

ReactJS. Базовый курс

Жизненный цикл компонента

[React 17.0.1]



На этом уроке

1. Познакомимся с понятием жизненного цикла компонента.
2. Узнаем о том, что такое хуки в React.
3. Познакомимся с понятием state.
4. Научимся работать со state и props.
5. Научимся рендерить списки из массивов данных
6. Узнаем о фрагментах React

Оглавление

[На этом уроке](#)

[Теория урока](#)

[State](#)

[Стейт в классовых компонентах](#)

[Контролируемые формы](#)

[Жизненный цикл компонента](#)

[Методы жизненного цикла](#)

[Хуки жизненного цикла](#)

[Props](#)

[Правила хуков. Другие хуки React.](#)

[Рендер массивов. Фрагмент](#)

[Глоссарий](#)

[Дополнительные материалы](#)

[Используемые источники](#)

Теория урока

Жизненный цикл компонента, состояние (state) и свойства (props) компонента.

State

На предыдущем уроке уже упоминалось, что компоненты могут не только отображать переданные им статические данные, но и инкапсулировать некоторую логику. Это достигается, в том числе, за счет использования переменных состояния, или стейта. Стейт (state) - специальная переменная, сохраняющая переданное ей значение внутри компонента.

Взгляните на следующий пример:

```
import React from 'react';

export function Counter() {
  return (
    <div>
      <span className="counter">0</span>
      <button className="counter-button">Click!</button>
    </div>
  )
}
```

Здесь на странице рендерится кнопка и некоторый счетчик - значение переменной counter. Добавим интерактивности - пусть по нажатию на кнопку отображаемое число увеличивается. Для этого сперва значение переменной count сохраним в стейт с помощью специальной функции-хука useState:

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

export function Counter() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <span className="counter">{count}</span>
      <button className="counter-button">Click!</button>
    </div>
  )
}
```

useState - один из основных хуков React (хуки - специальные функции, используемые в функциональных компонентах; подробно рассмотрены в разделе 4). Он принимает в себя начальное значение переменной и возвращает массив из двух значений - первым элементом этого массива является текущее значение переменной, а вторым - функция-сеттер для обновления значения этой переменной. Присвоение полученных из useState значений переменным обычно происходит с помощью деструктуризации массива:

```
const [variable, setVariable] = useState('initial');
```

```
// то же самое, что

const variableState = useState('initial');
const variable = variableState[0];
const setVariable = variableState[1];
```

Как правило, функцию-сеттер называют так же, как переменную, но с префиксом `set` (например, для переменной `messages` функцию называют `setMessages`). В эту функцию требуется передать значение, которое необходимо установить переменной стейта (в данном случае переменной `count`).

Добавим теперь функцию `updateCount` и назначим ее кнопке обработчиком события `onClick`:

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

export function Counter() {
  const [count, setCount] = useState(0);

  const updateCount = () => {
    setCount(count + 1);
  }

  return (
    <div>
      <span className="counter">{count}</span>
      <button className="counter-button" onClick={updateCount}>+1</button>
    </div>
  )
}
```

Теперь при нажатии на кнопку значение, хранящееся в переменной `count`, будет увеличено, компонент обновится и отобразит обновленное значение в браузере.

Внимание!

Функцию `setCount` можно вызывать не только со значением, которое требуется установить переменной, но и с коллбэком:

```
const updateCount = () => {
  setCount((prevCount) => prevCount + 1);
}
```

В этом случае аргументом в коллбэк будет передано (автоматически) предыдущее значение переменной (в данном случае `count`), а вернуть из функции требуется новое значение. Такой подход используют, как правило, когда необходимо изменить/обновить стейт, учитывая предыдущее его значение. Необходимость использования такой записи связана, в частности, с тем, что обновление состояния происходит асинхронно, и к моменту вызова `setCount(count + 1)`

переменная `count` может иметь устаревшее значение. Вызывая `setCount` с функцией, мы всегда будем иметь доступ к актуальному значению `count` (`prevCount` в примере).

