

КОМПОЗИЦИЯ КОМПОНЕНТОВ



Дмитрий
Федин



Дмитрий Федин

Frontend Engineer в Dasha.AI



ПЛАН ЗАНЯТИЯ

1. [Введение в композицию компонентов](#)
2. [Подходы к композиции компонентов](#)
3. [Передача компонентов в атрибуты](#)
4. [Вложенные компоненты - Children Components](#)
5. [Композиция списков](#)
6. [Функция как дочерний компонент](#)
7. [Spread оператор - оператор разворота](#)

ВСПЛЫВАЮЩИЙ БЛОК

Допустим у нас на проекте мы хотим показывать горячие предложения всплывающим блоком:

```
1 function TodayOfferModal() {  
2   return (  
3     <div className="modal">  
4       <div className="modal-body">  
5         <h2>Только сегодня и только сейчас!</h2>  
6         <a href="/today/">Узнать подробнее</a>  
7       </div>  
8       <button className="modal-close">закрыть</button>  
9     </div>  
10  );  
11 }
```

Компонент может выглядеть примерно так.

ЧТО ЕСЛИ ТАКИХ ПРЕДЛОЖЕНИЙ БУДЕТ МНОГО?

Чувствуете проблему, которая будет нарастать как снежный ком с каждой новой акцией?

```
1  function DiscountOfferModal() {  
2    return (  
3      <div className="modal">  
4        <div className="modal-body">  
5          <h2>Самые большие скидки!</h2>  
6          <a href="/discount/">Узнать подробнее</a>  
7        </div>  
8        <button className="modal-close">закрыть</button>  
9      </div>  
10   );  
11 }
```

ЗАДЕЙСТВУЕМ СИЛУ `props` !

Вынесем заголовок и ссылку в атрибуты нашего компонента:

```
1  function OfferModal(props) {  
2    return (  
3      <div className="modal">  
4        <div className="modal-body">  
5          <h2>{props.title}</h2>  
6          <a href="{props.link}">Узнать подробнее</a>  
7        </div>  
8        <button className="modal-close">закрыть</button>  
9      </div>  
10   );  
11 }
```

ПРОБЛЕМА РЕШЕНА!

Было:

```
1 <TodayOfferModal />
2 <DiscountOfferModal />
```

Стало:

```
1 <OfferModal
2   title="Только сегодня и только сейчас!"
3   link="/today/" />
4 <OfferModal
5   title="Самые большие скидки!"
6   link="/discount/" />
```

НО РАЗВИТИЕ ПРОЕКТА НЕ ОСТАНОВИТЬ

Со временем появится требование создать всплывающий блок с таким кодом:

```
1  <div className="modal">
2    <div className="modal-body">
3      <form action="/abonnement/">
4        <h2>Выбери свой абонемент!</h2>
5        <select name="type">
6          <option value="3">На 3 месяца</option>
7          <option value="6">На 6 месяцев</option>
8        </select>
9        <button type="submit">Выбрать</button>
10     </form>
11   </div>
12   <button className="modal-close">закрыть</button>
13 </div>
```


КТО ВИНОВАТ И ЧТО ДЕЛАТЬ?

Содержимое нашего всплывающего блока перестало подчиняться изначальному правилу «заголовок + ссылка» и стало свободным: просто какая-то разметка.

В каких направлениях мы можем двигаться:

- Создать отдельный компонент именно под эту акцию.
- Передавать в атрибуты компонент, вместо заголовка и ссылки.
- Передавать вообще HTML разметку как строку.

ОТДЕЛЬНЫЙ КОМПОНЕНТ

Выходит что мы лишь немного притормозили рост компонентов:

```
1  function AbonOfferModal(props) {  
2    return (  
3      <div className="modal">  
4        <div className="modal-body">  
5          <form action="/abonnement/">  
6            { /* ... */ }  
7          </form>  
8        </div>  
9        <button className="modal-close">закрыть</button>  
10     </div>  
11   );  
12 }
```

КОМПОНЕНТ В АТТРИБУТЫ

На первый взгляд всё вроде элегантно:

```
1 function OfferModal(props) {  
2   return (  
3     <div className="modal">  
4       <div className="modal-body">{props.body}</div>  
5       <button className="modal-close">закрыть</button>  
6     </div>  
7   );  
8 }
```

А вот на второй — отвратительно. Как вам JSX внутри JSX:

```
1 <OfferModal body={<LinkOfferModalBody />} />  
2 <OfferModal body={<AbonOfferModalBody />} />
```

ПЕРЕДАЧА HTML СТРОКОЙ

Вставка HTML-кода строкой возможна, но посмотрите как это выглядит:

```
1 | <div dangerouslySetInnerHTML={props.body} />
```

`dangerouslySetInnerHTML` (*dangerously* — опасно) — разработчики React встроили напоминание о том, что нужно стараться этого избегать в саму библиотеку.

