REACT. Состояние в react компонентах

Компонент класс

Создадим на основе функционального компонента аналогичный *компонент, основанный на классе*:
cc-tab - сниппет

```
import React from "react";  // Это сторонняя библиотека. Импортировать ее первой, перед импортом др. модулей

class ListItem extends React.Component {
 render() {
  return (
   <li className="todo-item">
    <span className="todo-item-text">{this.props.task.title}</span>
    <div className="btn-group">
     <button role="button" className="btn btn-outline-dark btn-sm">
      Важное
     </button>
     <button role="button" className="btn btn-outline-danger btn-sm">
      Удалить
     </button>
    </div>
   </li>
  );
 }
}

export default ListItem;
```

При создании компонента класса обязателен импорт react, так как компонент класс при создании за основу использует класс React.Component, расширяя его, extends. Для использования свойств, компонент класс обращается к ним через своё свойство **this.props**. Например если при вызове компонента в него передается свойство **title**, то обращение к нему будет **this.props.title**

Состояние для компонента класса

Состояние в классовом компоненте описывается как поле класса с именем **state**, в которое записывается объект с свойствами данного **состояния**. Обращаться к состоянию компонента можно через this.state. Используем состояние чтобы задавать определенные классы для разметки:

```
class ListItem extends React.Component {
 state = {
  important: false,
 };

 render() {
  let classNames = "todo-item";

  if (this.state.important) {
   classNames += " important";
  }

  return (
   <li className={classNames}>...  </li>
  );
 }
}
```

Обработка кликов

Далее, для работы с состоянием элемента, нам необходимо совершать какое-нибудь действие на странице, например нажимать на кнопку “**Важное**” чтобы менять состояние компонента и устанавливать статус задачи как “Важная”.

Как работает **событие в атрибуте**:
<button **onClick**={**console.log("click!")**}Важное</button> - выведет один раз click! при запуске
<button **onClick**={**() => {console.log("click!")}**}Важное</button> - выводит click! при клике
<button **onClick**={**func**}Важное</button> - выводит клике срабатывает функция func. Скобки для запуска функции не пишем!

Сокращенный код:

```
class ListItem extends React.Component {
```

```
 onImportantClick = () => {
   console.log("Click Important!");
 };

 render() {
   return (
       <button onClick={this.onImportantClick} role="button" className="btn btn-outline-dark btn-sm"> Важное </button>
   );
 }
}
```

Обновление состояния компонента

Сделаем так чтобы метод `onImportantClick` менял состояние компонента. Он будет менять значение свойства important с `false` на `true`. В react нельзя напрямую менять значение объекта `state`. Делать это необходимо с помощью метода `**setState**`, в который нужно передать объект с обновленными свойствами состояния, пример:

```
 state = {
   important: false,
 };

onImportantClick = () => {
   this.setState({ important: true });
 };
```

В метод setState мы передаем объект только с теми свойствами state которые нужно изменить. Свойства state которые остаются неизменными мы не передаем. Если необходимо будет обновить сразу несколько свойств состояния, тогда будет передан объект с несколькими свойствами

Обновление состояния основываясь на текущем состоянии.

В случае если нам необходимо обновить состояние компонента основываясь на его текущем состоянии, тогда в метод setState необходимо передать не объект, а функцию, с параметром state и опираясь на него менять состояние.

