

Урок 3

Состояние. Хук useState.

План урока


1. Состояние React-компонента. useState.
2. Жизненный цикл компонента. Хуки.
3. Начинаем работу над приложением.
4. Модальное окно. Формы.

Состояние React-компонента. useState


Из прошлого урока стало понятно, что мы используем пропсы для передачи информации любому дочернему компоненту. При использовании пропсов есть важное правило: компонент не может изменять их в процессе своей работы. Функциональные React-компоненты — это [чистые функции](#). Чтобы инкапсулировать поведение компонента и сделать его интерактивным (то есть позволить ему меняться в ответ на действия пользователя), используют **состояние** компонента.

Попробуем написать простой компонент, который считает, сколько раз на него кликнули, и отображает эту информацию. Компонент будет состоять из текста и кнопки. Напомним, что в обычном HTML кнопке можно назначить атрибут **onclick** с произвольным JS-кодом. В React это делается похожим образом, только используется camelCase стиль.

Первое, что приходит в голову, — это объявить переменную внутри компонента и функцию, которая инкрементирует эту переменную и передается в onClick.



```
1  const Counter = () => {
2    let clickedTimes = 0;
3
4    const handleClick = () => {
5      ++clickedTimes;
6      console.log('Clicked!', clickedTimes);
7    }
8
9    return (
10     <div>
11       <h2>I was clicked {clickedTimes} times</h2>
12       <button onClick={handleClick}>Click me</button>
13     </div>
14   );
15 };
16
17 export default Counter;
18
```



```
1  import Counter from "./Counter";
2
3  function App() {
4    return (
5      <div>
6        <Counter/>
7      </div>
8    );
9  }
10
11 export default App;
12
```

Если вы попытаете это запустить, то обнаружите, что при клике текст не меняется, однако в консоли значение переменной все же увеличивается. Это объясняется тем, как React рендерит компоненты. После того, как мы объявили компоненты (в формате JSX или с помощью `React.createElement()`), React построит дерево элементов, которое затем передаст в React-DOM. React-DOM занимается переводом дерева элементов в обычное DOM-дерево, понятное браузеру. Когда в каких-то узлах дерева элементов происходят изменения, они не переносятся в DOM-дерево сразу. Вместо этого React строит новое дерево элементов с обновленными значениями и сравнивает это дерево с предыдущим. И только когда React находит разницу, происходит фаза рендеринга в браузере.

В случае изменения значения какой-то переменной, в нашем случае это `clickedTimes`, React не понимает, что в конкретном компоненте `Counter` произошли какие-то изменения, и не строит новое дерево элементов. Ему нужно явно сообщить, что “в этой функции мы меняем значение переменной, обновись”. Для этого и придумали состояние, меняя которое, мы сообщаем React о введенных изменениях.

Доступ к состоянию предоставляет функция `useState`, которую нужно импортировать из `'react'`. Она принимает начальное значение и возвращает пару значений: текущее состояние и функцию, которую нужно использовать, чтобы обновить состояние. К изначальному компоненту добавим еще и кнопку `Reset`, которая сбрасывает счетчик.

```
1 import { useState } from 'react';
2
3 const Counter = () => {
4   const [clickedTimes, setClickedTimes] = useState(0);
5
6   return (
7     <div>
8       <h2>I was clicked {clickedTimes} times</h2>
9       <button onClick={() => setClickedTimes((prev) => prev + 1)}>Click me</button>
10      <button onClick={() => setClickedTimes(0)}>Reset</button>
11    </div>
12  );
13 };
14
15 export default Counter;
16
```