Поэтому этот вариант даже не будем рассматривать.



ВЛОЖЕННЫЕ ЭЛЕМЕНТЫ

ВЛОЖИМ ТЕЛО В КОМПОНЕНТ

До этого мы использовали наши компоненты как одиночный самодостаточный элемент. Но ведь в HTML у элементов есть дочерние элементы. А что же JSX? Вложим тело в компонент:

```
1 <OfferModal>
2   <h2>{props.title}</h2>
3   <a href="{props.link}">Узнать подробнее</a>
4 </OfferModal>
```

```
1 <OfferModal>
2   <form action="/abonnement/">
3     { /* ... */ }
4   </form>
5 </OfferModal>
```

ПОХОЖЕ ЧТО ЭТО ВОЗМОЖНО

Ошибок такой код с нашим компонентом не вызывает:

```
1 function OfferModal(props) {  
2   return (  
3     <div className="modal">  
4       <div className="modal-body">{props.body}</div>  
5       <button className="modal-close">закрыть</button>  
6     </div>);  
7   }
```

Но и чуда не случается. Получаем такой HTML-код:

```
1 <div class="modal">  
2   <div class="modal-body"></div>  
3   <button class="modal-close">закрыть</button>  
4 </div>
```

ПОЛУЧАЕМ ВЛОЖЕННЫЕ ЭЛЕМЕНТЫ

Так как у нас функциональный компонент, и до этого мы имели дело только с аргументом `props`, начнем поиски с него:

```
1 function OfferModal(props) {  
2   console.log(props);  
3   return (  
4     <div className="modal">  
5       <div className="modal-body">{props.body}</div>  
6       <button className="modal-close">закрыть</button>  
7     </div>  
8   );  
9 }
```


СРАЗУ В ЯБЛОЧКО!

Видим в консоле что в `props` есть свойство `children` и оно *массив*:

```
Object  
└─ children: Array(2)
```

В массиве два элемента, каждый из которых объект:

```
Array (2)  
└─ 0 { type: "h2", ...}  
    └─ 1 { type: "a", ...}
```

Это и есть наши вложенные элементы. Как их вывести?

ВЫВОДИМ ВЛОЖЕННЫЕ ЭЛЕМЕНТЫ

Мы уже не раз выводили в JSX массив элементов. Помните занятия по спискам?

Мы можем просто вставить массив в нужное место кода нашего компонента:

```
1  function OfferModal(props) {  
2    return (  
3      <div className="modal">  
4        <div className="modal-body">  
5          {props.children}  
6        </div>  
7        <button className="modal-close">закрыть</button>  
8      </div>  
9    );  
10 }
```

СВОБОДА ВЫБОРА

Стоит отметить, что мы вообще не обязаны куда-то выводить `props.children`. Мы так пробовали, и это не привело к ошибкам. Мы решаем — может ли компонент иметь вложенные элементы, и где они будут выводиться.

Но мы так же можем решить вывести только первый:

```
1  function OfferModal(props) {  
2    return (  
3      <div className="modal">  
4        <div className="modal-body">  
5          {props.children[0]}  
6        </div>  
7        <button className="modal-close">закрыть</button>  
8      </div>  
9    );  
10 }
```

ИТОГИ ПО ВЛОЖЕННЫМ ЭЛЕМЕНТАМ

Возможность вкладывать элементы в компонент открывает перед нами новые возможности по созданию более гибких и универсальных компонентов. Подведем итоги:

- Созданный нами компонент можно использовать как одиночный `<OfferModal />`, так и двойной `<OfferModal>...</OfferModal>`.
- Если в наш компонент вложить элементы, то они передаются в компонент через аргумент функции `props` в его свойстве `children`.
- Свойство `children` — *массив*, каждый элемент которого является React-элементом.
- Мы сами определяем где и как выводятся вложенные элементы из `children`. Например, мы можем их вообще не выводить.
- В компонентах на основе классов вложенные элементы доступны через `this.props.children`. И работают так же.



