НЕТОЛОГИЯ

СОБЫТИЯ И СОСТОЯНИЕ

АЛЕКСАНДР ШЛЕЙКО

АЛЕКСАНДР ШЛЕЙКО

Программист в Яндекс

a.shleyko@yandex.ru vk.com/shleiko @dustyo_O

ПЛАН ЗАНЯТИЯ

1. Обработчики событий
2. Состояние
3. Состояние в Functional компонентах
4. Состояние в Class-based компонентах
5. Передача состояния

СОБЫТИЯ

ИТОГИ ПРОШЛОЙ ЛЕКЦИИ

Компоненты — основа React.

В основе библиотеки React лежит использование компонентов. Компоненты представляют собой части пользовательского интерфейса, которые могут обладать собственным состоянием и поведением, а также содержать дочерние компоненты.

JSX помогает декларативно описывать UI компонентов.

СОБЫТИЯ

На прошлой лекции мы с вами научились пользоваться инструментом create-react-app, создавать собственные компоненты и отображать данные, используя JSX.

Естественно, этого не достаточно, хотелось бы реагировать на события, генерируемые пользователем - для начала самые простые вроде кликов, а затем уже и обрабатывать формы.

LIKE BUTTON

Создадим компонент `LikeButton`, который будет реагировать на наши действия и при каждом клике выводить сообщение в консоль.

Напоминаем сокращение VSCode для быстрого создания функционального компонента: `rfcp + TAB`

```
// файл components/LikeButton.js
function LikeButton(props) {
  let clicks = 0;
  return ( <button>Click me: {clicks}</button> );
}
// файл App.js
function App() {
  return ( <LikeButton /> );
}
```

Когда мы нажимаем на кнопку, ничего не происходит - это логично, т.к. не установлен обработчик события `click`.

ОБРАБОТКА СОБЫТИЙ

В чистом JS мы бы обрабатывали события следующим образом:

```
evtTarget.addEventListener('click', evt => { ... });
```

Но в JSX для большинства элементов (не всех EventTarget'ов, а именно элементов) принят другой подход: обработчики событий записываются как атрибуты в нотации camelCase:

```
function LikeButton(props) {
  let clicks = 0;
  const handleClick = evt => {
    console.log(evt);
    clicks++;
  };
  return (
    <button onClick={handleClick}>Click me: {clicks}</button>
  );
}
```

ОБРАБОТКА СОБЫТИЙ

Да-да, это не ошибка, прямо в JSX пишется обработчик `onClick`, в который кладётся ссылка на функцию-обработчик.

В чистом JS такой подход считается нерекомендуемым (когда вы в HTML пишете обработчик `onclick`). Но, во-первых, это не HTML, а во-вторых, React сам позаботится о том, чтобы добавить обработчик события:

ВАЖНО!

Обратим еще раз внимание на то, что мы просто передаём функцию, но не вызываем ее!

```
function LikeButton(props) {
  const handleClick = evt => {
    ...
  };
  return (
    <button onClick={handleClick()}>Click me: {clicks}</button>
  );
}
```

Этот код содержит ошибку! Т.к. мы просто вызвали функцию, а в `onClick` передали `undefined` (т.к. наша функция ничего не возвращает).

Соответственно, при кликах ничего в консоль выводится не будет.

ОБРАБОТКА СОБЫТИЙ

Мы также можем передавать стрелочную функцию, в которой вызывать обработчик:

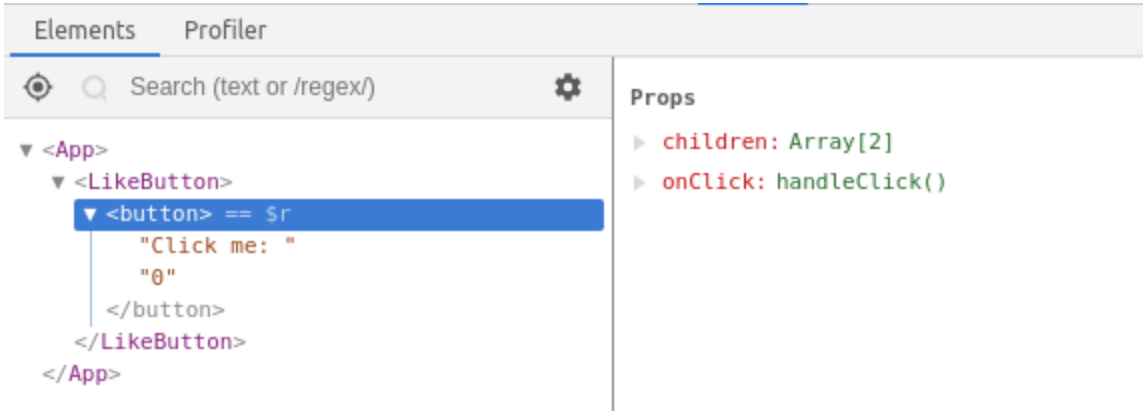
```
function LikeButton(props) {
  let clicks = 0;
  const handleClick = evt => {
    console.log(evt, clicks);
    clicks++;
  };
  return (
    <button
      onClick={evt => handleClick(evt)}
    >Click me: {clicks}</button>
  );
}
```

Чаще всего этот синтаксис используется, если вы хотите передать в обработчик ещё какие-либо параметры кроме самого события.

СПИСОК ОБРАБОТЧИКОВ

Обработчиков достаточно много, все они перечислены на странице документации React.

Посмотреть на все установленные обработчики вы можете в React Developer Tools:



ОБРАБОТКА СОБЫТИЙ

Ключевые моменты:

- в консоль выводится не `MouseEvent`, как было в чистом JS, а какой-то другой объект
- число на кнопке не обновляется, хотя в консоли мы видим, что значение реально увеличивается (а нам обещали, что всё будет "обновляться автоматически")

Давайте разбираться по порядку.

SYNTHETIC EVENT

В классическом DOM браузер передает объект события первым аргументом функции обработчика события. И мы можем для этого события отменить действие по умолчанию, вызвав метод события `preventDefault`.

В React мы тоже получаем объект события в обработчике события первым аргументом.

SYNTHETIC EVENT

Однако в React в функцию-обработчик передаётся не оригинальное событие DOM. Мы получаем *синтетический объект события*.

Он называется *синтетическим*, потому что React создает специальную обертку для нативного объекта, чтобы обеспечить единый интерфейс во всех браузерах.

Оригинальный объект события браузера доступен в свойстве `nativeEvent`.

Сделано это для того, чтобы добиться кросс-браузерной совместимости.

ДОСТУПНЫЕ АТРИБУТЫ

В возвращаемом объекте события мы получаем множество атрибутов, которые описывают наше произошедшее событие, среди которых:

- тип события `type`;
- отмена действия по умолчанию `preventDefault`
- предотвращение "продвижение события" `stopPropagation`
- на каком элементе производится обработка события `target`
- на каком элементе оно было перехвачено `currentTarget`
- состояние всплытия события `bubbles`
- отменено ли действие браузера по умолчанию `isDefaultPrevented`
- и другие.

Важно: в большинстве случае не нужно обращаться к `nativeEvent`.

КОНТЕКСТ ВЫЗОВА ОБРАБОТЧИКА

Еще одно важное отличие: контекст вызова самого обработчика события. В DOM обработчик события вызывается в контексте элемента, на котором он был установлен.

В React контекст вызыва обработчика событий не задан и равен `undefined`. Поэтому используем свойство события `currentTarget`, если нам нужен сам DOM-элемент:

```
1  const handleClick = evt => {
2    console.log(this);
3    console.log(evt.currentTarget);
4  };
```

РЕЗУЛЬТАТ В КОНСОЛИ

При клике в консоли браузера мы увидим следующий вывод:

```
undefined
<button>...</button>
```

ИТОГИ ПО SYNTHETIC EVENT

Таким образом, мы получаем обёртку над большим количеством событий браузера, позволяющую нам свободно взаимодействовать с ними.

