Введение в JSX

ЭЛЕМЕНТЫ (ОНИ НЕ ИЗМЕНЯЕМЫ!!!)

Создав однажды элемент, вы не сможете изменить его дочерние элементы или атрибуты. Элемент похож на один кадр в фильме: он представляет собой пользовательский интерфейс в определённый момент времени.

**Элементы — это то, что из чего компоненты «сделаны»**

Самый простой пример кода для React:

ReactDOM.render(

<h1>Привет, мир!</h1>,

document.getElementById('root')

);

Рассмотрим объявление такой переменной.

const ***element*** = <h1>Hello world</h1>

Такой код называется JSX, и это расширение синтаксиса JavaScript. Он не обязателен для JS, Но оч рекомендуем.

В приведённом ниже примере мы объявляем переменную с именем name, а затем используем ее внутри JSX, обернув ее в фигурные скобки:

const ***name*** = "Josh Perez"  
const ***element*** = <h1>Hello, {***name***}</h1>  
  
ReactDOM.render(  
 ***element***,  
 ***document***.getElementById('root')  
);

В фигурных скобках JSX вы можете поместить любое корректное [выражение JavaScript](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Expressions_and_Operators#Expressions).

Т.е. мы создали переменную name, const, использовали данные из переменной, чтобы отрисовать element

Может сделать так:

function formatName(user) {  
 return user.firstName + " " + user.lastName;  
}  
  
const ***user*** = {  
 firstName: "Harper",  
 lastName: "Perez"  
}  
  
const ***element*** = (  
 <h1>Hello, {formatName(***user***)}!</h1>  
);  
  
ReactDOM.render(  
 ***element***,  
 ***document***.getElementById('root')  
)

После компиляции выражения JSX становятся обычными вызовами функций JavaScript и вычисляются в объекты JavaScript.

Это означает, что вы можете использовать JSX внутри операторов if иfor, присваивать его переменным, принимать его в качесиве аргументов и возвращать из функций:

**Установка атрибутов с помощью JSX**

Вы можете использовать кавычки для указания строковых литералов в качестве атрибутов:

Вы также можете использовать **фигурные скобки для вставки JavaScript**-выражения в атрибут:

const ***avatars*** = {  
 avatarUrl: "hhtps://mail.ru"  
}  
  
const ***element*** = <img src={***user***.avatarUrl}></img>

Имена свойств - camelCase

Теги JSX могут содержать **дочерние элементы:**

const ***element*** = (  
 <div>  
 <h1>Hello!</h1>  
 <p>Nice to meet you!</p>  
 </div>  
)

**Babel компилирует JSX в вызовы React.createElement().**

ОТРИСОВКА ЭЛЕМЕНТОВ

Чтобы отрисовать элементы, нужно поместить их в корневой узел ReactDOM, например

переменная НЕ ВСПЛЫВАЕТ!!!

const ***element*** = (  
 <div>  
 <h1>Hello!</h1>  
 <p>Nice to meet you!</p>  
 </div>  
)  
  
ReactDOM.render(  
 ***element***,  
 ***document***.getElementById('root')  
)

Компоненты не изменяемы, поэтому единственный способ обновить содержимое корневого узла root - это вызвать reactDOM.render() определенное количество раз. Например:

function tick() {  
 const element = (  
 <div>  
 <h1>Привет, мир!</h1>  
 <h2>Сейчас {new ***Date***().toLocaleTimeString()}.</h2>  
 </div>  
 );  
 ReactDOM.render(element, ***document***.getElementById('root'));  
}  
  
setInterval(tick, 1000);

DOM React сравнивает элемент и его дочерние элементы с предыдущими и применяет только обновления DOM, необходимые для преобразования DOM в желаемое состояние.

Несмотря на то, что мы создаём элемент, описывающий всё дерево пользовательского интерфейса на каждом тике, только текстовый узел, содержимое которого изменилось, обновляется в DOM React.

COMPONENTS

**Компоненты** позволяют разделить пользовательский интерфейс на независимые, повторно используемые части и работать с каждой из частей отдельно.

