

## Урок 12.

# Работа с событиями и реализация UI компонентов. Хук useRef

querySelector в React	2
Определение useRef	5
Задание для закрепления	7
Кнопка, которая "убегает" при наведении	10
Задание для закрепления	12
useState	14
Задание для закрепления	17
Управление фокусом и выбором элементов	20
Задание для закрепления	23
Комбинирование useRef и useEffect	26
Задание для закрепления	29
Создание продвинутых UI компонентов	32
useRef: управление состоянием анимации и DOM элементами	35
Задание для закрепления	37
Продвинутые концепции useRef	39

## querySelector в React

Сегодня мы поговорим о том, как работать с событиями в React и как реализовывать пользовательские интерфейсы (UI компоненты). Мы узнаем, как правильно обрабатывать события, такие как клики, движения мыши и другие, и как с помощью этих событий можно управлять поведением наших компонентов.

Также мы познакомимся с хуком `useRef`, который помогает нам работать с DOM-элементами непосредственно, сохраняя ссылки на них. Это важно для выполнения определённых задач, таких как анимации или управление фокусом.

Когда мы работаем с обычным JavaScript, для доступа к элементам на странице часто используется метод `querySelector`. Например, чтобы найти кнопку на странице и повесить на неё обработчик события клика, мы можем написать что-то вроде:



```
const button = document.querySelector('.my-button');
button.addEventListener('click', () => {
  console.log('Button clicked');
});
```

Однако в React использование `querySelector` не рекомендуется по нескольким причинам:

1. Противоречит философии React:

React был создан для того, чтобы разбивать наше приложение на небольшие независимые компоненты. Каждый компонент имеет своё состояние и логику. Когда мы используем `querySelector`, мы нарушаем этот принцип, потому что `querySelector` ищет элемент во всём документе (или в его части), а не в конкретном компоненте.

2. React-приложения динамические:

В React состояние и рендеринг компонентов могут меняться в зависимости от действий пользователя или других событий. Элементы могут добавляться,

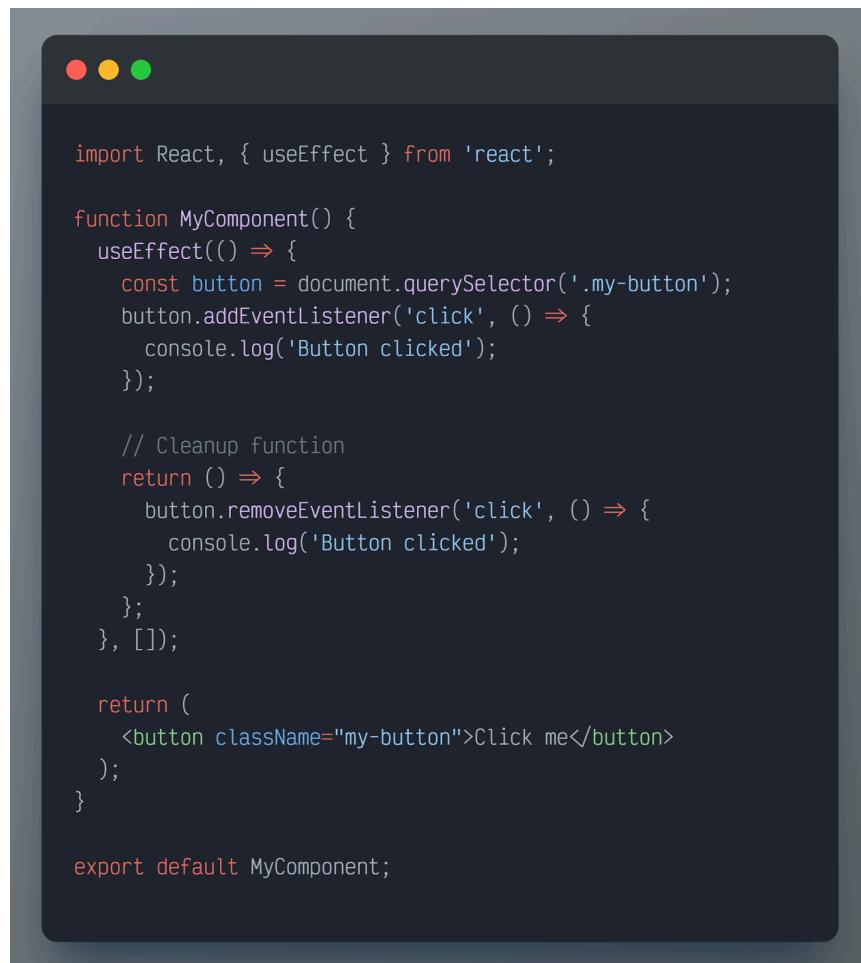
удаляться или изменяться. Это делает использование `querySelector` сложным и ненадежным, потому что селекторы, которые мы написали, могут перестать работать при изменении состояния приложения.

### 3. Трудности с подбором селекторов:

В больших приложениях сложно подобрать уникальные селекторы для каждого элемента. Например, у нас может быть несколько кнопок с одним и тем же классом. В таком случае `querySelector` вернёт первую кнопку, которую найдёт, а не ту, с которой мы хотим работать.

Пример использования `querySelector` в React:

Допустим, у нас есть компонент с кнопкой:



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

function MyComponent() {
  useEffect(() => {
    const button = document.querySelector('.my-button');
    button.addEventListener('click', () => {
      console.log('Button clicked');
    });
  });

  // Cleanup function
  return () => {
    button.removeEventListener('click', () => {
      console.log('Button clicked');
    });
  };
}

return (
  <button className="my-button">Click me</button>
);
}

export default MyComponent;
```

В этом примере мы используем `querySelector` для поиска кнопки и добавления к ней обработчика события. Но такой подход имеет недостатки, описанные выше.

Вместо этого мы можем использовать React-хуки, такие как `useRef` и `useEffect`, чтобы добиться того же результата, но более "реактивным" способом, который соответствует философии React.

## Определение useRef



**Хук useRef - это инструмент в React, который позволяет создавать "ссылки" на DOM-элементы или другие значения в функциональных компонентах.**

Он представляет собой функцию, которая возвращает мутабельный объект ref.

Основные концепции:

1. Создание ссылки: При помощи хука useRef можно создать переменную, которая будет содержать ссылку на DOM-элемент.
2. Доступ к DOM-элементам: После создания ссылки можно обращаться к DOM-элементам напрямую без использования селекторов или обхода компонентов.
3. Сохранение мутируемых значений: В дополнение к DOM-элементам, useRef можно использовать для сохранения мутируемых значений, которые должны сохраняться между рендерами компонента.
4. Не вызывает повторных рендеров: Изменение значения ref.current не вызывает повторный рендер компонента, в отличие от изменения состояния с помощью useState.
5. Применение в побочных эффектах: Часто useRef используется вместе с хуками useEffect для сохранения ссылок на элементы и работы с ними внутри побочных эффектов.

Преимущества использования useRef:

- Прямой доступ к DOM: useRef позволяет получить доступ к DOM-элементам напрямую без использования селекторов.
- Отсутствие перерисовок: Изменение значения useRef не вызывает повторный рендер компонента, что полезно для сохранения состояний без изменения визуального представления.
- Простота использования: Хук useRef прост в использовании и идеально подходит для задач, связанных с работой с DOM.

Принцип работы useRef:

Давайте разберём, как работает хук `useRef` на практике.

## 1. Создание ссылки с useRef и присвоение её элементу:

Когда мы используем `useRef`, мы создаём переменную-ссылку, которая может хранить ссылку на DOM-элемент или любое другое значение. Например:



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

function MyComponent() {
  const myRef = useRef(null);

  // ...
}
```

В этом примере `myRef` является переменной-ссылкой, которая изначально содержит `null`.

