

11. Продвинутый дизайн компонентов

Содержание урока

- Обзор;
- Классовые и функциональные компоненты;
- Анализ классового и функционального компонентов;
- Какой тип компонента использовать?;
- Переиспользование компонентов;
- Компоненты высшего порядка;
- Порталы;
- Подведём итоги.

Обзор

Привет! 🖐️ В этой части конспекта мы разберём методики композиции компонентов и некоторые продвинутые паттерны, следуя которым можно повысить эффективность своего приложения в разы. 🚀

К делу! 🦊

Классовые и функциональные компоненты

До сих пор мы рассматривали классовые компоненты как предпочтительные к использованию. Теперь, располагая определённой базой в понимании важных концепций механики React, мы можем сделать полноценный анализ каждого типа компонента, а также их сравнить для того, чтобы полностью понять, когда и зачем стоит задействовать тот или иной тип. 🧑

Функциональный компонент — это «облегчённая» версия классового компонента. То есть у функциональных компонентов отсутствуют возможности, присущие классовым компонентам.

Одна из этих возможностей — стейт. В отличие от классовых компонентов, функциональные не располагают приватным состоянием и, соответственно, не имеют методов для управления состоянием : `useState` , `useEffect` .

🔍 Хозяйке на заметку:

Иногда функциональные компоненты называют `stateless functional component`.

Также функциональные компоненты не поддерживают ни одного метода "жизненного цикла", доступного классовому компоненту, что ограничивает гранулированный контроль функциональных компонентов.

Однако пропсы доступны как классовым компонентам, так и функциональным, с той лишь разницей, что у классовых компонентов пропсы доступны через обращение к `this.props`, а у функциональных — через обращение к первому параметру в замыкании. 🐸

Резюмировав, можно составить таблицу-сравнение классовых и функциональных компонентов.

Анализ классового и функционального компонентов

Компонент-класс	Компонент-функция
✅ Создаётся в виде ES2015-класса	✅ Создаётся в виде JavaScript-функции
✅ Пропсы доступны через обращение к <code>this.props</code>	✅ Пропсы доступны из первого параметра в замыкании
✅ <code>propTypes</code> и <code>defaultProps</code> объявляются в виде статических свойств класса	✅ <code>propTypes</code> и <code>defaultProps</code> объявляются посредством аксессуара <code>.</code>
✅ Располагают приватным состоянием и методами для управления им	❌ Не располагают состоянием
✅ Располагают методами "жизненного цикла"	❌ Не располагают методами "жизненного цикла"
✅ Разметка возвращается из метода <code>render</code>	✅ Разметка возвращается компонентом непосредственно
❌ Сложнее тестировать	✅ Легче тестировать
❌ Многословный	✅ Компактный

Несмотря на то, что фактическое количество плюсов и минусов является равным, минусы функционального компонента существенней по весу: отсутствие стейта и методов "жизненного цикла" серьёзно ограничивают возможности при разработке.

А достоинство компактности, 💎 несомненно, значимое, особенно для любителей красивого кода, 🎨 однако не является практичным и никак не перекрывает существенные недостатки, перечисленные выше. 🙄

🔍 Хозяйке на заметку:

Существует распространённый миф о том, что функциональные компоненты обладают более высоким потенциалом производительности, учитывая факт своей «легковесности», что конечно же неправда, учитывая то, что классовый компонент, по сути, тоже является функцией.

Технически функциональный компонент можно наоборот расценить как менее производительный, с учётом отсутствия метода "жизненного цикла"

`shouldComponentUpdate`, предназначение которого — отладка производительности. ⚖️

Какой тип компонента использовать?

Здесь нет «золотой» линейки, чтобы определить, что лучше использовать.

С практической точки зрения классовый компонент выигрывает у функционального почти всухую. Хотя функциональный компонент может одолеть классовый в споре о стиле кода. 🚒

Во втором варианте существует сценарий, у которого есть право на жизнь. «Open source»-комьюнити взрослеет с каждым месяцем в невероятно позитивном темпе. В «open source» можно найти целый спектр полезнейших инструментов-компаньонов для твоего приложения.

