

# 7. Всегда в курсе происходящего с «состоянием» React

## Содержание урока

- Обзор;
- Что такое «состояние»?;
- Обновление «состояния» в обычном режиме;
- Обновление «состояния» в принудительном режиме;
- «Контролируемые» и «неконтролируемые» компоненты;
- Рефы;
- Подведём итоги.

## Обзор

Привет! 🙌 В React существует ещё один крайне важный концепт `«инкапсулированного состояния компонента»`. Чаще всего к нему ссылаются по термину `«стейт»` (англ. `«state»`), кто бы мог догадаться?! 🤔).

В этом уроке нам и предстоит разузнать об этом концепте, и что представляет из себя система управления состоянием React в целом.

К делу! 🦊

## Что такое `«состояние»`?

`«Состояние»` (далее `«стейт»`) — это обычный JavaScript-объект, похожий на пропсы, с поправкой на приватность и полную подконтрольность компонентом. 🧑 `«Стейт»` предназначен для хранения и управления моделью `состояния` компонента.

🔍 Хозяйке на заметку:

`«Стейт»` — одна из фич, доступных классовым компонентам и недоступных функциональным.

`«Стейт»` описывает форму данных, от которых зависит компонент, и эта форма данных может меняться с течением времени.

! Важно:

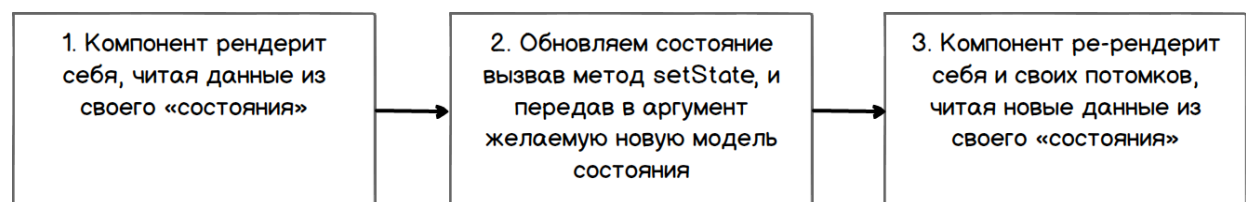
«Стейт» — инкапсулированная структура данных, приватная для каждого экземпляра классового компонента.

«Стейт» — динамический (как и почти любое состояние в природе). Для того, чтобы изменить «стейт» компонента, можно использовать метод `setState`, находящийся в распоряжении у каждого классового компонента, наследующего от `React.Component`.

Подпись метода `setState` выглядит следующим образом:

```
1 setState(updater[, callback])
```

Вызов метода `setState` всегда приводит к ре-рендеру компонента, за исключением случая, когда метод "жизненного цикла" `shouldComponentUpdate` возвращает `false`. Но об этом мы поговорим в следующих частях данного коспекта.



Первый аргумент метода `setState` — функция со следующей подписью:

```
1 (prevState, props) => stateChange
```

Где `prevState` — ссылка на объект «стейт» до обновления. Мутация этой ссылки запрещена. 🛑 Вместо этого необходимо описать изменения состояния, создав новый объект состояния с учётом объекта `prevState` и `props` (если это необходимо).

Например:

```
1 this.setState((prevState, props) => {
2
3     return {
4         counter: prevState.counter + props.step
5     };
6 });
```

Объекты `prevState` и `props` гарантированно содержат актуальные данные. А результат выполнения `setState` объединяется с `prevState` в «неглубоком» (англ. «shallow») режиме.

Второй аргумент метода `setState` — `callback`, необязательная функция-коллбэк. Данная функция будет вызвана после завершения операции `setState` и ре-рендера компонента.

# Обновление «СОСТОЯНИЯ» в обычном режиме

Иногда Оскар проводит вечера за написанием RPG-игр. Сегодня Оскар поделится с нами своими набросками его нового персонажа-колдуна, чтобы лучше разобраться со

«стейтом» React. 🧙‍♂️

🖥️ Пример кода 7.1:

```
1  export default class Wizzard extends Component {
2      state = {
3          name:      'Octavian',
4          spell:     'Wall of Fire',
5          research:  ''
6      }
7
8      render () {
9          const { name, spell, research } = this.state;
10
11         setTimeout(() => this.setState({
12             research: 'And my power grows as I gain new knowledge!'
13         }), 5000);
14
15         return (
16             <>
17                 <h1>Greetings!</h1>
18                 <p>
19                     I am { name }, a mighty wizzard! I can create {
20                     spell } on the fly! { research }
21                 </p>
22             </>
23         );
24     }
```

🔍 Хозяйке на заметку:

Использование таймера `setTimeout` в методе `render` не является хорошей практикой. В примерах кода данный таймер использован в демонстрационных целях и к использованию не рекомендуется.

