

Урок 4

Хуки. Стилизация. JSON Server.

Обзор

1. Хук `useRef`. Доступ к DOM элементу.
2. Хук `useEffect`.
3. CSS-модули.
4. JSON Server.

Хук `useRef`. Доступ к DOM элементу

Следующий хук, который мы рассмотрим, — это хук `useRef`. Он принимает начальное значение и возвращает объект. У этого объекта есть свойство `current`, по которому мы можем получить это самое значение. Особенность хука состоит в том, что этот объект хранится в течение всего жизненного цикла компонента и изменение значения не влечет за собой перерендер. По этой причине его используют для хранения данных, которые не должны зависеть от циклов обновления компонента и изменение которых не должно приводить к обновлению. Например, добавим `useRef` в наш компонент `Counter`. Напомним, что при нажатии на кнопку компонент перерендеривается каждый раз. Но что если нам нужно не отображать количество нажатий, а просто собирать статистику? Нам бы не хотелось, чтобы в процессе сбора статистика компонент перерисовывался.

```
1 import { useRef } from 'react';
2
3 const Counter = () => {
4   const clickedTimes = useRef(0);
5
6   console.log('Counter rendered', clickedTimes.current);
7
8   const gatherStats = () => {
9     ++clickedTimes.current;
10    console.log('Button clicked', clickedTimes.current);
11  }
12
13  return (
14    <div>
15      <h2>Gathering stats...</h2>
16      <button onClick={gatherStats}>Click me</button>
17    </div>
18  );
19 };
20
21 export default Counter;
22
```

Теперь сообщение о рендере выводится только один раз, при монтировании компонента, однако значение `clickedTimes` меняется.

`useRef` часто используется для ручного доступа к DOM. Ранее мы говорили о том, что, используя реактивность, нам не нужно вручную манипулировать DOM, однако могут возникать ситуации, когда вам требуется императивно изменить дочерний элемент, обойдя обычный поток данных. React предоставляет лазейку для такого случая.

Документация React указывает следующие ситуации, в которых подобное использование `useRef` является оправданным:

- Управление фокусом, выделение текста или воспроизведение медиа.
- Императивный вызов анимаций.
- Интеграция со сторонними DOM-библиотеками.

При этом желательно избегать использования `useRef` в ситуациях, когда задачу можно решить декларативным способом.

Допустим, нам нужно реализовать валидацию формы создания слова и при пустом поле выделить поле красной рамкой и перевести на него фокус. Чтобы получить доступ к DOM-элементу надо в компоненте создать переменную `myRef` с помощью `useRef()` и затем нужному HTML-элементу передать атрибут `ref={myRef}`. После того, как компонент будет смонтирован, в `myRef.current` будет лежать нужный DOM-элемент.

Однако в нашем случае `Input` — это React-компонент, а не обычный HTML-элемент, поэтому просто передать ему атрибут `ref` мы не можем. Это логично, ведь в компоненте может быть много HTML-элементов, и React не сможет понять, к какому из них добавить `ref`. Поэтому мы используем такой “хак”: в компонент `Input` передадим пропс `innerRef` и уже в нем присвоим `ref` элементу `<input>`.

Также мы динамически меняем `className` в зависимости от значений полей при отправке формы.

```
1  const Input = (props) => {
2    const {innerRef, ...restProps} = props;
3    return (
4      <input type="text" ref={innerRef} {...restProps} />
5    );
6  }
7
8  export default Input;
9
```

```

1 import { useState, useRef } 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   const [formSubmitted, setFormSubmitted] = useState(false);
9
10  const wordInput = useRef();
11  const translationInput = useRef();
12
13  const submitForm = () => {
14    if (!translation) {
15      translationInput.current.focus();
16    }
17    if (!word) {
18      wordInput.current.focus();
19    }
20    setFormSubmitted(true);
21    if (word && translation) {
22      onCreateWord({word, translation});
23      setWord('');
24      setTranslation('');
25      setFormSubmitted(false);
26    }
27  };
28
29  return (
30    <div className="form">
31      <h2 className="form-title">Создать слово</h2>
32      <Input
33        value={word}
34        placeholder="Введите слово"
35        onChange={(e) => setWord(e.target.value)}
36        className={`form-input ${formSubmitted} ? ((word) ? 'valid' : 'invalid') : ''}`
37        innerRef={wordInput}
38      />
39      <Input
40        value={translation}
41        placeholder="Введите перевод"
42        onChange={(e) => setTranslation(e.target.value)}
43        className={`form-input ${formSubmitted} ? ((translation) ? 'valid' : 'invalid') : ''}`
44        innerRef={translationInput}
45      />
46      <Button
47        text="Готово"
48        onClick={submitForm}
49      />
50    </div>
51  );
52 }
53
54 export default CreateWordForm;
55