Стейт в классовых компонентах

Аналогично, данный функционал можно реализовать на классовых компонентах. В этом случае используется объект `this.state`, хранящий значения всех переменных состояния, а для обновления используется метод `this.setState`:

```
import React from 'react';

export class Counter extends React.Component {
  state = {
    count: 0, // начальное значение
  };

  updateCount = () => {
    const { count } = this.state; // деструктуризация стейта
    this.setState({ count: count + 1 });
  };

  render() {
    return (
      <div>
        <span className="counter">{this.state.count}</span>
        <button className="counter-button" onClick={this.updateCount}>
          Click!
        </button>
      </div>
    );
  }
}
```

Важно отметить, что при наличии нескольких свойств в `this.state`, при обновлении одной из них не требуется передавать весь объект стейта:

```
import React from 'react';

export class Counter extends React.Component {
  state = {
    count: 0, // начальное значение
    name: 'Alexander'
  };

  updateCount = () => {
    const { count } = this.state; // деструктуризация стейта
    this.setState({ count: count + 1 });
    // this.state.name не изменится
  };
}
```

```

render() {
  return (
    <div>
      <span className="name">{this.state.name}</span>
      <span className="counter">{this.state.count}</span>
      <button className="counter-button" onClick={this.updateCount}>
        Click!
      </button>
    </div>
  );
}
}

```

Здесь в стейте хранятся переменные count и name, и происходит вызов setState с объектом, содержащим только count. Однако после такого вызова name не исчезает и по-прежнему равна установленному ранее значению.

Кроме того, как и в функциональных компонентах, можно вызывать setState с функцией.

Как уже упоминалось, setState выполняется асинхронно. Из этого следует две важные вещи:

```

updateCount = () => {
  this.setState({ count: 3 });
  this.setState({ name: 'noname' });
  this.setState({ count: 2 });
  console.log(this.state);
};

```

1. В данном примере React сможет объединить вызовы setState и выполнить одно обновление компонента (эквивалентно одному вызову такого вида: `this.setState({ count: 2, name: 'noname' });`);
2. `console.log` выполнится до обновления стейта.

Если существует необходимость выполнить некоторую функцию после того как обновился стейт, в классовой компоненте следует передать эту функцию вторым аргументом `this.setState`:

```

updateCount = () => {
  this.setState({ count: 2, name: 'noname' },
    () => {
      console.log(this.state);
    });
};

```

Внимание!

Стейт следует изменять **только** с помощью вызова `setState` (в классовых компонентах) или функции-сеттера, возвращаемой из `useState` (в функциональных). Не изменяйте переменные состояния напрямую! Использование записи вида `this.state = { count: 1 }` допустимо только в конструкторе классowego компонента. После вызова `setState` React, при необходимости, обновит UI в соответствии с изменениями состояния. При присваивании значений в стейт напрямую такого обновления выполнено не будет.

```
export class Counter extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      count: 0,
      name: 'Alexander'
    }
  }

  render() { /*...*/ }
}
```

Контролируемые формы

Достаточно часто стейт используется для создания контролируемых форм ввода.

```
export function Counter() {
  const [value, setValue] = useState('');

  const handleChange = (event) => {
    setValue(event.target.value);
  }

  return (
    <div>
      <input type="text" value={value} onChange={handleChange} />
    </div>
  )
}
```

Здесь элементу `input` устанавливаются атрибуты `value` - значение, отображаемое в поле - и `onChange` - функция, вызываемая при изменении этого значения. В качестве `value` в этом случае устанавливается значение переменной стейта, а для `onChange` - функция, изменяющая эту

переменную. В этом случае значение, введенное пользователем в поле ввода, хранится внутри компонента, и для доступа к нему нет необходимости получать его из DOM.

Обратите также внимание на то, что при изменении значения в поле ввода (событие `onChange`) аргументом в функцию-обработчик попадает специальный объект события. Это т.н. `SyntheticEvent` - специальная обертка над браузерным событием, которую React использует для поддержания кросс-браузерности.

