Разработка интернет-приложений

Лабораторная работа 5

Библиотека React. Формы. Знакомство с хуками. Хук useState.

4 часа

Цель работы: изучение библиотеки React и получение практических навыков при работе с ней.

Задачи:

- изучить основы работы с формами при использовании библиотеки React;

- изучить механизм работы хуков при использовании библиотеки React;

- согласно варианту выполнить задание.

Теоретическая часть

Работа с формами

Работа элементов HTML-форм в React немного отличается от работы других DOM-элементов. Это связано с тем, что элементы форм по своей природе обладают некоторым внутренним состоянием. К примеру, данная форма в нативном HTML принимает только имя:

```
<form>
 <label>
   Name: <input type="text" name="name" />
 </label>
 <input type="submit" value="Submit" />
</form>
```

Представленная форма имеет поведение HTML-формы по умолчанию: просмотр новой страницы, когда пользователь посылает форму. Если такое поведение вам необходимо и в React, то оно работает как обычно. Но в большинстве случаев нам удобно иметь JavaScript-функцию, которая имеет доступ к данным, которые пользователь ввел в форму и обрабатывает её отправку. Для этой цели есть стандартный подход, под названием **«контролируемые компоненты»**.

По умолчанию в HTML элементы формы, такие как `<input>, <textarea>` и `<select>`, хранят свое собственное состояние и обновляют его на основании пользовательского ввода. Но в React модифицируемое состояние, как правило, является собственностью компонентов и обновляется только с помощью **`setState()`**.

Мы можем скомбинировать обе эти особенности, делая состояние React “единственным источником достоверной информации (истины)”. В свою очередь React-компонент, который отрисовывает форму, также контролирует, что происходит на этой форме

в ответ на последующий ввод пользователя. Элемент ввода формы (например, input), значение которого контролируется React, в этом случае называется «контролируемый компонент».

Например, у нас есть компонент UserForm, который представляет форму для ввода имени пользователя с возможностью условной отправки:

```
   render() {
        return (
            <form onSubmit={this.handleSubmit}>
                <p>
                    <label>Имя:</label><br />
                    <input type="text" value={this.state.name} on-
Change={this.onChange}/>
                </p>
                <input type="submit" value="Отправить" />
            </form>
        );
      }
    }
    ReactDOM.render(
        <UserForm />,
        document.getElementById("app")
    )
```

Чтобы контролировать введенные значения, в конструкторе устанавливается объект **state**:

```
     constructor(props) {
        super(props);
        this.state = {name: ""}; //вот здесь устанавливается объект

        this.onChange = this.onChange.bind(this);
        this.handleSubmit = this.handleSubmit.bind(this);
      }
```

При определении поля ввода каждое поле связывается с определенным значением в **state**:

```
      <input type="text" value={this.state.name} onChange={this.onChange} />
```

Так, источником значения для поля ввода имени является объект **this.state.name**.

Для отслеживания изменений в поле ввода нам надо определить обработчик для события **change** с помощью атрибута **onChange**. Этот обработчик будет срабатывать при каждом нажатии клавиши клавиатуры. Если мы не определим для поля подобный обработчик, то это поле ввода будет доступно только для чтения.

Суть каждого обработчика заключается в изменении значений в **this.state**:

```
    onChange(e) {
    var val = e.target.value;
    this.setState({name: val});
    }
```

С помощью **e.target.value** получаем введенное значение. После обновления новое значение **this.state.name** отобразится в поле ввода.

Для условной отправки устанавливаем обработчик у формы для события submit, который выводит в окне введенные значения.

Обработка множества input

Когда вам нужно обрабатывать множество контролируемых элементов **input**, вы можете добавить атрибут **name** на каждый элемент и позволить функции-обработчику выбрать, что делать, на основании значения **event.target.name**.

