

PROPS



Дмитрий
Бронских



ДМИТРИЙ БРОНСКИХ

FullStack-developer в ООО "Телеком-Нейва"



[Дмитрий Бронских](#)

ПЛАН ЗАНЯТИЯ

1. [Props](#)
2. [Атрибуты без значений](#)
3. [defaultProps](#)
4. [Условный рендеринг](#)
5. [Списки элементов и атрибут `key`](#)
6. [Render Props](#)
7. [Компоненты без состояния](#)



PROPS

PROPS

Вспомним некоторые ключевые моменты про `props` с предыдущих лекций:

1. `props` передаются в качестве атрибутов в элемент: `<Row item={item}/>`
2. `{}` позволяют передавать сам объект, в противном случае он передастся в качестве строки
3. в `props` можно передавать всё, что угодно: примитивы, объекты, массивы и функции
4. библиотека `prop-types` позволяет контролировать тип `props`
5. внутри компонента `props` можно получить либо как первый аргумент функции, либо как первый аргумент конструктора



АТРИБУТЫ БЕЗ ЗНАЧЕНИЙ

АТТРИБУТЫ БЕЗ ЗНАЧЕНИЙ В HTML

В [стандарте HTML](#) для некоторых DOM-элементов определены атрибуты, которые не содержат своего значения. К примеру:

```
<textarea cols="30" rows="5" disabled>  
<!-- Или -->  
<input name="user" placeholder="Ваше имя" required>
```

В данном случае `disabled` запрещает вносить изменения в поле ввода `textarea`, а `required` помечает поле ввода `input` как обязательное при отправке формы. Таким образом получается, что данные атрибуты конфигурируют DOM-элемент, но не содержат в себе никакого значения (на самом деле значение - пустая строка).

Спецификация JSX также позволяет вносить такого рода атрибуты в ваш компонент, тем самым конфигурировать его удобным способом.



МОДАЛЬНОЕ ОКНО

Рассмотрим на примере модального окна. Предположим, что оно может быть в двух состояниях:

- по умолчанию, где выводит переданную ему информацию
- с ошибкой, где информирует пользователя об ошибке

Как бы вы реализовали такой компонент?

РЕАЛИЗАЦИЯ МОДАЛЬНОГО ОКНА

Скорее всего, ваш компонент выглядел бы так:

```
1 function Modal(props) {  
2   const modalClass = props.danger ? 'danger' : 'default';  
3  
4   return (  
5     <div className={`modal ${modalClass}`}>{ props.text }</div>  
6   )  
7 };  
8 Modal.propTypes = {  
9   danger: PropTypes.bool,  
10  text: PropTypes.string.isRequired,  
11 }
```

И в случае ошибки использовался бы таким образом:

```
1 function App(props) {  
2   const message = 'На стороне сервера произошла ошибка!';  
3   const modal = <Modal text={message} danger={true} />  
4   return (  
5     modal  
6   );  
7 }
```

НАШ КОД МОЖНО УЛУЧШИТЬ

Данный вариант является верным, но с JSX мы можем его улучшить:

```
1  function App(props) {  
2    const message = 'На стороне сервера произошла ошибка!';  
3    const modal = <Modal text={message} danger />  
4  
5    return (  
6      modal  
7    );  
8  }
```

Таким образом, мы просто задали атрибут `danger` без указания значения и получили такой же результат.