```

Чтобы не делать такой “хак” с передачей `innerRef`, в React существует “перенаправление рефов” с помощью `React.forwardRef`. Это позволяет взять `ref` из атрибутов компонента, и передать («перенаправить») его одному из дочерних компонентов.

```
1 import { forwardRef } from "react";
2
3 const Input = forwardRef((props, ref) => {
4   return (
5     <input type="text" ref={ref} {...props} />
6   );
7 });
8
9 export default Input;
10
```

```
1 <Input
2   value={word}
3   placeholder="Введите слово"
4   onChange={(e) => setWord(e.target.value)}
5   className={`form-input ${formSubmitted} ? ((word) ? 'valid' : 'invalid') : ''}`
6   ref={wordInput}
7 />
```

Хук useEffect

Хук `useEffect` позволяет совершать действия, напрямую не связанные с отображением компонента. Команда React называет такие действия “побочными эффектами”, отсюда название хука. `useEffect` чаще всего используют для

- сетевых запросов;
- подписки на изменения состояния;
- вызов `setTimeout` и `setInterval`.

В классовых компонентах можно было определить методы `componentDidMount`, `componentDidUpdate`, и `componentWillUnmount`, которые вызывались в соответствующие стадии жизненного цикла. Хук `useEffect` может заменить все эти методы.

`useEffect` помещается в тело компонента и принимает 2 аргумента: функцию, которая вызывается в ответ на изменение состояния, и массив зависимостей (необязательный аргумент). Функция будет вызвана тогда, когда зависимости в этом массиве поменялись. Чтобы было более понятно, рассмотрим пример. Вернемся к компоненту `Counter` и добавим в него `useEffect`:

```
1 import { useState, useEffect } from 'react';
2
3 const Counter = () => {
4   const [clickedTimes, setClickedTimes] = useState(0);
5
6   console.log('Counter rendered', clickedTimes);
7
8   useEffect(() => console.log('Counter useEffect'), []);
9
10  return (
11    <div>
12      <h2>I was clicked {clickedTimes} times</h2>
13      <button onClick={() => setClickedTimes((prev) => prev + 1)}>Click me</button>
14      <button onClick={() => setClickedTimes(0)}>Reset</button>
15    </div>
16  );
17 };
18
19 export default Counter;
20
```

В консоли браузера увидим, что лог с `useEffect` выводится позже, чем лог `'Counter rendered'`. Если мы обратимся к картинке из раздела “Жизненный цикл”, то увидим, что `useEffect` выполняется на `'Commit phase'` уже после `'Render phase'`, что объясняет такой порядок вывода. К тому же, мы увидим, что при клике на кнопки выводится лог `'Counter rendered'`, что логично, так как мы обновляем состояние, но не выводится лог с `useEffect`. Все верно, ведь мы передали пустой массив зависимостей. Значит, функция в `useEffect` сработает только на первоначальный `render` компонента и не вызовется в процессе обновления состояния. Если же мы добавим в массив переменную `clickedTimes`, то функция в `useEffect` будет также выполняться каждый раз, когда обновляется `clickedTimes`.