КОМПОЗИЦИЯ КОМПОНЕНТОВ



КОМПОЗИЦИЯ КОМПОНЕНТОВ

Композиция компонентов — это процесс комбинирования двух или более компонентов с целью создания новой более сложного компонента.

Композиция React компонентов схожа с композицией функций, так как содержит те же принципы.

СООБЩЕНИЕ

Допустим у нас есть компонент для отображения всплывающих сообщений:

```
1 function Message(props) {  
2   return (  
3     <div className={`message message-${props.type}`}>  
4       {props.text}  
5     </div>  
6   );  
7 }
```

Как поступить, если нам довольно часто нужно выводить сообщения об ошибке:

```
<Message type="error" text="Пароли не совпадают" />
```

БОЛЕЕ СПЕЦИАЛИЗИРОВАННЫЙ КОМПОНЕНТ

Всегда можно сделать так:

```
1 function ErrorMessage(props) {  
2   return (  
3     <div className="message message-error">  
4       {props.text}  
5     </div>  
6   );  
7 }
```

Но так мы нарушаем принцип DRY. А так нет:

```
1 function ErrorMessage(props) {  
2   return <Message type="error" text={props.text} />;  
3 }
```


ЭТО ИСКУСТВО

Создание компонентов React — это искусство. И сама библиотека очень в этом помогает.

Представим что по какой-то причине наш компонент `Message` принимает множество аргументов:

```
1 <Message
2   type="error"
3   icon="/path/to/error.png"
4   position="topright"
5   timeout="60"
6   priority="100"
7   text="Пароли не совпадают" />
```

Как нам быть с `ErrorMessage`, если мы хотим в нём зафиксировать только один-два параметра?

ПЛОХОЕ РЕШЕНИЕ

Мы можем просто «пробросить» все известные нам атрибуты в

`Message`:

```
1 function ErrorMessage(props) {  
2   return (  
3     <Message  
4       type="error" icon="/path/to/error.png"  
5       position={props.position} timeout={props.timeout}  
6       priority={props.priority} text={props.text} />  
7   );  
8 }
```

Решение принесет первые проблемы сразу как только в компоненте

`Message` появятся новые атрибуты, и кто-то попытается их использовать в `ErrorMessage`.

ОПЕРТОР РАЗВОРОТА В JSX

Для начала давайте вспомним про оператор разворота (*spread*) `...`.

Его можно использовать в JSX для передачи объекта целиком в качестве атрибутов. Допустим у нас есть объект:

```
1  const message = {  
2    type: 'info',  
3    text: 'Информационное сообщение'  
4  };
```

С помощью оператора разворота мы можем записать вот так:

```
1  <Message {...message} />
```

Все свойства объекта `message` передадутся в `props` нашего компонента или в свойства элемента.

ПОРЯДОК ИМЕЕТ ЗНАЧЕНИЕ

Если атрибут с именем, которое есть в объекте идет перед оператором разворота, его значение будет переписано. Тут получим сообщение типа `info`:

```
1 | <Message type="error" {...message} />
```

Если атрибут идет после оператора разворота, будет использовано его значение. Тут получим сообщение типа `error`:

```
1 | <Message {...message} type="error" />
```

ПРИМЕНИМ ОПЕРАТОР РАЗВОРАТА

Пробросим все атрибуты и переопределим те что нам нужны:

```
1 function ErrorMessage(props) {  
2   return (  
3     <Message  
4       {...props}  
5       type="error"  
6       icon="/path/to/error.png" />  
7   );  
8 }
```

ЧИСТОТА JSX

Рекомендуется использовать `Object.assign` для создания свойств для проброски в компонент, вместо переопределения их в JSX:

```
1 function ErrorMessage(props) {  
2   const base = {  
3     type: 'error',  
4     icon: '/path/to/error.png'  
5   };  
6   const newProps = Object.assign({}, props, base);  
7   // либо const newProps = {...props, ...base};  
8   return <Message {...newProps} />;  
9 }
```

Если вам вдруг пришла в голову идея переопределить свойства в `props`, напомним что в React это делать нельзя.

