**Разработка интернет-приложений**

**Лабораторная работа 4**

**Основы работы с библиотекой React**

**4 часа**

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

**Задачи:**

- изучить основы JSX и библиотеки React;

- согласно варианту выполнить задание.

**Теоретическая часть**

React — это библиотека JavaScript, которая используется для создания пользовательского интерфейса. Первоначально React предназначался для веба, для создания веб-сайтов, однако позже появилась платформа. React Native, которая уже предназначалась для мобильных устройств. Большинство приложений на React представляют собой SPA (одностраничное приложение).

**Отличительные особенности React**

* библиотека работает именно с виртуальным DOM, а не обычным;
* использование и концентрация на компонентах - мы можем создать отдельные компоненты и затем их легко переносить из проекта в проект;
* использование JSX.

**Виртуальный DOM**

Виртуальный DOM представляет легковесную копию обычного DOM (Document Object Model- организация элементов html, которыми мы можем манипулировать, изменять, удалять или добавлять новые). И отличительной особенностью React является то, что данная библиотека работает именно с виртуальным DOM, а не обычным.

Если приложению нужно узнать информацию о состоянии элементов, то происходит обращение к виртуальному DOM.

Если необходимо изменить элементы веб-страницы, то изменения вначале вносятся в виртуальный DOM. Потом новое состояние виртуального DOM сравнивается с текущим состоянием. И если эти состояния различаются, то React находит минимальное количество манипуляций, которые необходимы до обновления реального DOM до нового состояния и производит их.

**Основы JSX**

JSX представляет способ описания визуального кода посредством комбинации кода на JavaScript и разметки XML. В реальности для разработки под React использование JSX не требуется, однако применение JSX является рекомендуемым способом создания интерфейса.

JSX также позволяет вводить в эти элементы код JavaScript. С помощью фигурных скобок в код html можно вводить код на JavaScript, который затем выполняется, а на веб-станице мы увидим его результат.

Например,

ReactDOM.render(

<h2>2 + 2 = {2 + 2}</h2>, #JSX элемент

document.getElementById("root")

)

При работе с JSX следует учитывать ряд моментов, в частности, в JSX для установки класса применяется атрибут **className**, а не **class**. Второй момент: атрибут **style** в качестве значения принимает объект javascript. И третий момент: в JSX используется camel-case, то есть если мы хотим определить стилевое свойство для шрифта, например, свойство **font-family**, то соответствующее свойство в объекте стиля будет называться **fontFamily**, то есть дефис отбрасывается, а следующая часть слова начинается с заглавной буквы.

Например,

ReactDOM.render(

<div className={userClassName} style= {color:'red', fontFamily:'Verdana'}}>

<p>Полное имя: User </p>

<p>Возраст: 17 </p>

</div>,

document.getElementById("root")

)

**Рендеринг элементов**

Мельчайшими блоками приложения React являются элементы. Определение простейшего элемента:

const element = <h1>Hello, React</h1>;

По сути элементы в React - это обычные объекты JavaScript, с которыми быстрее работать, чем с обычными элементами на веб-странице.

Для рендеринга элементов в React применяется метод **ReactDOM.render()**. Этот метод принимает три параметра:

* **ReactElement**: элемент для рендеринга;
* **DOMElement**: элемент DOM, в который надо добавить ReactElement;
* **Callback**: опциональная функция обратного вызова.

Допустим, в нашем HTML-файле есть **<div>**:

<div id="root"></div>

Мы назовём его «корневым» узлом DOM, так как React DOM будет управлять его содержимым.

Обычно в приложениях, написанных полностью на React, есть только один корневой элемент. При встраивании React в существующее приложение вы можете рендерить столько независимых корневых элементов, сколько посчитаете нужным.

Для рендеринга React-элемента в корневой узел DOM вызовите **ReactDOM.render()** с React-элементом и корневым DOM узлом в качестве аргументов:

const element = <h1>Hello, world</h1>;

ReactDOM.render(element, document.getElementById('root'));

На странице будет написано «Hello, world».

