top
КОМПЬЮТЕРНАЯ АКАДЕМИЯ

Создание веб-приложений с использованием Angular и React

Урок № 9

React: расширенные приемы

Содержание

Маршрутизация и строка запроса 3
Переадресация маршрутов 9
Жизненный цикл компоненты 20
Домашнее задание 31

Маршрутизация и строка запроса

Мы продолжаем наше знакомство с маршрутами. К уже изученным механизмам добавим работу со строкой запроса. Что такое строка запроса? Вы видели её много раз. Самый типичный пример — поисковая форма:

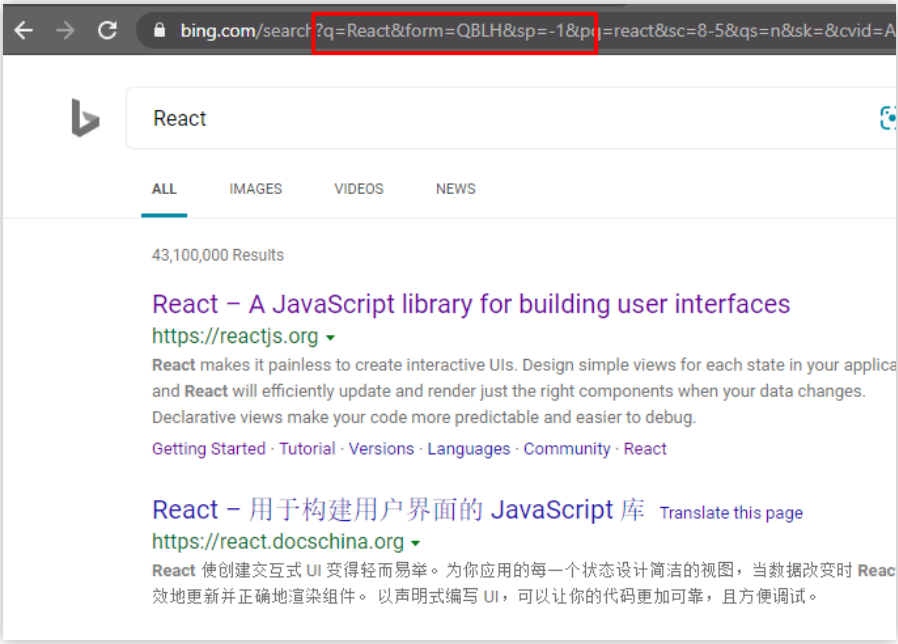

Рисунок 1

На странице результатов поиска в Bing можно увидеть строку запроса. Мы имеем дело с методом GET, поэтому нам видна строка запроса. В строке запроса указаны значения. Например, q=React, где q — имя параметра запроса,

a React значение. Строка запроса помогает передать набор значений. Эти значения можно использовать для разных целей. В нашем случае Bing с помощью строки запроса получает информацию о том, что мы ищем.

Как мы можем использовать строку запроса в наших приложениях? Например, с помощью неё можно передавать набор значений, которые будут применены для фильтрации информации.

Рассмотрим пример. Наше приложение будет отображать данные об автомобилях (рис. 2).