## 2. Добавление обработчиков событий через эту ссылку:

Мы можем использовать эту ссылку для доступа к DOM-элементу и добавления обработчиков событий или для других манипуляций с элементом. Например:



```
import React, { useRef, useEffect } from 'react';

function MyComponent() {
  const myRef = useRef(null);

  useEffect(() => {
    if (myRef.current) {
      myRef.current.addEventListener('click', () => {
        console.log('Button clicked');
      });
    }
  }, []);

  return <button ref={myRef}>Click me</button>;
}
```

В этом примере мы добавляем обработчик события клика к кнопке, используя ссылку `myRef`.

## ⭐ Задание для закрепления

Создание компонента ClickCounter с использованием useRef:

### 1. Начало работы

Создайте новый React-компонент с названием `ClickCounter`.

### 2. Импорт необходимых модулей

Импортируйте `React` и хуки `useState`, `useRef`, и `useEffect` из библиотеки `react`.

### 3. Создание переменных состояния и ссылки

Используя `useState`, создайте переменную состояния с именем `clickCount` и инициализируйте её значением `0`.

Используя `useRef`, создайте переменную-ссылку с именем `buttonRef` и присвойте ей значение `null`.

### 4. Использование useEffect для стилизации кнопки

Используйте хук `useEffect`, чтобы применить стили к кнопке при загрузке компонента.

Проверьте, что ссылка на кнопку (`buttonRef.current`) не равна `null`, и измените стили кнопки (например, цвет фона и отступы).

### 5. Добавление обработчика события

Создайте функцию `handleClick`, которая будет вызываться при клике на кнопку.

Внутри функции увеличьте счётчик кликов на 1, обновив состояние `clickCount`.

### 6. Использование переменной-ссылки в JSX

Верните JSX-разметку кнопки в компоненте `ClickCounter`.

Укажите атрибут `ref` кнопки равным `buttonRef`.

Добавьте атрибут `onClick` для вызова функции `handleClick`.

Выведите значение счётчика внутри кнопки.

## 7. Тестирование

Импортируйте компонент `ClickCounter` в основной файл вашего приложения и добавьте его на страницу.

Проверьте, что при каждом клике на кнопку счетчик увеличивается, а значение выводится под кнопкой. Также убедитесь, что стили кнопки применяются при загрузке компонента.

JavaScript

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

function ClickCounter() {
  const [clickCount, setClickCount] = useState(0);
  const buttonRef = useRef(null);

  useEffect(() => {
    // Пример очень простой стилизации кнопки при загрузке компонента
    if (buttonRef.current) {
      buttonRef.current.style.backgroundColor = 'lightblue';
      buttonRef.current.style.padding = '10px';
    }
  }, []); // Этот эффект будет запущен только один раз при монтировании
          // компонента

  const handleClick = () => {
    setClickCount(prevCount => prevCount + 1);
  };

  return (
    <div>
      <button ref={buttonRef} onClick={handleClick}>
```

```
    Click Me
  </button>
  <p>Click count: {clickCount}</p>
</div>
);
}

export default ClickCounter;
```

## Кнопка, которая "убегает" при наведении

Теперь давайте посмотрим на пример, где мы используем `useRef` для создания кнопки, которая "убегает" при наведении на неё.

### 1. Создание кнопки:

Мы создадим простую кнопку в React:

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

function FleeButton() {
  const buttonRef = useRef(null);

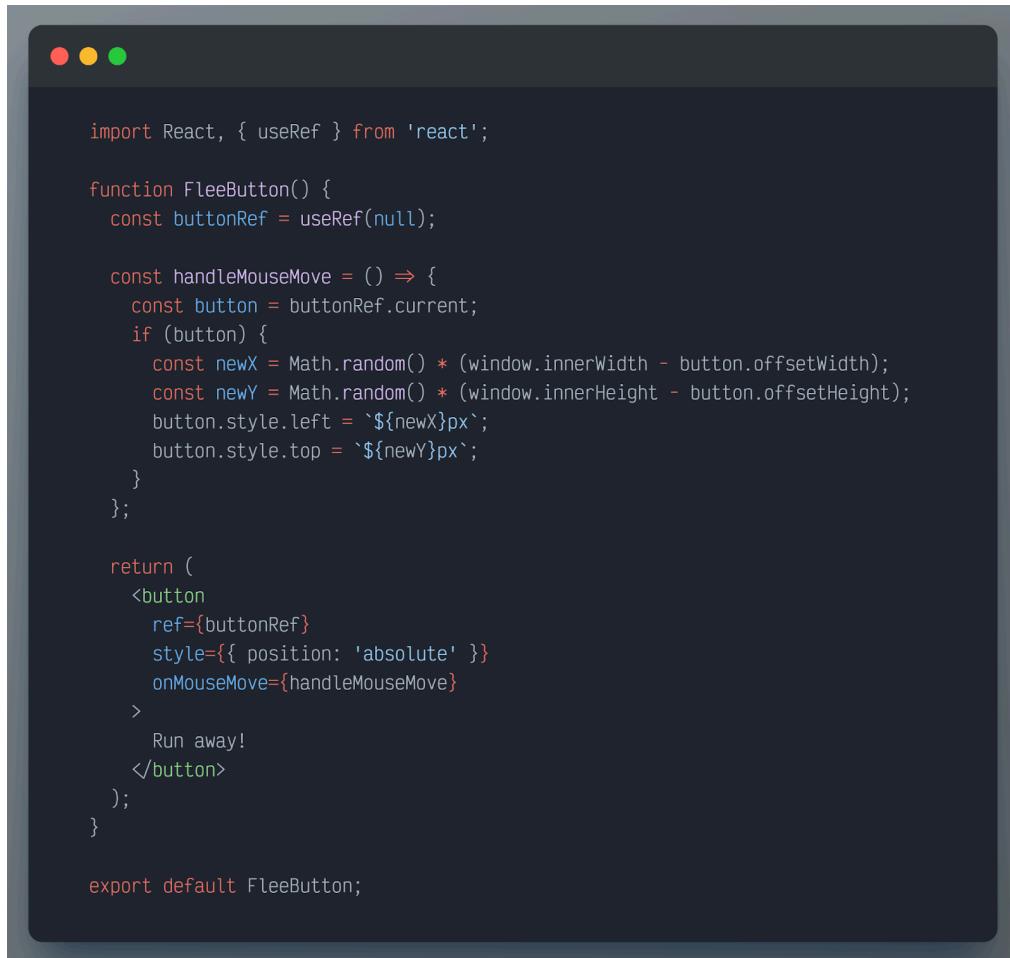
  return (
    <button ref={buttonRef} style={{ position: 'relative' }}>
      Run away!
    </button>
  );
}

export default FleeButton;
```

Здесь мы используем `useRef` для создания ссылки `buttonRef`, которая будет хранить ссылку на кнопку.