CLASS-BASED КОМПОНЕНТЫ

С class-based компонентами всё немного сложнее (поэтому functional-компоненты и являются рекомендуемыми).

Давайте посмотрим на варианты обработки событий.

Для этого создадим компонент `LikeButtonClassBased` и обработаем точно такое же событие в нём (напоминаем, удобное сокращение VSCode для создания class-based компонента: `rpce + TAB`).

CLASS-BASED КОМПОНЕНТ

```
export class LikeButtonClassBased extends Component {
  static propTypes = { }
  constructor(props) {
    super(props);
    this.clicks = 0;
  }
  handleClick(evt) {
    console.log(evt, this.clicks);
  }
  render() {
    return (
      <button onClick={this.handleClick}>Click me: 0</button>
    );
  }
}
```

CLASS-BASED КОМПОНЕНТ

Но если мы выполним этот код, то получим чудесную ошибку:

Означает она только одно: что в методе `handleClick` `this` не ссылается на объект компонента.

CLASS-BASED КОМПОНЕНТ

Вариантов у нас целых три:

1. привязать `this`
2. воспользоваться синтаксисом, который ещё не принят в стандарт
3. воспользоваться стрелочными функциями

ВАРИАНТ ПЕРВЫЙ

```
export class LikeButtonClassBased extends Component {
  static propTypes = { }
  constructor(props) {
    super(props);
    this.clicks = 0;
    this.handleClick = this.handleClick.bind(this);
  }
  ...
}
```

И так придётся поступать для каждого обработчика!

ВАРИАНТ ВТОРОЙ

```
export class LikeButtonClassBased extends Component {
  static propTypes = { }
  constructor(props) {
    super(props);
    this.clicks = 0;
  }

  handleClick = evt => {
    console.log(evt, this.clicks);
  };
}
```

ВАРИАНТ ТРЕТИЙ

```
export class LikeButtonClassBased extends Component {
  static propTypes = { }
  constructor(props) {
    super(props);
    this.clicks = 0;
  }
  handleClick(evt) {
    console.log(evt, this.clicks);
  }
  render() {
    return (
      <button onClick={evt => this.handleClick(evt)}>
        Click me: 0
      </button>
    );
  }
}
```

CLASS-BASED КОМПОНЕНТ

Какой из вариантов использовать - решать вам. Но мы настоятельно рекомендуем использовать Functional компоненты, в которых такой проблемы нет.

СОСТОЯНИЕ

СОСТОЯНИЕ

React нам предлагает концепцию локального состояния компонента, изменение которого (состояния) ведёт к перерисовке компонента (рендерингу).

Естественно, рендеринг должен происходить эффективно, для этого используется механизм Reconciliation, о котором мы говорили на прошлом занятии.

Важная деталь: как мы видели выше - недостаточно просто обновлять локальную переменную (React не знает, что она хранит состояние), нужно как-то сообщить React'у об этом.

СОСТОЯНИЕ

Первое - что в состоянии нужно хранить только те данные, которые непосредственно влияют на отображение компонента (т.е. не надо хранить в состоянии вообще всё).

Второе - компонент должен хранить в состоянии только те данные, хозяином которых он является.

Давайте смотреть на примерах.

СОСТОЯНИЕ В FUNCTIONAL КОМПОНЕНТАХ

ХУКИ

Для функциональных компонентов React предоставляет хуки - специальные функции, предоставляющие нам возможность "встроиться" в состояние и жизненный цикл **функциональных** React-компонентов.

Важно: хуки, которые мы будем рассматривать в данной лекции, работают только для функциональных компонентах, в Class-based компонентах они работать не будут.

Хуки можно активно использовать начиная с React 16.8, если в вашем проекте используется более старая версия React, вам придётся использовать Class-based компоненты.