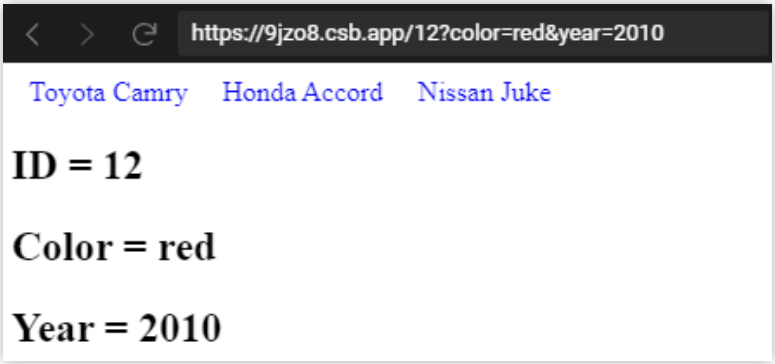

Рисунок 2

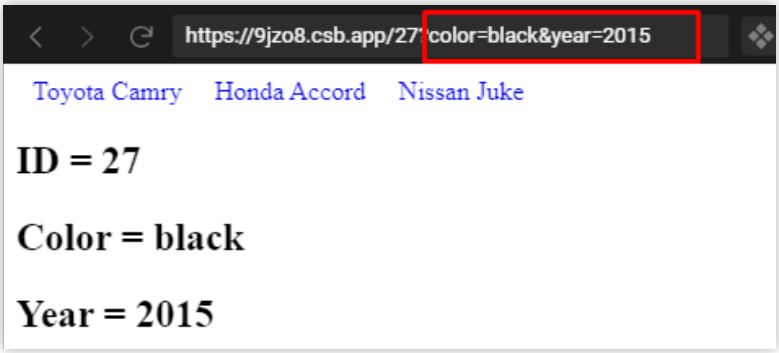

Рисунок 3

При клике на конкретный автомобиль отображаются его данные. В адресной строке отображаются параметры запроса (рис. 3).

Они идут после вопросительного знака. В нашем примере имена параметров запроса: color и year.

Код *App.js*:

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

function Main(props) {
  console.log(props.match);
  console.log(props.location);
  return (
    <div>
      <h2>ID = {props.match.params.id}</h2>
      <h2>Color = {
         new URLSearchParams(props.location.search).
                    get("color")}</h2>
      <h2>Year = {
         new URLSearchParams(props.location.search).
                    get("year")}</h2>
    </div>
  );
}

export default function App() {
  return (
    <>
      <Router>
        <nav>
          <Link to="/12?color=red&year=2010">
              Toyota Camry
          </Link>
```

```
            <Link to="/27?color=black&year=2015">
               Honda Accord
            </Link>
            <Link to="/27?color=yellow&year=2018">
               Nissan Juke
            </Link>
         </nav>

         <Switch>
            <Route path="/:id?" component={Main} />
         </Switch>
      </Router>
    </>
  );
}
```

Код нашего приложения относительно прост.

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

Мы используем возможности роутинга, поэтому нужно обязательно подключить react-router-dom и важные части из него.

```
<Router>
    <nav>
        <Link to="/12?color=red&year=2010">
            Toyota Camry
        </Link>
        <Link to="/27?color=black&year=2015">
            Honda Accord
        </Link>
```

```
        <Link to="/27?color=yellow&year=2018">
            Nissan Juke
        </Link>
    </nav>
    <Switch>
       <Route path="/:id?" component={Main} />
    </Switch>
</Router>
```

Блок навигации содержит новый формат пути. С помощью такого формата мы формируем строку запроса.

```
<Link to="/12?color=red&year=2010">Toyota Camry</Link>
```

12 — идентификатор автомобиля, color и year параметры строки запроса. Они должны быть обязательно разделены знаком &.

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

Маршрут содержит параметр id и ?.

Это позволяет нашему маршруту обрабатывать обращения, содержащие ?

```
function Main(props) {
   console.log(props.match);
   console.log(props.location);

   return (
      <div>
         <h2>ID = {props.match.params.id}</h2>
         <h2>Color = {
           new URLSearchParams(props.location.search).
                                get("color")}</h2>
```

```
          <h2>Year = {
            new URLSearchParams(props.location.search).
                                    get("year")}</h2>
        </div>
    );
}
```

В коде компоненты мы для наглядности отображаем в консоль содержимое match и location.

Параметры строки запроса содержатся внутри props.location.search.

Для получения значения параметра мы используем URLSearchParams.