### 2. Реализация анимации убегания кнопки:

Теперь давайте добавим эффект, который будет перемещать кнопку в случайное место, когда мы наводим на неё курсор:



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

function FleeButton() {
  const buttonRef = useRef(null);

  const handleMouseMove = () => {
    const button = buttonRef.current;
    if (button) {
      const newX = Math.random() * (window.innerWidth - button.offsetWidth);
      const newY = Math.random() * (window.innerHeight - button.offsetHeight);
      button.style.left = `${newX}px`;
      button.style.top = `${newY}px`;
    }
  };

  return (
    <button
      ref={buttonRef}
      style={{ position: 'absolute' }}
      onMouseMove={handleMouseMove}
    >
      Run away!
    </button>
  );
}

export default FleeButton;
```

Здесь мы добавляем обработчик события `onMouseMove`, который вызывается при движении мыши над кнопкой. Когда мышь движется, кнопка перемещается в случайное место на экране.

Таким образом, мы использовали `useRef` для создания ссылки на кнопку, а затем добавили обработчик события для реализации анимации перемещения кнопки при наведении мыши.

## ☆ Задание для закрепления

Создание простой анимированной кнопки

1. Начало работы:

Создайте новый React-компонент с названием `AnimatedButton`.

2. Импорт необходимых модулей:

Импортируйте `React` и хук `useRef` из библиотеки `react`.

3. Создание переменной-ссылки:

Используя `useRef`, создайте переменную-ссылку с именем `buttonRef` и присвойте ей значение `null`.

4. Добавление обработчика события:

Создайте функцию `handleMouseMove`, которая будет вызываться при движении мыши над кнопкой.

Внутри функции получите доступ к элементу кнопки через `buttonRef.current`.

Вычислите новые координаты для кнопки в случайном месте на экране.

Измените свойства `left` и `top` кнопки, чтобы переместить её в новое положение.

5. Использование переменной-ссылки в JSX:

Верните JSX-разметку кнопки в компоненте `AnimatedButton`.

Укажите атрибут `ref` кнопки равным `buttonRef`.

Добавьте атрибут `style` для задания начального положения кнопки.

6. Тестирование:

Импортируйте компонент `AnimatedButton` в основной файл вашего приложения и добавьте его на страницу.

Проверьте, что кнопка перемещается в случайное место при наведении мыши.

JavaScript

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

function FleeButton() {
  const buttonRef = useRef(null);

  const handleMouseMove = () => {
    const button = buttonRef.current;
    if (button) {
      const newX = Math.random() * (window.innerWidth - button.offsetWidth);
      const newY = Math.random() * (window.innerHeight - button.offsetHeight);
      button.style.left = `${newX}px`;
      button.style.top = `${newY}px`;
    }
  };

  return (
    <button
      ref={buttonRef}
      style={{ position: 'absolute' }}
      onMouseMove={handleMouseMove}>
      Run away!
    </button>
  );
}

export default FleeButton;
```

## 7. \*Дополнительные шаги:

Добавьте анимацию или изменения стилей при перемещении кнопки.

Реализуйте другие интересные эффекты при наведении мыши.

## useState

Давайте сравним, как работают хуки `useState` и `useRef` для решения задач анимации.

1. useState меняет значения и вызывает перерисовку:

Хук `useState` используется для создания состояния в компоненте. При изменении значения состояния компонент перерисовывается.

Если мы используем `useState` для анимации, каждое изменение состояния приведет к перерисовке компонента, даже если изменения касаются только стилей.

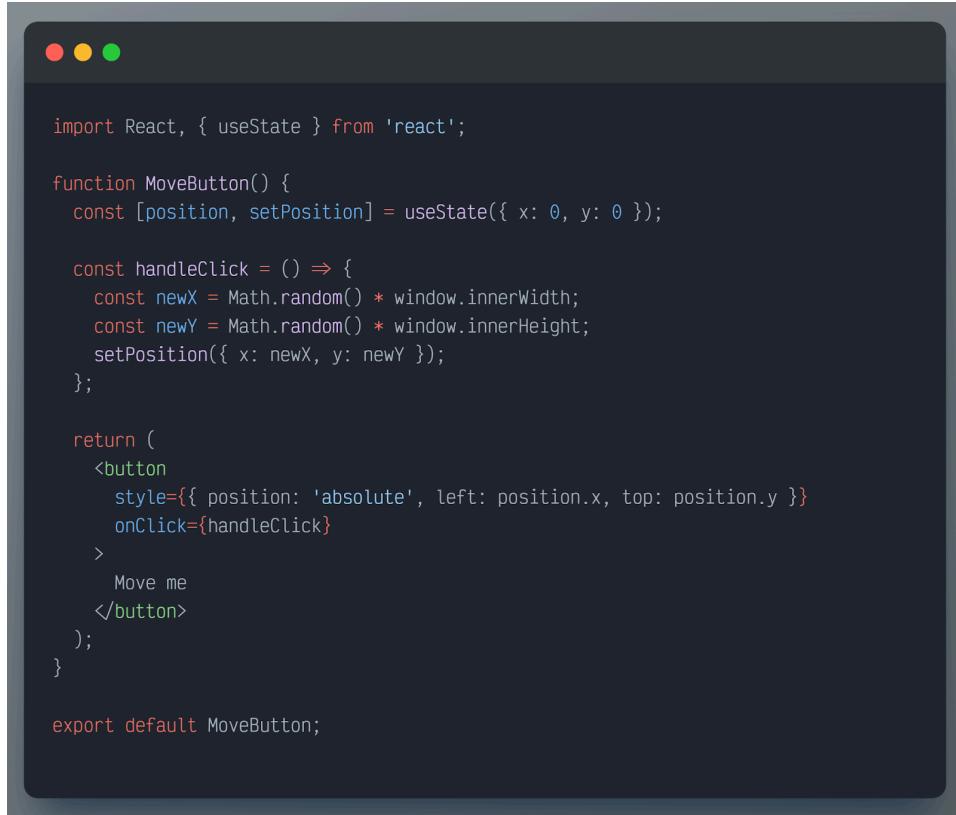
2. useRef сохраняет ссылку на элемент без перерисовки:

Хук `useRef` используется для создания ссылки на DOM-элемент или другое значение, которое не вызывает перерисовку компонента при изменении.