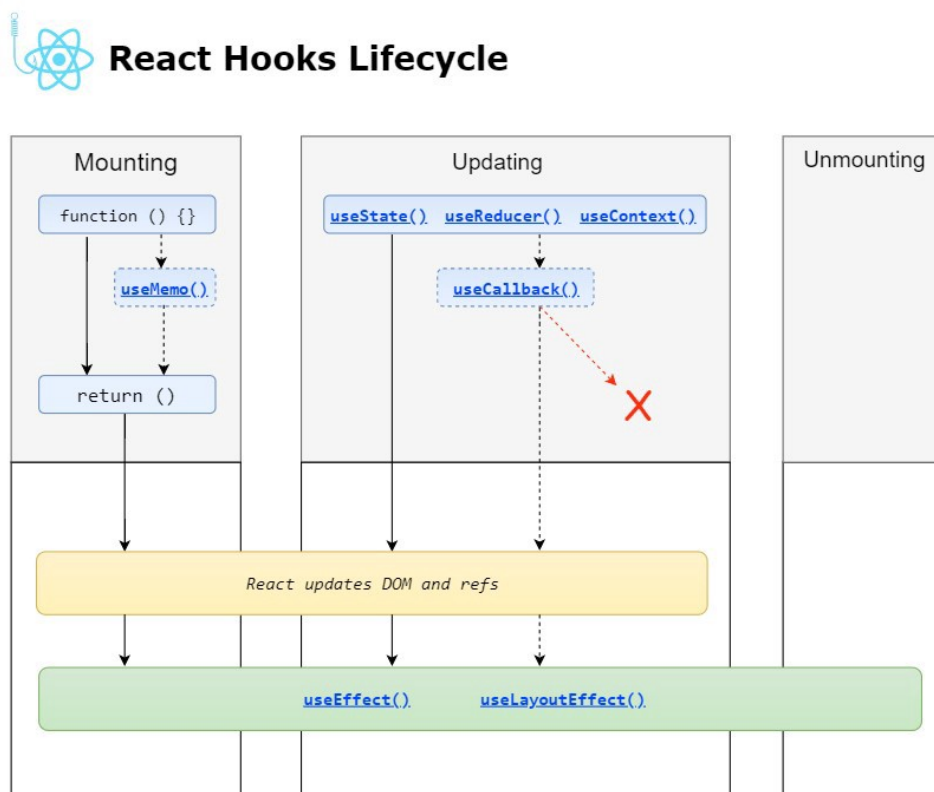
Теперь при вызове `setClickedTimes` React будет знать, что этот компонент надо будет перерендерить. Текст меняется корректно.

Жизненный цикл компонента. Хуки

Обновление React компонентов возможно благодаря жизненному циклу. Условно его можно разбить на 3 группы:

- Монтирование — первичное появление компонента в DOM-дереве;
- Обновление компонента — изменение отображения после изменения состояния;
- Демонтирование — удаление элемента из DOM-деревя.

Каждая из этих групп может быть описана с помощью двух этапов: этап рендеринга и этапа фиксации. Этап рендеринга нужен для того, чтобы React сформировал внутреннее дерево элементов, а этап фиксации — для применения изменений уже в DOM-дереве.




Чтобы убедиться в том, что жизненный цикл существует, создадим специальный компонент `VisibilityManager`, который может показать или скрыть наш компонент `Counter`. Добавим также `console.log()` внутри обоих компонентов.

```
1 import Counter from './Counter';
2 import { useState } from 'react';
3
4
5 const VisibilityManager = () => {
6   const [isVisible, setIsVisible] = useState(false);
7
8   console.log('VisibilityManager rendered');
9
10  return (
11    <div>
12      <button onClick={() => setIsVisible(true)}>Show</button>
13      <button onClick={() => setIsVisible(false)}>Hide</button>
14      {isVisible && <Counter/>}
15    </div>
16  );
17 };
18
19 export default VisibilityManager;
20
```

Можно заметить несколько вещей:

1. `VisibilityManager` выводит сообщение при загрузке страницы. Происходит монтирование.
2. `Counter` выводит сообщение при нажатии `Show`. Происходит монтирование.
3. Сообщение `Counter` выводится при нажатии кнопок. Происходит обновление.
4. Только `VisibilityManager` выводит сообщение при нажатии `Hide`. Происходит демонтирование `Counter`. Именно поэтому при следующем нажатии `Show` значение счетчика сбрасывается, так как `Counter` монтируется заново.

В классовых компонентах этапы жизненного цикла можно отслеживать в методах `componentDidMount()` и `componentWillUnmount()`, но в функциональных компонентах вместо этого используются **хуки**. Хук — это специальная функция, которая позволяет «подцепиться» к возможностям `React`. Например, хук `useState` предоставляет



функциональным компонентам доступ к состоянию React. Названия хуков всегда начинаются с “use”. Мы узнаем про другие хуки чуть позже.

Начинаем работу над приложением

Теперь мы знаем достаточно о React, чтобы начать разрабатывать настоящее приложение, а не учебные примерчики. В течение следующих уроков мы будем писать приложение “Личный словарь”.

Приложение позволяет хранить словарь из слов и выражений английского языка. Описание частей приложения:

1. Страница словаря.

Пользователь видит список добавленных слов в формате слово-перевод. С этой страницы можно открыть форму добавления нового слова. Нужно указать пару слово-перевод. В следующем уроке мы используем API какого-нибудь переводчика для автоматического перевода слова.