Подумаем, как применить `useEffect` в нашем приложении. Давайте при каждом изменении слова в форме делать запрос к API какого-нибудь переводчика и подставлять результат в форму перевода. Отправку запроса с помощью стандартного `fetch` вынесем в `utils/translate.js`:

```
1  const translate = (word) => {
2    return fetch(`https://api.mymemory.translated.net/get?q=${word}&langpair=en|ru`);
3  };
4
5  export default translate;
6
```

```
1  useEffect(() => {
2    if (!word) return;
3
4    const translateWord = async () => {
5      const resp = await translate(word);
6      const respJSON = await resp.json();
7      const matches = respJSON.matches;
8      if (matches.length) {
9        setTranslation(matches[0].translation);
10     }
11   }
12
13   const timer = setTimeout(() => translateWord(word), 800);
14
15   return () => clearTimeout(timer);
16 }, [word]);
```

Подробнее рассмотрим тело функции в `useEffect`: мы создаем и вызываем `async` функцию `translateWord`, которая совершает запрос, переводит ответ в формат JSON и затем присваивает значение полю `translation` с помощью хука `useState`. Поля ответа `matches` и `translation` получены путем анализа ответа от API. Такой вариант был бы уже вполне рабочим, но внесем одно улучшение. Печатая слово, мы отправляем запрос на каждую букву. То есть, для перевода слова из 10 букв нам пришлось отправить 10 запросов. Чтобы избежать этого, создадим таймер, который переводит слово спустя 800 мс. Также `useEffect` позволяет функции вернуть другую функцию “сброса”, которая срабатывает при демонтировании, либо перед следующим рендером. В эту функцию помещают очистку ресурсов, чтобы избежать утечек памяти. Поэтому получается такая цепочка действий:

1. Вводим букву в поле `word`.
2. Создается таймер, который переведет слово через 800 мс.
3. В это время пользователь печатает вторую букву.
4. Состояние изменилось. Компонент должен перерендериться. Вызывается функция “сброса”. Таймер удаляется.

5. Пользователь перестает печатать.
6. Таймер успешно срабатывает, мы видим переведенное слово.

При этом вся эта цепочка записывается всего в несколько строк. Еще один пример яркой декларативности React.

CSS-модули

Перейдем к вопросам стилизации компонентов. Сейчас мы используем один файл `index.css` для стилизации всех компонентов приложения. Использовать один глобальный файл стилей достаточно плохая идея по нескольким причинам:

- в крупных проектах стили занимают тысячи строк, найти конкретный стиль в одном файле очень сложно;
- жесткая связка между всеми файлами компонентов и файлом `index.css`;
- могут возникнуть коллизии имен, если два разработчика использовали два одинаковых селектора, но стилизовали их по-разному.

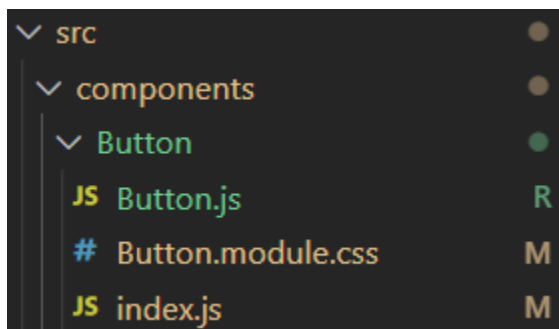
Для борьбы с коллизиями имен была придумана методология именования классов [БЭМ](#) от Яндекса.

CSS-модули предлагают другой подход. CSS модуль — это CSS файл, в котором все имена классов и анимаций имеют локальную область видимости по умолчанию. Чтобы избежать коллизий, достаточно в каждом компоненте использовать свой CSS-модуль. Механизм их действия достаточно прост: к имени класса добавляется уникальный хэш, который отличает его от остальных классов в глобальной области видимости. Например, если в двух компонентах используется элемент `<input className="some-input" ... />` и для обоих компонентов написаны CSS-модули, то на самой странице мы увидим классы `"some-input__<хэш1>"` и `"some-input__<хэш2>"`.

Чтобы начать использовать CSS-модули в нашем проекте, воспользуемся одним алгоритмом:

1. Для каждого компонента создадим отдельную директорию.
2. Поместим туда `.js` файл и создадим в ней файл `<название_компонента>.module.css`.
3. В `<название_компонента>.module.css` скопируем все стили, относящиеся к компоненту.
4. В `.js` файле импортируем объект со стилями и используем их в коде.