USESTATE

`useState` - хук, который позволяет сообщить React, что пара переменная и функция используются для хранения локального состояния и его изменения соответственно.

```
// сокращение useState в ES7 React snippets
// array destructuring
const [state, setState] = useState(initialState);
```

Где:

- `state` - имя переменной, которая хранит состояние
- `setState` - имя функции, которая будет устанавливать новое состояние
- `initialState` - значение начального состояния

USESTATE

На примере нашего компонента:

```
import React, { useState } from 'react'

function LikeButton(props) {
  const [clicks, setClicks] = useState(0);

  const handleClick = evt => {
    setClicks(prevClicks => prevClicks + 1);
  };
  return (
    <button onClick={handleClick}>Click me: {clicks}</button>
  );
}
```

После этого счётчик наконец заработал!

ХУКИ

Самое важное:

1. Хуки должны располагаться в самом начале вашей функции-компонента
2. Хуки не могут быть включены в if и другие конструкции (должны быть на верхнем уровне)

PREVIOUS STATE

В рассмотренном выше примере мы передаём в `setClicks` функцию, поскольку следующее значение количества кликов рассчитывается на базе предыдущего.

Вообще говоря, `state` может изменяться асинхронно, и React может группировать несколько изменений состояния. Поэтому чтобы всегда получать правильное предыдущее значение, стоит использовать предыдущее состояние.

Это не относится к сценариям, когда мы используем фиксированное значение, допустим: `setClicks(0)` - т.к. это значение не зависит от предыдущего.

ХУКИ

`useState` - не единственный хук, доступный в React. Остальные мы будем проходить по мере необходимости.

ВАЖНО

Ключевая особенность при работе с состоянием в React: мы не изменяем состояние, мы каждый раз создаём новое состояние.

React сам следит за состоянием и при изменении состояния заново отрисовывает компонент.

Важно, что меняем меняем состояние только через **set**-функцию, а не напрямую.

ИТОГИ

1. Состояние хранится в первом объекте, возвращаемом из `useState`
2. Функция, изменяющая состояние, хранится во втором объекте, возвращаемом из `useState`
3. Для использования предыдущего состояния стоит использовать аргумент `prev...` в функции, изменяющей состояние

СОСТОЯНИЕ В CLASS-BASED КОМПОНЕНТАХ

CLASS-BASED КОМПОНЕНТЫ

Давайте посмотрим, как это реализовать в Class-based компоненте.

Здесь всё снова немного сложнее, чем в Functional компоненте.

Мы должны объявить свойство `state`, в котором мы будем хранить наше состояние.

`state`

Объявить его (поле `state`) мы можем двумя способами:

```
export class LikeButtonClassBased extends Component {
  static propTypes = { }
  state = { clicks: 0 } // способ первый
  constructor(props) {
    super(props);
    this.state = { clicks: 0 }; // способ второй
  }
  ...
}
```

`state`

Важное замечание: инициализировать состояние можно так только один раз.

Все дальнейшние изменения должны делаться только с помощью метода `setState`.

setState

Независимо от выбранного способа объявления состояния, мы можем обновить (по факту - вернуть новое состояние) с помощью метода `setState`, который принимает в качестве аргумента предыдущее состояние и возвращает новое:

```
export class LikeButtonClassBased extends Component {
  ...
  handleClick(evt) {
    this.setState(prevState => {clicks: prevState.clicks + 1});
  }
  ...
  render() {
    return (<button onClick={evt => this.handleClick(evt)}>
      Click me: {this.state.clicks}
    </button>);
  }
}
```

`prevState` здесь ипользуется по тем же причинам, что и в функциональном компоненте.

ВАЖНЫЕ НЮАНСЫ СВОЙСТВА `state`

1. Свойство `state` только для чтения, если вы хотите изменить текущее состояние, то необходимо использовать метод `setState`
2. React следит за текущим состоянием и при каждом его изменении вызывает метод `render`, который перерисовывает компонент
3. Так как при изменении состояния вызывается метод `render`, то вызов в нём (в `render`) `setState` приведет к бесконечному циклу обновлений (React при этом выбросит ошибку)
4. Следует держать состояние компонента в чистоте и вызывать метод `setState` только по необходимости

ИТОГИ

1. Состояние хранится в переменной `state`
2. Изменить (по факту - установить новое состояние) можно только через `setState`
3. Для использования предыдущего состояния стоит использовать аргумент `prevState` в `setState`

ПЕРЕДАЧА СОСТОЯНИЯ

PURCHASE LIST

Пример с кнопкой-лайкером слишком простой, давайте попробуем сделать что-то посложнее, например, Purchase List (список покупок), в котором можно отмечать уже сделанные покупки или удалять ненужные.