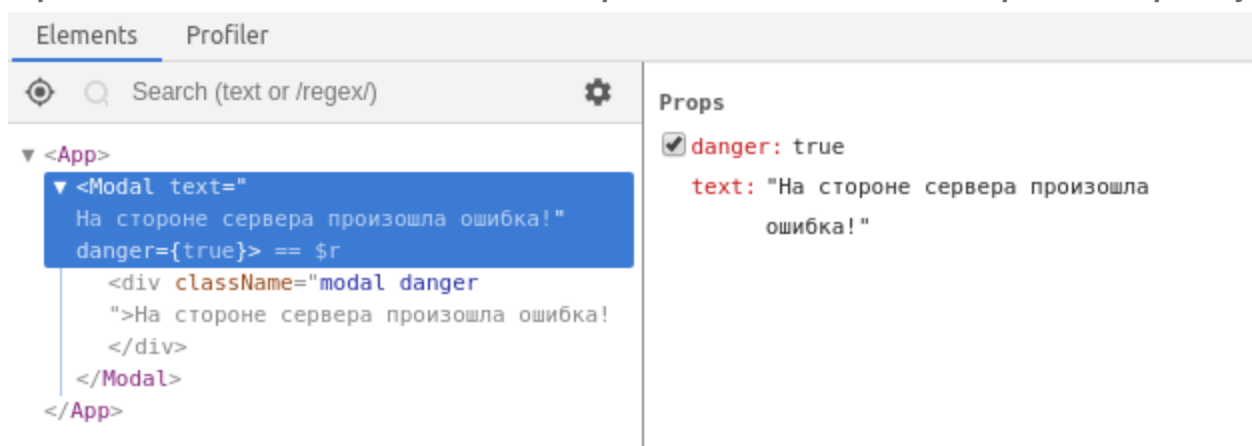
АТТРИБУТ БЕЗ ЗНАЧЕНИЯ РАВЕН `true`

В JSX атрибут без указания значения автоматически преобразуется в атрибут типа `attr={true}`, то есть, принимает значение `true`.

Если мы явно не укажем значение атрибута, оно получит значение `true`. Данный подход часто применяется при конфигурации компонентов под определенные заранее заданные действия.

ПРОСМОТР АТТРИБУТОВ

Часто при разработке нам бы хотелось узнать, что компонент получает в виде атрибутов. Такая задача часто встречается, и в данном случае нам просто необходимо Developer Tools и посмотреть атрибуты:





defaultProps

defaultProps

Свойство компонента `defaultProps` позволяет задать значения атрибутов по умолчанию. Для этого достаточно присвоить объект в это свойство компонента, где ключ будет именем атрибута, а значение — значением, которое будет передаваться в `props` компонента, если атрибут не задан. Рассмотрим на примере:

```
1 // object destructuring из props
2 function UserInfo({name, img}) {
3   return (
4     <div className='UserInfo'>
5       <img src={img} />
6       <div>{name}</div>
7     </div>
8   );
9 };
10 UserInfo.defaultProps = {
11   img: 'img/no-avatar.png'
12 };
13 UserInfo.propTypes = {
14   img: PropTypes.string
15 };
```

ПЕРЕДАДИМ В `UserInfo` ТОЛЬКО ИМЯ

```
1 // файл App.js
2 const user = {
3   name: 'Елена Иванова',
4   img: 'user/elena.png'
5 }
6 return (
7   <UserInfo name={user.name} />,
8 );
```

В этом примере мы создали компонент, который показывает имя и изображение профиля пользователя. Так как изображение есть не у всех пользователей, то нужно задать значение по умолчанию, в нашем случае `img: 'img/no-avatar.png'`. Мы получим такое DOM-дерево:

```
1 <div class="UserInfo">
2   
3   <div>Елена Иванова</div>
4 </div>
```

ПЕРЕДАДИМ АРГУМЕНТ `img`

А что будет, если мы передадим аргумент `img` ? Попробуем:

```
1 // файл App.js
2 const user = {
3   name: 'Елена Иванова',
4   img: 'user/elena.png'
5 }
6
7 return (
8   <UserInfo name={user.name} img={user.img} />,
9 );
```

Теперь подставлена фотография пользователя:

```
1 <div class="UserInfo">
2   
3   <div>Елена Иванова</div>
4 </div>
```


А ЕСЛИ СВОЙСТВА `img` НЕТ?

А что если у пользователя не будет свойства `img` ? Проверим:

```
1  const user = {  
2    name: 'Елена Иванова'  
3  };  
4  
5  return (  
6    <UserInfo name={user.name} img={user.img} />,  
7  );
```

Получается, мы передаем `undefined` в компонент. И в `props` в этом случае приходит значение атрибута по умолчанию, что удобно:

```
1  <div class="UserInfo">  
2      
3    <div>Елена Иванова</div>  
4  </div>
```

`img` РАВНО `null` ИЛИ `false`

Но если передать `null` или `false`, то значение атрибута по умолчанию использоваться не будет:

```
1  const user = {  
2    name: 'Елена Иванова',  
3    img: null,  
4  };  
5  
6  return (  
7    <UserInfo name={user.name} img={user.img} />  
8  );
```

И мы получим пустой атрибут `src` у картинки в DOM:

```
1  <div class="user">  
2    <img>  
3    <div>Елена Иванова</div>  
4  </div>
```



УСЛОВНЫЙ РЕНДЕРИНГ

&& ДЛЯ УСЛОВНОГО РЕНДЕРИНГА

Тот факт, что `false`, `null` и `undefined` не дают никакого вывода, можно использовать для условного рендеринга.