```
new URLSearchParams(props.location.search).get("color")
```

Количество значений и имена в строке запроса зависят только от вашей фантазии и задач в рамках конкретного проекта.

▷ Код проекта можно посмотреть по ссылке.

Переадресация маршрутов

Как часто во время серфинга в интернете вы встречались с ситуацией, когда при заходе на одну страницу вы переадресовывались на другую? Причины такого поведения могут быть различны. Например, информация на искомой вами странице больше неактуальна или сайт переехал на новый адрес и т.д.

Для встраивания переадресации в React-приложение, применяется Redirect из react-router-dom.

Синтаксис использования:

```
<Redirect from="откуда" to="куда" />
```

В атрибуте from нужно указать первоначальный адрес, в атрибуте to путь для переадресации. Создадим приложение с переадресацией.

Внешний вид нашего приложения:

Рисунок 4

При клике на ссылку Archive приложение будет перенаправлено на News.

Код *App.js*:

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

import "./styles.css";

function Main() {
  return <h1>Main page</h1>;
}

/*
  Все обращения к Archive будут переведены на News
*/
function News() {
  return <h1>News page</h1>;
}

export default function App() {
  return (
    <div>
      <Router>
        <div>
          <nav>
            <Link to="/">Main</Link>
            <Link to="/news">News</Link>
            <Link to="/archive">Archive</Link>
          </nav>
        </div>
        <Switch>
          <Route exact path="/" component={Main} />
```

```
          <Route path="/news" component={News} />
          <Redirect from="/archive" to="/news" />
        </Switch>
      </Router>
    </div>
  );
}
```

Мы импортировали Redirect для использования. Раздел кода по маршрутизации:

```
<Router>
    <div>
        <nav>
            <Link to="/">Main</Link>
            <Link to="/news">News</Link>
            <Link to="/archive">Archive</Link>
        </nav>
    </div>

    <Switch>
        <Route exact path="/" component={Main} />
        <Route path="/news" component={News} />
        <Redirect from="/archive" to="/news" />
    </Switch>

</Router>
```

У нас есть две компоненты Main и News. Компоненту для ссылки Archive мы не создавали, так как используем переадресацию. Для неё мы вставили Redirect в описание маршрутов.

```
          <Redirect from="/archive" to="/news" />
```

При обращении к /archive будет выполнена переадресация на /news.

▹ Код проекта можно посмотреть по ссылке.

При переадресации может возникнуть проблема передачи данных. Для её решения используются уже знакомые вам механизмы. Добавим в наш уже существующий пример, передачу параметров. Внешний вид приложения:

Рисунок 5

При клике на News мы передаём идентификатор 56, при переадресации с Archive на News мы передадим 44:

Рисунок 6

Код *App.js*:

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

import "./styles.css";

function Main() {
   return <h1>Main page</h1>;
}

/*
  Все обращения к Archive будут переведены на News
*/
function News(props) {
   return (
      <>
         <h1>News page</h1>
         <h2>ID = {props.match.params.id}</h2>
      </>
   );
}

/*
  Перенаправляем на News и передаём полученный
  параметр
*/
function Archive(props) {
    return <Redirect to={`/news/${props.match.
                                  params.id}`} />;
}
```

```
export default function App() {
  return (
    <div>
      <Router>
        <div>
          <nav>
            <Link to="/">Main</Link>
            <Link to="/news/56">News</Link>
            <Link to="/archive/44">Archive</Link>
          </nav>
        </div>
        <Switch>
          <Route exact path="/" component={Main} />
          <Route path="/news/:id" component={News} />
          <Route path="/archive/:id"
                 component={Archive} />
          <Route children={() => <h1>404 page</h1>} />
        </Switch>
      </Router>
    </div>
  );
}
```

Блок кода для маршрутизации отличается от предыдущего примера.

```
<Router>
    <div>
        <nav>
          <Link to="/">Main</Link>
          <Link to="/news/56">News</Link>
          <Link to="/archive/44">Archive</Link>
        </nav>
    </div>
    <Switch>
        <Route exact path="/" component={Main} />
```

```
        <Route path="/news/:id" component={News} />
        <Route path="/archive/:id"
               component={Archive} />
        <Route children={() => <h1>404 page</h1>} />
    </Switch>
</Router>
```

При описании ссылок News и Archive мы указываем значения для параметра id (56 для News, 44 для Archive). Теперь в блоке маршрутизации мы не используем Redirect.