**Концептуально компоненты похожи на функции JavaScript.** Они принимают произвольные входные данные (**называемые «props» или свойствами**) и возвращают React-элементы, описывающие, что должно появиться на экране.

***Функциональные компоненты***

Самый простой способ определить компонент — написать JavaScript-функцию:

function Welcome(props) {

return <h1>Привет, {props.name}</h1>;

}

такие компоненты мы называем «функциональными», потому что они являются буквально **функциями JavaScript.**

***Классовые компоненты***

class Welcome extends React.Component {

render() {

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

}

}

Два вышеуказанных компонента эквивалентны с точки зрения React.

ОТРИСОВКА КОМПОНЕНТОВ

Когда React видит элемент, представляющий **пользовательский компонент**, он передаёт JSX-**атрибуты** этому компоненту **в виде единственного объекта**. Мы называем этот объект **«props».**

Например, этот код отображает «Привет, Сара» на странице:

function Welcome(props) {  
 return <h1>Hello, {props.name}</h1>  
}  
  
const ***element*** = <Welcome name="Sara"/> //пользовательский компонент передает атрибуты этого компонента в виде объекта  
  
ReactDOM.render(***element***, ***document***.getElementById('root'))

Давайте посмотрим, что происходит в этом примере:

1. Мы вызываем ReactDOM.render() с элементом <Welcome name="Сара" />.
2. React вызывает компонент Welcome с объектом {name: 'Sara'} как props.
3. Наш компонент Welcome возвращает элемент <h1>Hello, Sara</h1> в качестве результата.
4. React DOM эффективно обновляет DOM, чтобы соответствовать <h1>Hello, Sara</h1>.

**Примечание: Всегда именуйте компоненты с заглавной буквы.**

Компоненты могут ссылаться на другие компоненты в своём выводе.

Порядок передачи данных в компонент:

1. Например, мы создали объект с данными:

const ***comment*** = {  
 date: new ***Date***(),  
 text: "Hope, you like React",  
 author: {  
 name: "Hello Kitty",  
 avatarUrl: 'https://placekitten.com/g/64/64'  
 }  
}

Нам нужно использовать эти данные для их представления в функциональном компоненте.

1. Вызов рендера ReactDOM.render(). Ниже вызов без данных

ReactDOM.render(<Comment />, ***document***.getElementById('root'))

При вызове рендера мы передаем в отрисовку компонент, чтобы передать в этот компонент данные из объекта comment нам нужно сделать следующее.

1. Используя атрибуты компоненты <Comment/>, мы КАК БЫ СОЗДАЕМ НОВЫЙ ОБЪЕКТ “PROPS” из атрибутов. Атрибуты и значения атрибутов, например name = {object.name} будут ключами-значениями передаваемого в компонент данных.

Т.е. у нас есть объект:

const ***comment*** = {  
 date: new ***Date***(),  
 text: "Hope, you like React",  
 author: {  
 name: "Hello Kitty",  
 avatarUrl: 'https://placekitten.com/g/64/64'  
 }  
}

на его основе мы создаем новый объект из ключей-значений:

ReactDOM.render(<Comment   
 date={***comment***.date}   
 text={***comment***.text}   
 author={***comment***.author}/>, ***document***.getElementById('root'))

(Атрибуты это будут ключ-значения нового объекта)ю Мы не видим создание этого объекта.

const ***props*** = {  
 date: ***comment***.date,  
 text: ***comment***.text,  
 author: ***comment***.author  
}

Такой объект props затем передается в компонент для отрисовки, т.е.