Например, вот так выглядит функция обработчик:

```
onChangeInput(event) {
      const name = event.target.name;
      this.setState({[name]: value});
    }
```

А вот сама форма:

```
<form>
          <label>First Name: <input name="firstName"  type="text"
                                value={this.state.firstName} onChange={this.on-
ChangeInput}/></label>
          <label> Last Name: <input name="lastName"  type="text"
                                value={this.state.lastName} onChange={this.on-
ChangeInput}/></label>
</form>
```

Валидация форм

Реализации валидации форм в React.js предусматривает проверку их введенных значений, и если эти значения соответствуют требованиям, тогда происходит изменение состояния компонента.

Например, мы имеем форму, которое имеет поле для ввода возраста и имени. Дополнительные значения **nameValid** и **ageValid** позволяют установить корректность введенных имени и возраста соответственно. Эти значения понадобятся для стилизации полей. Так, если введенное значение некорректно, то поле ввода будет иметь красную границу, иначе зеленую:

```
var nameColor = this.state.nameValid===true?"green":"red";
var ageColor = this.state.ageValid===true?"green":"red";
```

При определении поля ввода каждое поле связывается с определенным значением в **state**:

```
<input type="text" value={this.state.name} onChange={this.onNameChange}
style={{borderColor:nameColor}} />

<input type="number" value={this.state.age} onChange={this.onAgeChange}
style={{borderColor:ageColor}} />
```

И для каждого поля ввода определен свой обработчик **onChange**, в котором происходит валидация и изменение введенного значения.

Хуки

Хуки позволяют определять и использовать состояние и другие возможности React без создания классов. По сути, хуки представляют функции, которые позволяют подключиться к состоянию и другим возможностям, которые есть в React.

Мы можем создавать свои хуки, однако React по умолчанию уже предоставляет ряд встроенных хуков:

- **useState:** предназначен для управления состоянием компонентов
- **useEffect:** предназначен для перехвата различного рода изменений в компонентах, которые нельзя обработать внутри компонентов
- **useContext:** позволяет подписываться на контекст React
- **useReducer:** позволяет управлять локальным состоянием сложных компонентов
- **useCallback:** позволяет управлять функциями обратного вызова
- **useMemo:** предназначен для управления мемоизированными (грубо говоря кэшированными) значениями
- **useRef:** возвращать некоторое изменяемое значение, например, ссылку на html-элементы DOM, которыми затем можно управлять в коде JavaScript
- **useImperativeHandle:** настраивает объект, который передается родительскому компоненту при использовании ref
- **useLayoutEffect:** аналогичен хуку useEffect(), но вызывается синхронно после всех изменений в структуре DOM
- **useDebugValue:** предназначен для отображения некоторого значения в целях отладки

Правила хуков

Хуки имеют ряд ограничений при определении и использовании:

- Хуки вызываются только на **верхнем уровне** (top-level) компонента. Они НЕ вызываются внутри циклов, условных конструкций, внутри стандартных функций javascript.
- Хуки можно вызывать только из функциональных компонентов React, либо из других хуков. Но их нельзя вызывать из классов-компонентов.

Хук useState

Например, мы имеем код с **использованием классов**, в котором сначала в конструкторе определено состояние **this.state** как **{ count: 0 }**. Каждый раз, когда пользователь кликает, мы увеличиваем **state.count** на единицу, вызывая **this.setState():**

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }
  render() {
    return (
      <div>
        <p>Вы кликнули {this.state.count} раз(а)</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Нажми на меня
        </button>
      </div>
    );
  }
}
```

В функциональном компоненте нам недоступен **this**, поэтому мы не можем задать или считать состояние через **this.state**. Вместо этого мы вызываем хук **useState** напрямую изнутри нашего компонента:

```
import React, { useState } from 'react';

function Example() {
  // Объявление новой переменной состояния «count»
  const [count, setCount] = useState(0);
```

Что делает вызов useState? Он объявляет «переменную состояния». Мы называли переменную count, но могли дать ей любое имя, хоть банан. Таким образом мы можем «сохранить» некоторые значения между вызовами функции. **useState** — это новый способ использовать те же возможности, что даёт **this.state** в классах. Обычно переменные «исчезают» при выходе из функции. К переменным состояния это не относится, потому что их сохраняет React.

Какие аргументы передавать useState? Единственный аргумент **useState** — это исходное состояние. В отличие от случая с классами, состояние может быть и не объектом, а строкой или числом, если нам так удобно. Поскольку в нашем примере отслеживается количество сделанных пользователем кликов, мы передаём 0 в качестве исходного значения переменной. (Если нам нужно было бы хранить два разных значения в состоянии, то пришлось бы вызвать **useState()** дважды.)

Что возвращается из useState? Вызов **useState** вернёт пару значений: текущее состояние и функцию, обновляющую состояние. Поэтому мы пишем **const [count, setCount] = useState().** Это похоже на **this.state.count** и **this.setState** в классах, с той лишь разницей, что сейчас мы принимаем их сразу в паре. Если вам незнаком использованный синтаксис, мы вернёмся к нему ближе к концу страницы.

Когда мы хотим отобразить текущее состояние счётчика в классе, мы обращаемся к **this.state.count**:

```
<p>Вы кликнули {this.state.count} раз(а)</p>
```

В итоге мы получаем следующий код, используя хуки вместо классов:

```
import React, { useState } from 'react';

function Example() {
  // Объявление новой переменной состояния «count»
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Вы кликнули {count} раз(а)</p>
      <button onClick={() => setCount(count + 1)}>
        Нажми на меня
      </button>
    </div>
  );
}
```

Задание на лабораторную работу

1) Выполнить задание №1 и задание №2 по работе с формами и хуком useState.

   Задание №1

   Необходимо создать компонент формы, используя библиотеку React, со следующими обязательными и валидируемыми полями для ввода:

   - Имя
   - Фамилия
   - Email
   - Номер телефона
   - Выпадающий список с выбором пола

   При нажатии кнопки «Отправить» информация из формы должна отображаться во всплывающем окне/консоли в виде текста/объекта. При пустом значении поля вывести предупреждающий текст (можно под полем). Допускается использование дополнительных библиотек для создания компонентов.

   При выполнении задания **использовать классы**, без использования хуков.

   Задание№2

   Выполнить задание№1, но использовать хук useState.

2) Выполнить задание№3 согласно варианту по работе с хуком useState.

   Необходимо создать текстовое поле и компонент списка, элементы которого добавляются при помощи ввода значения в текстовое поля при нажатии на пробел/кнопку. Например, ввел в текстовое поле значение, далее нажал пробел и значение появилось снизу списка. Начальное состояние должно быть пустым массивом. При выполнении задания использовать хук useState.

Вариант	Тема
1	Список марок машин
2	Список учеников класса
3	Список школьных предметов
4	Список книг по React

3) Выполнить задание№4 согласно варианту

Вариант	Задание
1	Создать компонент, состоящий из 2 полей ввода (например, имя и фамилия) и текста под ним. При наборе текста в поле ввода, он сразу же дублируется под ним. Выполнить задание, используя хук useState.

2	Создать компонент «Комната» с кнопкой «Выключатель света» и текстом, описывающим «В комнате светло» или «В комнате темно». Нажатие кнопки должно включать и выключать свет, а также обновлять текст. Используйте хук useState, чтобы сохранить состояние выключателя.
3	Создать компонент «Рандомный лист», который состоит из кнопки и списка случайных чисел. При нажатии на кнопку, в список добавляется еще одно случайное число. Сохраняйте массив чисел с помощью useState. Начальное состояние должно быть пустым массивом.
4	Создать компонент «Счетчик кликов», который состоит из кнопки и текста-счетчика (кол-во кликов на кнопку). При нажатии на кнопку к счетчику добавляется 1. Выполнить задание, используя хук useState.