top
КОМПЬЮТЕРНАЯ АКАДЕМИЯ

Создание веб-приложений с использованием Angular и React

Урок № 7

React: расширенные приемы

Содержание

Возвращение в JavaScript

Для изучения нового материала по React требуется немного вспомнить JavaScript. Нам понадобится метод map, который возможно вам уже знаком. Его используют для трансформации массивов. Упрощенный синтаксис этого метода:

```
имя_массива.map(функция)
```

Принцип работы метода:

1. Для каждого элемента массива, который вызвал map, вызывается функция, указанная в качестве параметра. Например, если в массиве три элемента, функция будет вызвана трижды.
2. map возвращает новый массив. Элементами этого массива будут являться значения, полученные после воздействия функции-параметра на каждое значение из оригинального массива.
3. Оригинальный массив в результате вызова map не изменяется.
4. Функция-параметр вызывается только для элементов оригинального массива со значениями.

Рассмотрим пример использования. В нём для каждого элемента массива будет вызвана функция подсчёта квадратного корня.

```
/*
  Оригинальный массив
*/
var arr = [9, 4, 16];

/*
  9 4 6
*/
console.log(arr);

/*
  Для каждого элемента массива arr вызывается функция
  Math.sqrt
  Из map возвращается новый массив элементами,
  которого являются элементы arr, после вызова
  Math.sqrt для каждого из них
  Оригинальный массив не изменяется
*/
var mArr = arr.map(Math.sqrt);
/*
  3 2 4
*/
console.log(mArr);
```

При вызове map для каждого элемента arr была вызвана функция подсчёта квадратного корня. Из map вернулся новый массив с значениями из arr после подсчёта квадратного корня. Именно, поэтому в mArr содержатся элементы со значениями 3 2 4.

В этом примере мы использовали стандартную функцию sqrt, а если нам необходимо вызвать пользовательскую функцию? Как это можно сделать? Для решения этой задачи можно использовать следующий синтаксис map:

```
имя_массива.map(пользовательская_функция(текущий элемент)
{
    тело
}
)
```

Мы передадим собственную функцию в вызов map. Она будет вызвана для каждого элемента массива. В качестве параметра наша функция будет принимать текущий, анализируемый элемент массива. Этот параметр будет заполняться автоматически.

В нашем следующем примере, пользовательская функция будет возводить в квадрат каждый элемент массива.

```
/*
  Оригинальный массив
*/
var arr = [3, 4, 7];

/*
  3 4 7
*/
console.log(arr);

/*
  Для каждого элемента массива arr вызывается
  функция Math.pow
  Из map возвращается новый массив элементами,
  которого являются элементы arr, после
  вызова Math.pow для каждого из них
  Оригинальный массив не изменяется
*/
var mArr = arr.map(function(item) {
  return Math.pow(item, 2);
});
```

```
/*
  9 16 49
*/
console.log(mArr);
```

У нашей пользовательской функции нет названия. Хотя ничто не мешало нам, создать функцию и передать её имя в качестве параметра. Для каждого элемента массива arr будет вызван код функции. В нашем случае это Math.pow(item,2).

Обратите внимание, что пользовательская функция должна возвращать значение.

Изменим этот пример. Мы воспользуемся более новым синтаксисом для создания функций:

```
/*
  Оригинальный массив
*/
var arr = [3, 4, 7];

/*
  3 4 7
*/
console.log(arr);

/*
  Для каждого элемента массива arr
  вызывается функция Math.pow
  Из map возвращается новый массив элементами,
  которого являются элементы arr, после
  вызова Math.pow для каждого из них
  Оригинальный массив не изменяется
*/
var mArr = arr.map((item) => {
```

```
  return Math.pow(item, 2);
});
/*
  9 16 49
*/
console.log(mArr);
```

Зачем писать function, если его можно пропустить ☺. Тело функции не изменилось.

Мы можем не писать return, так как в теле нашей функции одна строка. Новый код примера:

```
/*
  Оригинальный массив
*/
var arr = [3, 4, 7];
/*
  3 4 7
*/
console.log(arr);

var mArr = arr.map(item => Math.pow(item, 2));
/*
  9 16 49
*/
console.log(mArr);
```

В коде примера мы не указываем (), {}, return. Мы не указываем (), так как у нашей функции один параметр, а это значит, что () можно упустить. Мы не указываем {}, так как у нашей функции одна строка, а это значит, что {} можно упустить. Ключевое слово return будет подставлено автоматически при этой форме синтаксиса.