2. Карточка слова.

Отображается полная информация о слове. Карточка открывается без перехода на новую страницу, в виде модального окна.

Начнем разработку со страницы словаря. Для начала сделаем таблицу со списком слов и кнопкой для добавления нового слова. Создадим компоненты `WordsList.js`, `WordRow.js` и базовый компонент `Button.js`. Все стили написаны в файле `index.css`, который можно найти в репозитории к этому уроку. `WordsList` поместим в папку `pages`, остальные компоненты — в папку `components`.

```

1  import WordRow from "../components/WordRow";
2  import Button from "../components/Button";
3
4  const words = [
5    {
6      word: 'cat',
7      translation: 'кошка'
8    },
9    {
10     word: 'dog',
11     translation: 'собака'
12   },
13   {
14     word: 'code',
15     translation: 'код'
16   },
17 ]
18
19 const WordsList = () => {
20   return (
21     <div>
22       <div className="card center">
23         <h1 className="heading">Список слов</h1>
24         <Button text="Добавить слово"/>
25         <table className="content-table">
26           <thead>
27             <tr>
28               <th>№</th>
29               <th>Слово</th>
30               <th>Перевод</th>
31             </tr>
32           </thead>
33           <tbody>
34             {words.map((word, index) =>
35               <WordRow index={index + 1} key={word.word} {...word}/>)}
36             </tbody>
37           </table>
38         </div>
39       </div>
40     );
41   };
42
43   export default WordsList;
44

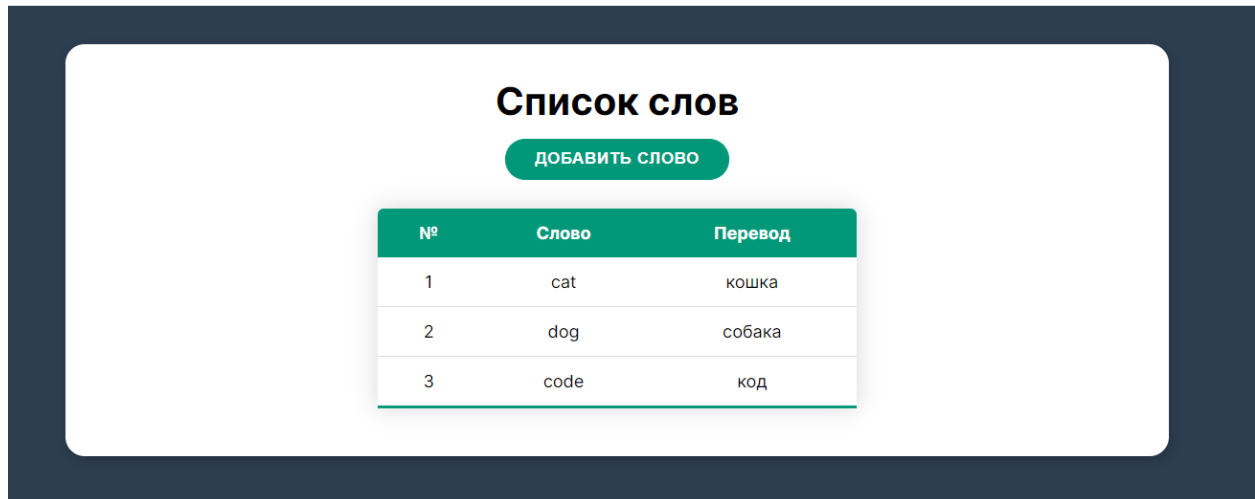
```

```
1  const WordRow = ({ index, word, translation }) => {
2    return (
3      <tr>
4        <td>{index}</td>
5        <td>{word}</td>
6        <td>{translation}</td>
7      </tr>
8    );
9  };
10
11  export default WordRow;
12
```

```
1  const Button = ({ text }) => {
2    return (
3      <button className="btn primary" onClick={() => console.log('clicked button')}>{text}</button>
4    );
5  }
6
7  export default Button;
8
```

```
1  import WordsList from "../pages/WordsList";
2
3  function App() {
4    return (
5      <main className="app">
6        <WordsList/>
7      </main>
8    );
9  }
10
11  export default App;
12
```


Результат:



К кнопке мы пока добавили только логирование при клике. Также обратите внимание на пропс `key` при использовании компонента `WordRow`. Это очень желательный пропс при рендеринге списков. Ключи помогают React определять, какие элементы были изменены, добавлены или удалены. Эта информация нужна, чтобы React мог сопоставлять элементы массива в течение времени и эффективно менять DOM-дерево. Если вы не укажете `key`, React будет выводить предупреждения в консоли браузера.