Элементы React являются неизменяемыми (immutable). После создания элемента нельзя изменить его атрибуты или дочерние элементы. И единственный способ изменить интерфейс, определенный в элементе, это создать новый элемент и передать его в функцию **ReactDOM.render()**.

**Компоненты и создание компонентов**

Компоненты аналогичны функциям JavaScript. Они хранят состояние с помощью свойств и возвращают элементы React, которые затем появляются на веб-странице.  
Компоненты можно определить различными способами. Первый способ - функциональный. Например:

function Hello() {

return <h1>Привет, Eugene</h1>;

}

Здесь определяется компонент Hello. Название компонентов должно начинаться с заглавной буквы.

Второй способ определения компонентов предполагает использование классов ES6:

class Hello extends React.Component {

render() {

return <h1>Привет, Eugene</h1>;

}

}

Для рендеринга компонента в классе компонента обязательно должен быть определен метод **render()**, который возвращает создаваемый элемент на JSX.

Также для определения мы можем использовать стрелочные функции (arrow functions):

var Hello =() => {

return (<h1>Привет, Eugene</h1>);

}

## Props

Props представляет коллекцию значений, которые ассоциированы с компонентом. Эти значения позволяют создавать динамические компоненты, которые не зависят от жестко закодированных статических данных.

Функциональный подход:  
function Hello(props) {

return <div>

<p>Имя: {props.name}</p>

<p>Возраст: {props.age}</p>

</div>;

}

Параметр **props**, который передается в функцию компонента, инкапсулирует свойства объекта. В частности, свойство name и age. При рендеринге мы можем создать набор компонентов Hello, но передать в них разные данные для name и age. И таким образом, получим набор однотипной разметки html, наполненной разными данными.

Использование классов ES6:

class Hello extends React.Component {

render() {

return <div>

<p>Имя: {this.props.name}</p>

<p>Возраст: {this.props.age}</p>

</div>;

}

}

Класс компонента также извне получает объект свойств, который доступен через this.props.

Использование стрелочных функций (arrow functions):  
const Hello = (props) => {

const {name, age} = props;

return(<div>

<p>Имя: {name}</p>

<p>Возраст: {age}</p>

</div>);

}

**Обновление props**

Сам **this.props** представляет неизменяемый объект, который предназначен только для чтения. Поэтому не следует его изменять, например:

class Hello extends React.Component {

render() {

this.props.name = "Tom";

return <h1>Привет, {this.props.name}</h1>;

}

}

**События**

Обработка событий элементов в React во многом похожа на обработку событий элементов DOM с помощью обычного JavaScript. В то же время есть небольшие отличия:

* События в React используют camelCase (в стандартном html "onclick", в React - "onClick")
* В JSX в обработчик события передается функция компонента, а не строка

В React можно применять разные способы определения и вызова событий.

Главная сложность при использовании событий — это работа с ключевым словом **this**, которое указывает на текущий объект, в данном случае компонент. По умолчанию в функцию обработчика не передается текущий объект, поэтому this будет иметь значение undefined. И ни к каким свойствам и методам компонента через this мы обратиться не сможем. И чтобы в метод press корректно передавалась ссылка на текущий объект через this, в конструкторе класса прописывается вызов:

class ClickButton extends React.Component {

    constructor(props) {

        super(props);

        this.press = this.press.bind(this); #вызов события

    }

    press(){

        console.log(this);

        alert("Hello React!")

    }

    render() {

        return <button onClick={this.press}>Click</button>;

}

}

**State**

Объект state описывает внутреннее состояние компонента, он похож на props за тем исключением, что состояние определяется внутри компонента и доступно только из компонента.

Если props представляет входные данные, которые передаются в компонент извне, то состояние хранит такие объекты, которые создаются в компоненте и полностью зависят от компонента.

Также в отличие от props значения в state можно изменять.

И еще важный момент - значения из state должны использоваться при рендеринге. Если какой-то объект не используется в рендерниге компонента, то нет смысла сохранять его в state.

