React JS (2020)

**JSX**

React базируется на двух концепциях – элементы и компоненты. С элементами связано понятие JSX. Концепция JSX позволяет писать разметку внутри JS, используя все его возможности.

import React from 'react'

import ReactDOM from 'react-dom'

const phase = 'Home'

const element = <h1>Hello, { phase }</h1>

ReactDOM.render(

element,

document.getElementById('root')

)

Переменная element содержит внутри себя разметку, но при запуске все это преобразуется в понятные для JS элементы, поэтому внутри можно использовать JS выражения внутри фигурных скобок.

Обязательное условие для того, чтоб можно было использовать JSX – это подключение библиотеки React. Она всегда поймет, когда используется JSX, но если выражение внутри JSX большое, то его следует обвернуть в круглые скобки.

const element = (

<div>

<h1>Hello, { phase }</h1>

<h2>Come Back</h2>

</div>

)

Здесь можно заметить, что есть несколько тегов, но они обвернуты корневым тегом. Это также обязательное условие, если несколько и более тегов, то они должны быть обвернуты.

У тегов по-прежнему сохраняются атрибуты, но все они будут в CamelCase и значение для них также можно передать в фигурных скобках.

const element = (

<div>

<h1>Hello, { phase }</h1>

<img src={'...'}/>

<button className={'btn'}>Submit</button>

</div>

)

Видно, что здесь место обычного атрибута класса написано className. Это потому, что class в js – это ключевое слово и иначе код сломается.

**Элементы**

Элемент – это то, что видно пользователю. Может показаться, что это DOM элементы, но React элементы – это описание того, как должен выглядеть DOM элемент. Эта прослойка позволяет ускорить работу в разы.

Элементы создаются так:

const element = React.createElement('h1', null, 'Hello, Home')

Но это неудобно, поэтому есть JSX.

const element = <h1>Hello Home</h1>

Эти две записи здесь эквиваленты, но вариант с JSX просто понятнее.

Как видно, что это простые JS объекты, которые необходимо поместить в DOM, т.е. провести рендеринг. Рендеринг осуществляет библиотека ReactDOM путем метода render:

const element = <h1>Hello, Home</h1>

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

В первый передаем элемент, а во второй нужно передать DOM элемент, в который будет рендерится элемент. По-умолчанию React приложение содержит один html файл с де-факто одним тегом #root, в который по-умолчанию будут рендерится все элементы приложения.

**Иммутабельность и обновление кода**

Иммутабельность – это когда нельзя изменить элемент, поэтому если нужно будет обновить элемент, то нужно будет создать его сначала и отрендерить его.

function tick() {

const element = <h2>It is {new Date().toLocaleTimeString()}.</h2>

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

}

setInterval(tick, 1000)

Притом страница не будет перезагружаться полностью, потому что React сравнивает обновленные элементы со старыми элементами и точечно обновляет их. В этом помогает иммутабельность.

Конечно, пример сверху неправильный, т.к. здесь появляется логика и здесь должны быть компоненты.

**Компоненты**

Компоненты позволяют разбить программу на независимые куски, содержащие внутри какую-то логику. Компонент бывают двух видов:

Функциональный:

function Welcome(props) {

return <h1>Привет, {props.name}</h1>;

}

Классовый:

class Welcome extends React.Component {

render() {

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

}

}

Они созданы для разных целей, поэтому их возможности разные. Но общее, что они получают данные (пропсы) и должны вернуть элементы.

В функциональных компонентах пропсы получаются напрямую через параметры, и элементы возвращаются, как обычный результат.

В классах есть отдельный метод render, который должен вернуть элементы, а пропсы можно получить через this.props.

Отрендерить можно их, как обычный закрывающий тег, а параметры передать, как атрибут:

function Welcome(props) {

return <h1>Привет, {props.name}</h1>;

}

ReactDOM.render(

<Welcome name='Alex' />,

document.getElementById('root')

)

Компоненты никак не должны изменять пропсы, т.е. вести себя, как чистые функции:

function Sum({ a, b }) {

a += b

return <p>{a}</p>

}

Такой компонент не валидный. Здесь использовалась деструктуризация объекта на его свойства для пропсов.