В примере кода 7.1:

- На строке кода 2: объявлен объект «стейт» с двумя свойствами: `name` и `spell`. Доступ к «стейту» можно получить лишь непосредственно из тела компонента, обратившись к `this.state`;
- На строке кода 8: использован синтаксис деструктурирующего присваивания для изъятия необходимых данных из «стейта»;
- На строке кода 11: объявлен таймер `setTimeout`, дающий инструкцию вызвать

функцию `setstate` через `5000` миллисекунд. Спустя 5 секунд после изначального рендера сработает вызов функции `setstate`.

Вызов функции `setState` «ставит в очередь» изменение объекта состояния «стейт» компонента и даёт инструкцию React перерендерить данный компонент, а также все дочерние компоненты после того, когда «стейт» будет обновлён. 🧑

Это основной механизм обновления UI в ответ на действия пользователя и сетевые запросы.

👉 Совет бывалых:

Расценивай `setState` больше как `асинхронный запрос`, а не как `синхронную операцию`. Дело в том, что React может «отложить» выполнение функции `setState` в целях оптимизации производительности, «собрав» несколько последовательных вызовов `setState` в один, перередерив компонент и его детей только один раз вместо нескольких.

Поэтому React не гарантирует, что вызов `setState` будет обработан мгновенно, и чтение из состояния сразу после его обновления может не дать ожидаемого результата. Для чтения из состояния после его обновления лучше использовать второй аргумент метода `setState` — коллбэк, выполняющийся сразу после того, как `setState` обновит «стейт» компонента новым значением. Хотя для этого существует более подходящий механизм — метод "жизненного цикла" `componentDidUpdate`, но об этом чуть позже.

Существует ещё одна форма первого аргумента метода `setState` — объект. Передав объект первым аргументом методу `setState`, React одноуровнево объединит текущий «стейт» компонента с аргументом вызова. Такая форма `setState` тоже является `асинхронной`.

👉 Совет бывалых:

React всегда ссмерживает новое и старое «состояние» на один уровень, поэтому на первый взгляд может показаться, что изначальное состояние (англ. «`initial state`») указывать необязательно, однако хорошей практикой является указание изначального состояния для любого значения «состояния». Это добавляет ясности тому, что происходит в компоненте.


🖥️ Пример кода 7.2:

```
1 export default class PotionsStore extends Component {
2   state = {
3     potionsType: 'Empowering potions'
4   }
5
6   render () {
7     const { potionsType, potions } = this.state;
8     const potionsList = potions && potions.map((potion, index) => {
9
```

```

10         return <li key = { index }>{ potion }</li>;
11     });
12
13     setTimeout(() => this.setState({
14         potions: ['Potion of Great Power', 'Potion of Great Wisdom']
15     }), 5000);
16
17     return (
18         <section>
19             <h1>Potions Store</h1>
20             <h4>{ potionsType }</h4>
21             <ul>
22                 { potionsList }
23             </ul>
24         </section>
25     );
26 }
27 }

```

 Хозяйке на заметку:

Обрати внимание, что в примере кода выше использован альтернативный подход использования метода `setState` — с передачей объекта, описывающего обновление модели состояния вместо функции.

В примере кода 7.2 описывается компонент-магазин магических зелий.

- На строке кода 7 объявлена деструктуризация необходимых свойств «стейта», включая `potions` — массив с магическими зельями, не имеющий изначального (`initial`) состояния;
- Из-за этого на строке кода 8 приходится делать проверку на то, что `potions` — не `undefined`, описав инструкцию `potions && potions.map...`. Если не сделать этой проверки, будет ошибка.

Данный пример иллюстрирует неудобство ввиду необходимости дополнительных проверок свойств состояния перед их использованием. Поэтому рекомендуется всегда задавать изначальное состояние для каждого свойства состояния.