Рассмотрим ещё один пример. В нём мы будем оперировать массивом объектов. Наш объект будет содержать информацию о фамилии и возрасте. Пользовательская функция будет увеличивать значение возраста на 10 лет.

```
var users = [{ lastName: "Brown", age: 30 },
{ lastName: "Davids", age: 40 }];

/*
  В нулевом элементе возраст будет равен 40
  В первом элементе 50
*/
var mUsers = users.map(item => {
  item.age += 10;
  return item;
});
```

Массив users содержит информацию о пользователях. Каждый элемент массива — это объект. Пользовательская функция принимает в качестве параметра элемент массива (это конкретный объект). Она будет вызвана для каждого объекта из массива users. Внутри функции мы увеличиваем возраст текущего элемента на 10 лет. Измененный элемент возвращается из функции. В результате работы map будет возвращен новый массив с объектами внутри. У каждого объекта величина возраста будет увеличена на 10 лет по сравнению с соответствующим элементом внутри users. Оригинальный массив users не изменится.

Использование массивов в React-приложениях

Мы пока не работали с массивами данных в наших React-приложениях. Теперь настал момент, когда мы будем использовать массив данных и отображать значения из него на экран. Когда это может пригодиться? Например, мы можем получить массив данных после запроса на веб-сервис.

Перед погружением в новую тему вспомним о деструктуризации объектов. Мы работали с ней в прошлом уроке. Рассмотрим код:

```
let user = {name:"Dan",lastName:"Brown"};
let {...concreteU} = user;

console.log(concreteU);
```

В этом коде мы создали объект user. Внутри у него два поля: name и lastName. С помощью синтаксиса деструктуризации {…concreteU} мы создаём новый объект concreteU. В него будет скопировано содержимое и устройство объекта user. Это значит, что внутри concreteU будет два поля: name и lastName. В name будет Dan. В lastName будет Brown. Этот синтаксис вы ещё увидите в этом уроке.

Начнем с базового примера по использованию массива. Отобразим элементы массива в приложении. В коде ниже у нас есть массив объектов с информацией о писателях (имя и фамилия). Мы будем показывать эту информацию на экран. Внешний вид нашего приложения:

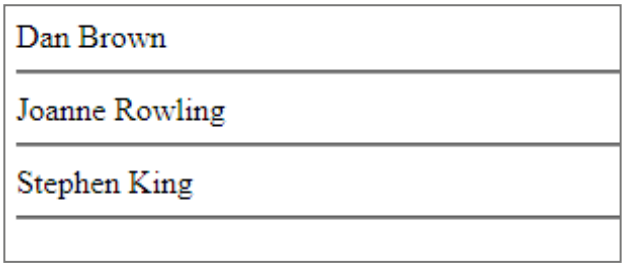

Рисунок 1

Для решения этой задачи создадим несколько дополнительных функциональных компонент. Первая компонента будет содержать информацию об одном писателе. Мы назовём её Writer. Вторая компонента будет показывать список всех писателей с информацией о них. Её имя будет WritersList.

Код файла *App.js*:

```
import React from "react";
import "./styles.css";
/*
  Массив объектов
*/
let writers = [
  { name: "Dan", lastName: "Brown" },
  { name: "Joanne", lastName: "Rowling" },
  { name: "Stephen", lastName: "King" }
];

/*
  Компонента с информацией о писателе
*/
function Writer(props) {
  return (
    <>
```

```
      <div>
        {props.name} {props.lastName}
      </div>
      <hr />
    </>
  );
}

/*
  Список писателей
*/
function WritersList() {
  return (
    <div>
      <Writer {...writers[0]} />
      <Writer {...writers[1]} />
      <Writer {...writers[2]} />
    </div>
  );
}

export default function App() {
  return (
    <>
      <WritersList />
    </>
  );
}
```

В нашем коде три компонента и один массив. Начнем анализ примера с массива.

```
/*
  Массив объектов
*/
let writers = [
```

```
  { name: "Dan", lastName: "Brown" },
  { name: "Joanne", lastName: "Rowling" },
  { name: "Stephen", lastName: "King" }

];
```

В массиве объектов writers содержится информация о писателях. Каждый элемент — это отдельный писатель. Массив writers объявлен на глобальном уровне в нашем коде. Это сделано для упрощения кода примера.

Компонент Writer отвечает за отображение данных писателя.

```
/*
  Компонента с информацией о писателе
*/

function Writer(props) {
  return (
    <>
      <div>
        {props.name} {props.lastName}
      </div>
      <hr/>
    </>
  );

}
```

Информация о писателе будет передана внутрь компоненты через props.