КОМПОНЕНТ БЕЗ КОМПОЗИЦИИ

Нам нужно создать компонент профиля пользователя. Как бы мы реализовали это без композиции компонентов?

Примерно так:

```
1  const Avatar = props => (  
2    <div>  
3      <img src={`https://photo.yoursite.com/${props.username}`} />  
4      <a href={`https://yoursite.com/${props.username}`}>  
5        {props.username}  
6      </a>  
7    </div>  
8  )
```

Все вроде хорошо и работает. В чем тогда смысл перетруждать себя?

На лекции с HTTP мы рассмотрим, как для этой цели можно использовать environment.

КОМПОНЕНТЫ

Напоминаем, что компоненты (функциональные) - это просто функции и не важно, как вы их записываете в исходном файле:

```
1 | const Avatar = props => (...);  
2 | // или  
3 | function Avatar(props) { return (...); }
```


ПРОФИЛЬ ПОЛЬЗОВАТЕЛЯ

Допустим, вы хотите создать профиль пользователя, тогда компонент будет выглядеть так:

```
1  const Profile = props => (  
2    <div>  
3      <img src={`https://photo.yoursite.com/${props.username}`} />  
4      <a href={`https://yoursite.com/${props.username}`}>{props.username}</a>  
5      <div>{props.age}</div>  
6      <div>{props.email}</div>  
7    </div>  
8  )  
9  
10 // в App.js  
11 return (  
12   <Profile username='coollogin' age={24} email='coollogin@yoursite.com' />  
13 )
```

Вопрос: какие недостатки у данного подхода?

ГЛАВНЫЙ НЕДОСТАТОК ПОДХОДА

Если мы захотим изменить ссылку на сайт, то нам придется пройти по всем частям программы где используется изображение профиля и ссылка на него. В нашем случае таких частей две `Avatar` и `Profile`.

А что если их было бы намного больше?

А если бы тесты для них были бы разработаны и зависели от ссылок в явном виде?

ВВЕДЁМ ДОПОЛНИТЕЛЬНЫЕ КОМПОНЕНТЫ

```
1  const ProfilePic = props =>
2    <img src={`https://photo.yoursite.com/${props.username}`} />
3  const ProfileLink = props =>
4    <a href={`https://yoursite.com/${props.username}`}>{props.username}</a>
5
6  const Avatar = props => (
7    <div>
8      <ProfilePic username={props.username} />
9      <ProfileLink username={props.username} />
10   </div>
11 )
12
13 const Profile = props => (
14   <div>
15     <ProfilePic username={props.username} />
16     <ProfileLink username={props.username} />
17     <div>{props.age}</div>
18     <div>{props.email}</div>
19   </div>
20 )
```

РЕЗУЛЬТАТ ИЗМЕНЕНИЙ

Наш компонент `Avatar` состоит из компонентов `ProfilePic` и `ProfileLink`, как и компонент `Profile`.

Данный подход позволил нам существенно улучшить код и избежать в будущем проблем с изменением компонентов.

ПОДХОДЫ К КОМПОЗИЦИИ КОМПОНЕНТОВ

ВЫДЕЛЕНИЕ КОМПОНЕНТОВ

Рассмотрим вторую задачу: у нас есть боковая панель сайта (`sidebar`) на которой у нас расположены виджеты.

Наша задача выделить в данном коде компоненты написать их реализацию.

```
<aside id="sidebar">
  <div class="widget">
    <div>Поиск</div>
    <form method="GET" action="/search/" class="search">
      <input type="text" name="query" placeholder="Поиск..." value="">
    </form>
  </div>
  <div class="widget">
    <div>Тег</div>
    <div class="tag-block">
      <a href="/articles/tag/js/" class="tag-link clear">JavaScript</a>
      <a href="/articles/tag/nodejs/" class="tag-link clear">Node.JS</a>
    </div>
  </div>
  ... продолжение на следующем слайде ...
</aside>
```

```
<aside id="sidebar">
  ... начало на предыдущем слайде ...
  <div class="widget">
    <div>Социальные сети</div>
    <div class="social-block">
      <ul>
        <li><a href="https://vk.com/username" class="clear social-link">VK</a></li>
        <li><a href="mailto:username@gmail.com" class="clear social-link">Email</a></li>
        <li><a href="https://github.com/username" class="clear social-link">Github</a></li>
      </ul>
    </div>
  </div>
</aside>
```




ПОДХОДЫ ДЛЯ ВЫДЕЛЕНИЯ КОМПОНЕНТОВ

Существуют три подхода для выделения компонентов:

- От большего к меньшему;
- От меньшего к большему;
- Комбинирование.