Нередко state описывает какие-то визуальные свойства элемента, которые могут изменяться при взаимодействии с пользователем. Например, кнопку нажали, и соответственно можно изменить ее состояние - придать ей какой-то другой цвет, тень и так далее. Кнопку нажали повторно - можно вернуть исходное состояние.

Единственное место, где можно установить объект state — это конструктор класса:

class Hello extends React.Component {

            constructor(props) {

                super(props);

                this.state = {welcome: "Добро пожаловать на сайт!"};

            }

            render() {

                return <h1>{this.state.welcome}</h1>;

            }

        }

        ReactDOM.render(

            <Hello />,

            document.getElementById("app")

        )

Для обновления состояния вызывается функция **setState()**.

Изменение состояния вызовет повторный рендеринг компонента, в соответствии с чем веб-страница будет обновлена.

В то же время не стоит изменять свойства состояния напрямую, например:

this.state.welcome = "Привет React";

В данном случае изменения повторного рендеринга компонента происходить не будет.

При этом нам не обязательно обновлять все его значения. В процессе работы программы мы можем обновить только некоторые свойства. Тогда необновленные свойства будут сохранять старые значения.

**Асинхронное обновление состояния**

При наличии нескольких вызовов setState() React может объединять их в один общий пакет обновлений для увеличения производительности.

Так как объекты this.props и this.state могут обновляться асинхронно, не стоит полагаться на значения этих объектов для вычисления состояния.

Для обновления надо использовать другую версию функции setState(), которая в качестве параметра принимает функцию. Данная функция имеет два параметра: предыдущее состояние объекта state и объект props на момент применения обновления:

this.setState(function(prevState, props) {

return {

counter: prevState.counter + props.increment

};

});

**Жизненный цикл компонента**

В процессе работы компонент проходит через ряд этапов жизненного цикла. На каждом из этапов вызывается определенная функция, в которой мы можем определить какие-либо действия:

* **constructor(props):** конструктор, в котором происходит начальная инициализация компонента;
* **static getDerivedStateFromProps(props, state):** вызывается непосредственно перед рендерингом компонента. Этот метод не имеет доступа к текущему объекту компонента (то есть обратиться к объкту компоненту через this) и должен возвращать объект для обновления объекта state или значение null, если нечего обновлять.
* **render():** рендеринг компонента;
* **componentDidMount():** вызывается после рендеринга компонента. Здесь можно выполнять запросы к удаленным ресурсам;
* **componentWillUnmount():** вызывается перед удалением компонента из DOM.

Кроме этих основных этапов или событий жизненного цикла, также имеется еще ряд функций, которые вызываются при обновлении состояния после начального рендеринга компонента, если в компоненте происходят обновления:

* **static getDerivedStateFromProps(props, state);**
* **shouldComponentUpdate(nextProps, nextState):** вызывается каждый раз при обновлении объекта props или state. В качестве параметра передаются новый объект props и state. Эта функция должна возвращать true (надо делать обновление) или false (игнорировать обновление). По умолчанию возвращается true. Но если функция будет возвращать false, то тем самым мы отключим обновление компонента, а последующие функции не будут срабатывать.
* **render():** рендеринг компонента (если shouldComponentUpdate возвращает true);
* **getSnapshotBeforeUpdate(prevProps, prevState):** вызывается непосредственно перед компонента. Он позволяет компоненту получить информацию из DOM перед возможным обновлением. Возвращает в качестве значения какой-то отдельный аспект, который передается в качестве третьего параметра в метод componentDidUpdate() и может учитываться в componentDidUpdate при обновлении. Если нечего возвращать, то возвращается значение null
* **componentDidUpdate(prevProps, prevState, snapshot):** вызывается сразу после обновления компонента (если shouldComponentUpdate возвращает true). В качестве параметров передаются старые значения объектов props и state. Третий параметр - значение, которое возвращает метод getSnapshotBeforeUpdate

**Создание нового React-приложения**

[Create React App](https://github.com/facebookincubator/create-react-app) — удобная среда для изучения React и лучший способ начать создание нового [**одностраничного**](https://ru.reactjs.org/docs/glossary.html#single-page-application) (SPA) приложения на React.