```
<Switch>
    <Route exact path="/" component={Main} />
    <Route path="/news/:id" component={News} />
    <Route path="/archive/:id"
           component={Archive} />
    <Route children={() => <h1>404 page</h1>} />
</Switch>
```

Вместо этого указываем компонент Archive. Переадресация будет проходить внутри этого компонента. Код компонента Archive:

```
function Archive(props) {
   return <Redirect to={`/news/${props.match.params.
                                  id}`} />;
}
```

При переадресации внутри пути мы указываем значение полученного параметра.

▷ Код проекта можно посмотреть по ссылке.

Добавим в наш механизм переадресации, проверку полученного идентификатора.

Если параметр равен 44, тогда мы будем переадресовывать на News. Если мы получим другое значение, переадресация будет выполнена на главную страницу.

Рисунок 7

Получен идентификатор 44.

Рисунок 8

Мы указали в адресной строке путь с идентификатором 99, после нажатия на Enter будет выполнена переадресация на главную страницу.

Рисунок 9

Код *App.js*:

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

import "./styles.css";

function Main() {
   return <h1>Main page</h1>;
}
/*
  Все обращения к Archive будут переведены на News
*/
function News(props) {
   return (
      <>
         <h1>News page</h1>
         <h2>ID = {props.match.params.id}</h2>
      </>
```

```
  );
}

/*
  Перенаправляем на News и передаём полученный
  параметр
*/
function Archive(props) {
  if (Number.parseInt(props.match.params.id, 10) ===
        44)
    return <Redirect to={`/news/${props.match.
                                  params.id}`} />;
    else return <Redirect to="/" />;
}

export default function App() {
  return (
    <div>
      <Router>
        <div>
          <nav>
            <Link to="/">Main</Link>
            <Link to="/news/56">News</Link>
            <Link to="/archive/44">Archive</Link>
          </nav>
        </div>
        <Switch>
          <Route exact path="/" component={Main} />
          <Route path="/news/:id" component={News} />
          <Route path="/archive/:id"
                 component={Archive} />
          <Route children={() => <h1>404 page</h1>} />
        </Switch>
      </Router>
    </div>
  );
}
```

Код компонента Archive в нашем примере устроен по-новому:

```
function Archive(props) {
  if (Number.parseInt(props.match.params.id, 10) === 44)
    return <Redirect to={`/news/${props.match.
                          params.id}`} />;
  else return <Redirect to="/" />;
}
```

Мы проверяем идентификатор. Если он равен 44, переадресация производится на News, иначе мы переходим на главную страницу.

▷ Код проекта можно посмотреть по ссылке.

Мы надеемся, что вы полностью поняли механизм маршрутизации. Однако, настоящее понимание приходит только во время выполнения практики. Поэкспериментируйте с новым аппаратом в ваших проектах.

Жизненный цикл компоненты

Мы многое уже знаем о компонентах. Мы умеем их создавать, передавать им параметры, работать с состоянием и многое другое. Сейчас мы поговорим о ещё одном важном аспекте компонентов — жизненном цикле. Понятие жизненного цикла возможно знакомо вам из курса школьной биологии.

Жизненный цикл компонента — это набор этапов, через которые проходит компонент во время своей жизни. Мы можем создать функции для обработки того или иного этапа. Обработка событий жизненного цикла отличается для классовых и функциональных компонент. Для начала рассмотрим классовые компоненты.

Список самых популярных функций для обработки событий жизненного цикла в классовых компонентах:

- constructor(props) — конструктор классового компонента. Он вызывается до того, как компонент прикреплен внутрь DOM. В нём может быть произведена инициализация состояния, прикрепление обработчиков событий и т.д.
- render() — используется для рендеринга компонента. Также вызывается, когда изменяются props и state.
- componentDidMount() — вызывается после прикрепления компонента. DOM уже существует. Внутри функции можно взаимодействовать с DOM, таймера-

ми, делать http-запросы, менять состояние, начинать работать с фреймворками, и т.д.

- componentDidUpdate() — вызывается после обновления DOM. Не вызывается в процессе начальной инициализации.
- componentWillUnmount() — вызывается перед удалением компоненты из DOM. Тут можно чистить полученные и выделенные ресурсы.

Это не все доступные функции для обработки состояния. Однако, нам хватит знакомства с ними для понимания концепции жизненного цикла. Об остальных функциях вы можете прочесть в официальной документации по React.