Предположим, нам нужно создать прелоадер, управляемый атрибутом `isLoading`:

```
1  function LoadingIndicator ({ isLoading }) {  
2    return (  
3      <div>  
4        { isLoading && <p>Загрузка...</p> }  
5      </div>  
6    );  
7  }  
8  LoadingIndicator.propTypes = {  
9    isLoading: PropTypes.bool,  
10 }
```

РЕЗУЛЬТАТ ПРЕЛОАДЕРА

```
1  // в App.js
2  return render(
3    <LoadingIndicator isLoading={true} />
4  );
```

Мы видим такой DOM в документе:

```
<p>Загрузка...</p>
```

ОСОБЕННОСТЬ ОБРАБОТКИ `&&` В JS

Почему это работает?

В Javascript выражение типа `true && 42` вернет значение `42` — значение последнего элемента выражения. А выражение типа `false && 42` вернет `false` — значение первого элемента.

Это связано с тем, что JavaScript интерпретирует выражение слева направо и конечно же экономит на действиях:

- если левый операнд приводится к `true`, то он вернет значение правого операнда
- если же левый операнд приводится к `false` (`null`, `undefined`), дальше проверять смысла уже нет, и он просто вернет левый операнд как есть

РАЗНЫЕ ЗНАЧЕНИЯ `isLoading`

В React часто используется данная особенность для условного рендеринга, ведь значение `false` не преобразуется в DOM-элемент или строку, также как и `true`, `undefined` или `null`.

```
1 // Показывается
2 <LoadingIndicator isLoading={true} />
3
4 // Не показывается
5 <LoadingIndicator isLoading={false} />
6
7 // Не показывается (isLoading = undefined)
8 <LoadingIndicator />
```

ОПЕРАТОР ||

Другой особенностью JavaScript является обработка выражения `false || 42`. Данное выражение будет равно `42`. Причина все та же. JavaScript проверит левый операнд и если он приводится к `false`, то вернет правый. А выражение `true || 42` вернет `true`. Потому что левый операнд приводится к `true`, и нет смысла проверять дальше, он вернет левый операнд как есть.

Попробуем в JSX:

```
1  function UserInfo (props) {  
2    return <div>{props.name || 'Гость'}</div>  
3  };
```


ПЕРЕДАДИМ ПАРАМЕТРЫ В `UserInfo`

Если мы передадим атрибут `name`, то получим его значение в DOM:

```
1 // в App.js
2 return render(
3   <UserInfo name="Иван" />
4 );
5 // <div>Иван</div>
```

Иначе получим «Гость» в DOM, так как `props.name` будет `undefined`, что приводит к `false`:

```
1 // в App.js
2 return render(
3   <UserInfo />
4 );
5 // <div>Гость</div>
```

ТЕРНАРНЫЙ ОПЕРАТОР ДЛЯ УСЛОВНОГО РЕНДЕРИНГА

Не всем и не всегда эти «хаки» очевидны. И последний пример будет выглядеть понятнее при использовании тернарного оператора:

```
1 function UserInfo (props) {  
2   return <div>{ props.name ? props.name : 'Гость' }</div>  
3 };
```

Но так же, как и в JavaScript, тернарный оператор хорош только в простых небольших выражениях.



СПИСКИ ЭЛЕМЕНТОВ



ПРИМЕР СПИСКА В REACT

Списки используются часто в React-приложении при отображении каких-либо однотипных элементов.

Предположим, у нас есть список дел, и мы хотим представить его в интерфейсе:

РЕАЛИЗУЕМ КОМПОНЕНТ `TodoList`

```
1  function TodoList(props) {
2    const [todos, setTodos] = useState([
3      'Забрать машину',
4      'Купить молоко',
5      'Позвонить на работу'
6    ]);
7    const todoItems = todos.map((item) => <li>{item}</li>);
8    return (<div>
9      <h2>Список дел:</h2>
10     <ul>
11       {todoItems}
12     </ul>
13   </div>);
14 }
```

Обратите внимание: мы вставляем в JSX список React-элементов.