значение **props.date** ссылается на значение **comment.date**

значение **props.author.avatarUrl** ссылается на значение **comment. author.avatarUrl**

**и т.д.**

function Comment(props) {  
 return (  
 <div className="Comment">  
 <div className="UserInfo">  
 <div className="Avatar">  
 <img src={props.author.avatarUrl} alt={props.author.name}/>  
 <div className="UserInfo-name">  
 {props.author.name}  
 </div>  
 </div>  
 <div className="Comment-text">  
 {props.text}  
 </div>  
 </div>  
 <div className="Comment-date">  
 {formatDate(props.date)}  
 </div>  
 </div>  
 );  
}

ИЗВЛЕЧЕНИЕ КОМПОНЕНТОВ

Можно разделать компоненты на более мелкие компоненты

Например, у нас есть такой компонент

function Comment(props) {  
 return (  
 <div className="Comment">  
 <div className="UserInfo">  
 <div className="Avatar">  
 <img src={props.author.avatarUrl} alt={props.author.name}/>  
 <div className="UserInfo-name">  
 {props.author.name}  
 </div>  
 </div>  
 <div className="Comment-text">  
 {props.text}  
 </div>  
 </div>  
 <div className="Comment-date">  
 {formatDate(props.date)}  
 </div>  
 </div>  
 );  
}

Мы можем разделить его на части,

извлечем Avatar и другие компоненты

КЛАССОВЫЕ КОМПОНЕНТЫ

Компоненты, определённые как классы, имеют некоторые дополнительные возможности. **Локальное состояние** — это как раз одно из них: эта возможность доступна только классам.

Преобразуем функциональный компонент:

function Clock(props) {  
 return (  
 <div>  
 <h1>Hello, world</h1>  
 <h2>Now is {props.date.toLocaleTimeString()}</h2>  
 </div>  
 )  
}  
  
function tick() {  
 ReactDOM.render(  
 <Clock date={new ***Date***()}/>,  
 ***document***.getElementById('root')  
 )  
}  
  
setInterval(tick, 1000)

В классовый компонент.

***Преобразование функции в класс:***

Преобразовать функциональный компонент, такой как Clock, в классовый компонент можно за пять шагов:

1. Создать ES6-класс с тем же самым именем, который расширяет React.Component.
2. Добавить к нему пустой метод render().
3. Перенести тело функции в метод render().
4. Заменить props на this.props в теле render().
5. Удалить оставшиеся пустое объявление функции.

// БЫЛО

function Clock(props) {  
 return (  
 <div>  
 <h1>Привет, мир!</h1>  
 <h2>Сейчас {props.date.toLocaleTimeString()}.</h2>  
 </div>  
 );  
}

//СТАЛО

class Clock extends React.Component {  
 render() {  
 return (  
 <div>  
 <h1>Hello, world</h1>  
 <h2>Now is {this.props.date.toLocaleTimeString()}</h2>  
 </div>  
 );  
 }  
}

**Clock** теперь определён как класс, а не функция.

Теперь добавим СОСТОЯНИЕ к классу CLOCK

1. Заменить this.props.date на this.state.date в методе render():

class Clock extends React.Component {  
 render() {  
 return (  
 <div>  
 <h1>Hello, world</h1>  
 <h2>Now is {this.state.date.toLocaleTimeString()}</h2>  
 </div>  
 );  
 }  
}

1. Добавить конструктор класса, который устанавливает начальное состояние в this.state.

Обратите внимание, что мы передаём props базовому (родительскому) конструктору!

class Clock extends React.Component {  
 constructor(props) {  
 super(props);  
 this.state = {date: new ***Date***()}  
 }  
  
 render() {  
 return (  
 <div>  
 <h1>Hello, world</h1>  
 <h2>Now is {this.state.date.toLocaleTimeString()}</h2>  
 </div>  
 );  
 }  
}

1. Удалить свойство date из элемента <Clock />:

Позже мы добавим код таймера обратно к самому компоненту.

ReactDOM.render(  
 <Clock />,  
 ***document***.getElementById('root')  
 )

Итого получаем(часы статичны, не изменяются) :

class Clock extends React.Component {  
 constructor(props) {  
 super(props);  
 this.state = {date: new ***Date***()}  
 }  
  
 render() {  
 return (  
 <div>  
 <h1>Hello, world</h1>  
 <h2>Now is {this.state.date.toLocaleTimeString()}</h2>  
 </div>  
 );  
 }  
}  
  
ReactDOM.render(  
 <Clock/>,  
 ***document***.getElementById('root')  
)

***Добавление методов жизненного цикла в класс***

В приложениях с множеством используемых компонентов очень важно освобождать ресурсы, занятые при их удалении.