Жизненный цикл компонента

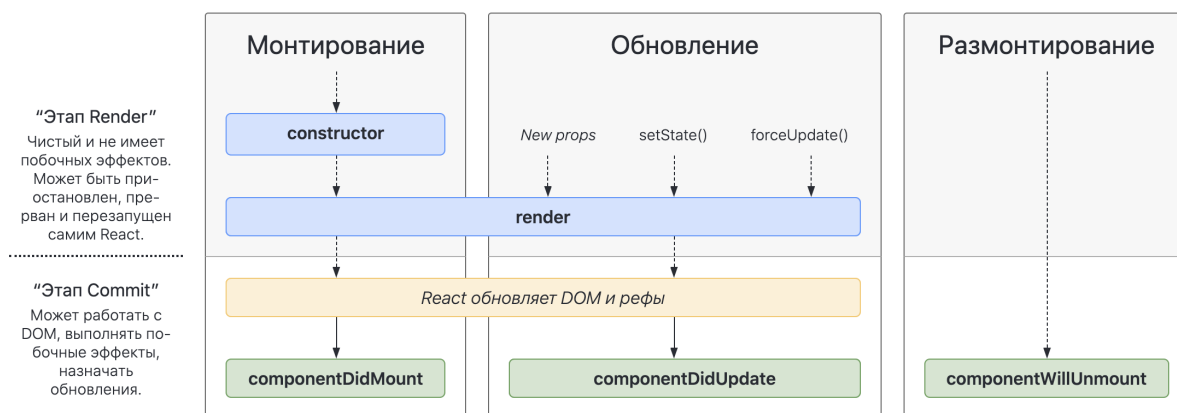
Как мы увидели из предыдущих примеров, при изменении переменной стейта с помощью вызова `setState` происходит обновление компонента. Компонент обновляется за счет внутренних механизмов React, при этом могут происходить некоторые дополнительные действия (например, React вызовет некоторые функции или методы компонентов).

Каждый компонент за время своего существования проходит определенные этапы, которые называют стадиями жизненного цикла:

1. Монтирование (`mount`): компонента не было на странице и он добавляется туда в первый раз (либо он существовал, был удален и появляется заново).
2. Обновление (`update`): компонент уже существует, но обновились данные, от которых он зависит - стейт или пропсы.
3. Размонтирование (`unmount`): компонент существует и происходит его удаление из DOM.

Методы жизненного цикла

На каждом этапе жизненного цикла React вызывает соответствующие методы классовых компонентов или коллбэки хуков в функциональных. Рассмотрим сперва на примере классowego компонента:



Как видно, на начальном этапе будет вызван конструктор класса, затем метод `render`, затем `componentDidMount` (обратите внимание - `componentDidMount` вызывается **после** того как компонент

отрендерен и существует в DOM). В этом методе, как правило, запускаются побочные эффекты компонента - отправляются запросы на сервер, ставятся таймеры, подписки и т.п.

При изменении стейта (через `setState`) или пропсов будут вызваны методы `render` и `componentDidUpdate`. Этот метод первыми двумя аргументами получает значения пропсов и стейта до обновления, и при необходимости здесь можно выполнить сравнение пропсов или стейта до и после обновления.

При размонтировании (перед тем как компонент будет удален из памяти) будет вызван метод `componentWillUnmount` - он служит в первую очередь для отмены ранее отправленных в компоненте запросов, таймеров, подписок и т.п. Это позволяет избежать утечек памяти.