На компоненты еще накладывается ограничения в названии – они должны начинаться с заглавной буквы, иначе React преобразует его к элементу.

Компоненты можно вызывать неограниченно количество раз и передавать разные пропсы:

ReactDOM.render(

<div>

<Welcome name='Alex' />

<Welcome name='Bob' />

<Welcome name='John' />

</div>,

document.getElementById('root')

)

Можно помещать один компонент в другой, т.е. проводит композицию компонентов. Например, есть компонент кнопки, который можно положить в компонент формы и в компонент заголовка.

Из-за композиции принято делать один главный компонент (обычно App), который внутри себя содержит все остальные компоненты.

function Welcome(props) {

return <h1>Привет, {props.name}</h1>;

}

function App() {

return <div>

<Welcome name='Alex' />

<Welcome name='Bob' />

<Welcome name='John' />

</div>

}

ReactDOM.render(

<App />,

document.getElementById('root')

)

**Состояние**

Состояние похожи на пропсы, но только они полностью доступны одному компоненту и их можно изменять. Здесь будет рассказано про состояние в классе, а в функциональных компонентах будет «Хуках».

Создать состояние надо в конструкторе в переменной state и передать туда объект:

class App extends React.Component {

constructor(props) {

super(props)

this.state = {

name: 'Bob'

}

}

render() {

return <div>

{ this.state.name }

</div>

}

}

В этот объект положили свойство name, которое можно получить через this.state.name.

Конструктор получает пропсы и так как здесь есть наследование, то нужно проинициализировать родителя через super.

Теперь есть соблазн в процессе жизни как-то поменять состояние, но делать это напрямую нельзя, т.к. все в React иммутабельно, поэтому есть отдельная функция setState, которую нужно передать новый объект, который смаржится с прошлым.

componentDidMount() {

this.setState({ name: 'John' })

}

Этот вариант простого изменения состояния, т.к. здесь оно задается напрямую, а если новое значение зависит от состояния и пропсов, то здесь нужно изменять через callback.

class Counter extends React.Component {

constructor(props) {

super(props)

this.state = {

value: 0

}

}

componentDidMount() {

this.setState((state, props) => ({ value: state.value + props.incr }))

}

render() {

return <div>

{ this.state.value }

</div>

}

}

ReactDOM.render(

<Counter incr={2}/>,

document.getElementById('root')

)

Данная функция callback принимает состояние и пропсы.

**Обработка событий**

В React обработка событий отличается от стандартных событий тем, что событие именуется в CamelCase и обработчик передается, как функция.

function App() {

function buttonHandler(event) {

console.log('Click')

}

return <div>

<button onClick={buttonHandler}>Click</button>

</div>

}

Также нужно сказать то, что отменить действия браузера по-умолчанию нельзя через возврат false.

В классовых компонентах у функций нет this, поэтому контекст надо привязать в конструкторе:

class Toggle extends React.Component {

constructor(props) {

super(props)

this.state = { mode: true }

this.toggleMode = this.toggleMode.bind(this)

}

toggleMode() {

this.setState((state, props) => ({ mode: !state.mode }))

}

render() {

return <div>

<button onClick={this.toggleMode}>

{this.state.mode ? 'Включено' : 'Выключено'}

</button>

</div>

}

}

Есть и другие способы создания обработчиков в React. Самый удобный вариант – это объявление, как функция стрелка.

toggleMode = () => {

this.setState((state, props) => ({ mode: !state.mode }))

}

Третий же способ заключается, в том, что передавать в событие callback, который вызовет обработчик.

toggleMode() {

this.setState((state, props) => ({ mode: !state.mode }))

}

render() {

return <div>

<button onClick={() => this.toggleMode()}>

{this.state.mode ? 'Включено' : 'Выключено'}

</button>

</div>

}

В принципе можно использовать любой вариант, но 2 наиболее предпочтительный, так что использоваться будет далее именно он.