 Пример кода 7.3:

```

1  // PotionsOrder.js
2  import React, { Component } from 'react';
3  import Potion from './Potion';
4
5  export default class PotionsOrder extends Component {
6      render () {
7
8          return (
9              <>

```

```

10         <Potion initialOrder = { 2 } />
11     </>
12     );
13 }
14 }

```

```

1  // Potion.js
2  import React, { Component } from 'react';
3
4  export default class Potion extends Component {
5      state = {
6          potionsCount: 0,
7          total: 0
8      }
9
10     _orderMorePotions = () => {
11         this.setState({
12             potionsCount: this.state.potionsCount + 1
13         });
14     }
15
16     _checkout = () => {
17         this.setState((prevState, props) => ({
18             total: prevState.potionsCount + props.initialOrder
19         }));
20     }
21
22     render () {
23         const { potionsCount, total } = this.state;
24
25         return (
26             <>
27                 <p>You have { potionsCount } potions!</p>
28                 <p>Your order totals (initial order included): { total }
potions.</p>
29                 <button onClick = { this._orderMorePotions }>More
potions!</button>
30                 <button onClick = { this._checkout }>Checkout!</button>
31             </>
32         );
33     }
34 }
35 }

```

Пример кода 7.3 описывает компонент `PotionsOrder`, рендерящий компонент `Potion` с передачей ему пропса с числовым значением, описывающим количество зелий в заказе. Компонент описывает механизм увеличения количества зелий в заказе. Метод `_checkout` на строке кода 16 подсчитает желаемое количество зелий из «стейта» компонента и изначальное количество зелий в заказе, взяв это значение из пропсов компонента.

🧠 Ошибка:

Распространённым заблуждением является возможность использования унарного оператора `++` в реализации метода `setState`. Использование такого подхода приведет к багу. Взяв за пример компонент `Potion` из примера кода 7.3, использовав инструкцию `this.state.potionsCount++` в методе `_orderMorePotions`, состояние не будет обновлено. Так происходит потому, что унарный оператор `++` неявно мутирует свойство `potionsCount` объекта «стейт».

❌ Плохая практика:

```
1 _orderMorePotions () {
2   this.setState({
3     potionsCount: this.state.potionsCount++
4   });
5 }
```

✅ Хорошая практика:

```
1 _orderMorePotions () {
2   this.setState({
3     potionsCount: this.state.potionsCount + 1
4   });
5 }
```

! Важно:

Никогда не мутируй «стейт». Относись к «стейт», как к иммутабельной структуре данных.

## Обновление «состояния» в принудительном режиме


Дефолтное поведение компонента — это вызов метода `render` всякий раз при изменении пропса или «стейта». Но ты можешь инструктировать React отрендерить компонент принудительно в тех случаях, когда метод `render` зависит от некоторых внешних данных.




Для этого существует метод `forceUpdate`.

```
1 | component.forceUpdate(callback)
```


Аргументом метода `forceUpdate` является функция-коллбэк. Данный коллбэк вызовется после того, как метод `forceUpdate` отрендерит компонент принудительно.

 Хозяйке на заметку:

Вызов метода `forceUpdate` принудительно отрендерит компонент, пропуская проверку метода "жизненного цикла" `shouldComponentUpdate`, при этом у всех дочерних компонентов методы "жизненного цикла" сработают по обычному сценарию. Мы рассмотрим тему методов "жизненного цикла" компонента в соответствующей части конспекта.

В целом лучше не использовать `forceUpdate` без необходимости. Хорошей идеей будет опираться на механизм ре-рендеринга, основанный на изменении «стейт» или пропс. `forceUpdate` стоит использовать лишь тогда, когда это действительно необходимо. 

## «Контролируемые» и "неконтролируемые" компоненты

Преимущественно при работе с формами основной операционный поток включает в себя получение, хранение и обновление данных, введенных пользователем. React предоставляет удобную привязку «стейта» к элементам-инпутам. 