Например, библиотека [recompose](#) предоставляет набор утилит в виде «компонентов высшего порядка», дополняющих возможности функциональных компонентов.

🔍 Хозяйке на заметку:

«Компонент высшего порядка» — это фирменный паттерн React. Мы рассмотрим механику работы данного паттерна далее в этой части конспекта.

В частности, «компонент высшего порядка» [withState](#) позволяет функциональным компонентам располагать и управлять приватным состоянием. А [lifecycle](#) открывает доступ к методам "жизненного цикла".

`Recompose` предоставляет ещё много «компонентов высшего порядка» на разные случаи жизни, в очень удобной форме. 😎


Однако, строго говоря, использование `recompose` в качестве «возмещения» недостающих возможностей функциональных компонентов — не что иное, как «лечение симптома», нежели «искоренение причины». Поэтому стоит хорошо взвесить все «за» и «против» перед движением в эту сторону. 🤔

А сократив критерии выбора к минимуму, можно вывести следующую формулу выбора типа компонента для использования:

- Если ты практичен 🧑🏻 и следуешь подходу «главное, чтобы работало надежно и стабильно» — используй `классовые компоненты`;
- Если ты экзотик 🦄 и следуешь подходу «люблю экспериментировать и делать вычурные вещи» — используй `функциональные компоненты`.

Переиспользование компонентов

React обладает мощной композиционной моделью, позволяющей переиспользовать компоненты во многих местах, избежав дублирования.

 Пример кода 11.1:

```
1 // LoginForm.js
2 import React, { Component } from 'react';
3 import Input from './Input';
4
5 export default class LoginForm extends Component {
6   state = {
7     login: '',
8     password: ''
9   }
10
11   _onLoginChange = (event) => {
12     this.setState({
13       login: event.target.value
14     });
15   }
16
17   _onPasswordChange = (event) => {
18     this.setState({
19       password: event.target.value
20     });
21   }
22
23   render () {
24     const { login, password } = this.state;
25
26     return (
27       <form>
28         <Input
29           darkTheme
30           id = 'login'
31           label = 'Login: '
32           placeholder = 'Your login'
33           type = 'text'
34           value = { login }
35           onChange = { this.onLoginChange }

```

```

36         />
37         <Input
38             darkTheme
39             id = 'password'
40             label = 'Password: '
41             placeholder = 'Your password'
42             type = 'password'
43             value = { password }
44             onChange = { this.onPasswordChange }
45         />
46         <Input
47             darkTheme
48             type = 'submit'
49             value = 'Log in'
50         />
51     </form>
52 );
53 }
54 }

```

```

1 // Input.js
2 import React, { Component } from 'react';
3 import Styles from './styles';
4
5 export default class Input extends Component {
6     render () {
7         const {
8             id,
9             checked,
10            label,
11            placeholder,
12            type,
13            value,
14            onChange,
15            darkTheme
16        } = this.props;
17
18        return (
19            <label
20                label
21                className = { darkTheme
22                    ? Styles.darkTheme
23                    : Styles.defaultTheme }
24            htmlFor = { id }>
25                <input
26                    checked = { checked }
27                    className = { darkTheme
28                        ? Styles.darkTheme

```

```

29         : Styles.deafultTheme }
30         id = { id }
31         placeholder = { placeholder }
32         type = { type }
33         value = { value }
34         onChange = { onChange }
35     />
36 </label>
37 );
38 }
39 }

```

В примере кода 11.1 представлено два компонента: `LoginForm` и `Input`. Компонент `Input` организован как переиспользуемая сущность: поведение компонента меняется в зависимости от пропсов, переданных родительским компонентом. Следуя данному несложному подходу, можно создавать воистину эффективные проекты, объём код-базы которых без переиспользования разрастался бы в разы.

Переиспользуемость компонентов проста в понимании и лёгкая в имплементации. Поэтому данное качество расценивается как одно из «хороших» частей React. 📌

Компоненты высшего порядка

Компонент высшего порядка (англ. «Higher order component» или «НОС») — продвинутая техника React, помогающая эффективно организовывать переиспользуемую логику.

