем отображать страницу с информацией о филиалах, страницу конкретного филиала, новости конкретного филиала.

Внешний вид общей страницы о филиалах:

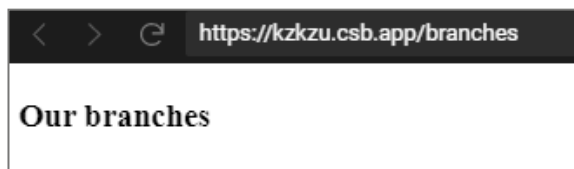


Рисунок 21

Мы не прорабатывали бизнес-логику происходящего. Поэтому у нас на странице только надпись **Our branches**. Страница конкретного филиала:

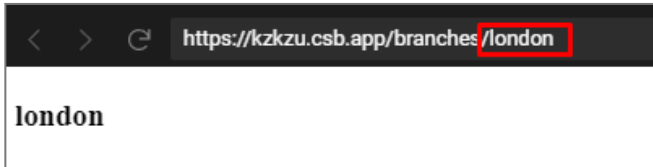


Рисунок 22

London в нашем пути это параметр, который мы подставили. На месте **London** может быть любое название города.

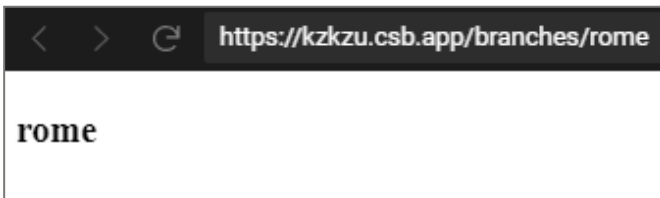


Рисунок 23

Страница конкретной новости конкретного филиала:

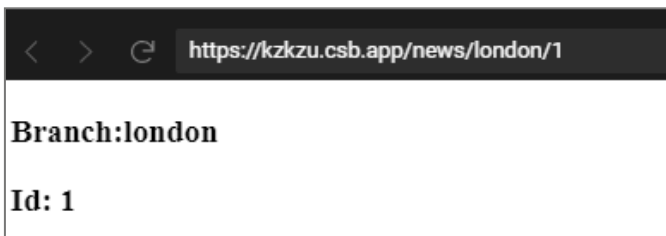


Рисунок 24

Если бы мы создали логику обработки такого запроса, нам потребовалось бы показать новость номер один

по филиалу в London. Название филиала и номер новости являются параметрами нашего маршрута.

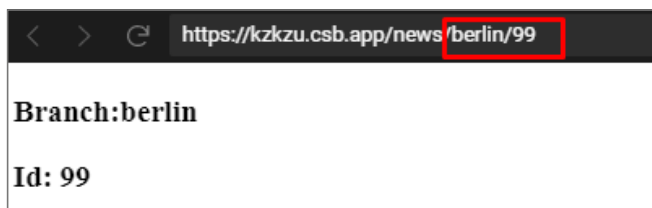


Рисунок 25

В данном случае параметр филиал равен **Berlin**, номер новости **99**.

Реализация в *App.js*:

```
import React from "react";
import { BrowserRouter as Router, Route, Switch }
  from "react-router-dom";

function Main() {
  return <h2>Main page</h2>;
}

function BranchList() {
  return <h3>Our branches</h3>;
}

/*
  Отображаем название филиала
*/
function Branch(props) {
  return <h3>{props.match.params.name}</h3>;
}

/*
  Проверяем два маршрута
  Если маршрут к branches отображаем список филиалов
*/
```



```

    Если маршрут к branches и содержит параметр,
    отображаем конкретный филиал
  */
function Branches() {
  return (
    <Switch>
      <Route exact path="/branches"
              component={BranchList} />
      <Route path="/branches/:name" component={Branch} />
    </Switch>
  );
}
/*
  Отображаем название филиала и id новости
*/
function News(props) {
  const branch = props.match.params.branch;
  const id = props.match.params.id;

  return (
    <div>
      <h3>Branch:{branch}</h3>
      <h3>Id: {id}</h3>
    </div>
  );
}

export default function App() {
  return (
    <>
      <Router>
        <Switch>
          <Route exact path="/" component={Main} />
          <Route path="/branches" component={Branches} />
          <Route path="/news/:branch/:id"
                  component={News} />
          <Route children={() => <h2>Not Found</h2>} />
        </Switch>
      </Router>
    </>
  );
}

```

```

    </Switch>
  </Router>
</>
);
}

```

В коде много уже знакомых вам конструкций. Пройдемся по новым аспектам.

```

<Route path="/news/:branch/:id" component={News} />

```

Для указания параметра в пути используется синтаксис `путь/:имя_параметра`.