 Пример кода 7.4:

```
1 | export default class WizzardEmail extends Component {
2 |     state = {
3 |         email: ''
4 |     }
5 |
6 |     _handleChange = (event) => {
7 |         this.setState({
8 |             email: event.target.value
9 |         });
10 |    }
11 |
12 |    render () {
13 |        const { email } = this.state;
14 |
15 |        return (
16 |            <form>
17 |                <div>Your email is: { email }</div>
18 |                <input type = 'email' value = { email } onChange = {
19 |                    this._handleChange } />
                </form>
```



```

20         );
21     }
22 }

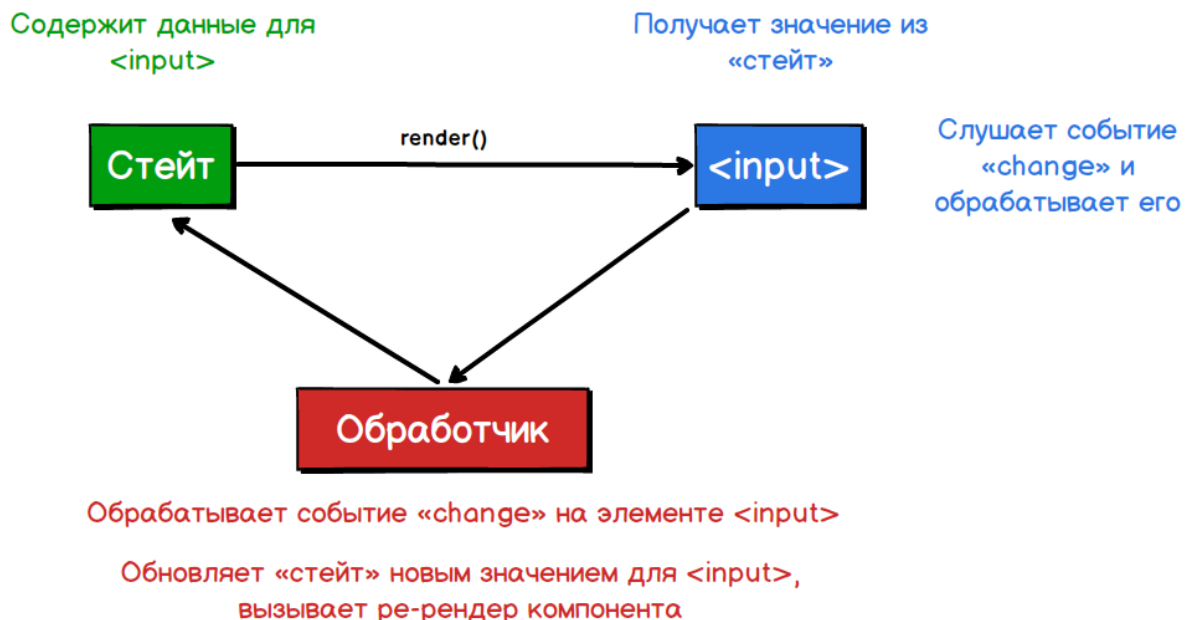
```

🔍 Хозяйке на заметку:

Вышеописанный пример кода содержит ещё нерассмотренную нами концепцию «синтетических событий», представленную идентификатором `event`. Мы рассмотрим концепцию «синтетических событий» React в последующих частях конспекта.

В примере кода 7.4 на строке кода 18 описан элемент `<input>`, получающий свое значение посредством привязки к атрибуту `value` значения, деструктурированного из «стейта» — `email`. Также у элемента `<input>` зарегистрирован слушатель события `change` с привязанным обработчиком данного события `_handleChange`, описывающим обновление «стейта» компонента при изменении значения элемента `<input>`.

Происходящее можно иллюстрировать схематически:



Иными словами, «контролируемый компонент» — это компонент, значение которого:

1. Определяет React;
2. Обрабатывает React.

Следуя подходу использования «контролируемых компонентов», разработчик всегда может быть уверен в «источнике правды» и не опасаться «сайд-эффектов», учитывая «одностороннюю направленность потока данных» React. 💎

«Неконтролируемый компонент» — это противоположность «контролируемого компонента» ввиду отсутствия его привязки к «стейту».

🖥️ Пример кода 7.5:

```

1 export default class WizzardLogin extends Component {

```

```

2      render () {
3
4          return (
5              <form>
6                  <label htmlFor = 'name'>
7                      Name:
8                      <input defaultValue = 'Merlin' id = 'name' type =
'text' />
9                  </label>
10                 <br />
11                 <label htmlFor = 'remember' >
12                     <input defaultChecked id = 'remember' type =
'checkbox' />
13                     Remember me
14                 </label>
15                 <br />
16                 <input type = 'submit' value = 'Log in' />
17             </form>
18
19         );
20     }
21 }

```

Пример кода 7.5 описывает «неконтролируемый компонент». Атрибуты `defaultValue` и `defaultChecked` задают дефолтное состояние элемента на UI, но никак не заданное программатически. Иными словами, состояние «неконтролируемого компонента» управляется DOM напрямую.

## Рефы

React поддерживает особый атрибут, доступный для применения к любому элементу — «реф» (англ. «ref», сокращенно от «reference», что в переводе означает «ссылка»). Данный атрибут может быть функцией или объектом.

Атрибут «Реф» открывает доступ к DOM-нодам, созданным методом `render` компонента.