ВЫВОДИМ СПИСОК В КОМПОНЕНТЕ APP

```
1 function App() {  
2   return (  
3     <TodoList />  
4   );  
5 }
```

Но при выполнении возникла проблема — мы получим предупреждение:

```
✖ Warning: Each child in a list should have a unique "key" prop.  
  
Check the render method of `TodoList`. See https://fb.me/react-warning-keys for more information.  
    in li (at TodoList.js:10)  
    in TodoList (at App.js:8)  
    in App (at src/index.js:7)
```

ПРИЧИНЫ ПРЕДУПРЕЖДЕНИЯ

Мы вставили в JSX список элементов. У этих элементов не указан атрибут `key`.

Данный атрибут позволяет однозначно идентифицировать элементы списка в виртуальном дереве. И в дальнейшем при обновлении определить, какие из элементов были добавлены, какие изменены, а какие удалены.

Рассмотрим подробнее.

СПИСОК ИЗМЕНИЛСЯ

Во время работы программы в начало списка был добавлен новый пункт:

```
1  function TodoList(props) {
2    const [todos, setTodos] = useState([
3      'Забрать машину',
4      'Купить молоко',
5      'Позвонить на работу'
6    ]);
7    const todoItems = todos.map((item) => <li>{item}</li>);
8
9    const handleModify = () => {
10     setTodos(prevTodos => ['Новый самый важный пункт', ...prevTodos.slice(1)]
11   );
12
13   return (
14     <div>
15       <h2>Список дел:</h2>
16       <ul>{todoItems}</ul>
17       <button onClick={handleModify}>Modify</button>
18     </div>
19   );
20 }
```


НУЖНО ОБНОВИТЬ DOM-ДЕРЕВО

Без кода сложно понять: `Новый самый важный пункт` — это новый элемент или измененный старый `Забрать машину` ?

Без указания атрибута `key` задача сопоставления элементов списка при обновлении усложняется настолько, что гораздо проще пересоздать все элементы списка заново. Таким образом, существенно снижается производительность.

Так мы теряем преимущество Virtual DOM, которое заключается в том, что React проходит по виртуальному DOM-дереву и выявляет изменения в компонентах, после чего происходит перерисовка только измененных элементов. А ведь элементы списка могут быть достаточно большими.

КЛЮЧИ

Если в предыдущем примере указать атрибут `key` для каждого элемента списка, то React вставит новый элемент, не затрагивая остальных.

При наличии этого аргумента React сравнивает `key` значения новых узлов дерева с `key` значениями текущего и выявляет изменения.

Основное требование к этому атрибуту — быть уникальным в контексте родительского узла и оставаться неизменным. То есть, можно использовать одинаковые ключи для узлов, имеющих разных родителей.

ПРАВИЛЬНОЕ ИСПОЛЬЗОВАНИЕ `key`

Основная задача атрибута `key` — быть уникальным в контексте родительского узла. Поэтому такие ключи допустимо использовать (например, если возьмём список людей):

```
1  <ul>
2    <li key="1">Ларичев Иван</li>
3    <li key="2">Морозов Алексей</li>
4    <li key="3">Бурмистров Роман</li>
5  </ul>
6  <ul>
7    <li key="1">Болдырева Анастасия</li>
8    <li key="2">Чичев Александр</li>
9    <li key="3">Егоров Николай</li>
10 </ul>
```

НЕПРАВИЛЬНОЕ ИСПОЛЬЗОВАНИЕ `key`

Но нельзя использовать одинаковые ключи для узлов, у которых один и тот же родитель:

```
1 <ul>
2   <li key="1">Ларичев Иван</li>
3   <li key="1">Морозов Алексей</li>
4   <li key="2">Бурмистров Роман</li>
5 </ul>
```

ИДЕНТИФИКАЦИЯ ЭЛЕМЕНТА СПИСКА

Лучший способ подобрать ключ — использовать строку, которая однозначно идентифицирует элемент списка среди остальных.

Чаще всего в качестве ключей вы будете использовать идентификаторы из ваших данных:

```
const todoItems = props.items.map(item =>  
  <li key={item.id}>{item.value}</li>);
```

В данном примере `item` является объектом со свойствами `id` и `value`.