В маршруте новостей два обязательных параметра: `branch` и `id`.

Когда мы говорим, что параметр обязательный это значит, что для активизации маршрута его надо обязательно передать, иначе путь будет не найден и отобразится компонента `NotFound`.

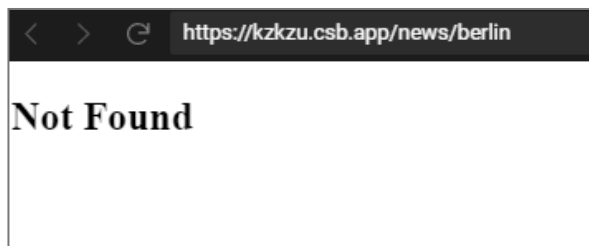


Рисунок 26

Мы не передали номер новости и маршрут не был найден.

Для получения доступа к параметрам маршрута в теле компонента используется `props.match.params`.

```
function News(props) {
  const branch = props.match.params.branch;
  const id = props.match.params.id;

  return (
    <div>
      <h3>Branch: {branch}</h3>
      <h3>Id: {id}</h3>
    </div>
  );
}
```

Для доступа к конкретному параметру мы должны указать его имя. Например, `props.match.params.branch`.

Код для маршрутов филиалов:

```
function Branches() {
  return (
    <Switch>
      <Route exact path="/branches"
              component={BranchList} />
      <Route path="/branches/:name"
              component={Branch} />
    </Switch>
  );
}
```

Если мы не указываем параметр в пути, будет загружен компонент `BranchList`. Если параметр был указан, загружается компонент `Branch`.

Внимательно проработайте этот пример для лучшего понимания.

Ссылка на проект: <https://codesandbox.io/s/route5-kzkzu>.

Необязательные параметры

Как вы могли догадаться, есть и не обязательные параметры. Это те параметры, которые можно не передать.

Добавим к нашему примеру маршрут [Management](#). Он будет отвечать за отображение менеджмента филиала. В этом маршруте мы будем использовать необязательные параметры: название филиала и фамилию менеджера.

UI нашего маршрута:

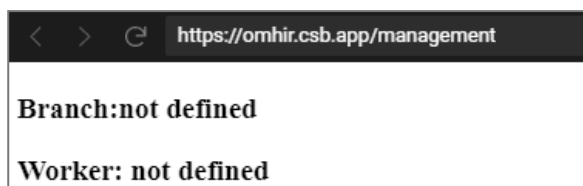


Рисунок 27

Ни один параметр не был указан:

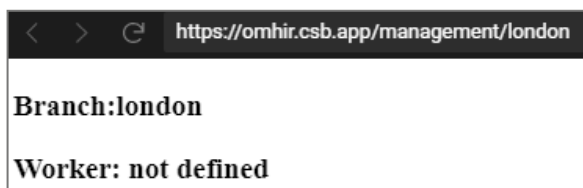


Рисунок 28

Название филиала указали, фамилию менеджера для отображения не указали:

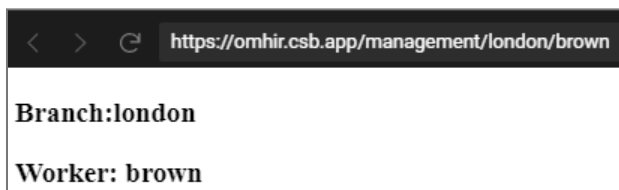


Рисунок 29

В пути мы указали название города и фамилию менеджера. В этот раз мы не будем приводить полностью код *App.js*. Разберем только новые места:

```
export default function App() {
  return (
    <>
      <Router>
        <Switch>
          <Route exact path="/" component={Main}/>
          <Route path="/branches" component={Branches}/>
          <Route path="/news/:branch-:id(\d+)"
            component={News} />
          <Route path="/management/:branch?/:worker?"
            component={Management} />
          <Route children={ () => <h2>Not Found</h2>} />
        </Switch>
      </Router>
    </>
  );
};
```

Мы создаём маршрут с необязательными параметрами, как и обычный маршрут внутри *Router*:

```
<Route path="/management/:branch?/:worker?"
  component={Management} />
```

Для указания, что параметр необязательный мы используем `?` после имени параметра. Например, `:branch?`

Код компоненты *Management*:

```
function Management(props) {
  let branch = "not defined";
  let worker = "not defined";
```

```

if (typeof props.match.params.branch !== "undefined")
  branch = props.match.params.branch;
if (typeof props.match.params.worker !== "undefined")
  worker = props.match.params.worker;
return (
  <div>
    <h3>Branch: {branch}</h3>
    <h3>Worker: {worker}</h3>
  </div>
);
}

```

Внутри мы проверяем определен ли параметр. Для этого мы используем `typeof`. Если он не определен, `typeof` вернет `undefined`.