Атрибут «Реф» в виде функции принимает коллбэк в виде значения. В первом аргументе этот коллбэк получит ссылку на DOM-элемент React-элемента, на котором «реф» применен.

 Пример кода 7.6:

```

1  export default class WizzardLogIn extends Component {
2      state = {
3          password: ''
4      }

```

```

5
6   _handlePasswordChange = (event) => {
7       this.setState({
8           password: event.target.value
9       })
10  }
11
12  _handleFocus = (event) => {
13      event.preventDefault();
14      this.passwordElement.focus();
15  }
16
17  render () {
18      const { password } = this.state;
19
20      return (
21          <form>
22              <input
23                  type = 'password'
24                  onChange = { this._handlePasswordChange }
25                  ref = { (passwordElement) =>
26                      this.passwordElement = passwordElement }
27                  value = { password }
28              />
29              <button onClick = { this._handleFocus }>Focus</button>
30              <input type = 'submit' value = 'Log in' />
31          </form>
32      );
33  }
34
35 }

```

В примере кода 7.6 на строке кода 24 описан атрибут «ref» со значением в виде коллбека, и аргументом этого коллбека является ссылка на DOM-элемент `<input>`, на котором этот «ref» применён. Это открывает возможность привязать эту ссылку к свойству класса `passwordElement` и использовать ссылку на элемент в виде свойства на строке кода 13, для императивного фокуса на элементе `<input>`.

В React v16.3 ввели ещё один более лаконичный способ использовать «ref».

 Пример кода 7.7:

```

1  import React from 'react';
2
3  export default class WizzardLogIn extends React.Component {
4      state = {
5          password: '',
6      };
7

```

```

8     passwordElement = React.createRef();
9
10    _handlePasswordChange = event => {
11        this.setState({
12            password: event.target.value,
13        });
14    };
15
16    _handleFocus = event => {
17        event.preventDefault();
18        this.passwordElement.current.focus();
19    };
20
21    render() {
22        const { password } = this.state;
23
24        return (
25            <form>
26                <input
27                    type = 'password'
28                    onChange = { this._handlePasswordChange }
29                    ref = { this.passwordElement }
30                    value = { password }
31                />
32                <button onClick = { this._handleFocus }>Focus</button>
33                <input type = 'submit' value = 'Log in' />
34            </form>
35        );
36    }
37 }

```

В примере кода 7.7 на строке кода 8 мы создаём «реф» посредством вызова `React.createRef()`. Теперь мы можем привязать `<input>` к данному рефу на строке кода 29. Данный вариант использования «реф» является более лаконичным, хотя вариант с коллбэком технически более мощный. Важная разница между «реф» в виде объекта и «реф» в виде функции: в первом варианте ссылка на желаемый элемент присваивается к свойству объекта `current`. Это можно заметить на строке кода 18. 🐶

👉 Совет бывалых:

«Рефы» стоит использовать обдуманно. «Рефы» открывают окно к императивным манипуляциям DOM, что контрастирует с декларативным подходом React. Используйте «рефы» только тогда, когда необходимо:

- Стригерить фокус (выделение текста или воспроизведение аудио);
- Стригерить императивную анимацию;
- Интегрировать приложение с вендорной библиотекой для манипуляции DOM.

# Подведём итоги

---

В этой части конспекта мы познакомились еще с несколькими ключевыми концепциями React, открывающими возможность хранить и управлять состоянием компонента. 🧑🏻💻

Давай подведём итог изученного:

- «Стейт» — это объект, хранящий приватные данные о состоянии компонента;
- «Стейт» нельзя мутировать;
- Для обновления «стейт» можно использовать метод `setState`, принимающий в первый аргумент функцию или объект, а во второй — коллбэк, который вызовется после того, как `setState` выполнится;
- Вызов метода `setState` влечёт за собой ре-рендер компонента;
- При обновлении объекта «state» происходит одноуровневый («shallow») мерж свойств объекта;
- «Контролируемый компонент» — это элемент, значение которого менеджит React;
- «Неконтролируемый компонент» — это элемент, значение которого не менеджит React;
- «Реф» — это ссылка на DOM-элемент.

Спасибо, что остаёшься с нами! 🙌 В следующей части этого конспекта мы разберёмся с «методами жизненного цикла» компонента. До встречи! 🐼

Мы будем очень признательны, если ты оставишь свой фидбек в отношении этой части конспекта на нашу электропочту [hello@lectrum.io](mailto:hello@lectrum.io).