ПОПЫТАЕМСЯ ЗАРАНЕЕ ОПРЕДЕЛИТЬ **key** ЭЛЕМЕНТА

А что если в цикле у нас вставляется наш компонент? Где следует определить атрибут **key**? Попробуем добавить атрибут **key** его корневому элементу:

```
1  function TodoItem(props) {  
2    return (  
3      <li key={props.item.id}>  
4        {props.item.value}  
5      </li>  
6    );  
7  };
```

ПОДСТАВЛЯЕМ СПИСОК КОМПОНЕНТОВ

```
1 function TodoList (props) {  
2   const todoItems = props.items.map(item =>  
3     <TodoItem item={item} />);  
4  
5   return <ul>{todoItems}</ul>;  
6 };
```

В итоге мы получим то же предупреждение:

Warning: Each child in an array or iterator should have a unique "key" prop.

КЛЮЧ ЗАДАЁТСЯ У ЭЛЕМЕНТА СПИСКА

Отсюда следует правило, что `key` атрибут имеет смысл лишь в контексте списка создаваемых элементов или компонентов. И именно этим элементам следует добавлять атрибут `key`.

ЗАДАДИМ КЛЮЧ САМОМУ `TodoItem`

Перепишем наш пример:

```
1  function TodoItem(props) {  
2    return <li>{props.item.value}</li>  
3  };  
4  
5  function TodoList(props) {  
6    const todoItems = props.items.map(item =>  
7      <TodoItem key={item.id} item={item} />)  
8  
9    return <ul>{todoItems}</ul>  
10  };
```

Теперь предупреждения нет. Потому что в цикле создается именно `<TodoItem>`, и `key` теперь является его атрибутом.

КЛЮЧИ НЕ МОГУТ БЫТЬ ИСПОЛЬЗОВАНЫ КАК ПАРАМЕТР КОМПОНЕНТА

Стоит также учитывать, что ключи в React служат для идентификации DOM-узла, но они не доступны в самом компоненте из `props`. То есть, такая реализация не даст ожидаемого результата:

```
1  function TodoItem(props) {  
2    return <li>{props.key}: {props.value}</li>  
3  };  
4  
5  function TodoList(props) {  
6    const todoItems = props.items.map(item =>  
7      <TodoItem key={item.id} value={item.value} />);  
8  
9    return <ul>{todoItems}</ul>  
10  }
```

КЛЮЧ МОЖНО ПЕРЕДАТЬ ОТДЕЛЬНЫМ АРГУМЕНТОМ

`key` не передаётся в `props`. Поэтому мы передали атрибут `id`:

```
1 function TodoItem(props) {
2   return <li>{props.id}: {props.value}</li>;
3 }
4
5 function TodoList(props) {
6   const todoItems = props.items.map((item) =>
7     <TodoItem
8       key={item.id}
9       id={item.id}
10      item={item.value} />
11   );
12
13   return <ul>{todoItems}</ul>;
14 }
```

ИСПОЛЬЗОВАНИЕ ИНДЕКСА МАССИВА В КАЧЕСТВЕ КЛЮЧА НЕ РЕКОМЕНДУЕТСЯ

Использование индекса массива в качестве `key` является плохой практикой.

Вы догадываетесь, почему?

ЧТО ЛУЧШЕ ИСПОЛЬЗОВАТЬ В КАЧЕСТВЕ КЛЮЧЕЙ

Рекомендуется использовать уникальный идентификатор объекта из базы данных. Но что делать, если мы получаем данные не из базы данных, или у них нет идентификаторов?

Тогда перед использованием списка можно сгенерировать уникальный и статичный идентификатор для каждого элемента. Для таких задач разработаны модули `shortid` и `react-key-index`. Рассмотрим, как их использовать.

react-key-index

```
npm install react-key-index
```

```
yarn add react-key-index
```

```
1  import keyIndex from 'react-key-index';
2
3  // Список продуктов
4  const groceries = ["яйца", "хлеб", "сыр"];
5
6  // Генерируем новый список
7  const output = keyIndex(groceries, 1);
8
9  // Использование
10 const list = output.map((grocery) =>
11   <li key={grocery._id}>{grocery.value}</li>
12 );
```

СГЕНЕРИРОВАННЫЙ СПИСОК С `_id`

Переменная `output` содержит массив объектов, где для каждого из них определены свойства `_id` для идентификации и `value` для значения. В итоге для нашего примера `output` будет выглядеть так:

```
1  [{
2    value: "яйца",
3    _id: "jRfM"
4  }, {
5    value: "хлеб",
6    _id: "kRf0"
7  }, {
8    value: "сыр",
9    _id: "lYfo"
10 }];
```

А ЕСЛИ В МАССИВЕ ОБЪЕКТЫ?