Рассмотрим следующий пример:

```
export class Counter extends React.Component {
  constructor(props) {
    super(props);

    console.log('constructor');
  }

  componentDidMount() {
    console.log('componentDidMount');
  }

  render() {
    console.log('render');

    return (
      <div>rendered!</div>
    );
  }
}
```

При такой записи в консоли отобразится следующее:

```
constructor
```

```
render
```

```
componentDidMount
```

Усложним пример, добавив дочерний компонент:

```
export class Counter extends React.Component {
  constructor(props) {
```

```

    super(props);

    console.log('constructor');
  }

  componentDidMount() {
    console.log('componentDidMount');
  }

  render() {
    console.log('render');

    return (
      <div>
        rendered!
        <Child />
      </div>
    );
  }
}

class Child extends React.Component {
  constructor(props) {
    super(props);

    console.log("child constructor");
  }

  componentDidMount() {
    console.log("child componentDidMount");
  }

  render() {
    console.log("child render");

    return <div>rendered!</div>;
  }
}

```

В консоли будет выведено:

`constructor`

`render`

`child constructor`

`child render`

`child componentDidMount`

`componentDidMount`

Обратите внимание, что сперва вызывается метод `componentDidMount` дочернего, а затем родительского компонента.

Хуки для работы с жизненным циклом

В функциональных компонентах не существует методов жизненного цикла, однако аналогичное поведение достигается использованием хука `useEffect`. Он принимает два аргумента - коллбэк и массив зависимостей. Коллбэк `useEffect` будет выполнен:

1. При монтировании компонента
2. При изменении одной из зависимостей

То есть, при использовании следующей записи

```
export function Example(props) {
  const { name } = props.name;

  useEffect(() => {
    console.log('useEffect');
  }, [name]);

  return (
    <div>
      {name}
    </div>
  )
}
```

коллбэк будет выполнен при первом вызове функции-компонента, а затем только при изменении пропса `name`.

Если указать вторым аргументом пустой массив - функция будет выполнена один раз, только после монтирования компонента (аналогично `componentDidMount`). Если не передавать второй аргумент -

коллбэк выполняется при каждом обновлении (аналогично `componentDidUpdate`, но, в отличие от этого метода, коллбэк `useEffect` выполнится и на первом рендере тоже).

```
export function Example(props) {
  const { name } = props.name;

  useEffect(() => {
    console.log('like didMount');
  }, []);

  useEffect(() => {
    console.log("like didUpdate");
  });

  return (
    <div>
      {name}
    </div>
  )
}
```

Если из коллбэка `useEffect` вернуть функцию, то она будет вызвана на этапе размонтирования компонента:

```
useEffect(() => {
  return () => {
    console.log('like willUnmount')
  }
}, []);
```

С использованием хука эффекта можно легко решить задачу выполнения некоторой функции в ответ на изменение переменной стейта:

```
export function Counter() {
  const [count, setCount] = useState(0);

  const updateCount = () => {
    setCount((prevCount) => prevCount + 1);
  }

  useEffect(() => {
    console.log(count);
  }, [count]);

  return (
    <div>
      <span className="counter">{count}</span>
      <button className="counter-button" onClick={updateCount}>Click!</button>
    </div>
  )
}
```

```
    </div>
  )
}
```

`console.log(count)` будет выполняться каждый раз, когда изменяется значение переменной `count`.

Внимание!

Из `useEffect` можно вернуть только функцию. Т.к. функции, объявленные с помощью `async`, всегда неявно возвращают промис, то коллбэк `useEffect` объявлять `async` нельзя. Вместо этого можно объявить отдельную асинхронную функцию и вызвать ее внутри `useEffect`.

Кроме перечисленных, существуют и другие, менее используемые методы жизненного цикла.

Подробнее узнать о них можно [здесь](#) и [здесь](#).