🔍 Хозяйке на заметку:

`НОС` не является API React, а скорее паттерн, вытекающий из богатой композиционной природы.

По определению `НОС` — это функция, принимающая аргументом компонент и возвращающая новую, улучшенную версию этого компонента.

Подпись `НОС` можно описать так:

```

1 const EnhancedComponent = higherOrderComponent(WrappedComponent);

```

Где:

- `WrappedComponent` — компонент, функционал которого требуется «улучшить» или «расширить»;
- `higherOrderComponent` — компонент высшего порядка;
- `EnhancedComponent` — новая, улучшенная версия `WrappedComponent`.

👉 Совет бывалых:

Паттерн компонентов высшего порядка является очень распространённым в комьюнити React. Например, на основе компонентов высшего порядка работают такие библиотеки, как `react-redux`, `react-router`, `recompose`, `react-intl`, `relay` и многие другие. Поэтому данный паттерн настоятельно рекомендуется к изучению.

Оскар очень любит читать и ~~кодировать~~, настолько сильно, что даже создал свою собственную интернет-лавку по торговле книгами. 📖 Но однажды, столкнувшись со злым хакером, 🧑🏻 Оскар понял, что лавку с книгами нужно защитить. Давай напишем компонент высшего порядка, который защитит каждую книгу Оскара.

🖥️ Пример кода 11.2:

```
1 import React from 'react';
2 import Book from './Book';
3
4 const protect = WrappedComponent => {
5   return class Enhancer extends WrappedComponent {
6     render() {
7       return this.props.purchased
8         ? super.render()
9         : 'Sorry, you need to purchase this book first!';
10    }
11  };
12 };
13
14 export default protect(Book);
```

В примере кода 11.2 описан компонент высшего порядка `protect`. При вызове `protect` принимает в аргумент компонент, который необходимо будет защитить от взлома. Реализация нового компонента, возвращаемого компонентом высшего порядка `protect`, предусматривает наличие у расширяемого компонента булевого пропса `purchased`, говорящего о том, что компонент был «приобретён» честным путем. Если это так, то он отрендерится как ни в чём не бывало, а если нет — отобразится фолбэкный UI, говорящий о том, что не так. 🧑🏻

Порталы

Композиционные возможности React расширяет ещё одна интересная функциональность — порталы. 🌟

Портал предоставляет механизм рендеринга дочерних элементов в DOM-ноде, существующей вне DOM-области, подконтрольной приложению React.

Подпись портала можно выразить так:

```
1 ReactDOM.createPortal(child, container)
```

Где:

- Первый аргумент (`child`) — это дочерний элемент: любая сущность, которую React может отрендерить, или «renderable»;
- Второй аргумент (`container`) — целевая нода DOM, к которой портал «присоединит» «renderable», переданное в первый аргумент `React.createPortal`.

Типичный юзкейс использования портала — модальное окно с участием родительского компонента, имплементирующего CSS наподобие `overflow: hidden` или модифицированный `z-index`. В таком случае необходимо, чтобы дочерний компонент «вырвался наружу» из своего родительского контейнера. Такая потребность может также возникнуть при работе с диалоговыми окнами, тултипами и другими элементами смежной категории. 🗨️

Подведём итоги

В этом уроке мы разузнали о различиях типов компонентов React, а также о его композиционных возможностях, а именно:

- Классовый компонент — обладает полным спектром возможностей;
- Функциональный компонент — обладает урезанным спектром возможностей;
- Для того, чтобы эффективно переиспользовать компонент, необходимо организовать его внутренность так, чтобы его желаемое поведение определялось передачей соответствующих пропсов;
- Компонент высшего порядка — это паттерн, позволяющий эффективно работать с композиционной моделью React;
- Портал — это механизм, позволяющий отрендерить рендеримую сущность в области DOM, находящейся вне области React-приложения.

Спасибо, что остаёшься с нами! 🙌 В следующей части конспекта мы рассмотрим подходы анимирования React-компонентов. До встречи! 🐔

Мы будем очень признательны, если ты оставишь свой фидбек в отношении этой части конспекта на нашу электропочту `hello@lectrum.io`.