Для этого, помимо `App` создадим ещё два компонента: `PurchaseList`, `PurchaseItem`, а также класс, описывающий саму покупку - `PurchaseModel`.

PURCHASEMODEL

Файл `/src/models/PurchaseModel.js`:

```
1  class PurchaseModel {
2    constructor (id, name, done = false) {
3      this.id = id;
4      this.name = name;
5      this.done = done;
6    }
7  }
8
9  export default PurchaseModel;
```

PURCHASELIST

Файл `/src/components/PurchaseList.js`:

```
1  import React from 'react'
2  import PropTypes from 'prop-types'
3
4  function PurchaseList(props) {
5    return (
6      <ul></ul>
7    )
8  }
9
10 PurchaseList.propTypes = {
11
12 }
13
14 export default PurchaseList
```

STATE

А теперь давайте думать - список объектов это состояние или нет?

Это непосредственно влияет на отображение компонента и компонент является хозяином этих данных.

Если это состояние, тогда мы должны использовать хук `useState`, чтобы хранить его:

```
const [items, setItems] = useState([
  new PurchaseModel(1, 'Pizza'), new PurchaseModel(2, 'Juice')
]);
```

STATE

Отображать список мы будем с помощью компонентов `PurchaseItem` (файл `PurchaseList.js`):

```
1  import React from 'react'
2  import PropTypes from 'prop-types'
3
4  function PurchaseList(props) {
5    const [items, setItems] = useState([
6      new PurchaseModel(1, 'Pizza'), new PurchaseModel(2, 'Juice')
7    ]);
8
9    return (
10     <ul>{items.map(o => <PurchaseItem key={o.id} item={o} />)}</ul>;
11   )
12 }
13
14 PurchaseList.propTypes = {}
15
16 export default PurchaseList;
```

PurchaseItem

```
function PurchaseItem(props) {
  const {name, done} = props.item;

  return (
    <li>{name}
      <button>{done ? 'Uncheck' : 'Check'}</button>
      <button>Remove</button>
    </li>
  )
}

PurchaseItem.propTypes = {
  item: PropTypes.instanceOf(PurchaseModel).isRequired
}

export default PurchaseItem;
```

`PurchaseItem`

А теперь давайте подумаем, нужно ли хранить в локальном состоянии `item`, который приходит в `props`?

Каково ваше мнение на этот счёт?

`PurchaseItem`

Правильный ответ: **не нужно**.

Потому что, во-первых, компонент не является хозяином этих данных (их ему присылают в `props`), а во-вторых, с первой лекции вы знаете, что `props` - read only.

Хорошо, но если мы не можем менять состояние, как же тогда реагировать на клики по кнопкам?

`props`

В `props` компоненту мы можем передавать не только "обычные" свойства, но так же и функции, которые компонент может вызвать по своем усмотрению (файл `PurchaseList.js`):

```
const onItemToggle = item => { ... };
const onItemRemove = item => { ... };

return (
  <ul>
    {items.map(
      o => <PurchaseItem key={o.id} item={o}
            onToggle={onItemToggle} onRemove={onItemRemove} />
    )}
  </ul>
);
```

Т.е. мы фактически придумали собственные обработчики событий и передали их в компонент в качестве `props`.

PurchaseItem

```
function PurchaseItem(props) {
  const {name, done} = props.item;

  const onToggle = () => {
    props.onToggle(props.item); // передаём объект
  }

  const onRemove = () => {
    props.onRemove(props.item); // передаём объект
  }

  return (<li>{name}
    <button onClick={onToggle}>{done ? 'Uncheck' : 'Check'}</button>
    <button onClick={onRemove}>Remove</button>
  </li>);
}
```

`props`

На самом деле, React никак не обрабатывает события на наших компонентах. И по сути, это просто результат передачи функции в `props`.