Props

На прошлом уроке мы уже использовали пропсы, чтобы передавать дочернему компоненту статические данные. Рассмотрим теперь случай, когда пропсами передается значение переменной стейта:

```
function Child(props) {
  return (
    <span>{props.count}</span>
  )
}

export function Example() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <Child number={count} />
    </div>
  )
}
```

Теперь дочерний компонент получает значение переменной из стейта родителя. Мы можем передавать эту переменную и другим дочерним компонентам. Кроме того, пропсами можно передавать функции - например, для изменения переменной стейта. Рассмотрим пример, где один из дочерних компонентов получает и отображает значение переменной, а другой - коллбэк для ее изменения.

```
function Child(props) {
  return (
```

```

    <span>{props.count}</span>
  )
}

function Button(props) {
  return (
    <div className="button" onClick={props.onClick}>Click!</div>
  )
}

export function Example() {
  const [count, setCount] = useState(0);

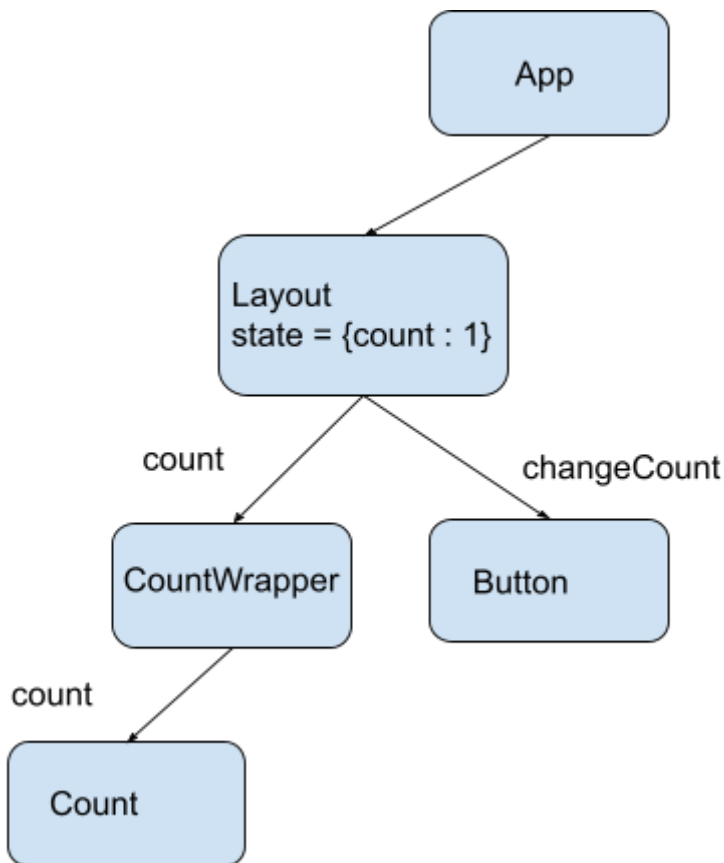
  const changeCount = () => {
    setCount(prevCount => prevCount + 1);
  }

  return (
    <div>
      <Child number={count} />
      <Button onClick={changeCount} />
    </div>
  )
}

```

Заметьте, что, хотя переменная `count` отображается в компоненте `Child`, ее значение хранится в родительском компоненте `Example` - а компонент `Button` получает функцию для изменения этой переменной. Такой подход называется поднятие стейта - если два компонента должны иметь доступ к одной и той же переменной стейта, ее помещают в ближайшего общего родителя этих компонентов.

Компоненты должны хранить и передавать переменные только сверху вниз - от родительских компонентам дочерним. Изменение данных родительского компонента из дочернего следует осуществлять только с помощью переданных дочернему компоненту пропсами коллбэков.



Компонентам Count и Button, не связанным между собой, требуется доступ к переменной count. Эта переменная помещена в стейт их ближайшего общего предка - Layout. Из него необходимые данные передаются пропсами соответствующим компонентам.

Как видно из схемы жизненного цикла компонента (в предыдущем разделе), при вызове `setCount` произойдет обновление компонента и вызов функции-компонента `Example` (для классовых компонентов - вызов метода `render`). При этом все переменные, созданные в теле функционального, создаются заново. Исключение составляют переменные и функции, созданные с использованием хуков - таким образом переменная `count`, созданная с помощью `useState`, сохраняет свое значение между рендерами.

Правила хуков. Другие хуки React.