Например при первом клике на кнопку “Важное” - мы отмечаем задачу как важная, задавая important: true однозначно и напрямую. Однако при повторном клике на кнопку “Важное” мы бы хотели снимать флаг important, то есть сказать что новое состояние задачи будет противоположным от текущего. (**this.state.important = !this.state.important**). Напрямую так делать нельзя. Сделаем это правильным способом:

```
import React from "react";

class ListItem extends React.Component {
// Состояние компонента
 state = {
   important: false,
 };

// Метод для изменения свойства important в состоянии компонента
 onImportantClick = () => {
   this.setState((state) => {
     return {
       important: !state.important,
// Внимание! Выше мы обращаемся именно к параметру state, а не к общему состоянию this.state
     };
   });
 };

 render() {
   let classNames = "todo-item";

   if (this.state.important) {
     classNames += " important";
   }

   return (
     <li className={classNames}>
       <span className="todo-item-text">
         {this.props.task.title}
       </span>
```

```
      <div className="btn-group">
       <button onClick={this.onImportantClick} role="button" className="btn btn-outline-dark btn-sm">
        Важное
       </button>
       <button role="button" className="btn btn-outline-danger btn-sm">
        Удалить
       </button>
      </div>
     </li>
    );
   }
}
```

Когда мы меняем состояние компонента, этот компонент **перерисовывается**. В данном случае перезапускается метод **render** заново.

Переносим данные приложения в state компонента App

В компоненте List все данные (список задач) описаны в массиве. Переделаем **app** в компонент класс, сделаем в нем state, и в его состоянии опишем все данные нашего приложения:

```
class App extends React.Component {
 state = {
  todoData: [
   { id: 0, title: "Выпить кофе" },
   { id: 1, title: "Сделать React приложение" },
   { id: 2, title: "Позавтракать" },
  ],
 };

 render() {
  return (
   <div>
    ...
     <List data={this.state.todoData} />...
   </div>
  );
 }
}
```

В List.js:

```
function List(props) {
 const render = props.data.map((task) => {
  return <ListItem key={task.id} task={task} />;
 });
...
 return <ul className="todo-list">{props.data.length > 0 ? render : emptyList}</ul>;
}
```

Переносим свойства задач в общий state

В **ListItem.js** у нас описывается каждая задача, есть state. Перенесем state в **App.js:**

```
class App extends React.Component {
 state = {
  todoData: [
   { id: 0, title: "Выпить кофе", important: false, done: false },
...
  ],
 };...
```

В ListItem.js:

```
 render() {
  let classNames = "todo-item";

  if (this.props.task.important) {
   classNames += " important";
  }

  if (this.props.task.done) {
   classNames += " done";
  }

  return (
```

```
    <li className={classNames}>
      <span onClick={this.onTitleClick} className="todo-item-text">
        {this.props.task.title}
      </span>
      <div className="btn-group">
        <button onClick={this.onImportantClick} role="button" className="btn btn-outline-dark btn-sm">
          Важное
        </button>
        <button role="button" className="btn btn-outline-danger btn-sm">
          Удалить
        </button>
      </div>
    </li>
  );
 }
```

В List.js было:

```
function List(props) {
 const render = props.data.map((task) => {
  return <ListItem key={task.id} task={task} />;
 });...
```

Запускаем метод внутри другого компонента

Клик по кнопке в **ListItem.js** меняет состояние. В этом файле он не будет работать, т.к. state у нас в app.js.

App (данные в state) - определим метод, меняющий состояние

List (функция) - передадим из App метод как свойство props внутрь компонента List

ListItem - клик по кнопке. Также передаем через props

App.js

```
 onToggleImportant = (id) => {
  console.log("onToggleImportant!", id);
 };

 render() {
  return (
   <div>...
    <List data={this.state.todoData} onToggleImportant={this.onToggleImportant} /> // передаем свойство onToggleImportant
... </div>
  );
 }
}
```

List.js

```
function List(props) {
 const render = props.data.map((task) => {
  return <ListItem onToggleImportant={props.onToggleImportant} key={task.id} task={task} />; // также передаем свойство onToggleImportant
 });...
}
```

ListItem.js

```
 render() {
...
  return (
   <li className={classNames}>
...
    <div className="btn-group">
     <button
      onClick={() => {
       this.props.onToggleImportant(this.props.task.id);
      }}
      role="button"
      className="btn btn-outline-dark btn-sm">
      Важное
     </button>
...
    </div>
   </li>
  );
 }
}
```