Компонента WritersList будет отображать массив писателей. Это означает, что внутри её кода мы будем

создавать объект конкретного писателя (объект компоненты Writer).

```
/*
  Список писателей
*/

function WritersList() {
  return (
    <div>
      <Writer {...writers[0]} />
      <Writer {...writers[1]} />
      <Writer {...writers[2]} />
    </div>

  );
```

Главная новинка кода компоненты WritersList это конструкция вида:

```
      <Writer {...writers[0]} />
```

Мы создаём объект Writer. В качестве его props мы передаём ему объект writers[0]. Для этого использован синтаксис деструктуризации. Внутрь props будем скопировано полностью содержимое writers[0]. Это значит, что внутри props будут созданы свойства name и lastName. Значения в них будут скопированы из соответствующих полей writers[0].

В нашем массиве три элемента, поэтому мы повторяем создание компоненты Writer трижды в коде с разными индексами (0,1,2).

Код компоненты App привычен для нас:

```
export default function App() {
  return (
    <>
      <WritersList />
    </>
  );
}
```

Мы создали объект списка писателей и ничего более.

Ссылка на код проекта: https://codesandbox.io/s/array-lfz9o.

Код нашего примера по работе с массивами содержит небольшой изъян. Каждый элемент массива мы указываем явно в WritersList. В коде выше мы делали это три раза. Это неудобно и не масштабируемо. Исправим этот недостаток в новой версии примера. Интерфейс приложения у нас не изменится. Все изменения произойдут в логике отображения.

Код файла *App.js*:

```
import React from "react";
import "./styles.css";

var writers = [
  { name: "Dan", lastName: "Brown" },
  { name: "Joanne", lastName: "Rowling" },
  { name: "Stephen", lastName: "King" }
];

function Writer(props) {
  return (
    <>
      <div>
```

```
        {props.name} {props.lastName}
      </div>
      <hr />
    </>
  );
}

function WritersList(props) {
  return (
    <div>
      {props.data.map(item => <Writer {...item} />)}
    </div>
  );
}

export default function App() {
  return (
    <>
      <WritersList data={writers} />
    </>
  );

}
```

В нашем коде две новинки. Первая внутри кода компоненты App:

```
export default function App() {
  return (
    <>
      <WritersList data={writers} />
    </>
  );

}
```

Мы передаём массив writers через свойство data внутрь компоненты WritersList.

Также у нас изменения внутри WritersList.

```
function WritersList(props) {
  return (
    <div>
      {props.data.map(item => <Writer {...item} />)}
    </div>
  );
}
```

У нас уже нет кода с каждым элементом массива писателей. Вместо него мы используем уже знакомый нам метод map. Его вызов возвращает массив компонент Writer.

```
{props.data.map(item => <Writer {...item} />)}
```

Этот синтаксис может смотреться зловеще, но с каждой из его частей, мы уже встречались раньше. Рассмотрим по частям:

```
props.data.map(
```

Вызываем метод map для свойства data (ссылается на массив writers)

```
item => <Writer {...item} />
```

Для каждого элемента массива writers вызывается этот код. В этом коде возвращается элемент для нового массива. Фактически, мы создаём новый элемент Writer и заполняем его props значениями свойств текущего, анализируемого элемента.

На место вызова map будет вставлен новый, созданный массив объектов типа Writer.

Благодаря map мы избавились от необходимости указывать каждый элемент вручную.

Ссылка на проект: https://codesandbox.io/s/array2-5eygm.

Ключ

Возможно, вы заметили, что с нашим примером по работе с массивами данных есть небольшая проблема ☺.

Если не заметили, сейчас мы вам покажем. В онлайн-редакторе CodeSandbox есть блок интерфейса, который отображает консоль. Если открыть наш пример с массивом в нижнем правом углу можно увидеть нечто подобное:

Рисунок 2

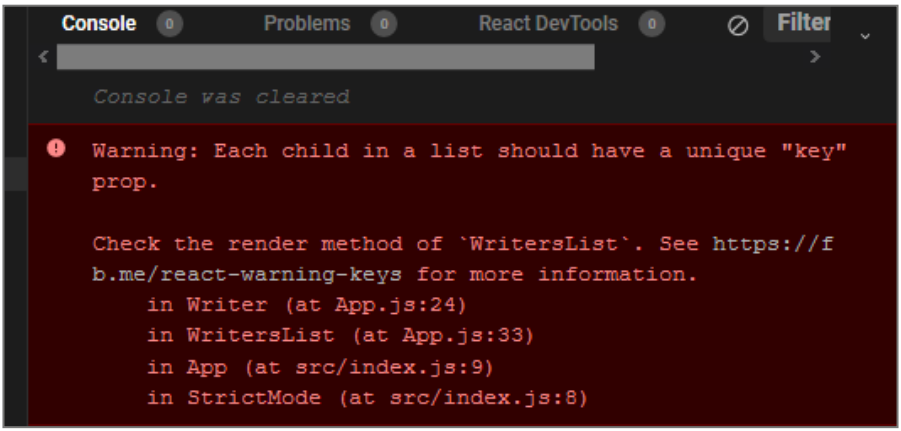

Рисунок 3

Красная единица говорит о том, что у нас есть какая-то одна проблема. Открываем консоль и видим следующее сообщение (рис. 3). О чём нам оно говорит? React предупреждает нас, что у каждого элемента в списке должен быть свой уникальный ключ. Мы задавали значение для элемента. Значения для ключей мы не задавали.