Хуки - специальные функции, которые вызываются в теле функционального компонента и используются, в первую очередь, для сохранения значений переменных между вызовами этой функции. Кроме уже описанных выше `useState` и `useEffect`, существует множество других, наиболее часто встречающимися из которых являются:

1. `useCallback` - используется для оборачивания функций, объявленных в теле функционального компонента.

```
const changeCount = useCallback(() => {  
  setCount(1)  
})
```

```
}, []);
```

Функция, обернутая в `useCallback`, “запоминается” и будет пересоздаваться только в случае изменения любой из зависимостей - переменных, указанных в массиве, переданным вторым аргументом. Как правило, `useCallback` используют для функций, которые:

- а. Передаются пропсами другому компоненту
 - б. Используются в других хуках в том же компоненте
2. `useMemo` - служит для мемоизации значений. К примеру, если в компоненте используется результат вычисления достаточно “тяжелой” функции (проход по массиву и т.п.), имеет смысл обернуть это вычисление в `useMemo`, чтобы не выполнять его на каждом рендере. Как и `useCallback`, и `useEffect`, функция будет выполняться только на первом рендере, а затем при изменении одной из зависимостей, однако, в отличие от `useCallback`, который возвращает переданную ему функцию, `useMemo` возвращает результат вычисления переданной функции.

```
const elem = useMemo(  
  () => props.hugeArray.find(el => el.show),  
  [props.hugeArray]  
);
```

3. `useRef` - возвращает специальный объект с мутируемым свойством `current`. Зачастую используется для получения ссылки на DOM-элемент (подробнее рефы будут рассмотрены в дальнейших уроках)

```
export function Example(props) {  
  
  const elem = useRef(null);  
  return (  
    <div ref={elem}>  
      Refs  
    </div>  
  )  
}
```

Для корректной работы с хуками следует соблюдать следующие правила:

1. Хуки должны вызываться только из функциональных компонентов на верхнем уровне (не из вложенных функций) или из других хуков.
2. Внутри функционального компонента хуки должны вызываться всегда в одном и том же порядке. То есть, нельзя оборачивать вызов хука в условия или циклы, а также после

условного return. Для реализации работы хуков React отслеживает порядок их вызовов в компоненте, и нарушение этого порядка вызовет ошибку.

3. Указывайте в зависимостях хуков все переменные и пропсы, которые используются в коллбэке хука, в том числе и те, которые созданы с помощью других хуков. Исключение составляют переменные, объявленные вне тела компонента, а также функция-сеттер, возвращаемая из useState.

```
export function Example(props) {
  const badIdea = () => {
    const err = useCallback(() => {
      // вызовет ошибку
    }, [])
  }

  for (let i = 0; i < 10; i++) {
    // вызовет ошибку
    const res = useMemo(() => null, []);
  }

  if (!props.show) {
    return null
  }

  // ошибка!
  const dont = useRef(null);

  return (
    <div>
      Hooks
    </div>
  )
}
```

```
function Button(props) {
  const [count, setCount] = useState(0);

  const changeCount = useCallback(() => {
    // здесь переменная count всегда равна 0,
    // т.к. коллбэк не обновляется при ее изменении
    console.log(count);
  }, []);

  const changeCountCorrect = useCallback(() => {
    // здесь переменная count всегда актуальна,
    console.log(count);
  }, [count]);
}
```

```
return (  
  <div className="button" onClick={props.onClick}>Click!</div>  
)  
}
```

Обратите внимание на то, что хуки можно вызывать из других хуков. Это связано с одним важным преимуществом хуков - вы можете создавать свои, “кастомные” хуки и переиспользовать их в различных компонентах. К примеру, с помощью следующего кастомного хука можно получить значение переменной на предыдущем рендере:

```
function usePrevious(value) {  
  const ref = useRef();  
  
  useEffect(() => {  
    ref.current = value;  
  }, [value]);  
  
  return ref.current;  
}
```

Использование такого хука выглядит следующим образом:

```
function Button() {  
  const [count, setCount] = useState(0);  
  
  const prevCount = usePrevious(count);  
  
  useEffect(() => {  
    if (prevCount !== count) {  
      /* ... */  
    }  
  }, [prevCount, count]);  
  
  const changeCount = useCallback(() => {  
    setCount(1);  
  }, []);  
  
  return (  
    <div className="button" onClick={changeCount}>  
      Click!  
    </div>  
  );  
}
```

Данная реализация позволяет использовать значение переменной count до последнего обновления, аналогично значению prevProps или prevState в методе componentDidUpdate классического компонента.