Изменение данных в state. Отмечаем "важные" задачи

App.js

```
 onToggleImportant = (id) => {
  this.setState((state) => {
    // 1. Найти индекс задачи в массиве todoData
    const index = state.todoData.findIndex((el) => el.id === id);

    // 2.  Сформировать новый объект данной задачи с обратным значением important
    const oldItem = state.todoData[index];
    const newItem = { ...oldItem, important: !oldItem.important };

    // 3. Формируем новый массив todos внедряя в него новый {} с задачей на тоже место, где был предшествующий
    const part1 = state.todoData.slice(0, index);
    const part2 = state.todoData.slice(index + 1);
    const newArray = [...part1, newItem, ...part2];

    return {
      todoData: newArray,
    };
  });
 };
```

Контролируемые элементы

Контролируемые элементы - элемент, который привязан к состоянию компонента.

```
class Footer extends React.Component {
 state = {
  taskTitle: "",
 };

 onInputChange = (e) => {
  console.log("Change!!!");

  this.setState({
    taskTitle: e.target.value,
  });
 };

 onSubmit = (e) => {
  e.preventDefault();
  if (this.state.taskTitle.trim()) {
    this.props.addItem(this.state.taskTitle);
  }
  this.setState({
    taskTitle: "",
  });
 };

 render() {
  return (
    <form onSubmit={this.onSubmit} className="footer">
      <input
        onChange={this.onInputChange}
        type="text"
        placeholder="Что необходимо сделать"
        className="form-control me-2"
        value={this.state.taskTitle}
      />
      <button type="submit" className="btn btn-primary">
        Добавить
      </button>
    </form>
  );
 }
}
```

Рефакторинг

Деструктуризация при импорте

```
import React from "react";
```

Меняем на

```
import {Component} from "react";
```

Деструктуризация props

```
const { task, onToggleDone, onToggleImportant, deleteItem } = this.props;
```

Далее меняем **this.props.task** на **task** и т.д.

Деструктуризацию можно сделать и сразу в параметре функции-компонента, например:

```
const BlogList = ( {blog} ) => {...}
```

Сокращенный вариант deleteItem

```
    deleteItem = (id) => {
        console.log('Delete', id);
        this.setState((state) => {
            const index = state.todoData.findIndex((el) => el.id === id);
            const part1 = state.todoData.slice(0, index);
            const part2 = state.todoData.slice(index + 1);
            const newArray = [...part1, ...part2];

            return {
                todoData: newArray,
            };
        });
    };
```

Меняем на

```
deleteItem = (id) => {
  this.setState((state) => {
    return {
      todoData: state.todoData.filter((el) => el.id !== id),
    };
  });
};
```

Сокращенный вариант onToggleImportant и onToggleDone

```
    onToggleImportant = (id) => {
        console.log('onToggleImportant click!', id);
        this.setState((state) => {
            // 1. Найти индекс задачи в массиве todoData
            const index = state.todoData.findIndex((el) => {
                return el.id === id;
            });

            // 2. Сформировать новый {} но с обратным значением important
            const oldItem = state.todoData[index];
            const newItem = { ...oldItem, important: !oldItem.important };

            // 3. Формируем новый массив todos внедряя в него новый {} с задачей на тоже место, где был предшествующий
            const part1 = state.todoData.slice(0, index);
            const part2 = state.todoData.slice(index + 1);
            const newArray = [...part1, newItem, ...part2];

            return {
                todoData: newArray,
            };
        });
    };
```

Меняем на

```
onToggleImportant = (id) => {
  this.toggleParam(id, "important");
```

```
};


toggleParam = (id, param) => {
  this.setState((state) => {
    const newArray = state.todoData.map((task) => {
      return {
        ...task,
        [param]: task.id === id ? !task[param] : task[param],
      };
    });
    return {
      todoData: newArray,
    };
  });
};
```