В данном случае массив состоял только из строк, а что будет, если он будет состоять из объектов, к примеру, данного типа:

```
1  [{  
2    item: "молоко",  
3    cost: 10000  
4  }, {  
5    // ...  
6  }];
```


УНИКАЛЬНЫЙ `_id` ДЛЯ КАЖДОГО СВОЙСТВА

Тогда на выходе для каждого такого объекта будет сопоставляться новый со следующими свойствами. Таким образом, для каждого свойства из объекта создан свой уникальный идентификатор:

```
1  [{  
2    item: "молоко",  
3    itemId: "kwH4H0HgH8HZfV",  
4    cost: 10000,  
5    costId: "X7HGHrHLf8cEf3"  
6  }, {  
7    // ...  
8  }];
```

shortid

```
npm install shortid
```

```
yarn add shortid
```

Данный модуль также легко использовать, только отличие в том, что мы сами задаем свойство, где будет находиться идентификатор элемента. Рассмотрим также для нашего списка:

```
1  const groceries = ["яйца", "хлеб", "сыр"];
2
3  const groceriesIndex = groceries.map((grocery) =>
4    ({id: shortid.generate(), value: grocery }));
5
6  const list = groceriesIndex.map((grocery) =>
7    <li key={grocery.id}>{grocery.value}</li>);
```

Использование `shortid` является более предпочтительным, так как вы можете сами задать свойства с уникальным значением ключа.

ВЫВОДЫ ПО `key`

- `key` позволяет однозначно идентифицировать элемент среди однотипных элементов в контексте списка;
- `key` имеет смысл в контексте списка и не может быть определен в самом компоненте;
- `key` атрибут не может быть использован в самом компоненте, нужно задавать атрибут `id`;
- `key` атрибут должен быть неизменяемым и уникальным во время выполнения программы.

ОТОБРАЖЕНИЕ СПИСКА ПОКУПОК

Используя значения атрибутов по умолчанию и навыки по отображению списков, мы можем сделать наши компоненты еще более универсальными.

Например, давайте реализуем возможность настраивать шапку и используемую валюту в компоненте `CartTable`.

CARTTABLE

```
1 function CartTable({items, headings, currency}) {
2   const tableItems = items.map(item => (
3     <tr key={item.id}>
4       <td>{item.name}</td> <td>{item.price}</td> <td>{item.quantity}</td>
5     </tr>
6   ))
7   const header = <tr>{headings.map((title, index) =>
8     <td key={index}>{title}</td>)}</tr>
9   const getTotalPrice = items => items.reduce((prev, item) =>
10     prev + item.price * item.quantity, 0)
11
12   return (<table>
13     <thead> {header} </thead>
14     <tbody> {tableItems}
15     <tr><td colSpan={3}>К оплате: {currency} {getTotalPrice(items)}</td></tr>
16   </tbody>
17   </table>);
18 }
19
20 CartTable.defaultProps = {
21   headings: ['Название товара', 'Цена', 'Количество'],
22   currency: '$'
23 }
```

РЕНДЕРИМ CartTable

По умолчанию шапка таблицы имеет поля названия товара, цены и количества. Если нас это устраивает, мы можем их не задавать при использовании компонента. Мы определили их по умолчанию, также мы задали валюту по умолчанию в долларах.

```
1 // файл App.js
2 const cartItems = [
3   {id: 124, name: 'Iphone 8', price: 1034, quantity: 1},
4   {id: 221, name: 'MacBook Pro 13', price: 2720, quantity: 2},
5   {id: 371, name: 'Samsung Galaxy S8', price: 600, quantity: 3}
6 ];
7
8 return (
9   <CartTable items={cartItems} />
10 );
```

КАСТОМИЗАЦИЯ

А что делать, если нам надо изменить шапку или валюту?