Обратите внимание на ещё несколько новых приёмов в коде:

```

<Route path="/news/:branch-id(\d+)" component={News} />

```

При определении параметров вам необязательно указывать `/`. В коде выше мы использовали в качестве разделителя «-»

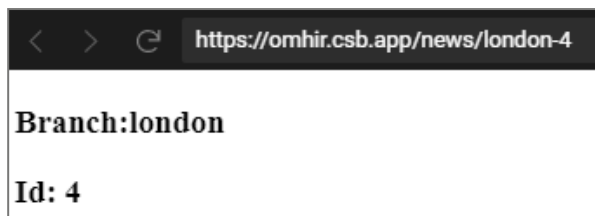


Рисунок 30

Тут видно, что теперь при передаче параметров мы указали `-`, так как он был указан в атрибуте `path`.

При описании параметра мы можем наложить ограничение на его содержимое. Для этого используются регулярные выражения, знакомые вам из курса JavaScript. В коде выше мы указали `:id(\d+)`. Это значит, что `id` может содержать только цифры. При попытке указать хотя бы одну букву, маршрут будет не найден:

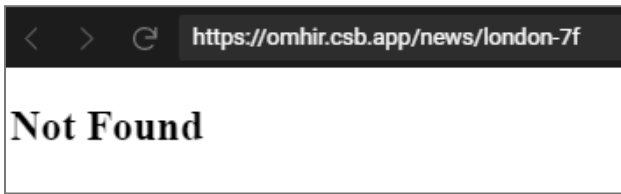


Рисунок 31

Ссылка на код проекта: <https://codesandbox.io/s/route6-omhir>.

Маршруты, ссылки, массивы

Создадим ещё один пример в котором объединим вместе изученные понятия. В приложении мы будем отображать список филиалов и при клике на имя филиала будем переходить на его страницу. В нашем приложении будет создан массив объектов филиалов.

Внешне приложение будет работать так:

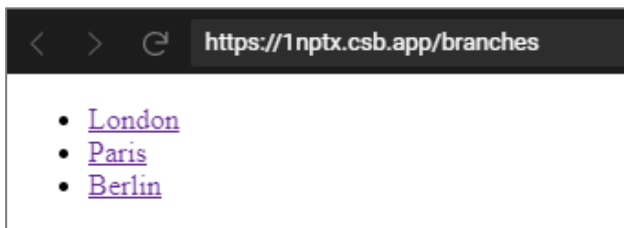


Рисунок 32

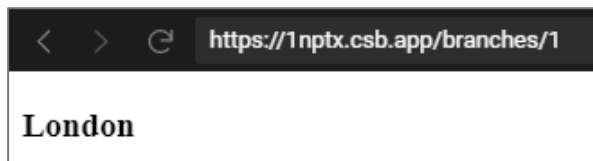


Рисунок 33

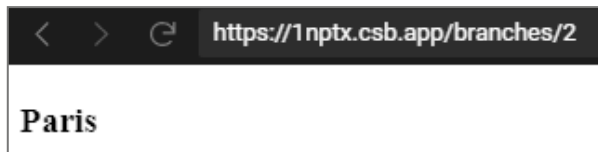


Рисунок 34

Код *App.js*:

```
import React from "react";
import {BrowserRouter as Router, Route, Switch, Link}
    from "react-router-dom";

/*
  Массив с данными для тестирования
*/
const branches = [
  { id: 1, name: "London" },
  { id: 2, name: "Paris" },
  { id: 3, name: "Berlin" }
];

function Main() {
  return <h2>Main page</h2>;
}

/*
  Отображаем список филиалов. Если кликнуть по ссылке
  откроется страница конкретного филиала
*/
```



```

function BranchList() {
  return (
    <ul>
      {branches.map(item => {
        return (
          <li key={item.id}>
            <Link to={'/branches/${item.id}'}>
              {item.name}</Link>
            </li>
          );
        })}
    </ul>
  );
}
/*
  Отображаем название филиала.
  Перед этим проверяем есть ли id филиала
  в нашем массиве объектов
*/
function Branch(props) {
  let branchId;
  let branch;
  branchId = parseInt(props.match.params.id, 10);

  for (let i = 0; i < branches.length; i++) {
    if (branches[i].id === branchId) {
      branch = branches[i];
      break;
    }
  }

  if (branch !== undefined) {
    return <h3>{branch.name}</h3>;
  } else {
    return <h3>Branch is not found!</h3>;
  }
}