Всегда по-умолчанию передается объект события, если же нужно явно указать какие-то параметры, то есть два способа:

class Counter extends React.Component {

constructor(props) {

super(props)

this.state = {

value: 0

}

}

setValue = (num, e) => {

this.setState(state => ({ value: state.value + num }))

console.log(e)

}

render() {

return <div>

<button onClick={this.setValue.bind(this, 1)}>Incr</button>

<button onClick={(e) => this.setValue(-1, e)}>Decr</button>

<h1>{ this.state.value.toString() }</h1>

</div>

}

}

Первый заключается в использования bind, притом React добавит событие, как последний параметр. Во втором же функция вызывается в callback, притом если нужно передать событие, то делать это нужно напрямую.

**Условный рендеринг**

Условный рендеринг в React – это обычное ветвление.

Если условие верно и нужно что-то отрендерить, то нужно воспользоваться оператором &&:

class Logged extends React.Component {

constructor(props) {

super(props)

this.state = { isLogged: false }

}

toggleMode = () => {

this.setState(state => ({ isLogged: !state.isLogged }))

}

render() {

return <div>

<button onClick={this.toggleMode}>

Logged

</button>

{this.state.isLogged &&

<div>

Welcome

</div>

}

</div>

}

}

Если же нужно что-то отрендерить в соответствии условия, то есть тернарный оператор:

render() {

return <div>

<button onClick={this.toggleMode}>Logged</button>

{this.state.isLogged

? <div>

Welcome

</div>

: <div>

Goodbye

</div>

}

</div>

}

Если же нужно не рендерить весь компонент, то нужно просто вернуть null, но притом сам компонент не удалится:

function Banner({ warn }) {

if (!warn)

return null

return <div>

Warning!

</div>

}

**Списки**

Если куча однотипных элементов / компонентов, то имеет смысл сами отличительные вещи вынести в отдельный массив и преобразовать его в то, что надо:

const arr = [1, 4, 6, 8]

const items = arr.map(el => <li>{el}</li>)

ReactDOM.render(

<ul>

{items}

</ul>,

document.getElementById('root')

)

Если список используется в компонентах, то нужно передать в параметр key уникальный идентификатор (желательно строку), чтоб React мог манипулировать элементами списка.

function NumberList({ arr }) {

return <ul>

{arr.map(el =>

<li key={el.toString()}>{el}</li>

)}

</ul>

}

ReactDOM.render(

<NumberList arr={[1, 5, 8, 2]} />,

document.getElementById('root')

)

Зачастую в key передает индекс элемента, но тогда каждая вставка / удаление массива заставит переопределить key, поэтому упадет производительность, т.к. React будет трудно сравнивать, чтоб точечно обновить.

**Формы**

Элементы формы содержат внутри собственное состояние, и они его же обновляют по надобности. Кроме того, формы при отправке делают переход на новую страницу.

Это неудобно, потому что обрабатывать формы через JS, имея доступ данным удобнее. Так что в React придумали управляемые компоненты.

Заключается это в том, что соединяется стандартное поведение элементов и пользовательского состояния.

class App extends React.Component {

constructor(props) {

super(props)

this.state = { value: '' }

}

handleChange = event => {

this.setState({ value: event.target.value })

}

handleSubmit = event => {

event.preventDefault()

console.log('send', this.state.value)

this.setState({ value: '' })

}

render() {

return <div>

<form onSubmit={this.handleSubmit}>

<label>

Имя:

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

</label>

<input type="submit" value="Отправить" />

</form>

</div>

}

}

Здесь создается свойство value в состоянии, которое при помощи атрибута value связывается, т.е. значением в этом поле будет this.state.value.

А обработчик handleChange обновляет это поле, когда есть изменения в элементе. Если не указать обработчик, то значение будет никак не поменять. Но есть исключение – null и undefined, если их передать, то само поле ввода можно будет редактировать.