Состояние в функциональных компонентах. Reach hook - useState

Раньше у функциональных компонентов не было state. Потом появился **хук useState**. Нужно его **импортировать из react**
Например: в функциональном компоненте состояние задается следующим образом: используем **хук useState(параметр)**, в параметре - стартовое значение свойства нашего состояния. Этот хук возвращает **массив** с двумя элементами, 1-ый э-т - переменная, которая ссылается на **значение состояния**, 2-ой э-т - **ф-ция**, которая используется для **обновления этого состояния.** Сделаем деструктуризацию массива. Если у нас несколько свойств, то **еще раз вызываем useState**

```
import { Component, useState } from "react";

function App() {
  const [counter, setCounter] = useState(0);
  const [name, changeName] = useState("Mike");

  const reset = () => {
    setCounter(0);  // обновление состояния на 0
  };

  const increase = () => {
    setCounter((value) => {  // обновление состояния, основываясь на его текущем значении
      return value + 1;
    });
  };

  const decrease = () => {
    setCounter((value) => {  // обновление состояния, основываясь на его текущем значении
      return value - 1 >= 0 ? value - 1 : 0;
    });
  };

  return (
    <div className="App">
      <h1>Counter</h1>
      <Counter counter={counter} />
      <Controls reset={reset} increase={increase} decrease={decrease} />
      <h2>{name}</h2>
      <button
        onClick={() => {
          changeName("Bob");
        }}></button>
    </div>
  );
}

export default App;
```

Еще пример **обновления состояния**, основываясь **на текущем с известным id**:

```
const increase = (id) => {
      setCart((cart) => {
            return cart.map((product) => {
                  if (product.id === id) {
                        return {
                              ...product,
```

```
                    count: ++product.count
                }
            return product
            }
        });
    })
}
```

! Заметим, что обновление состояния, основываясь на его текущем значении нужно делать именно **через внутреннюю функцию с параметром**, а не просто вставить туда значение, н-р, **counter (из const [counter, setCounter] = useState(0);)**

Проброска состояния. Props drilling

Часто бывает нужно "пробросить" какие-либо свойства через несколько компонентов (они могут быть как в одном файле, так и в разных). 1-ый вариант - через **пропсы**.

```
// App - secondComponent - thirdComponent
export default function App() {
    const text = 'Hello World!'
    return (
        <div className='App'>
            <secondComponent text={text} />
        </div>
    );
}

const secondComponent = ({ text }) => {
    return <thirdComponent text={text} />;
};

const thirdComponent = ({ text }) => {
    return <h1>{text}</h1>;
};
```

Другой вариант - использовать **контекст:**

```
import './App.css';
import { createContext, useContext } from 'react';

const AppContext = createContext(null);  // Нужен ли аргумент null?

export default function App() {
    const text = 'Hello World!'
    return (
        <div className='App'>
            <AppContext.Provider value={{text: text}}>
                <SecondComponent/>
            </AppContext.Provider>
        </div>
    );
}

const SecondComponent = () => {
    return <ThirdComponent/>;
};

const ThirdComponent = () => {
    const {text} = useContext(AppContext)
    return (<h1>{text}</h1>);
};
```

В примере выше был создан контекст `**AppContext**` и передан как **провайдер** для вызова компонента `SecondComponent`.В `AppContext` заносим значения, которые в последствии можно будет использовать в рамках данного контекста. И теперь не нужно передавать их как свойства от компонента к компоненту. Достаточно лишь вызвать хук `**useContext**` данного контекста и получить необходимые данные.

Теперь мы сразу можем получить значение переменной text внутри `ThirdComponent`, которая была определена в App, без пробрасывания её через `SecondComponent`.

Контекст - аналог **addEventListener** - он "слушает" изменение контекста. Все потребители, которые являются потомками контекста, будут повторно рендериться, как только проп value у Provider изменится.