Это полезно для случаев, когда мы хотим сохранить ссылку на элемент и манипулировать им без перерисовки.

Пример с использованием useState для изменения позиции кнопки без анимации:

Давайте рассмотрим пример, где мы используем `useState` для изменения позиции кнопки без анимации:



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

function MoveButton() {
  const [position, setPosition] = useState({ x: 0, y: 0 });

  const handleClick = () => {
    const newX = Math.random() * window.innerWidth;
    const newY = Math.random() * window.innerHeight;
    setPosition({ x: newX, y: newY });
  };

  return (
    <button
      style={{ position: 'absolute', left: position.x, top: position.y }}
      onClick={handleClick}
    >
      Move me
    </button>
  );
}

export default MoveButton;
```

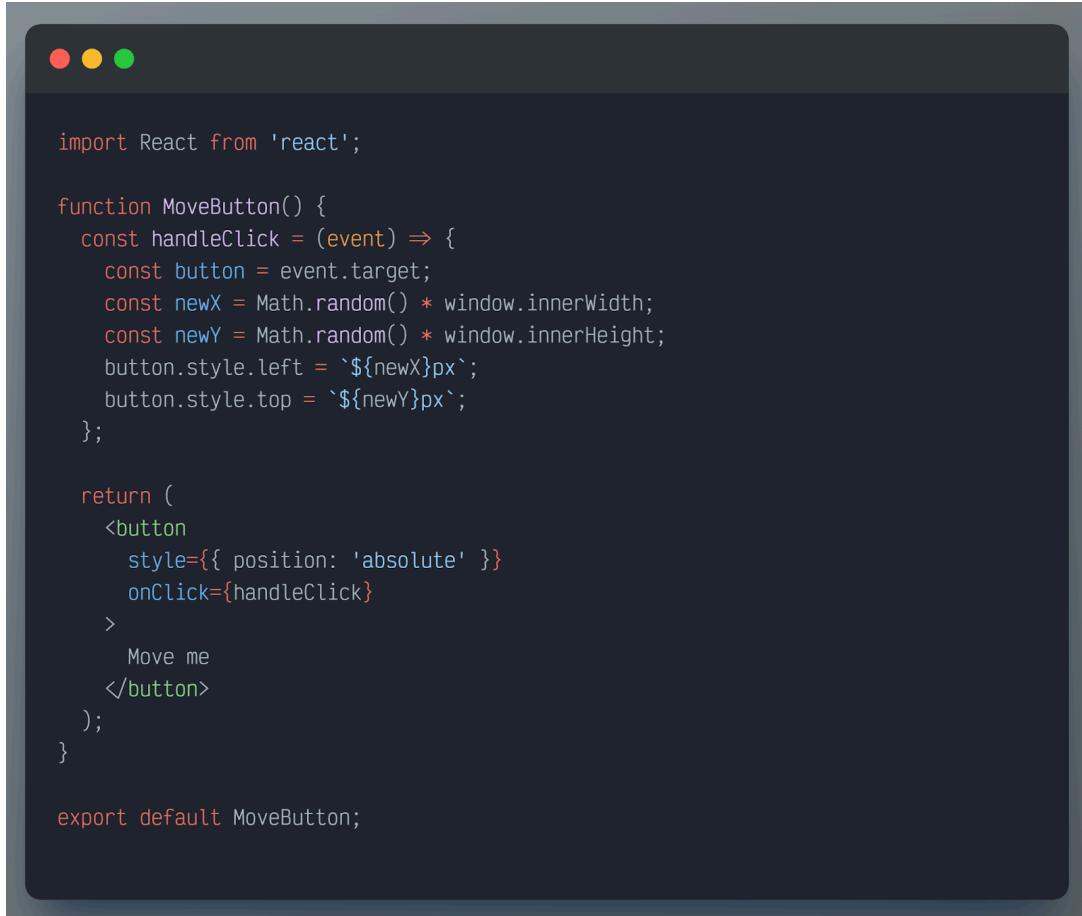
В этом примере мы используем `useState` для хранения позиции кнопки. Каждый раз при клике на кнопку мы генерируем новые случайные координаты для `x` и `y` и обновляем состояние кнопки. Это приводит к перерисовке кнопки с новыми координатами, но без какой-либо анимации.

Таким образом, использование `useState` для анимации может привести к лишним перерисовкам компонента и неэффективному использованию ресурсов. Вместо этого для анимации лучше использовать `useRef`, чтобы сохранить ссылку на элемент и изменять его стили напрямую, минуя перерисовку компонента.

Понимание, когда не нужно использовать `useRef`, так важно, как и знание, когда нужно. `useRef` чаще всего используется, когда нужно анимировано изменить другой элемент по событию:

`useRef` обычно применяется, когда нам нужно сохранить ссылку на DOM-элемент и манипулировать им, например, для анимации, изменения стилей и т. д.

Теперь давайте перепишем наш предыдущий пример, используя `event.target` вместо `useRef`.



```
import React from 'react';

function MoveButton() {
  const handleClick = (event) => {
    const button = event.target;
    const newX = Math.random() * window.innerWidth;
    const newY = Math.random() * window.innerHeight;
    button.style.left = `${newX}px`;
    button.style.top = `${newY}px`;
  };

  return (
    <button
      style={{ position: 'absolute' }}
      onClick={handleClick}
    >
      Move me
    </button>
  );
}