<input type="text" value={null} />

Такие обработчики сделаны, чтоб облегчить вадилацию или изменения полученного значения. Например, если нужно чтоб все было в высоком регистре, то сделается это изменением одной строчки:

handleChange = event => {

this.setState({ value: event.target.value.toUpperCase() })

}

Для отправки этой формы используется также обычный обработчик, но здесь он еще отменяет действия по-умолчанию.

В формах часто есть несколько однотипных полей и создавать отдельный обработчик для них утомляет, поэтому можно сократить код, используя атрибут name, в которое нужно передать свойство, которое нужно изменить.

class App extends React.Component {

constructor(props) {

super(props)

this.state = { name: '', lastname: '' }

}

handleChange = event => {

this.setState({

[event.target.name]: event.target.value

})

}

handleSubmit = event => {

event.preventDefault()

console.log('send', this.state.name, this.state.lastname)

}

render() {

return <div>

<form onSubmit={this.handleSubmit}>

<label>

Имя:

<input type='text' name='name' value={this.state.name} onChange={this.handleChange}/>

</label>

<label>

Фамилия:

<input type='text' name='lastname' value={this.state.value} onChange={this.handleChange}/>

</label>

<input type='submit' value='Отправить' />

</form>

</div>

}

}

Элемент textarea отличается от стандартного тега textarea тем, что значение надо передать через атрибут value.

<textarea value={this.state.value} onChange={this.handleChange}/>

Элемент select в отличие от стандартного select для выбора пункта нужно передать его значение в атрибут value.

this.state = {value: 'Milk', el: ['Apple', 'Banana', 'Milk']}

<select value={this.state.value} onChange={this.handleChange}>

{this.state.el.map (el =>

<option value={el} key={el}>{el}</option>

)}

</select>

**Вставка**

Обычно компонентам типа Modal и т.д. нужно как-то получить дочерние элементы. Это можно сделать через отдельное свойство пропса children.

function Modal({ children }) {

return <div style={{

border: '1px solid #ccc',

padding: '1.5rem'

}}>

{children}

</div>

}

function App() {

return <div>

<Modal>

<h1>Modal</h1>

</Modal>

}

Здесь переданный JSX будет рендерится внутри компонента, поэтому он может получить доступ к состоянию и пропсам компонента Modal, но не сможет получить этого у App.

Зачастую children хватает, но если нужно вставить несколько и более JSX, то нужно реализовать это через параметры:

function Card({ title, content }) {

return <div className='Card'>

<div className='Card-title'>{title}</div>

<div className='Card-content'>{content}</div>

</div>

}

function App() {

return <div>

<Card

title={

<h1>Title</h1>

}

content={

<div>...</div>

}

/>

</div>

}

**Специализация**

Компонент Password может содержать очень много схожий логики с компонентом Input, поэтому вместо копирования он может просто вернуть Input, немного его изменив. Такое поведение называется специализация.

function Password(props) {

return <Input type='password' {...props}/>

}

function Input({ type, value }) {

return <input type={type} value={value}/>

}

function App() {

return <div>

<Password value={'secret'}/>

</div>

}

**Наследование компонентов**

Классовые компоненты могут наследоваться:

class Button extends React.Component {

get content () {

return 'Content: ' + this.props.content

}

render () {

return <button>{this.content}</button>

}

}

class SuperButton1 extends Button {

get content () {

return super.content + ' (Super)'

}

}

Но это неправильно и влечет к большим потерям в производительности, и ухудшают поддержку кода. Такой код сложно изменить.

За место него используют композицию, специализацию и вставки. Если же повторяется функционал не связанный с внешним видом, то его следует вынести в отдельный файл и просто импортировать.

Тем более классовые компоненты сейчас нужно использовать только, как корневой компонент, а все остальное следует делать через функциональные компоненты.