Инструмент настраивает среду для использования новейших возможностей JavaScript, оптимизирует приложение для продакшена и обеспечивает комфорт во время разработки. Вам понадобятся [Node.js не ниже версии 10.16 и npm не ниже версии 5.6](https://nodejs.org/ru/) на вашем компьютере. Для создания проекта выполните команды:

npx create-react-app my-app

cd my-app

npm start

Create React App не обрабатывает бэкенд логику или базы данных, он только предоставляет команды для сборки фронтенда, поэтому вы можете использовать его с любым бэкэндом. «Под капотом» используются [Babel](https://babeljs.io/" \t "_blank) и [webpack](https://webpack.js.org/" \t "_blank), но вам не нужно ничего знать о них.

Когда ваше приложение готово к развёртыванию в продакшене, запуск команды npm run build создаст оптимизированную сборку вашего приложения в папке build. Вы можете узнать больше о Create React App [из его README](https://github.com/facebookincubator/create-react-app#_blank) и [его пользовательского руководства](https://facebook.github.io/create-react-app/).

**Задание на лабораторную работу**

1) Скачать Node.js и ознакомиться с возможностями менеджера пакетов npm.

2) Создать базовое одностраничное (SPA) react-приложение через create-react-app.

3) Выполнить задание №1 по выбранному варианту, используя библиотеку React.

|  |  |
| --- | --- |
| Вариант | Задание |
| 1 | Создать элементы: кнопка, картинка, текст, по умолчанию на кнопке текст «Вторая картинка». Когда мы нажимаем на кнопку, её текст меняется на «Первая картинка», меняется картинка и текст-описание (также работает и в обратную сторону). |
| 2 | Создать элементы: кнопка и параграф с текстом, по умолчанию на кнопке текст "Скрыть" и блок с текстом виден. Когда мы нажимаем на кнопку, её текст меняется на "Показать" и текст ниже пропадает. |
| 3 | Создать два квадрата, синий и зеленый, при клике на «зеленый» - зеленый пропадает, красный появляется, при клике на «синий» - синий пропадает, желтый появляется. При клике на «красный» - красный пропадает, зеленый появляется, при клике на «желтый» - желтый пропадает, синий появляется. |
| 4 | Создать элементы: кнопка и параграф с текстом По умолчанию цвет кнопки оранжевый, текст кнопки «Зажечь» и текст параграфа «Камин не горит». При нажатии на кнопку цвет кнопки становится голубой, текст кнопки «потушить» и текст параграфа «Камин горит». |
| 5 | Создать компонент, которому передаем текст (2-3 предложения), и кнопка «Подробнее». Компонент выводит только первое предложение. При клике на кнопку «Подробнее» выводиться вместо одного предложения весь текст. |
| 6 | Создать элементы: текст по умолчанию «Текущая температура:0», кнопку «Увеличить» и кнопку «Уменьшить». При нажатии на на «Увеличить» значение температуры увеличивается на 1, при нажатии на «Уменьшить» значение температуры уменьшается на 1. |
| 7 | Создать элементы: круг и кнопку. По умолчанию круг закрашен белым цветом и текст кнопки «Закрасить». При нажатии на «Закрасить» круг закрашивается черным и текст кнопки «Очистить». |

4) Выполнить задание№2 согласно выбранному варианту.

Создать список элементов(нумерованный/ненумерованный) список, каждый элемент списка должен представлять собой компонент. Список должен храниться в массиве и передаваться в компоненты через props.

|  |  |
| --- | --- |
| Вариант | Тема |
| 1 | Список марок машин |
| 2 | Список учеников класса |
| 3 | Список книг по React |
| 4 | Список покупок в магазине |
| 5 | Список школьных предметов |