```

```

/*
  Проверяем два маршрута
  Если маршрут к branches отображаем список филиалов
  Если маршрут к branches и содержит параметр,
  отображаем конкретный филиал
*/
function Branches() {
  return (
    <Switch>
      <Route exact path="/branches"
              component={BranchList}/>
      <Route path="/branches/:id"
              component={Branch} />
    </Switch>
  );
}
export default function App() {
  return (
    <>
      <Router>
        <Switch>
          <Route exact path="/" component={Main}/>
          <Route path="/branches" component={Branches}/>
          <Route children={() => <h2>Not Found</h2>}/>
        </Switch>
      </Router>
    </>
  );
}

```

В нашем коде определен массив объектов филиалов:

```

const branches = [
  { id: 1, name: "London" },
  { id: 2, name: "Paris" },
  { id: 3, name: "Berlin" }
];

```

Id нам понадобится для атрибута `key` при отображении конкретного элемента списка. Об атрибуте `key` говорили ранее.

Код компоненты `Branches`:

```
function Branches() {  
  return (  
    <Switch>  
      <Route exact path="/branches"  
              component={BranchList} />  
      <Route path="/branches/:id" component={Branch} />  
    </Switch>  
  );  
}
```

Если путь равен `/branches`, мы отображаем список филиалов. Если путь равен `/branches/:id`, мы отображаем информацию о конкретном филиале. Начнем с кода `BranchList`.

```
function BranchList() {  
  return (  
    <ul>  
      {branches.map(item => {  
        return (  
          <li key={item.id}>  
            <Link to={`/${branches}/${item.id}`}>  
              {item.name}</Link>  
            </li>  
          );  
        })}  
      </ul>  
    );  
}
```

С помощью уже известного вам метода `map` мы формируем список. Элементом списка является ссылка на конкретный город. Для создания ссылки мы используем `Link`

Теперь рассмотрим код `Branch`.

```
function Branch(props) {  
  let branchId;  
  let branch;  
  
  branchId = parseInt(props.match.params.id, 10);  
  
  for (let i = 0; i < branches.length; i++) {  
    if (branches[i].id === branchId) {  
      branch = branches[i];  
      break;  
    }  
  }  
  
  if (branch !== undefined) {  
    return <h3>{branch.name}</h3>;  
  } else {  
    return <h3>Branch is not found!</h3>;  
  }  
}
```

Мы проверяем есть ли полученный нами идентификатор в массиве объектов филиалов. Если есть, отображаем информацию о филиале, иначе сообщение об ошибке:

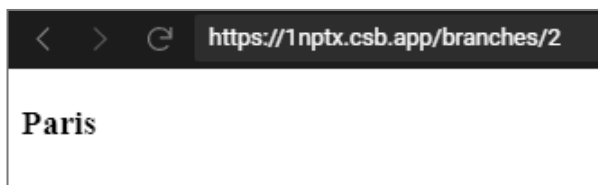


Рисунок 35

Ниже пример передачи неправильного `id`:

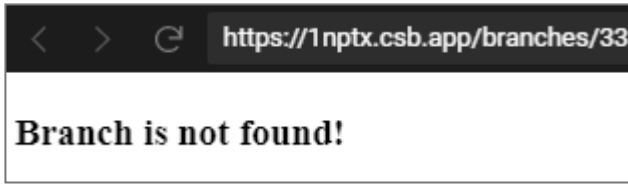


Рисунок 36

Ссылка на код проекта: <https://codesandbox.io/s/route7-1nptx>.

Домашнее задание

1. Используя механизм `routes` создайте приложение, посвященное известному художнику. Один маршрут должен вести на биографию художника, другой маршрут на его самую известную картину, третий маршрут на собрание его картин.
2. Добавьте к заданию 1 механизм ссылок, который позволит переходить с главной страницы по ссылкам на маршруты.
3. Добавьте к заданию 1 передачу параметров при переходе на маршрут.
4. Используя механизм `routes` создайте приложение, посвященное вашему городу. Один маршрут должен вести на информацию о городе, другой маршрут на его самую известную достопримечательность, третий маршрут на другие достопримечательности, четвертый на фотографии города.
5. Добавьте к заданию 4 механизм ссылок, который позволит переходить с главной страницы по ссылкам на маршруты.
6. Добавьте к заданию 4 передачу параметров при переходе на маршрут.