Мы хотим **настроить таймер всякий раз**, когда Clock отрисовывается в DOM в первый раз. Это **называется «монтированием» (установкой) в React.**

Мы также хотим **сбрасывать этот таймер** всякий раз, когда DOM, созданный Clock, удаляется. Это **называется «размонтированием» в React.**

Мы можем объявить специальные методы в классе-компоненте для выполнения кода, когда компонент устанавливается и удаляется:

componentDidMount()

componentWillMount()

Это называется **хуками (методами)** жизненного цикла.

Хук **componentDidMount()** запускается **после** того, как вывод компонента отрисован в DOM. Это хорошее место для установки таймера.

Обратите внимание, что мы сохраняем идентификатор таймера в this.

componentDidMount() {  
 this.timerID = setInterval(() => this.tick(), 1000);  
}

Мы удалим таймер в хуке жизненного цикла **componentWillUnmount():**

componentWillMount() {  
 clearInterval(this.timerID)  
}

Наконец, реализуем метод tick(), который компонент Clock будет запускать каждую секунду.

Он будет использовать this.setState() для планирования обновлений локального состояния компонента:

tick() {  
 this.setState({  
 date: new ***Date***()  
 });  
}

Давайте быстро повторим, что происходит, а также перечислим порядок, в котором вызываются методы:

1. Когда <Clock /> передаётся ReactDOM.render(), React вызывает конструктор компонента Clock. Так как Clock должен отображать текущее время, он инициализирует this.state с объектом, включающим текущее время. Позднее мы обновим это состояние.
2. Затем React вызывает метод render() компонента Clock. Вот как React узнаёт, что должно отображаться на экране. Потом React обновляет DOM, чтобы он соответствовал выводу отрисовки Clock.
3. Когда в DOM вставлен вывод Clock, React вызывает хук жизненного цикла componentDidMount(). Внутри него компонент Clock указывает браузеру настроить таймер для вызова метода tick() компонента один раз в секунду.
4. Каждую секунду браузер вызывает метод tick(). Внутри него компонент Clock планирует обновление пользовательского интерфейса, вызывая setState() с объектом, содержащим текущее время. Благодаря вызову setState() React знает, что состояние изменилось, и снова вызывает метод render(), чтобы узнать, что должно отображаться на экране. На этот раз this.state.date в методе render() будет другим, и поэтому вывод отрисованного компонента будет включать обновлённое время. React обновляет DOM соответствующим образом.
5. Если компонент Clock когда-либо удаляется из DOM, React вызывает хук жизненного цикла componentWillUnmount(), чтобы оставить таймер.

***Правильное использование состояния setState()***

Есть три детали о setState(), про которые нужно знать.

1. Нельзя изменять состояние напрямую

// Неправильно

this.state.comment = 'Привет';

// Правильно

this.setState({comment: 'Привет'});

1. Обновления состояния могу быть асинхронными

React может выполнять несколько вызовов setState() за одно обновление для лучшей производительности.

Поскольку this.props и this.state могут обновляться асинхронно, вы не должны полагаться на их значения для вычисления следующего состояния.

// Неправильно

this.setState({

counter: this.state.counter + this.props.increment,

});

// Правильно

this.setState((state, props) => ({

counter: state.counter + props.increment

}));

// Правильно (без стрелочной функции)

this.setState(function(state, props) {

return {

counter: state.counter + props.increment

};

});

1. Обновления состояния объединяются

Когда вы вызываете setState(), React объединяет объект, который вы предоставляете c текущим состоянием.

## Однонаправленный поток данных

Ни родительский, ни дочерний компоненты не могут знать, является ли какой-либо компонент с состоянием или без него, и им не важно, определен ли он как функция или класс.

Вот почему состояние часто называют локальным или инкапсулированным. Оно недоступно для любого компонента, за исключением того, который владеет и устанавливает его.