Рендер массивов. Фрагмент

Достаточно частой задачей является отображение списков (к примеру, список сообщений, товаров, и т.п.). Данные, как правило, хранятся в таком случае в массиве, а для рендера используется метод map:

```
export function MessagesList() {
  const [messages, setMessages] = useState([
    "message 1",
    "message 2",
    "message 3",
  ]);

  return messages.map((message) => <div>{message}</div>);
}
```

React не позволяет возвращать из компонента несколько элементов на одном уровне:

```
function Error() {
  return (
    <span>Будет ошибка!</span>
    <div>Will throw error!</div>
  );
}
```

Если из компонента требуется вернуть несколько сиблингов, а добавление еще одного оборачивающего элемента (напр. div) нежелательно, следует использовать фрагмент. Фрагмент - это специальный компонент React, он служит исключительно для группировки нескольких элементов. При рендере в DOM он не будет добавлен как элемент:

```
function FragmentExample() {
  return (
    <>
      <span>This is right!</span>
      <div>Краткая запись фрагмента</div>
    </>
  );
}
// или
```

```
function FragmentExample2() {
  return (
    <React.Fragment>
      <span>This is right!</span>
      <div>Полная запись фрагмента</div>
    </React.Fragment>
  );
}
```

Фрагмент может принимать только один проп - key, при этом при использовании краткой записи его указать невозможно.

Внимание!

При рендере массивов в консоли вы можете увидеть следующую ошибку: 'Each child in an array should have a unique "key" prop'. Причины ее возникновения и пути решения будут разобраны на следующем уроке.

Глоссарий

1. State (стейт) - набор переменных состояния компонента. Эти переменные инкапсулированы внутри компонента, и их обновление влияет на вызов методов жизненного цикла.
2. Хуки React - специальные функции, используемые внутри функциональных компонентов для инкапсуляции некоторой логики.
3. Контролируемые формы - подход к созданию форм, при котором значение, введенное пользователем в поле ввода, хранится в стейте компонента, и передается пропсами элементам DOM.
4. Поднятие стейта - помещение необходимой для нескольких компонентов переменной состояния в их ближайшего общего родителя.
5. Мемоизация - запоминание результатов вызова функции для избегания многократных ее вызовов.
6. Фрагмент - специальный компонент React, служащий для оборачивания нескольких компонентов при возвращении их из компонента. Никак не влияет на отображение элементов в DOM.

Домашнее задание

1. Добавить в компонент App поле стейта messageList. В нем хранить массив объектов - сообщений (объект должен содержать, как минимум, поля text и author). Начальное значение - пустой массив).
2. Рендерить список сообщений через map.

3. Добавить и стилизовать форму - поле для ввода текста и кнопка для отправки сообщения. При отправке сообщения обновление UI должно происходить за счет обновления стейта App.
4. Добавить в App useEffect - на каждое отправленное пользователем сообщение должен отвечать робот (должно автоматически отправляться сообщение с фиксированным текстом) - для этого необходимо проверять автора последнего сообщения.
5. * Сделать задержку ответа робота - сообщение от него должно отправляться через некоторый промежуток времени после отправки сообщения пользователя (напр., 1.5 сек).

Дополнительные материалы

1. [Статья "Понимание методов жизненного цикла"](#)
2. [Статья "Полное руководство по useEffect"](#)
3. [Статья "Понимание хуков в React"](#)
4. [Статья о фрагментах](#)

Используемые источники

1. [Официальная документация React](#)
2. [Схема жизненного цикла компонента React](#)