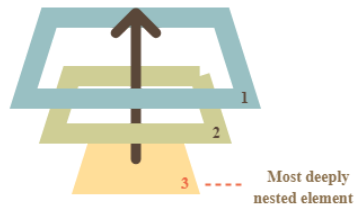
Теперь поработаем над добавлением элементов в список.

Модальное окно. Формы

При нажатии на кнопку “Добавить слово” будем показывать модальное окно с формой. Создадим компонент `Modal`. Управлять отображением будем с помощью переменной состояния `showCreateModal`. Также передадим пропс `onClick` в кнопку, чтобы при клике вызывать `setShowCreateModal(true)`.

Заметьте, что в `div` с содержимым модального окна мы добавили `onClick` со `stopPrapagation()`. Функция `stopPropagation()` останавливает дальнейшее “всплытие” события. Это необходимо, чтобы при клике на содержимое событие этого клика не передалось на внешний `div` и окно не закрылось.

Иллюстрация “всплытия” события:



```
1  const Modal = ({ active, setActive, children }) => {
2    return (
3      <div
4        className={({active}) ? 'modal active' : 'modal'}
5        onClick={() => setActive(false)}
6      >
7        <div
8          className={({active}) ? 'modal-content active' : 'modal-content'}
9          onClick={(e) => e.stopPropagation()}
10        >
11          {children}
12        </div>
13      </div>
14    );
15  };
16
17  export default Modal;
18
```

```
1  const Button = ({ text, onClick }) => {
2    return (
3      <button className="btn primary" onClick={onClick}>{text}</button>
4    );
5  }
6
7  export default Button;
8
```

```

1  const WordsList = () => {
2      const [showCreateModal, setShowCreateModal] = useState(false);
3
4      return (
5          <div>
6              <div className="card center">
7                  <h1 className="heading">Список слов</h1>
8                  <Button text="Добавить слово" onClick={() => setShowCreateModal(true)} />
9                  /* без изменений */
10             </div>
11             <Modal active={showCreateModal} setActive={setShowCreateModal}>
12                 Форма
13             </Modal>
14         </div>
15     );
16 };
17
18 export default WordsList;
19

```

Обратите внимание на пропс children в Modal. В него попадает все, что мы поместили “внутри” компонента, в нашем случае это текст “Форма”. Это помогает создавать переиспользуемые компоненты.

Теперь займемся непосредственно добавлением в список. Создадим базовый компонент Input и компонент формы CreateWordForm.

```

1  const Input = (props) => {
2      return (
3          <input className="form-input" type="text" {...props} />
4      );
5  }
6
7  export default Input;
8

```

```
1 import { useState } from "react";
2 import Button from "../Button";
3 import Input from "../Input";
4
5 const CreateWordForm = ({ onCreateWord }) => {
6   const [word, setWord] = useState('');
7   const [translation, setTranslation] = useState('');
8
9   return (
10     <div className="form">
11       <h2 className="form-title">Создать слово</h2>
12       <Input
13         value={word}
14         placeholder="Введите слово"
15         onChange={(e) => setWord(e.target.value)}
16       />
17       <Input
18         value={translation}
19         placeholder="Введите перевод"
20         onChange={(e) => setTranslation(e.target.value)}
21       />
22       <Button
23         text="Готово"
24         onClick={() => onCreateWord({word, translation})}
25       />
26     </div>
27   );
28 }
29
30 export default CreateWordForm;
31
```

```
1  const WordsList = () => {
2    const [showCreateModal, setShowCreateModal] = useState(false);
3
4    const createWord = (word) => {
5      words.push(word);
6      setShowCreateModal(false);
7    };
8
9    return (
10     <div>
11       {/* без изменений */}
12       <Modal active={showCreateModal} setActive={setShowCreateModal}>
13         <CreateWordForm onCreateWord={createWord} />
14       </Modal>
15     </div>
16   );
17 };
18
19 export default WordsList;
```

Заметьте, как мы организовываем работу с Input: при вводе пользователем символа создается событие change, поэтому в пропс onChange мы передаем функцию изменения состояния. Таким образом, любые изменения в обычном <input> влияют на состояние компонента. Также мы создали функцию createWord, в котором это состояние используется, чтобы добавить слово в массив и закрыть модальное окно. При этом слово автоматически добавится в таблицу! В этом и заключается “магия” реактивности.