Что такое ключ? Это некоторое уникальное значение, которое не будет изменяться. React использует ключи, чтобы понимать какие элементы были добавлены, удалены, изменены. Если вы хотите быть уверенными в правильном обновлении интерфейса, вам нужно обязательно задавать ключи.

Уникальность ключей должна поддерживаться в рамках одного списка. Это значит, что ключи в разных списках могут быть не уникальны. Для задания ключа используется атрибут key.

Рассмотрим пример по работе с ключами. В нашем примере будет массив городов, который мы будем отображать. У каждого элемента списка городов будет задан свой уникальный ключ:

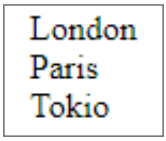

Рисунок 4

Код *App.js*:

```
import React from "react";
import "./styles.css";
const cities = ["London", "Paris", "Tokio"];
```

```
function City(props) {
  return <div>{props.value}</div>;
}


function CityList(props) {
  const citiesData = props.data.map(
        city => <City key={city} value={city} />
  );

  return <>{citiesData}</>;
}


export default function App() {
  return (
    <>
      <CityList data={cities} />
    </>
  );

}
```

Принцип работы этого примера схож с предыдущими. У нас есть компоненты для города и списка городов. Начнем анализ с массива городов cities и компоненты City.

```
const cities = [“London”, “Paris”, “Tokio”];

function City(props) {
  return <div>{props.value}</div>;


}
```

У нас объявлен массив названий городов. Компонент City отображает название конкретного города. Код компоненты CityList:

```
function CityList(props) {
  const citiesData = props.data.map(
        city => <City key={city} value={city} />
  );

  return <>{citiesData}</>;

}
```

В этом примере мы вынесли вызов map из JSX. Сделано это было для демонстрации другого, возможного подхода. Функция-параметр для map выглядит немного по-другому:

```
city => <City key={city} value={city} />
```

Мы говорим, что элемент списка городов имеет два свойства. Первое свойство value — название города. Второе свойство key — ключ для текущей, создаваемой строки. Ключ не отображается в интерфейсе. Пользователь его не видит. Этот ключ нужен для React. В этом примере мы в качестве ключа использовали название города. Если название города в массиве будет встречаться более одного раза, то тогда наш подход перестанет работать.

Для того, чтобы избежать потенциальных проблем в качестве ключа используют идентификаторы. Этот приём мы покажем в другом примере.

Мы не используем синтаксис деструктуризации, так как у нас массив строк, а не массив объектов.

И код компоненты *App*

```
export default function App() {
  return (
    <>
      <CityList data={cities} />
    </>
  );
}
```

Мы передали массив через свойство data.

Ссылка на проект: https://codesandbox.io/s/cities-7zpf6.

Очень важно запомнить, что ключ надо передавать в том месте, где создаётся список городов, а не внутри компонента City. Код ниже не является верным для задания ключа:

```
function City(props) {
  return <div key={props.value}>{props.value}</div>;
}

function CityList(props) {
  const citiesData = props.data.map(city =>
        <City value={city} />);
  return <>{citiesData}</>;
}
```

Мы задали атрибут key внутри компоненты City в теге div. Такой подход не позволяет React идентифицировать строки списка. Об этом сообщит вам предупреждение в консоли:

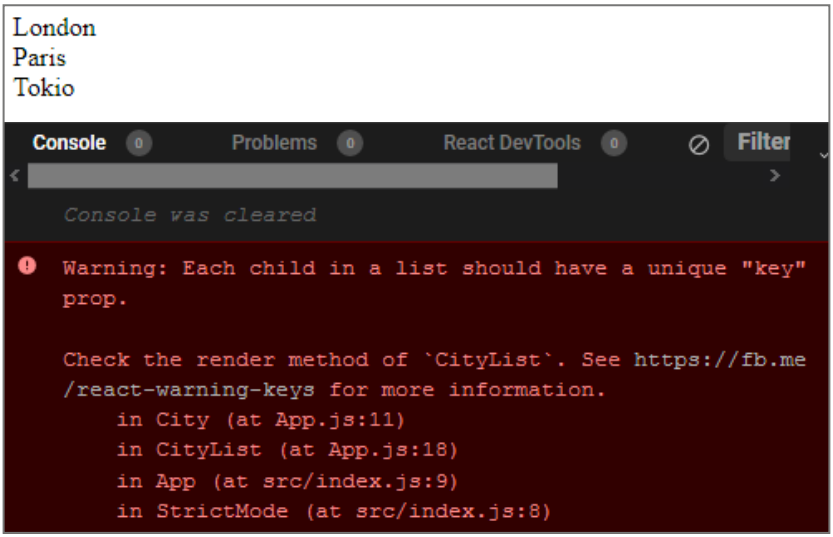

Рисунок 5

Изменим код примера с писателями. Внедрим поддержку ключей. В качестве ключа мы будем использовать уникальные идентификаторы. Для этого мы добавим свойство id внутрь объекта писателя.

```
import React from "react";
import "./styles.css";

var writers = [
  { id: 1, name: "Dan", lastName: "Brown" },
  { id: 2, name: "Joanne", lastName: "Rowling" },
  { id: 3, name: "Stephen", lastName: "King" }
];

function Writer(props) {
  return (
    <>
      <div>
```

```
        {props.name} {props.lastName}
      </div>
      <hr />
    </>
  );
}
function WritersList(props) {
  return (
    <div>
      {props.data.map(item => (
        <Writer key={item.id} {...item} />
      ))}
    </div>
  );
}

export default function App() {
  return (
    <>
      <WritersList data={writers} />
    </>
  );
}
```

Что изменилось в коде? Во-первых, внутри объекта писателя появился идентификатор: свойство id.

```
var writers = [
  { id: 1, name: "Dan", lastName: "Brown" },
  { id: 2, name: "Joanne", lastName: "Rowling" },
  { id: 3, name: "Stephen", lastName: "King" }
];
```

Напоминаем вам, что значения для id должны быть уникальными.

Во-вторых, изменился код компоненты WritersList.

```
function WritersList(props) {
  return (
    <div>
      {props.data.map(item => (
        <Writer key={item.id} {...item} />
      ))}
    </div>
  );
}
```

При описании компоненты Writer мы явно указываем атрибут key и задаём ему значение item.id.

Больше изменений в нашем коде нет. В результате использования key предупреждение пропало, но самое главное, что теперь React точно отличает одну строку списка писателей от другой.

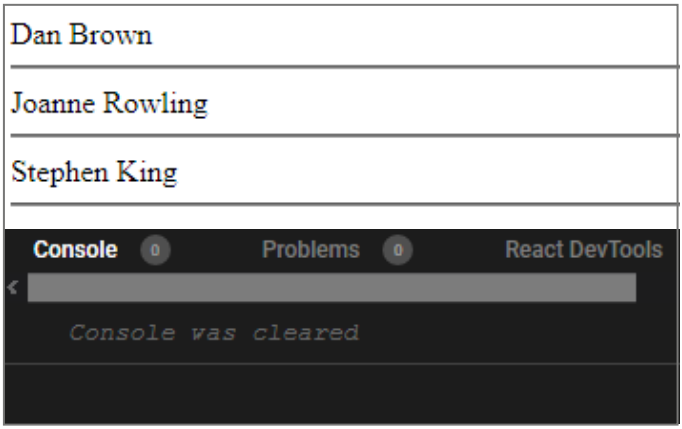

Рисунок 5

Ссылка на проект: https://codesandbox.io/s/array3-fwz9w.

Формы

Вы уже знакомы с механизмом форм в HTML. Настала очередь рассмотреть, как поддерживается работа с формами при помощи базовых механизмов React. Для старта нашего погружения возьмем базовый пример с отображением формы.

Рисунок 6

В нашем UI есть текстовое поле и кнопка. Ничего необычного. Код нашей формы:

```
function Form() {
  return (
    <form>
      <input type="text" />
      <input type="submit" value="Click me" />
    </form>
  );
}
```

Для создания формы мы используем типичные теги для работы с формами. Тег form объявляет форму. Два тега input создают текстовое поле и кнопку submit (используется для отсылки информации на сервер). В первом input атрибут type равен text — это значит, что мы создаём текстовое поле. Для создания кнопки submit мы указываем

submit при описании атрибута type. Если пользователь нажмет на кнопку submit произойдет отправка данных формы и страница перегрузится.

При описании тега form мы не использовали никаких атрибутов. Это значит, что мы не указали путь к серверному сценарию, который будет анализировать данные формы. В реальной жизни вы будете обязательно это делать с помощью атрибута action. Куда же отправляются наши данные в случае, если action не указан? Данные из формы на странице будут посылаться самой же странице. Например, если ваша форма находится в файле *myform.php* на сервере, данные после нажатия submit будут отосланы в *myform.php*.