ОТ БОЛЬШЕГО К МЕНЬШЕМУ

Используется когда вам надо выделить компоненты в достаточно объемном коде, где наблюдается существенная трудность для выделения всего и сразу.



ОТ МЕНЬШЕГО К БОЛЬШЕМУ

Используется когда область вашей задачи небольшая и вы вполне можете выделить маленькие части вашего интерфейса.



КОМБИНИРОВАНИЕ

Используется также когда надо выделить компоненты в достаточно большом коде, и представляет собой чередование подходов.



ПЕРЕДАЧА КОМПОНЕНТОВ В АТТРИБУТЫ

СОЗДАДИМ КОМПОНЕНТ ДЛЯ КОНТЕНТА

В нашем случае мы будем использовать подход "от большего к меньшему".

Первым делом создадим компонент в котором будут содержаться контент для боковой панели:

```
1  const Sidebar = props =>
2    <aside id={props.id}>{props.content}</aside>
3
4  // в родительском компоненте
5  const sidebarContent = (
6    // ... внутри всё, что было в aside
7    // ... все три виджета
8  )
9
10 return (
11   <Sidebar id='sidebar' content={sidebarContent}/>
12 )
```

DOM В КАЧЕСТВЕ АТТРИБУТА

В данном случае мы обернули нашу верстку в компонент `Sidebar` в котором определили два атрибута: `id` и `content`.

`content` содержит в себе DOM элементы являющиеся потомками для нашей боковой панели. Для этого мы определили переменную `sidebarContent` в которую поместили DOM.

Таким образом мы передали наш DOM в качестве атрибута. Но данный подход является неинтуитивным и мало чем схож с обычной версткой HTML, хотя изначально JSX таким себя позиционирует.

ВЛОЖЕННЫЕ КОМПОНЕНТЫ - CHILDREN COMPONENTS

ИСПОЛЬЗУЕМ `props.children`

Использовать атрибут `props.children` и наш компонент станет больше похожим на классическую HTML разметку.

```
1  const Sidebar = props =>
2    <aside id={props.id}>{props.children}</aside>
3
4  // в родительском компоненте
5  return (
6    <Sidebar id='sidebar'>
7      // ...
8    </Sidebar>
9  )
```

НЕЯВНОЕ ПОПАДАНИЕ В АТТРИБУТ

Дочерные компоненты, которые располагаются между открывающим и закрывающим тегом нашего компонента *неявно* попадают в атрибут `props.children`, который может быть в последующем вызван.

А МОЖНО БЕЗ `props.children` ?

А что будет если не указать в нашем компоненте `props.children`, а внутри тега передать ему дочерние компоненты?

```
const Sidebar = props => <aside id={props.id}></aside>  
  
// ...
```

БЕЗ `props.children` НЕ РАБОТАЕТ

Если мы не указываем `props.children`, то ничего не выведется на экран.

А ЕСЛИ НИЧЕГО НЕ ПЕРЕДАВАТЬ?

Что если указать в компоненте `props.children`, но дочерние компоненты ему не передать?

```
1  const Sidebar = props => {  
2    console.log(props.children);  
3  
4    return <aside id={props.id}>{props.children}</aside>  
5  }  
6  
7  // в родительском компоненте  
8  return (<Sidebar id='sidebar'></Sidebar>)
```

ТАК ТОЖЕ НЕ СРАБОТАЕТ

Также ничего не выведется, так как в консоли мы получим значение `props.children undefined`, что попросту не отображается React в DOM элемент (вспомним также: `null`, `false`, `true`, пустые значения: `[]`, `{}` и т.д.).

А ЕСЛИ ЯВНО УКАЗАТЬ АТТРИБУТ `children`

```
1  const Sidebar = props =>
2    <aside id={props.id}>{props.children}</aside>
3
4  // В родительском компоненте
5  return (
6    <Sidebar id='sidebar'
7      children={<b>Я боковая панель!</b>} >
8      <b>А я дочерний компонент!</b>
9    </Sidebar>
10  )
```

Что выведется на экране в таком случае?