export default MoveButton;
```

Объяснение изменений:

1. Использование `event.target`:

Вместо создания ссылки на кнопку с помощью `'useRef'`, мы теперь получаем ссылку на кнопку через `'event.target'` в обработчике события `'handleClick'`.

2. Применение стилей непосредственно к кнопке:

Мы изменяем стили кнопки напрямую, используя `'button.style.left'` и `'button.style.top'`, без обращения к `'useRef'`.

Таким образом, мы переписали наш пример, избегая использования `'useRef'` и вместо этого использовали `'event.target'`, чтобы получить доступ к элементу, с которым произошло событие. Это показывает, что `'useRef'` не всегда необходим, и в некоторых случаях его можно избежать, используя другие способы доступа к элементам.

## ⭐ Задание для закрепления

Создание компонента `ScrollBox` с использованием `useRef` для управления прокруткой:

1. Импортируйте React и useRef:

В файле `ScrollBox.js` начните с импорта React и хука `useRef`.

2. Создайте ссылку на элемент:

Создайте переменную с помощью `useRef`, чтобы сохранить ссылку на прокручиваемый элемент.

3. Создайте функцию для прокрутки:

Определите функцию, которая будет прокручивать элемент до нижней части, используя созданную ссылку.

4. Создайте разметку компонента:

Добавьте контейнер с прокручиваемым контентом.

Добавьте кнопку, при нажатии на которую будет выполняться прокрутка.

5. Привяжите ссылку к элементу:

Используйте атрибут `ref` для привязки созданной ссылки к прокручиваемому контейнеру.

6. Добавьте обработчик события:

Привяжите функцию прокрутки к событию `onClick` кнопки.

7. Импортируйте и используйте компонент `ScrollBox` в основном приложении:

Откройте файл `App.js` и импортируйте компонент `ScrollBox`.

Вставьте компонент `ScrollBox` в разметку вашего основного компонента.

8. Запустите проект:

В командной строке выполните команду `npm start`, чтобы запустить проект.

Перейдите в браузере по адресу `http://localhost:3000` и проверьте, что ваш компонент `ScrollBox` отображается корректно.

### 9. Проверьте функциональность:

Убедитесь, что длинный контент отображается в прокручиваемом контейнере.

Нажмите на кнопку и проверьте, что контейнер прокручивается до нижней части контента.

JavaScript

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

function ScrollBox() {
    // Создаем ссылку на прокручиваемый элемент
    const boxRef = useRef(null);

    // Функция для прокрутки элемента до нижней части
    const scrollToBottom = () => {
        if (boxRef.current) {
            boxRef.current.scrollTop = boxRef.current.scrollHeight;
        }
    };

    return (
        <div>
            <div
                ref={boxRef}
                style={{
                    width: '300px',
                    height: '200px',
                    overflowY: 'scroll',
                    border: '1px solid black',
                    padding: '10px',
                }}
            >
                Длинный контент, который будет прокручиваться
            </div>
        </div>
    );
}
```

```
        }
      >
      <p>text</p>
      <p>text</p>
      <p>text</p>
    </div>
    <button onClick={scrollToBottom} style={{ marginTop: '10px' }}>
      Scroll to Bottom
    </button>
  </div>
);
}

export default ScrollBox;
```

# Управление фокусом и выбором элементов

Пример: Создание формы, где useRef используется для автоматической фокусировки на инпуте при открытии

Когда пользователь открывает форму, хорошей практикой является автоматическая установка фокуса на первое поле ввода (инпут). Это улучшает удобство использования и позволяет пользователю сразу начать ввод данных. В React для управления фокусом на элементах используется хук `useRef`.

Как это сделать:

1. Создаем компонент формы.
2. Используем `useRef` для создания рефа на инпут.
3. Используем `useEffect` для установки фокуса на инпут при монтировании компонента.



```
import React, { useEffect, useRef } from 'react';

const AutoFocusForm = () => {
  // Создаем реф для инпута
  const inputRef = useRef(null);

  // Используем useEffect для установки фокуса при монтировании компонента
  useEffect(() => {
    if (inputRef.current) {
      inputRef.current.focus();
    }
  }, []);

  return (
    <form>
      <label>
        Имя:
        <input type="text" ref={inputRef} />
      </label>
      <button type="submit">Отправить</button>
    </form>
  );
};

export default AutoFocusForm;
```

В этом примере:

1. Создаем реф: `const inputRef = useRef(null);` - создаем реф для хранения ссылки на DOM элемент инпута.

2. Используем `useEffect`: Внутри `useEffect` проверяем, что `inputRef.current` не равен `null`, и вызываем метод `focus()`, чтобы установить фокус на инпут. `useEffect` с пустым массивом зависимостей выполняется один раз при монтировании компонента.
3. Привязываем реф к инпуту: Добавляем `ref={inputRef}` к инпуту, чтобы связать его с созданным рефом.

Лучшие практики работы с фокусом в формах:

1. Установка фокуса на первый интерактивный элемент:

Всегда устанавливайте фокус на первый интерактивный элемент формы при ее открытии. Это помогает пользователю сразу начать взаимодействие с формой.

2. Управление фокусом при валидации формы:

При обнаружении ошибки валидации, перемещайте фокус на первое поле с ошибкой. Это позволяет пользователю быстро увидеть и исправить ошибки.

3. Своевременная установка фокуса:

Устанавливайте фокус только после полного рендера компонента. Для этого используйте `useEffect` без дополнительных зависимостей или с зависимостями, которые меняются, когда компонент готов.

4. Использование рефов для сохранения фокуса:

Если форма скрывается и снова отображается, можно использовать рефы для сохранения текущего фокуса и восстановления его после повторного отображения формы.

5. Смена фокуса по событию:

Вы можете использовать `useRef` для смены фокуса при определенных событиях. Например, после успешного сабмита формы переместить фокус на кнопку закрытия.

Пример использования фокуса при валидации формы:



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

const ValidatedForm = () => {
  const inputRef = useRef(null);
  const [error, setError] = useState('');

  const handleSubmit = (e) => {
    e.preventDefault();
    if (!inputRef.current.value) {
      setError('Поле не может быть пустым');
      inputRef.current.focus();
    } else {
      setError('');
      alert('Форма отправлена');
    }
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Имя:
        <input type="text" ref={inputRef} />
      </label>
      {error && <p style={{ color: 'red' }}>{error}</p>}
      <button type="submit">Отправить</button>
    </form>
  );
};

export default ValidatedForm;
```

В этом примере, если поле пустое при сабмите, мы устанавливаем фокус обратно на это поле и показываем сообщение об ошибке. Это помогает пользователю быстро исправить ошибку и улучшает взаимодействие с формой.

## ⭐ Задание для закрепления

Создание формы с автоматической фокусировкой на инпуте при открытии.

Вам нужно создать React-компонент, представляющий форму, в которой при открытии автоматически устанавливается фокус на первом поле ввода (инпут). Для этого вам нужно использовать хук `useRef`.

### 1. Создание нового React проекта

Создайте новый React проект с помощью команды: `npx create-react-app auto-focus-form`.

Перейдите в созданную директорию проекта: `cd auto-focus-form`.

### 2. Создание компонента формы

Внутри директории `src` создайте новый файл `AutoFocusForm.js`.

Создайте структуру функционального компонента внутри этого файла.

### 3. Импорт необходимых хуков

Импортируйте `useRef` и `useEffect` из библиотеки React.

### 4. Создание рефа для инпута

Внутри компонента создайте реф с помощью хука `useRef`.

### 5. Установка фокуса при монтировании компонента

Используйте хук `useEffect` для установки фокуса на инпут при монтировании компонента.

Внутри `useEffect` вызовите метод `focus()` для инпута, используя созданный реф.

### 6. Привязка рефа к инпуту

Добавьте элемент формы (`<form>`), содержащий инпут (`<input>`) и кнопку отправки (`<button>`).

Привяжите созданный реф к инпуту с помощью атрибута `ref`.

JavaScript