Ссылка на проект: https://codesandbox.io/s/form1-35e09.

Можно ли затормозить отсылку данных после нажатия на кнопку submit? Конечно! Для этого мы будем использовать функцию preventDefault. Покажем это на примере:

```
import React from "react";
import "./styles.css";

function Form() {
  const handleSubmit = event => {
    /*
    Отменяем реакцию обработчика по умолчанию
    */
    event.preventDefault();
    /*
    Получаем доступ к текстовому полю
    */
    let uName = document.getElementById("userName");
    alert(uName.value);
  };
```

```
  return (
    <form onSubmit={handleSubmit}>
      <input type="text" id="userName" />
      <input type="submit" value="Click me" />
    </form>
  );
}

export default function App() {
  return (
    <>
      <Form />
    </>
  );
}
```

Внешний нашего приложения не изменился. Отличия только в коде.

```
<form onSubmit={handleSubmit}>
```

При описании формы мы указали, что обрабатываем отсылку данных. Событие возникает, когда пользователь нажимает на кнопку submit внутри формы.

```
const handleSubmit = event => {
    /*
      Отменяем реакцию обработчика по умолчанию
    */
    event.preventDefault();
    /*
      Получаем доступ к текстовому полю
    */
    let uName = document.getElementById("userName");
    alert(uName.value);
  };
```

В коде нашего обработчика мы отменяем стандартную обработку события с помощью вызова preventDefault. После чего мы получаем доступ к текстовому полю и отображаем его значение.

Для получения ссылки на текстовое поле мы используем уже знакомый вам механизм getElementById.

Ссылка на проект: https://codesandbox.io/s/form2-1mcfh.

А можно ли получить доступ к значению элемента формы без использования getElementById? Конечно! Ответ в следующем раздел.

Рефы

Рефы — это инструмент для связывания элемента форма с некоторой переменной. Благодаря такому связыванию вы сможете обмениваться значениями с элементом управления. За механизм такого взаимодействия отвечает React.

Для создания ref-переменной используется метод React.createRef(). Шаги по создания ref-переменной.

1. Создать ref-переменную с помощью вызова createRef.
2. Привязать её к конкретному элементу управления через атрибут ref.

После выполнения этих шагов наша переменная готова к работе!

Создадим приложение с таким интерфейсом:

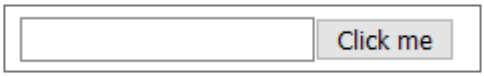

Рисунок 7

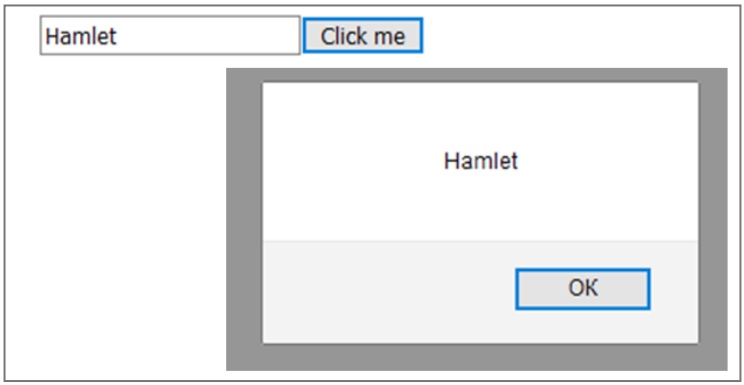

Рисунок 8

По нажатию на кнопку мы покажем окно сообщение со значением, введенным в текстовое поле. Данные на сервер отправлены не будут, так как в обработчике onSubmit мы вызовем preventDefault. Внешний вид приложения после нажатия на кнопку (рис. 8).

В нашем примере мы создадим функциональный компонент и ref-переменную.

Код *App.js*:

```
import React from "react";
import "./styles.css";

function Form() {
  let uRef = React.createRef();
  const handleSubmit = event => {
    event.preventDefault();
    alert(uRef.current.value);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" id="userName" ref={uRef} />
      <input type="submit" value="Click me" />
    </form>
  );
}

export default function App() {
  return (
    <>
      <Form />
    </>
  );

}
```

Важные этапы в коде.

```
let uRef = React.createRef();
```

Создание ref-переменной. Пока она не привязана ни к какому элементу управления.

```
const handleSubmit = event => {
    event.preventDefault();
    alert(uRef.current.value);
  };
```

В обработчике onSubmit мы получаем доступ к значению элемента управления, с которым связана ref-переменная. Для этого используется конструкция uRef.current.value.

```
      <input type="text" id="userName" ref={uRef} />
```

Тут мы связываем текстовое поле и ref-переменную. Связь происходит через атрибут ref. Все вопросы по пересылке данных решает React.