Для этого достаточно будет задать атрибут `currency` или `headings` или оба:

```
1  ...  
2  <CartTable  
3    items={cartItems}  
4    headings={['Товар', 'Цена в рублях', 'Количество']}  
5    currency='руб.' />  
6  ...
```

Таким образом, вместо значений по умолчанию мы получим значения атрибутов. В общем случае `defaultProps` отражают состояние компонента по умолчанию, то есть, представляют собой его дефолтную конфигурацию и являются нужным средством при создании действительно удобных и часто используемых компонентов.



RENDER PROPS

RENDER PROPS

Ещё одной техникой создания переиспользуемых компонентов, является техника Render Props, когда в качестве одно из `props` мы передаём функцию, результат которой используем для рендеринга.

Рассмотрим на примере: допустим, у нас есть компонент `AccountHolder`, который умеет получать данные (список счетов) с удалённого сервиса. И мы хотим научиться отображать его содержимое в различных форматах (таблица, карточки), но при этом сам компонент, мы изменять не хотим.

RENDER PROPS

```
1  function AccountsHolder(props) {
2    const [accounts, setAccounts] = useState([
3      {id: 1, name: '**** *464', balance: 10000}
4    ]);
5    return (
6      <Fragment>{props.render(accounts)}</Fragment>
7    );
8  }
9
10 function App() {
11   return (
12     <AccountsHolder render={accounts => (
13       <ul>
14         { accounts.map(o => <li key={o.id}>{o.name}: {o.balance}</li>) }
15       </ul>
16     )} />
17   );
18 }
```

RENDER PROPS

Таким образом, мы можем передавать в `render` любую функцию и менять результат (генерируемое отображение), не изменяя сам компонент `AccountHolder`.



КОМПОНЕНТЫ БЕЗ СОСТОЯНИЯ

КОМПОНЕНТЫ - СТРУКТУРНЫЕ ЕДИНИЦЫ ВАШЕГО ИНТЕРФЕЙСА

В React-приложении компоненты именуются согласно *UpperCamelCase* или *PascalCase* (ВерблюжийРегистр или ГорбатыйРегистр) — стиль написания составных слов, при котором несколько слов пишутся слитно без пробелов, при этом каждое слово внутри фразы пишется с заглавной буквы.

Например, если мы хотим определить компонент, который отвечает за вывод информации о пользователе, то нам следует написать его таким образом: `UserInfo`, а не таким `userInfo` или `User_info`.



ПОВТОРНОЕ ИСПОЛЬЗОВАНИЕ КОМПОНЕНТОВ

Одна из задач при построении приложения с использованием React состоит в том, чтобы была возможность повторного использования компонентов. Такой подход помогает контролировать разрастание приложения, а также существенно ускоряет разработку.

При этом React дает возможность создавать простые компоненты максимально простым способом — создавая функцию.

STATELESS

Stateless Components — это компоненты не имеющие собственного состояния и предназначенные только для работы с `props` (отображение, вызов функций и т.д.).

Напоминаем, `props` - read only и модифицировать их нельзя!

Раньше под Stateless Component'ами понимались Function Component'ы, т.к. до версии 16.8 они не могли иметь своего состояния. Но с выходом 16.8 появились хуки, которые позволяют работать с состоянием. Поэтому проводить параллель между Stateless Components и Functional components уже нельзя.

В рамках лекций, чтобы не запутаться, мы будем употреблять этот термин для Functional Component'ов, которые не имеют собственного состояния (не используют `useState`).



ПРЕИМУЩЕСТВО КОМПОНЕНТОВ БЕЗ СОСТОЯНИЯ

- проще поддерживать,
- легче читаются и понимаются,
- легко тестируются.

Разбивая приложение на маленькие простые компоненты, вы упрощаете его поддержку и развитие.

Рассмотрим простой пример. Допустим, на сайте нам нужно отобразить таблицу пользователей с их адресами электронной почты.

НАЧНЕМ С ДАННЫХ

Допустим, каждый пользователь представлен в виде объекта:

```
1 // файл App.js
2 const emails = [
3   {id: 124, name: 'Григорий', email: 'dikiigr@gmail.com'},
4   {id: 221, name: 'Алена', email: 'alenable@gmail.com'},
5   {id: 371, name: 'Сергей', email: 'sergiy@yandex.ru'}
6 ];
7
8 return render(
9   <EmailTable emails={emails} />
10 );
```

И реализуем вывод. Просто используем компонент `EmailTable` и передадим список адресов в атрибут `emails`.