Рассмотрим пример, в котором мы увидим последовательность запуска той или иной функции. Внешний вид приложения представлен на рисунке 10.

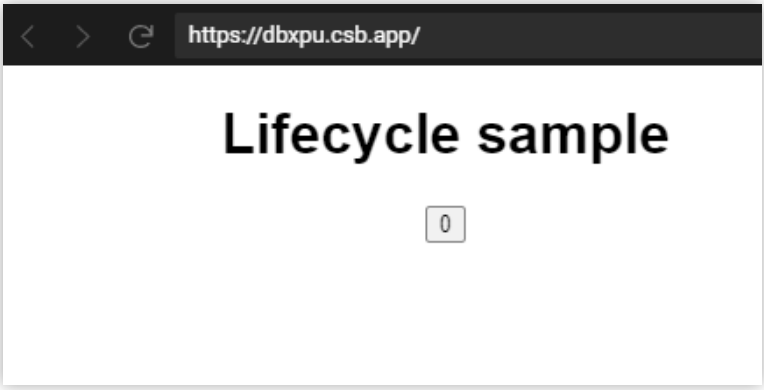

Рисунок 10

По нажатию на кнопку мы будем увеличивать значение счётчика (рис. 11-12).

Рисунок 11

Рисунок 12

В коде мы будем выводить информационные сообщения в консоль разработчика.

Код *App.js*:

```
import React from "react";

import "./styles.css";

export default class App extends React.Component {
  constructor(props) {
    super(props);
    console.log("Constructor");
    this.state = {
```

```
      counter: 0
    };
  }

  componentDidMount() {
    console.log("Component did mount");
  }

  componentDidUpdate() {
    console.log("Component did update");
  }

  componentWillUnmount() {
    console.log("Component will unmount");
  }

  render() {
    console.log("render");
    const clickHandler = () => {
      this.setState({ counter: this.state.counter + 1 });
    };

    return (
      <div className="App">
        <h1>Lifecycle sample</h1>
        <button onClick={clickHandler}>
                        {this.state.counter}
        </button>
      </div>
    );
  }
}
```

Для создания и обновления счётчика мы используем состояние. Самое интересное в этом примере для нас — очередность вызова функций жизненного цикла (рис. 13).