Ссылка на проект: https://codesandbox.io/s/formref-mpzbc.

Второй пример по работе с ref-переменными реализуем через классовые компоненты.

Внешний вид приложения:

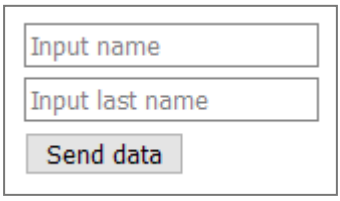

Рисунок 9

По нажатию на кнопку, отправка данных не произойдет. Мы отобразим окно с полученными от пользователя данными. Полный код *App.js*:

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

class Form extends Component {
  nameRef = React.createRef();
  lastNameRef = React.createRef();
  render() {
    const handlerSubmit = event => {
      event.preventDefault();
      let str = this.nameRef.current.value+" ";
      str += this.lastNameRef.current.value;
      alert(str);
    };

    return (
      <form onSubmit={handlerSubmit}>
        <div className="formElement">
          <input
            type="text"
            placeholder="Input name"
            ref={this.nameRef}
            required
          />
        </div>
        <div className="formElement">
          <input
            type="text"
            placeholder="Input last name"
            ref={this.lastNameRef}
            required
          />
        </div>
```

```
        <div className="formElement">
          <input type="submit" value="Send data" />
        </div>
      </form>
    );
  }
}

export default class App extends Component {
  render() {
    return (
      <>
        <Form />
      </>
    );
  }
}
```

Обращаем внимание на важные моменты кода.

```
import React, { Component } from "react";
import "./styles.css";
class Form extends Component {
```

Мы импортировали Component, чтобы сократить запись React.Component до Component при наследовании.

```
class Form extends Component {
  nameRef = React.createRef();
  lastNameRef = React.createRef();
```

Создание и использование ref-переменных работает также, как в функциональных компонентах. Мы создали

две ref-переменные, которые мы будем использовать для доступа к элементам управления.

```
const handlerSubmit = event => {
  event.preventDefault();
  let str = this.nameRef.current.value+" ";
  str += this.lastNameRef.current.value;
  alert(str);
};
```

Код обработчика onSubmit отличается от предыдущих примеров только наличием this.

```
<input
  type="text"
  placeholder="Input name"
  ref={this.nameRef}
  required
/>
```

Связывание элемента управления также производится через атрибут ref.

Ссылка на проект: https://codesandbox.io/s/classref-g3vws.

Состояние (State) и формы

Мы уже привыкли к тому, что React отвечает за состояние и обновления интерфейса нашего приложения, однако элементы управления в формах самостоятельно обновляют свой внешний вид. Например, когда пользователь вводит букву в текстовое поле, она автоматически появляется в нём.

Может ли нам понадобится React в этом процессе? Конечно. Причин может быть много.

Например, для общего подхода со всем UI-интерфейсом приложения. Возможно, мы захотим возложить на React задачу по отображению текста в поле только в том случае, если пользователь не ввёл запрещенного значения. Нам на помощь приходит уже известный механизм состояния.

Начнем с примера, в котором мы закрепим обновление текстового поля за React.

Внешний вид нашего приложения:

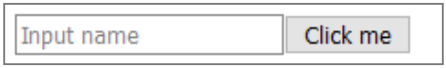

Рисунок 10

В этом примере, если вы попытаетесь ввести текст в текстовое поле ничего не произойдет.

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

function Form() {
```

```
  const [nameState, setNameState] = useState("");
  return (
    <form>
      <input type="text" placeholder="Input name"
             value={nameState} required />
      <input type="submit" value="Click me" />
    </form>
  );
}
export default function App() {
  return (
    <>
      <Form />
    </>
  );
}
```

В коде мы создаём переменную состояния.

```
  const [nameState, setNameState] = useState("");
```

Для закрепления переменной состояния за текстовым поле мы используем свойство value.

```
<input type="text" placeholder="Input name"
       value={nameState}
```

Для закрепления ref-переменной мы использовали свойство ref.

Почему текст не отображается? Это происходит из-за того, что мы закрепили в коде обновление элемента через React, но не уточнили логику этого процесса.

Ссылка на проект: https://codesandbox.io/s/formstate1-smopw.

Логика процесса обновления реализуется через обработчик onChange для текстового поля.

Теперь изменим пример, чтобы в текстовом поле появлялся набираемый текст.