Нам даже не обязательно их называть `on*`, и записывать в *lowerCamelCase*, и вешать на события React-элементов.

Но для читабельности кода лучше именовать атрибуты событий в том же стиле и сохранять интерфейс обработчика.

PurchaseItem PropTypes

Поскольку свойства `onToggle` и `onRemove` дожны быть в `props` и должны быть обязательными, мы можем их объявить в `PropTypes` (файл `PurchaseItem.js`):

```
PurchaseItem.propTypes = {
  item: PropTypes.instanceOf(PurchaseModel).isRequired,
  onToggle: PropTypes.func.isRequired,
  onRemove: PropTypes.func.isRequired,
}
```

`onItemToggle` и `onItemRemove`

Реализуем `onItemToggle` и `onItemRemove` (файл `PurchaseList.js`)

Ключевая вещь: мы не мутируем исходный массив (даже элементы в нём), а возвращаем новый (методы `map` и `filter` возвращают новый объект).

```
1  const onItemToggle = item => {
2    setItems(prevItems => prevItems.map(o => {
3      if (o.id === item.id) {
4        return {...o, done: !o.done};
5      }
6      return o;
7    }));
8  };
9
10 const onItemRemove = item => {
11   setItems(prevItems => prevItems.filter(o => o.id !== item.id));
12 };
```

ОБНОВЛЕНИЕ КОМПОНЕНТОВ

Стоит отдельно отметить, что устанавливая новый `state` в родительском компоненте, мы получаем перерисовку вложенных компонентов (поскольку `props` для них меняется).

ПРОМЕЖУТОЧНЫЕ ИТОГИ

Мы с вами на живом примере разобрали как:

1. Как передавать состояние вложенным компонентам (для них это не состояние, а `props`)
2. Как передавать в `props` функции, которые может вызывать дочерний компонент
3. Как создавать новое состояние для списков

CLASS-BASED КОМПОНЕНТЫ

```
// не забудьте import React, { Component } from 'react';
export class PurchaseListClassBased extends Component {
  static propTypes = { }
  state = {
    items: [new PurchaseModel(1, 'Pizza'), new PurchaseModel(2, 'Juice')]
  }
  onItemToggle = item => {
    this.setState(prevState => ({items: prevState.items.map(o => {
      if (o.id === item.id) { return {...o, done: !o.done}; }
      return o;
    })}));
  }
  onItemRemove = item => {
    this.setState(prevState => ({items: prevState.items.filter(o => o.id !== item.id)}));
  }
  render() {
    const {items} = this.state;
    return (
      <ul>
        {items.map(o => <PurchaseItemClassBased key={o.id} item={o}
                          onToggle={this.onItemToggle} onRemove={this.onItemRemove} />)}
      </ul>
    )
  }
}
```

CLASS-BASED КОМПОНЕНТЫ

```
export class PurchaseItemClassBased extends Component {
  static propTypes = {
    item: PropTypes.instanceOf(PurchaseModel).isRequired,
    onToggle: PropTypes.func.isRequired,
    onRemove: PropTypes.func.isRequired,
  }

  onToggle = () => {
    this.props.onToggle(this.props.item); // передаём объект
  }

  onRemove = () => {
    this.props.onRemove(this.props.item); // передаём объект
  }

  render() {
    const { name, done } = this.props.item;
    return (
      <li>{name}
        <button onClick={this.onToggle}>{done ? 'Uncheck' : 'Check'}</button>
        <button onClick={this.onRemove}>Remove</button>
      </li>
    )
  }
}
```

НЕТОЛОГИЯ

Задавайте вопросы и напишите отзыв о лекции!

АЛЕКСАНДР ШЛЕЙКО

 a.shleyko@yandex.ru

 vk.com/shleiko

 @dustyo_0