Например, через специализацию данный пример можно переписать так:

function formatDefault(text) {

return `content: ${text}`

}

function superFormat(text) {

return `super: ${text}`

}

function Button({ text, format=formatDefault}) {

return <button>{format(text)}</button>

}

function SuperButton({ text }) {

return <Button text={text} format={superFormat}/>

}

function App() {

return <div>

<Button text={'secret'} />

<SuperButton text={'secret'} />

</div>

}

**Компоненты высшего порядка (HOC)**

Компоненты высшего порядка HOC – это способ повторения какой-то логики, основанный на создании компонентов, которые никак не изменяют внешний вид, но добавляют логики.

В отличие от обычно компонента, HOC это чистая функция, которая получает компонент и возвращает новый компонент.

const withApi = Component => props => {

const [data, setData] = useState([12, 42, 42])

return <Component data={data} {...props} />

}

function List({ data, title }) {

return <div>

<h1>{title}</h1>

{data.join(' ')}

</div>

}

function App() {

const Component = withApi(List)

return <div>

<Component title='List' />

</div>

}

В этом примере был создан HOC withApi, который равен функции с параметром Component, которая возвращает функцию, принимающую пропсы.

Во вложенной функции создается необходимая логика, которая после передается новому компоненту.

**Фрагменты**

Компонент высшего порядка React.Fragment позволяет создавать компоненты, которые не содержать корневой элемент.

function Car({ title, year }) {

return <React.Fragment>

<h1>{title}</h1>

<h3>{year}</h3>

</React.Fragment>

}

function App() {

return <div>

<Car title='Podeba' year={1967} />

</div>

}

Здесь в компонент Fragment обворачиваются элементы, которые вставятся без лишней обвертки. В новых стандартах это облегчили до такого:

function Car({ title, year }) {

return <>

<h1>{title}</h1>

<h3>{year}</h3>

</>

}

Компонент Fragment очень простой. Его реализация занимает три строки:

function Fragment(props) {

return props.children

}

**Контекст**

Пропсы не всегда удобны, особенно когда передаем пропсы в компонент, чтоб эти пропсы получил его потомок. Здесь создается лишняя прослойка и увеличивается код.

Чтоб избежать этих прослоек были созданы контексты. Контекст сам по себе представляет просто хранилище методов или свойств. Контекст состоит из поставщика (Provider), который добавляет и потребителя (Consumer).

Например, при разработке приложения есть возможность темной темы. Передавать значение свойства темы во все параметры, всего приложения, будет трудно, поэтому значение темы можно положить в контекст и в элементах, которым нужно значение темы, они смогут его просто получать.

Для начала следует в отдельном файле создать сам контекст и его экспортировать.

import React from 'react'

const Context = React.createContext({})

export default Context

После этого в App передадим свойство темы через компонент Context.Provider:

import React from 'react'

import Button from './Button/Button'

import Context from './context'

class App extends React.Component {

state = {

theme: 'black'

}

render() {

return (

<Context.Provider value={this.state.theme}>

<div className='App'>

<Button text='Home' />

</div>

</Context.Provider>

)

}

}

export default App

Компоненту Context.Provider необходимо передать что-то в параметр value. После этого, все компоненты, объявленные внутри Context.Provider смогут через этот же контекст получить значение темы.

import React from 'react'

import Context from '../context'

class Button extends React.Component {

render() {

return <Context.Consumer>

{value => <button>{this.props.text} {value}</button>}

</Context.Consumer>

}

}

export default Button

Этот способ получения контекста заключается в использовании компонента Context.Consumer, которому нужно передать выражения для рендеринга.

Этот способ довольно-таки неудобный, т.к. в нем использование контекста ограниченно разметкой, поэтому есть более «глобальный».

import React from 'react'

import Context from '../context'

class Button extends React.Component {

static contextType = Context

render() {

return <button>{this.props.text} {this.context}</button>

}

}

export default Button

Здесь указываем контекст в статическом свойстве contextType, а потом сам контекст можно получить в свойстве context.

**Порталы**

Портал позволяет рендерить JSX в какой-либо DOM узел.