Приведем пример для компонента Button. Для остальных компонентов процесс абсолютно аналогичен.



```
1 import styles from './Button.module.css';
2
3 const Button = ({ text, onClick }) => {
4   return (
5     <button className={` ${styles.btn} ${styles.primary}`} onClick={onClick}>{text}</button>
6   );
7 }
8
9 export default Button;
10
```

```
1 export { default } from './Button';
2
```

Файл `index.js` не обязателен, но он позволяет писать

`import Button from "../components/Button";`

вместо

```
import Button from "../components/Button/Button";
```

Посмотрим на конечные классы в консоли разработчика:

```
<button class="Button_btn_U7rRG Button_primary_PwBqo">Добавить слово</button> == $0
```

Как видно, к нашим классам действительно добавляется хэш.

После всех подобных преобразований структура файлов в src выглядит так:

```
src
├── components
│   ├── Button
│   │   ├── Button.js
│   │   ├── Button.module.css
│   │   └── index.js
│   ├── CreateWordForm
│   │   ├── CreateWordForm.js
│   │   ├── CreateWordForm.module.css
│   │   └── index.js
│   ├── Input
│   │   ├── index.js
│   │   ├── Input.js
│   │   └── Input.module.css
│   ├── Modal
│   │   ├── index.js
│   │   ├── Modal.js
│   │   └── Modal.module.css
│   ├── Counter.js
│   ├── VisibilityManager.js
│   └── WordRow.js
├── pages \ WordsList
│   ├── index.js
│   ├── WordsList.js
│   └── WordsList.module.css
```

JSON Server

Сейчас наше приложение имеет один крупный недостаток: вся введенная информация теряется при перезагрузке страницы. Это логично, поскольку под капотом мы всего лишь манипулируем DOM и никуда не сохраняем полученные от пользователя данные. Чтобы решить эту проблему, необходимо провести некоторую работу на бэкенде:

- создать базу данных;
- написать сервер, который может брать данные из БД и сохранять в нее;
- настроить API на сервере.

Однако вместо этого используем инструмент под названием [JSON Server](#). Это библиотека, которая предназначена для быстрого создания простенького REST API на основе одного json файла. Ее часто используют фронтенд-разработчики, ~~которые не умеют в бэкенд~~ которым надо провести быстрое прототипирование или проверить какую-то фичу, пока бэкенд не готов.

Ставится JSON Server как обычный пакет (можно установить глобально с помощью флага -g):



После этого создадим файл `db.json` со списком слов:



```
1 {
2   "words": [
3     {
4       "id": 1,
5       "word": "cat",
6       "translation": "кошка"
7     },
8     {
9       "id": 2,
10      "word": "dog",
11      "translation": "собака"
12     },
13     {
14       "id": 3,
15       "word": "code",
16       "translation": "код"
17     }
18   ]
19 }
```

Чтобы запустить сервер, необходимо в директории с db.json выполнить команду:



```
$ json-server --watch db.json --port 8000
```

После запуска сервера по пути <http://localhost:8000/words> будет доступен список слов; через /id также можно получить конкретное слово (например, <http://localhost:8000/words/1>).

Используем это в нашем приложении. В файл `utils/words.js` вынесем функции отправки запросов. В компоненте `WordsList` получим слова внутри `useEffect` и сохраним добавленное слово в функции `createWord`:

```


1  const API_PATH = 'http://localhost:8000';
2
3  export const getWords = () => fetch(`${API_PATH}/words`);
4
5  export const postWord = (word) => {
6      return fetch(`${API_PATH}/words`, {
7          method: 'POST',
8          headers: {
9              "Content-Type": "application/json",
10         },
11         body: JSON.stringify(word),
12     })
13 }

```

```

1  import { getWords, postWord } from '../utils/words';
2
3  const WordsList = () => {
4      const [words, setWords] = useState([]);
5      const [showCreateModal, setShowCreateModal] = useState(false);
6
7      useEffect(() => {
8          getWords()
9              .then((res) => res.json())
10             .then((res) => setWords(res))
11             .catch((err) => console.log(err));
12     }, []);
13
14     const createWord = (word) => {
15         postWord(word)
16             .then(() => setWords([...words, word]))
17             .catch((err) => console.log(err));
18         setShowCreateModal(false);
19     };
20
21     return (
22         /* без изменений */
23     );

```



Теперь введенные слова не сбрасываются после перезагрузки страницы. Они будут доступны даже после перезапуска React-приложения или при входе в приложение с другого браузера на другом устройстве.