`children` ПЕРЕЗАПИШЕТСЯ

В итоге на экране мы получим следующее:

А я дочерний компонент!

Соответственно сделаем вывод, что наш значение нашего атрибута `children` перезапишется дочерними компонентами между тегами родительского компонента.

ВЫВОДЫ

- JSX позволяет интуитивно задавать дочерние элементы.
- Если у нас не используется `props.children`, то мы можем задать наш компонент с помощью одиночного тега.
- Если у нас был задан явно атрибут `children`, то он перезапишется тем, что между тегами родительского компонента, а если между этими тегами ничего не заданно, то будет взято значение из нашего атрибута.



ПРИНЦИП ЕДИННОЙ ОТВЕТСТВЕННОСТИ

Одним из ключевых принципов композиции является то, что мы должны выделить пересечение частей в нашем коде, чтобы в итоге выделить это значение в новый компонент. Данный подход широко используется в программировании и носит название *Принципа единной ответственности*.

ВЫДЕЛИМ КОМПОНЕНТ `Widget`

Для нашего примера можно выделить компонент `Widget` так как он часто повторяется, паттерн его поведения очевиден и прост.

```
1  const Widget = props => (  
2    <div className="widget">  
3      <div>{props.title}</div>  
4      {props.children}  
5    </div>  
6  )
```

АТТРИБУТЫ КОМПОНЕНТА `Widget`

Данный компонент принимает в качестве атрибутов только название виджета `title`, а также у него определен неявный атрибут `props.children` для контента.

ОБНОВИМ КОМПОНЕНТ **Sidebar**

```
1  // в родительском компоненте
2  return (
3    <Sidebar id='sidebar'>
4      <Widget title='Поиск'>
5        // ...
6      </Widget>
7      <Widget title='Теги'>
8        // ...
9      </Widget>
10     <Widget title='Социальные сети'>
11       // ...
12     </Widget>
13   </Sidebar>
14 )
```



КОМПОЗИЦИЯ СПИСКОВ

КОМПОНЕНТЫ СПИСКА

В практике создания интерфейсов часто используются списки и нам бы хотелось создать компонент для работы с ними. Определим его компоненты:

```
1  const List = props =>
2    <ul className={props.className} style={props.style}>
3      {props.children}
4    </ul>;
5  const ListItem = props =>
6    <li className={props.className} style={props.style}>
7      {props.children}
8    </li>;
9  // в родительском компоненте
10 return (
11   <List className='list'>
12     <ListItem>Написать отчет</ListItem>
13     <ListItem>Купить молоко</ListItem>
14     <ListItem>Забрать машину из ремонта</ListItem>
15   </List>
16 );
```



ВСЁ ХОРОШО...ВРОДЕ БЫ

Все хорошо, список отображается, но у нас тут также присутствуют недостатки.

Какие недостатки вы тут видите?



НЕ ОПРЕДЕЛЕНА СВЯЗЬ С ДОЧЕРНИМИ КОМПОНЕНТАМИ

Самый очевидный недостаток состоит в том, что не определена связь между родительским компонентом и его детьми.

Что если родителю нужно передать некоторую информацию детям?

Для решения этой задачи мы можем использовать функцию как дочерний компонент.



ФУНКЦИЯ КАК ДОЧЕРНИЙ КОМПОНЕНТ



ФУНКЦИЯ В КАЧЕСТВЕ ДОЧЕРНЕГО ЭЛЕМЕНТА

Мы можем передавать в качестве дочернего элемента функцию, которую в последствии можем вызывать в нашем родительском компоненте, причем передавая ей нужные данные.

УСЛОВИЯ РАБОТЫ СПИСКА

Предположим, что список представляет собой массив и берется из элемента `List`. Причем в списке должно выводиться количество элементов в нем.

Как бы вы это сделали?