Например, нужно разработать окно, чтоб в нем можно было выводить сообщение. Можно создать обычный компонент и поместить его в корень приложения, но здесь нужно реализовать все так, чтоб вы всем приложении был только один этот компонент.

Правильнее в данной ситуации будет в html еще один DOM узел, в который будет рендерится сообщение с помощью порталов. Синтаксис порталов:

**ReactDOM.createPortal(child, domNode)**

Это все, что касается порталов, но нужно помнить, что domNode – это именно элемент из document, а не React элемент.

const message = document.getElementById('message')

function Message({ children }) {

return ReactDOM.createPortal(

<div className='Message'>

{children}

</div>,

message

)

}

function App() {

const [isToggle, setToggle] = useState(false)

return <div>

{

isToggle

&& <Message>Hello World</Message>

}

<button onClick={() => setToggle(!isToggle)}>Show</button>

</div>

}

В данном примере содержимое компонента сообщения будет рендерится отдельный тег с id message.

**Рендер-пропсы**

Рендер-пропсы позволяют повторять код, когда есть одинаковая логика, но разный render.

Например, есть компоненты DataList и DataInline, которые по-разному выводят одинаковые данные и можно не повторять логику получения данных, а просто вынести в компонент Data, который с помощью рендер-пропсом выведет данные, как нужно.

function Data({ render }) {

const [data, setData] = useState([1, 6, 9, 15, 53])

return render({ data })

}

function DataList({ data }) {

return <div>

<ul>

{data.map(el => <li key={el}>{el}</li>)}

</ul>

</div>

}

function DataInline({ data }) {

return <p>{data.join(' ')}</p>

}

function App() {

return <div>

<Data render={props => <DataList {...props}/>}/> {/\* Рендер-пропс \*/}

<Data render={props => <DataInline {...props}/>}/> {/\* Рендер-пропс \*/}

</div>

}

**Рефы**

Рефы или Референции позволяют получить какой-то элемент напрямую.

class Car extends React.Component {

constructor(props) {

super(props)

this.inputRef = React.createRef()

}

inputHandle = event => {

this.props.setName(event.target.value)

}

componentDidMount() {

this.inputRef.current.focus()

}

render() {

return <div>

<h1>{this.props.name}</h1>

<input ref={this.inputRef} type='text' onChange={this.inputHandle} />

<h1>{this.props.year}</h1>

</div>

}

}

Сначала нужно создать объект референции через метод React.createRef, а потом указать его в атрибуте ref. Получить данный элемент можно вызвав свойство current.

Может возникнуть соблазн его использовать везде. От этой идеи лучше отказаться, ведь ref используется в редких случаях.

* работа с Canvas, SVG и т.д.
* фокусирование, видео
* для работы плагинов

**Неуправляемые компоненты**

Управляемые компоненты сочетали в себе стандартное поведение и пользовательское состояние, но есть неуправляемые компоненты, которые полностью полагаются, а данные берутся с помощью рефов.

function App() {

const inputRef = useRef()

function submit(event) {

event.preventDefault()

console.log(inputRef.current.value)

}

return <div>

<form onSubmit={submit}>

<input type='text' defaultValue='Vasy' ref={inputRef}/>

<button type='submit'>Submit</button>

</form>

</div>

}

Нужно отметить, что использование value в неуправляемом компоненте категорически запрещено, поэтому используется defaultValue. Так, когда же надо использовать неуправляемые компоненты:

* если элемент всегда неуправляем. Например, file
* если форма простоя

**Перенаправление рефов**

Перенаправление рефов позволяет управлять элементами дочерних компонентов, путем протаскивая рефа в компоненты, созданные через React.forwardRef.

Перенаправление рефа часто применяется при работе с Веб Компонентами, типа canvas, video и т.д.

const Canvas = React.forwardRef(

(props, ref) => <canvas ref={ref} />

)

function App() {

const canvasRef = useRef()

return <div>

<Canvas ref={canvasRef} />

</div>

}