```
import React, { useEffect, useRef } from 'react';

const AutoFocusForm = () => {
  const inputRef = useRef(null);

  useEffect(() => {
    if (inputRef.current) {
      inputRef.current.focus();
    }
  }, []);

  return (
    <form>
      <label>
        Имя:
        <input type="text" ref={inputRef} />
      </label>
      <button type="submit">Отправить</button>
    </form>
  );
};

export default AutoFocusForm;
```

## 7. Добавление стилей и завершение компонента

Добавьте стили для формы, если необходимо, чтобы улучшить внешний вид. Убедитесь, что компонент возвращает JSX с формой и инпутом.

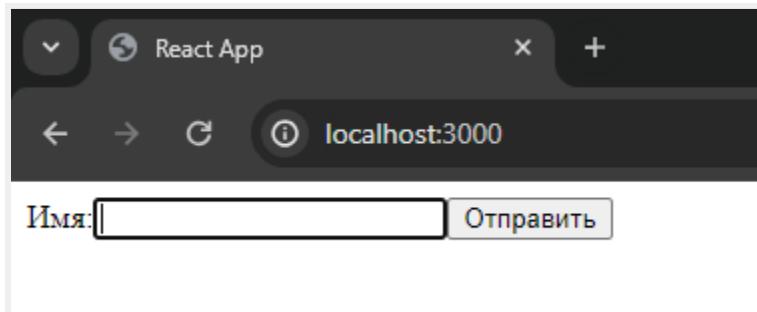
## 8. Использование компонента в приложении

Откройте файл `App.js`.  
Импортируйте компонент формы (`AutoFocusForm`).  
Вставьте компонент формы в JSX разметку `App`.

## 9. Запуск приложения

В терминале запустите приложение с помощью команды: `npm start`.

Убедитесь, что при открытии приложения фокус автоматически устанавливается на инпут формы.



## Комбинирование useRef и useEffect

В React хуки `useRef` и `useEffect` часто используются вместе для выполнения задач, связанных с доступом и манипуляцией DOM элементами или управления состоянием, которое не требует повторного рендеринга компонента.

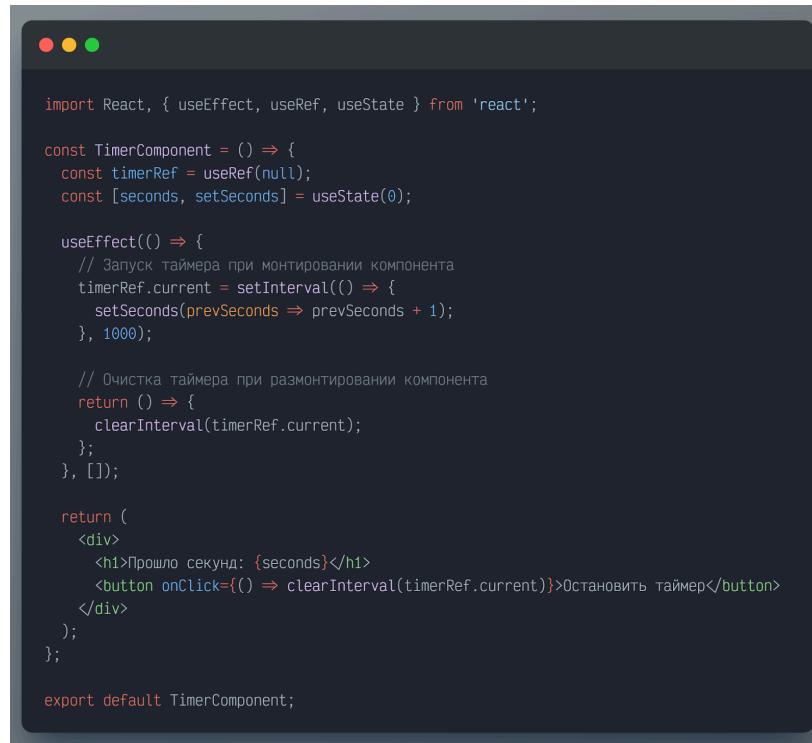
**useRef:** Позволяет сохранять ссылку на DOM элемент или сохранять изменяющиеся значения, которые не требуют повторного рендеринга компонента.

**useEffect:** Выполняет побочные эффекты, такие как вызовы API, подписки, изменение DOM и очистку, после рендеринга компонента.

Комбинируя `useRef` и `useEffect`, вы можете управлять элементами или состояниями, которые не должны влиять на рендеринг компонента, например, установка фокуса, управление таймерами или подписками.

Рассмотрим пример создания компонента с таймером, который запускается при монтировании и останавливается при размонтировании. Здесь `useRef` будет хранить идентификатор таймера, а `useEffect` управлять его запуском и остановкой.

1. Создание функционального компонента.
2. Используем `useRef` для хранения идентификатора таймера.
3. Используем `useEffect` для запуска и остановки таймера.



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

const TimerComponent = () => {
  const timerRef = useRef(null);
  const [seconds, setSeconds] = useState(0);

  useEffect(() => {
    // Запуск таймера при монтировании компонента
    timerRef.current = setInterval(() => {
      setSeconds(prevSeconds => prevSeconds + 1);
    }, 1000);

    // Очистка таймера при размонтировании компонента
    return () => {
      clearInterval(timerRef.current);
    };
  }, []);

  return (
    <div>
      <h1>Прошло секунд: {seconds}</h1>
      <button onClick={() => clearInterval(timerRef.current)}>Остановить таймер</button>
    </div>
  );
};

export default TimerComponent;
```

В этом примере:

1. Создание рефа: `const timerRef = useRef(null);` - создаем реф для хранения идентификатора таймера.
2. Запуск таймера в `useEffect`: В `useEffect` устанавливаем интервал, который увеличивает счетчик секунд каждую секунду и сохраняет идентификатор таймера в `timerRef.current`.
3. Очистка таймера: Возвращаем функция очистки из `useEffect`, которая очищает таймер при размонтировании компонента, используя `clearInterval(timerRef.current)`.
4. Управление таймером: Добавляем кнопку, которая позволяет вручную остановить таймер, вызывая `clearInterval(timerRef.current)`.

Оптимизация работы компонентов с использованием useRef и useEffect:

1. Избегание ненужныхrerендров:

Используйте `useRef` для хранения данных, которые не должны вызывать повторный рендеринг компонента при изменении. Например, идентификаторы таймеров, ссылки на DOM элементы или значения, которые должны оставаться неизменными между рендерами.

2. Чистка побочных эффектов:

Всегда возвращайте функцию очистки из `useEffect`, чтобы избежать утечек памяти и других проблем. Это особенно важно для таймеров, подписок на события и вызовов API.