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

function Form() {
  const [nameState, setNameState] = useState("");
  const handlerSubmit = event => {
    event.preventDefault();
    alert(nameState);
  };
  const handlerChange = event => {
    setNameState(event.target.value);
  };
  return (
    <form onSubmit={handlerSubmit}>
      <input
        type="text"
        placeholder="Input name"
        value={nameState}
        onChange={handlerChange}
        required
      />
      <input type="submit" value="Click me" />
    </form>
  );
}
export default function App() {
  return (
    <>
      <Form />
    </>
  );
}
```

При создании текстового поля мы указали, что у нас есть обработчик изменений в нём.

```
<input
        type="text"
        placeholder="Input name"
        value={nameState}
        onChange={handlerChange}
        required
      />
```

Это функция handlerChange.

```
const handlerChange = event => {
    setNameState(event.target.value);
  };
```

В теле функции мы берем текущий набранный текст через event.target.value и обновляем наше состояние через setNameState. После обновления переменной-состояния, React автоматически обновляет элемент управления, за которым она закреплена.

Ссылка на проект: https://codesandbox.io/s/formstate2-hzgy4.

В следующем примере мы будем анализировать, вводимый пользователем текст и отменять ввод, если текущее значение нас не устроит. Внешний вид приложения:

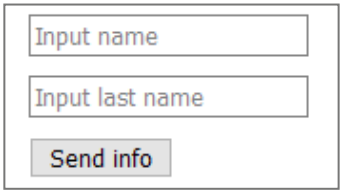

Рисунок 11

Мы запретим вводить test в качестве имени. Этот проект создан с помощью классовых компонент.

Код *App.js*:

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

class UserForm extends Component {
  state = {
    nameState: "",
    lastNameState: ""
  };

  render() {
    const handlerSubmit = event => {
      event.preventDefault();
      alert(this.state.nameState + " " +
            this.state.lastNameState);
    };

    const handlerNameChanged = event => {
      let name = event.target.value;
      if (name.trim().toUpperCase() === "TEST") {
        alert("Wrong name!");
        this.setState({ nameState: "" });
      } else {
        this.setState({ nameState:
                        event.target.value });
      }
    };

    const handlerLastNameChanged = event => {
      this.setState({ lastNameState:
                      event.target.value });
    };

    return (
      <form onSubmit={handlerSubmit}>
```

```
        <input
          type="text"
          className="formElement"
          placeholder="Input name"
          value={this.state.nameState}
          onChange={handlerNameChanged}
          required
        />
        <input
          type="text"
          className="formElement"
          placeholder="Input last name"
          value={this.state.lastNameState}
          onChange={handlerLastNameChanged}
          required
        />
        <input type="submit"
               className="formElement"
               value="Send info" />
      </form>
    );
  }
}

export default class App extends Component {
  render() {
    return (
      <>
        <UserForm />
      </>
    );
  }
}
```

Мы создаём две переменные состояния. По одной для каждого текстового поля.

```
class UserForm extends Component {
  state = {
    nameState: "",
    lastNameState: ""
  };
```

Мы используем задание свойства, а не конструктор, как мы много раз делали раньше.

Для проверки значения в текстовом поле мы используем обработчик события onChange:

```
const handlerNameChanged = event => {
      let name = event.target.value;
      if (name.trim().toUpperCase() === "TEST") {
        alert("Wrong name!");
        this.setState({ nameState: "" });
      } else {
        this.setState({ nameState: event.target.value
});
      }
    };
```

Если значение равно test, мы изменяем переменную состояния, записывая в неё пустое значение.

Если значение не равно test, тогда мы обновляем переменную состояния, записывая в неё текущее значение.

Разберите внимательно весь код и проверьте работу примера на практике.

Ссылка на проект: https://codesandbox.io/s/formstate3-2v4of.

Домашнее задание

1. Создайте приложение «Персональная страница». Для отображения части информации используйте текстовые поля. Например: ФИО, телефон, email, город проживания и т.д. Наличие текстовых полей позволяет пользователю модифицировать исходные данные. Также добавьте кнопку для возврата начальных значений. Используйте state, классовые компоненты.
2. Создайте регистрационную форму. Пользователь должен вводить: ник, электронный адрес, пол, возраст. Используйте возможности React для работы с формами. Используйте React для валидации, введенных значений.
3. Создайте форму для загрузки фотографии. Пользователь должен вводить: ник, пароль, электронный адрес, фотографию, описание фотографии, теги. Используйте возможности React для работы с формами. Используйте React для валидации, введенных значений.
4. Создайте приложение для отображения информации о городах вашей страны. Обязательно нужно использовать массивы объектов и функцию map
5. Создайте приложение для отображения информации о ваших любимых музыкальных группах. Обязательно нужно использовать массивы объектов, функцию map и классовые компоненты