КОМПОНЕНТ ТАБЛИЦЫ

Осталось реализовать сам компонент, отвечающий за вывод:

```
1  function EmailTable(props) {  
2    const tableItems = props.emails.map(item => (  
3      <tr key={item.id}>  
4        <td>{item.name}</td>  
5        <td>{item.email}</td>  
6      </tr>  
7    ));  
8  
9    return (  
10     <table>{tableItems}</table>  
11   );  
12 }
```

РАССМОТРИМ `EmailTable` ПОДРОБНЕЕ

Сам компонент довольно просто организован: список адресов на входе, таблица на выходе.

Но при этом компонент универсален. Он может показать нам любой список пользователей, который мы ему передадим в едином формате. Поэтому мы можем его повторно использовать его в любой части приложения, каждый раз, когда нам нужно показать такую таблицу.

ЛОГИЧЕСКИЙ ТИП, `null` И `undefined`

А что будет, если в нашем списке пользователей не будет части полей? Например, поля `name` или `email`. Или их значения будут равны `false`, `true` или `null`?

Возможно, что у нас определены следующие данные:

```
1  const emails = [  
2    {id: 124, name: 'Григорий', email: true},  
3    {id: 221, email: 'alenabl@gmail.com'},  
4    {id: 371, name: null, email: false}  
5  ];
```

РЕЗУЛЬТАТ ВЫВОДА

Вывод списка компонентом `EmailTable` даст такой результат:

```
1  <table>
2    <tr>
3      <td>Григорий</td> <td></td>
4    </tr>
5    <tr>
6      <td></td> <td>alenable@gmail.com</td>
7    </tr>
8    <tr>
9      <td></td> <td></td>
10   </tr>
11 </table>
```

Значения `true`, `false`, `null` и `undefined` не дают никакого вывода.

«ПУСТОЙ КОМПОНЕНТ»

Используем это в компоненте, который не должен рендериться при определенных условиях:

```
1 // файл Block.js
2 function Block(props) {
3   if (!props.visible) {
4     return null;
5   }
6   return <h1>Привет мир</h1>;
7 };
8
9 // файл App.js
10 return render(
11   <Block visible={false} />
12 );
```



СОСТОЯНИЕ

Старайтесь только в необходимых ситуациях хранить состояние, поскольку именно работа с состоянием является самой сложной частью, подверженной наибольшему количеству ошибок.



ИТОГИ

АТТРИБУТЫ И ПАРАМЕТРЫ ПО УМОЛЧАНИЮ

- В JSX атрибут без указания значения автоматически преобразуется в атрибут типа `attr={true}`, то есть принимает значение `true`.
- Свойство компонента `defaultProps` позволяет задать значения атрибутов по умолчанию.
- Используя значения атрибутов по умолчанию, мы можем сделать наши компоненты еще более универсальными.

УСЛОВЫЙ РЕНДЕРИНГ

- Значения `true`, `false`, `null` и `undefined` не приводятся к строкам или к чему-либо еще. Они просто не дают никакого вывода. Это можно использовать в компоненте, который не должен рендериться при определенных условиях.
- Операторы `&&` и `||` также могут быть использованы для условного рендеринга.
- Кроме того, вы можете внутри `render` использовать `if` делая код более понятным.

key

- `key` позволяет однозначно идентифицировать элемент среди однотипных элементов в контексте списка.
- `key` имеет смысл в контексте списка и не может быть определен в самом компоненте.
- `key` атрибут не может быть использован в самом компоненте, нужно задавать атрибут `id`.
- `key` атрибут должен быть неизменяемым и уникальным во время выполнения программы.



КОМПОНЕНТЫ БЕЗ СОСТОЯНИЯ

- Одна из задач при построении приложения с использованием React состоит в том, чтобы была возможность повторного использования компонентов.
- Компоненты без состояния отлично подходят для решения этой задачи: они более универсальны, их проще поддерживать, они легче читаются и понимаются.



Спасибо за внимание! Время задавать вопросы

ДМИТРИЙ БРОНСКИХ

 Дмитрий Бронских