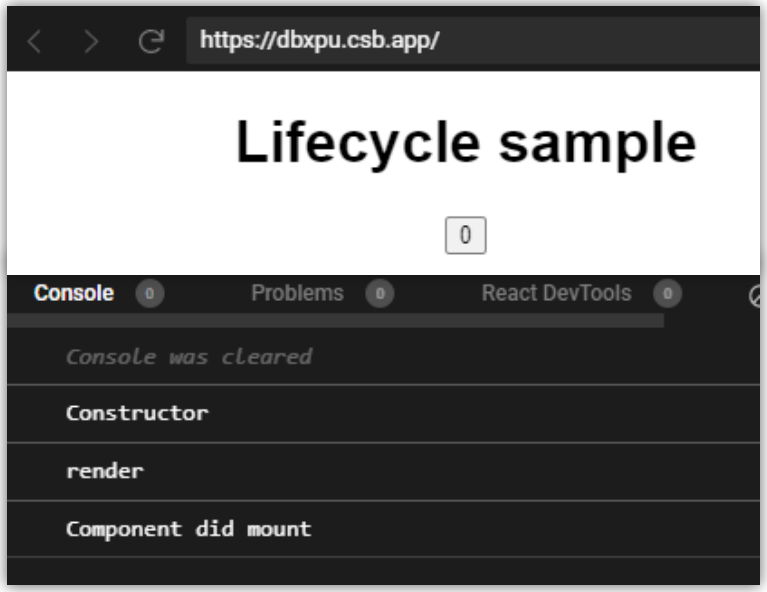

Рисунок 13

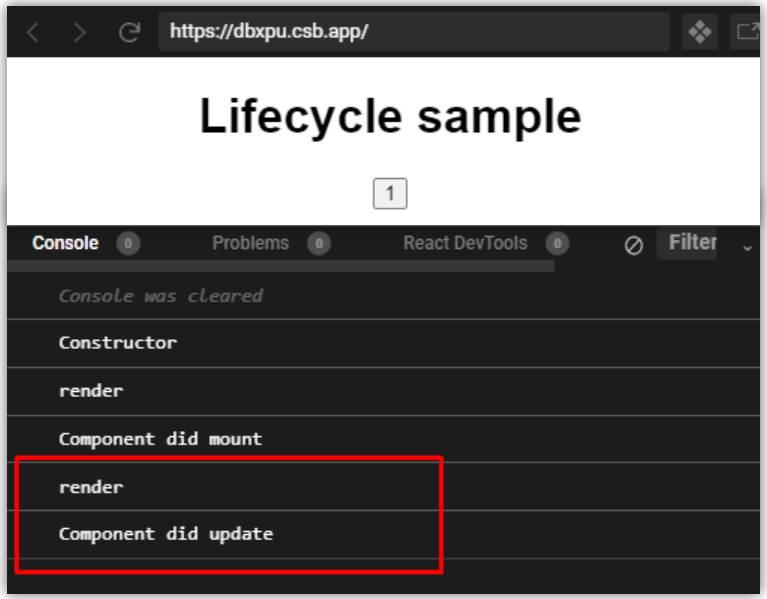

Рисунок 14

При старте приложения вызываются три функции: конструктор, render, componentDidMount. Нажмем на кнопку и посмотрим результат в консоли (рис. 14).

К уже полученному списку вызовов были добавлены два новых: render, componentDidUpdate. Функция render была вызвана, так как мы обновили состояние. Функция componentDidUpdate была вызвана, так как произошло обновление DOM. Если мы ещё раз нажмем на кнопку, будут снова вызваны эти же два метода.

▹ Код проекта можно посмотреть по ссылке.

Создадим новый проект для работы с жизненным циклом. В нём мы будем использовать возможности таймера. Наш код отображает текущее время. Обновление происходит каждую секунду (рис. 15).

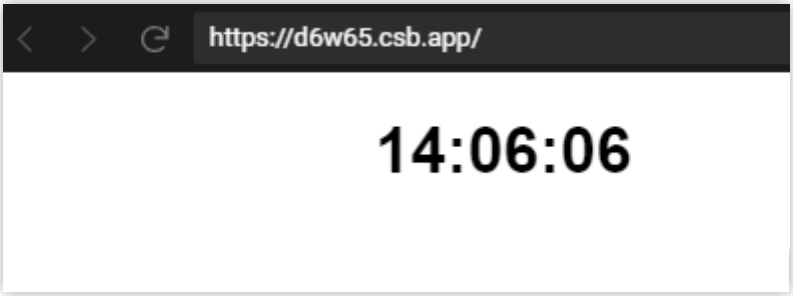

Рисунок 15

Зададимся вопросом: как это можно реализовать?

Время нужно хранить внутри состояния, чтобы его изменение приводило к перерисовке DOM. Какие функции нам понадобятся?

1. Конструктор для инициализации состояния. При желании можно инициализировать состояние с помо-

щью альтернативного синтаксиса без использования конструктора. Мы показывали этот способ в одном из примеров прошлых уроков.

2. Функция render для перерисовки компоненты. Она будет автоматически вызываться при изменении состояния.
3. Функция componentDidMount. В ней мы создадим таймер.
4. Функция componentWillUnmount. В ней мы очистим ресурсы, выделенные под таймер.

Код *App.js*:

```
import React from "react";
import "./styles.css";
export default class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      currDate: new Date()
    };
  }

  /*
    Ставим таймер. Он будет срабатывать каждую секунду
    Каждую секунду будет вызываться функция timerAction
  */
  componentDidMount() {
    this.handlerOfTimer = setInterval(() =>
          this.timerAction(), 1000);
  }

  /*
    Обработчик таймера. Изменяем состояние
  */
```

```
  timerAction() {
    this.setState({ currDate: new Date() });
  }

  componentWillUnmount() {
    clearInterval(this.handlerOfTimer);
  }

  render() {
    return (
      <div className="App">
        <h1>
            {this.state.currDate.toLocaleTimeString()}
        </h1>
      </div>
    );
  }
}
```