РЕАЛИЗАЦИЯ List

```
1  const List = props =>
2    <ul className={props.className}>{props.children}</ul>;
3
4  const ListItem = props =>
5    <li className={props.className}>{props.children}</li>;
6
7  // в родительском компоненте
8  const todos = ['Написать отчет', 'Купить молоко', 'Забрать машину из ремонта'];
9
10 const listItems = todos.map((item) => <ListItem>{item}</ListItem>);
11
12 return (
13   <List className='list'>
14     {listItems}
15     <li><b>Всего:</b> {listItems.length}</li>
16   </List>
17 );
```



НЕ СОБЛЮДЁН ПРИНЦИП ЕДИННОЙ ОТВЕТСТВЕННОСТИ

Данный вариант верный, но проблема заключается в том, что мы вынесли логику подсчета за пределы нашего компонента, а это означает, что если мы захотим создать еще раз такого рода список, то нам придется заново выполнить все пункты.

ПЕРЕНОСИМ ЛОГИКУ В **List**

```
1  const List = props => (  
2    <ul className={props.className}>  
3      {props.children(props.items)}  
4      <li><b>Всего:</b> {props.items.length}</li>  
5    </ul>  
6  );  
7  const ListItem = props =>  
8    <li className={props.className}>{props.children}</li>;  
9  
10 // в родительском компоненте  
11 const todos = ['Написать отчет', 'Купить молоко', 'Забрать машину из ремонта'];  
12  
13 return (  
14   <List className='list' items={todos}>  
15     {items => items.map((item, index) => <ListItem key={index}>{item}</ListItem>)}  
16   </List>  
17 );
```

ПОЧЕМУ ЭТО РАБОТАЕТ?

Все дело в том, что в атрибут `props.children` заносится функция, а не DOM элементы. В свою очередь в данной функции мы можем делать практически все, что пожелаем с дочерними компонентами, причем данные мы также можем брать от родительского компонента.



КОГДА ЭТО УДОБНО?

Это удобно когда вам нужно передать какие-то параметры от родительского компонента дочернему. К примеру если у вас для родителя определено какое-то внутреннее состояние, которое может меняться, то вы можете об этом "информировать" дочерние элементы.

ВОЗМОЖНОСТЬ СКРЫТИЯ КОНТЕНТА

Создадим файл `section.js` в директории `components`:

```
1 function Section(props) {  
2   const [hidden, setHidden] = useState(false);  
3  
4   toggleSection = () => { setHidden(prev => !prev); }  
5  
6   return (<section>  
7     <h1>{this.props.title}</h1>  
8     <div className='content'>  
9       {this.props.children(this.state.hidden)}  
10    </div>  
11    <button onClick={this.toggleSection}>  
12      Переключить  
13    </button>  
14  </section>);  
15 }
```

ЗАДАДИМ ФУНКЦИЮ КАК ДОЧЕРНИЙ ЭЛЕМЕНТ

В файле `index.js` импортируем данный компонент и зададим ему функцию, как дочерний элемент:

```
1 // в родительском компоненте
2 return (
3   <Section title='Истории из детства'>
4     {
5       (isHidden) => isHidden ? null : <p>Большой блок текста</p>
6     }
7   </Section>
8 );
```

ЛОГИКА КОМПОНЕНТА Section

Таким образом, информация о состоянии родителя может передаваться его дочерним элементам и если мы нажмем на кнопку "Переключатель", то и будет прятаться или показываться блок текста в зависимости от состояния нашего компонента.



SPREAD ОПЕРАТОР - ОПЕРАТОР РАЗВОРОТА



ОПЕРАТОР РАЗВОРОТА

Оператор разворота был стандартизирован в *ES6*. Он позволяет "разворачивать" свойства объекта, или элементы массива.

КОМПОНЕНТ С МНОЖЕСТВОМ АТТРИБУТОВ

У нас есть компонент, который выводит полную информацию о пользователе:

```
1  const User = props => (  
2    <ul>  
3      <li>{props.name}</li>  
4      <li>{props.age}</li>  
5      <li>{props.school}</li>  
6      <li>{props.university}</li>  
7      <li>{props.work}</li>  
8      <li>{props.hobby}</li>  
9    </ul>  
10  );
```

Как можно задать атрибуты для данного компонента?

СТАНДАРТНЫЙ СПОСОБ ЗАДАТЬ АТТРИБУТЫ

```
1  // в родительском компоненте
2  return (
3    <User
4      name='Петр Петрович'
5      age={28}
6      school='СПБ №33'
7      university='СПБГУ'
8      work='Главный инженер'
9      hobby='рыбалка, музыка, игры' />
10 );
```




ГРОМОЗДКО И СЛОЖНОЧИТАЕМО

Получается громоздко и сложночитаемо, так как атрибутов много и они не помещаются в заветные 80 символом.