Мы инициализируем состояние внутри конструктора:

```
constructor(props) {
   super(props);
   this.state = {
      currDate: new Date()
   };
}
```

Мы можем настраивать таймер, когда DOM уже создан и готов к работе. Именно поэтому нам приходится использовать componentDidMount:

```
componentDidMount() {
    this.handlerOfTimer = setInterval(() =>
         this.timerAction(), 1000);
  }
```

В коде мы устанавливаем таймер на каждую секунду. При тике будет вызываться timerAction. Кроме того, мы сохранили дескриптор таймера, так как нам нужно будет очистить ресурсы таймера по завершении приложения.

```
timerAction() {
    this.setState({ currDate: new Date() });
}
```

Обработчик таймера изменяет состояние. После изменения состояния React автоматически вызывает функцию render.

```
componentWillUnmount() {
    clearInterval(this.handlerOfTimer);
}
```

Мы должны освободить ресурсы таймера. Для этого мы используем componentWillUnmount.

▷ Код проекта можно посмотреть по ссылке.

Перейдем к функциональным компонентам. Для встраивания в жизненный цикл компонента используется хук useEffect, позволяющий реагировать на componentDidMount и componentDidUpdate для функционального компонента.

Синтаксис использования:

```
useEffect(обработчик)
```

Рассмотрим использование хука на примере. Создадим приложение, отображающее значение счетчика в документе, а также в заголовке окна браузера.

Внешний вид приложения:

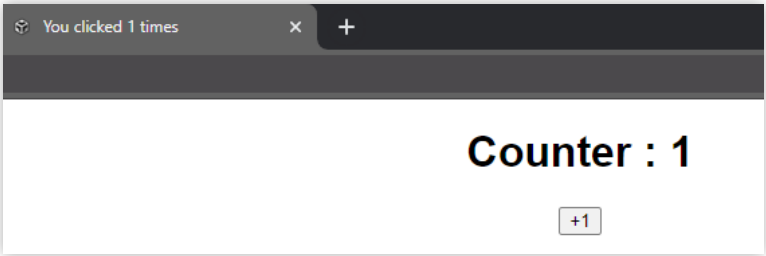

Рисунок 5

Код *App.js*:

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

export default function App() {
  const [counter, setCounter] = useState(0);
  useEffect(() => {
    document.title = `You clicked ${counter} times`;
  });

  return (
    <div className="App">
      <h1>Counter : {counter}</h1>
      <button onClick={() => setCounter(counter + 1)}>+1
      </button>
    </div>
  );
}
```

Для нас главный код это вызов useEffect.

```
useEffect(() => {
    document.title = `You clicked ${counter} times`;
});
```

Внутри вызова useEffect мы создали обработчик для componentDidMount и componentDidUpdate. Каждый раз, когда будет наступать время для этих функций наш код будет вызываться. Внутри кода мы отображаем в заголовке окна текущее значение переменной состояния. И как обычно код функционального компонента гораздо проще, чем код для классового компонента.

▷ Код проекта можно посмотреть по ссылке.

В нашем коде сейчас есть одна проблема. Обновление любой переменной состояния вызовет код обработчика, указанного в useEffect. Для того, чтобы этого не было нужно изменить вызов useEffect.

```
useEffect(() => {
    document.title = `You clicked ${counter} times`;
  },[counter]);
```

Во втором параметре useEffect мы указали массив переменных состояния, чьё изменение должно приводить к вызову кода в первом параметре. В нашем случае массив содержит только одну переменную counter.

Домашнее задание

1. Создайте приложение, отображающее текущее время. Обязательно используйте функциональные компоненты.
2. Создайте приложение «Таймер». Пользователь вводит количество секунд и нажимает кнопку «Старт». Сразу после нажатия начинается обратный отсчет. Пользователю также доступны кнопки «Стоп» и «Пауза».
3. Создайте приложение для транслитерации текста. Пользователь вводит текст в текстовое поле. В другом текстовом поле отображается транслитерированный текст.