Возникает вопрос как решить данную проблему?

ИСПОЛЬЗУЕМ ОПЕРАТОР РАЗВОРОТА

```
1 // в родительском компоненте
2 const props = {
3   name: 'Петр Петрович',
4   age: 28,
5   school: 'СПБ №33',
6   university: 'СПБГУ',
7   work: 'Главный инженер',
8   hobby: 'рыбалка, музыка, игры'
9 }
10
11 return (
12   <User {...props} />
13 );
```

ЧТО ПОД КАПОТОМ?

Мы задали объект `props` с определенными свойствами, а после с использованием оператора разворота, передали свойства находящиеся в нашем объекте в наш компонент, где ключ - название атрибута, а значение по ключу, значение атрибута.

`{...props}` СОЗДАЕТ НОВЫЙ ОБЪЕКТ

Важно понимать, что `{...props}` создает новый объект в который копируются все свойства `props`, и таким образом мы избавляемся от побочных эффектов "*принципа иммутабельности*" (о нем пойдет речь позже).

Таким образом компонент `User` обладает всеми нужными ему атрибутами, и читаемость кода существенно улучшилась.

КАК ПЕРЕПИСАТЬ АТТРИБУТ?

Возможно, что во время проектирования интерфейсов вам придется переписать одно из свойств там, где вы использовали *spread operator*. Для данной цели возможно просто переопределить атрибут после задания `{...props}`, то есть:

```
1 ReactDOM.render(  
2   <User {...props} age={30}/>,  
3   document.getElementById('root')  
4 );
```

КАК ЕЩЕ МОЖНО ЭТО ИСПОЛЬЗОВАТЬ?

Перепишем наш компонент таким образом, чтоб мы получали атрибуты `name` и `age` не в составе объекта `props`:

```
1  const User = ({ school, name, ...props }) => (  
2    <ul>  
3      <li>{name}</li>  
4      <li>{props.age}</li>  
5      <li>{school}</li>  
6      <li>{props.university}</li>  
7      <li>{props.work}</li>  
8      <li>{props.hobby}</li>  
9      </ul>  
10 );
```

"ВЫТАСКИВАЕМ" ЗНАЧЕНИЯ СВОЙСТВ

В данном случае из объекта `props` мы "вытаскиваем" значения свойств из переданного объекта, а оставшиеся свойства помещаются в объект `props` при помощи оператора расширения `...props` и соответственно в коде из можно использовать без указания `props.name` и `props.shool`.

А ЕСЛИ УКАЗАТЬ ИХ ЧЕРЕЗ `props` ?

А что если указать через `props` "вынимаемые" свойства?

```
1  const User = ({school, name, ...props}) => (  
2    <ul>  
3    <li>{props.name}</li>  
4    <li>{props.age}</li>  
5    <li>{props.school}</li>  
6    <li>{props.university}</li>  
7    <li>{props.work}</li>  
8    <li>{props.hobby}</li>  
9    </ul>  
10 );
```


СВОЙСТВА ПРИМУТ ЗНАЧЕНИЕ `undefined`

В данном случае `props.name` и `props.school` примут значение `undefined`.

ПОМЕНЯЕМ ПАРАМЕТРЫ МЕСТАМИ

Чтобы была возможность использовать свойства одновременно, принимаемые параметры должны быть обозначены так:

```
1  const User = ({...props, school, name}) => (  
2    <ul>  
3    <li>{props.name}</li>  
4    <li>{props.age}</li>  
5    <li>{props.school}</li>  
6    <li>{props.university}</li>  
7    <li>{props.work}</li>  
8    <li>{props.hobby}</li>  
9    </ul>  
10 );
```

ЧТО ИЗМЕНИЛОСЬ?

Во втором случае в объект `props` будут скопированы, а не извлечены все свойства передаваемого объекта (в нашем случае объекта с атрибутами), после чего в `school` и `name` извлекутся одноименные свойства.

☼ нетология

**Задавайте вопросы и
пишите отзыв о лекции!**

Дмитрий Федин