3. Правильное управление зависимостями:

Указывайте зависимости в массиве зависимостей `useEffect`, чтобы контролировать, когда эффект должен выполняться. Если массив зависимостей пустой, эффект выполняется только один раз при монтировании и очищается при размонтировании.

4. Сохранение состояний между рендерами:

Используйте `useRef` для хранения значений, которые должны сохраняться между рендерами, но не должны вызывать повторное рендеринг. Например, состояние ввода пользователя, которое должно сохраняться при каждом рендрере, но не должно вызывать обновление компонента.

## 5. Использование useEffect для побочных эффектов:

Используйте `useEffect` для выполнения побочных эффектов, таких как вызовы API, подписки на события и манипуляции с DOM, которые должны происходить после рендера компонента.

Эти практики помогут вам создавать более оптимизированные и производительные компоненты, которые эффективно управляют состояниями и побочными эффектами.

## ☆ Задание для закрепления

Создание приложения с таймером:

1. Создание компонента таймера

Внутри директории `src` создайте новый файл `TimerComponent.js`.

Создайте структуру функционального компонента внутри этого файла.

2. Импорт необходимых хуков

Импортируйте `useRef`, `useEffect`, и `useState` из библиотеки React.

3. Создание состояния для хранения времени

Внутри компонента создайте состояние для хранения количества секунд, используя хук `useState`.

4. Создание рефа для идентификатора таймера

Внутри компонента создайте реф с помощью хука `useRef` для хранения идентификатора таймера.

5. Запуск таймера при монтировании компонента

Используйте хук `useEffect` для установки интервала таймера при монтировании компонента.

Внутри `useEffect` создайте функцию, которая увеличивает счетчик времени каждую секунду и сохраняет идентификатор таймера в рефе.

Возвращайте функцию очистки из `useEffect`, которая будет очищать таймер при размонтировании компонента.

6. Создание кнопки для остановки таймера

Внутри компонента добавьте кнопку, которая при клике будет вызывать функцию для остановки таймера с использованием метода `clearInterval` и идентификатора таймера из рефа.

7. Добавление JSX разметки

Добавьте JSX разметку, которая будет отображать текущее количество секунд и кнопку для остановки таймера.

JavaScript

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

const TimerComponent = () => {
  const timerRef = useRef(null);
  const [seconds, setSeconds] = useState(0);

  useEffect(() => {
    timerRef.current = setInterval(() => {
      setSeconds(prevSeconds => prevSeconds + 1);
    }, 1000);

    return () => {
      clearInterval(timerRef.current);
    };
  }, []);

  return (
    <div>
      <h1>Прошло секунд: {seconds}</h1>
      <button onClick={() => clearInterval(timerRef.current)}>Остановить таймер</button>
    </div>
  );
};

export default TimerComponent;
```

## 8. Использование компонента в приложении

Откройте файл `App.js`.

Импортируйте компонент `TimerComponent`.

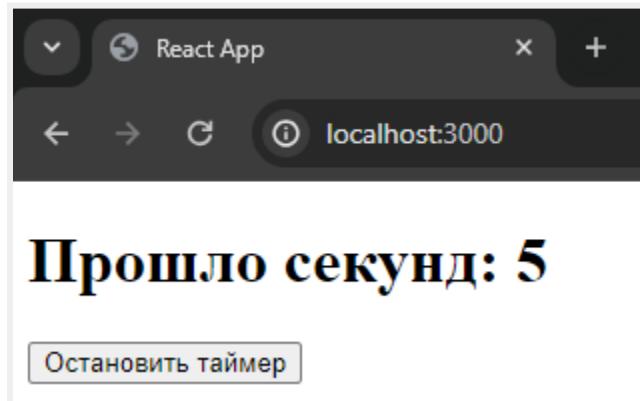
Вставьте компонент `TimerComponent` в JSX разметку `App`.

### 9. Запуск приложения

В терминале запустите приложение с помощью команды: `npm start`.

Убедитесь, что таймер запускается при открытии приложения и отображает количество секунд.

Проверьте, что кнопка останавливает таймер при нажатии.



# Создание продвинутых UI компонентов



**Модальное окно — это всплывающее окно, которое отображается поверх основного контента и требует взаимодействия с пользователем перед продолжением работы с приложением.**

Мы создадим модальное окно с анимацией открытия и закрытия, используя `useRef` для управления состоянием анимации и доступа к DOM элементам.

Шаги для создания компонента модального окна:

1. Создание структуры компонента модального окна

Внутри директории `src` создаем новый файл `Modal.js`.

Создаем функциональный компонент внутри этого файла, который будет представлять модальное окно.

2. Добавление состояния для управления видимостью модального окна

Используем хук `useState` для создания состояния, которое будет определять, отображается модальное окно или нет.

3. Использование `useRef` для доступа к DOM элементу модального окна

Создаем реф с помощью `useRef`, чтобы получить доступ к DOM элементу модального окна.

4. Использование `useEffect` для управления анимацией открытия и закрытия

Используем хук `useEffect` для добавления и удаления CSS классов, которые запускают анимацию при монтировании и размонтировании компонента.

5. Добавление стилей и анимации с помощью CSS

Создадим CSS файл для стилей и анимации модального окна.

Добавим классы для анимации открытия и закрытия модального окна.

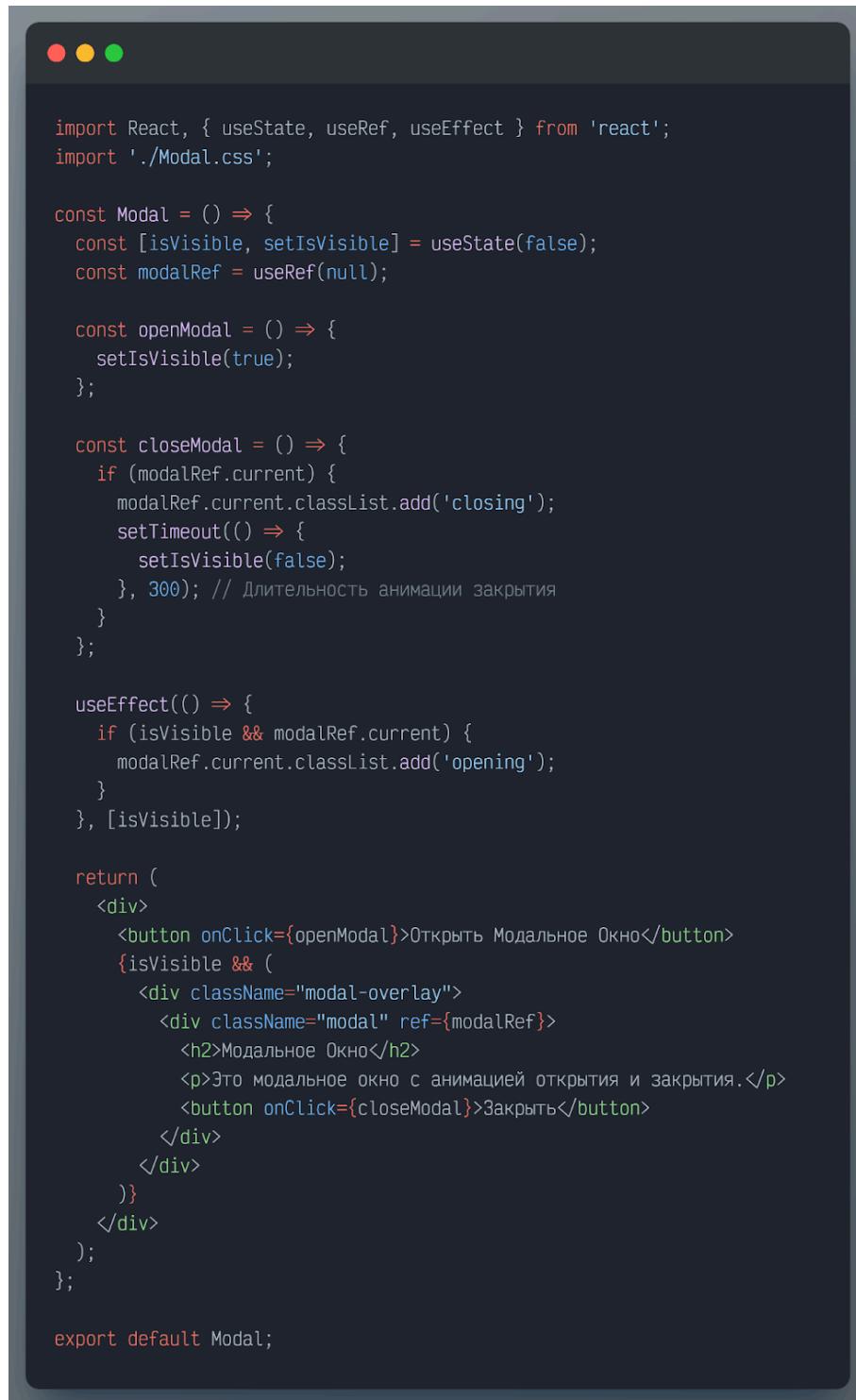
6. Добавление кнопок для открытия и закрытия модального окна

Добавим кнопку для открытия модального окна.

Добавим кнопку внутри модального окна для его закрытия.

Пример структуры и кода:

`src/Modal.js`



The screenshot shows a code editor window with a dark theme. At the top, there are three circular icons: red, yellow, and green. The code editor displays the following content:

```
import React, { useState, useRef, useEffect } from 'react';
import './Modal.css';

const Modal = () => {
  const [isVisible, setIsVisible] = useState(false);
  const modalRef = useRef(null);

  const openModal = () => {
    setIsVisible(true);
  };

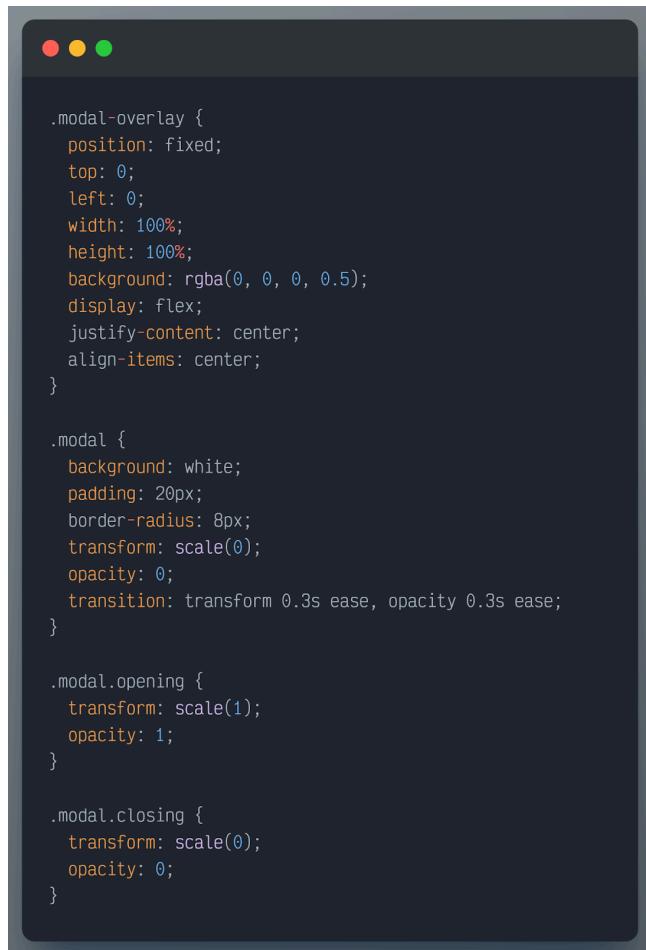
  const closeModal = () => {
    if (modalRef.current) {
      modalRef.current.classList.add('closing');
      setTimeout(() => {
        setIsVisible(false);
      }, 300); // Длительность анимации закрытия
    }
  };

  useEffect(() => {
    if (isVisible && modalRef.current) {
      modalRef.current.classList.add('opening');
    }
  }, [isVisible]);

  return (
    <div>
      <button onClick={openModal}>Открыть Модальное Окно</button>
      {isVisible && (
        <div className="modal-overlay">
          <div className="modal" ref={modalRef}>
            <h2>Модальное Окно</h2>
            <p>Это модальное окно с анимацией открытия и закрытия.</p>
            <button onClick={closeModal}>Закрыть</button>
          </div>
        </div>
      )}
    </div>
  );
};

export default Modal;
```

`src/Modal.css`



A screenshot of a code editor displaying CSS code for a modal overlay. The code defines four classes: .modal-overlay, .modal, .modal.opening, and .modal.closing. The .modal-overlay class sets a fixed position for the overlay, covering the entire screen with a semi-transparent background. The .modal class styles the modal itself, giving it a white background, padding, rounded corners, and a scale(0) transform. It also includes a transition for both transform and opacity. The .modal.opening class applies a scale(1) transform and an opacity of 1. The .modal.closing class applies a scale(0) transform and an opacity of 0.

```
.modal-overlay {
  position: fixed;
  top: 0;
  left: 0;
  width: 100%;
  height: 100%;
  background: rgba(0, 0, 0, 0.5);
  display: flex;
  justify-content: center;
  align-items: center;
}

.modal {
  background: white;
  padding: 20px;
  border-radius: 8px;
  transform: scale(0);
  opacity: 0;
  transition: transform 0.3s ease, opacity 0.3s ease;
}

.modal.opening {
  transform: scale(1);
  opacity: 1;
}

.modal.closing {
  transform: scale(0);
  opacity: 0;
}
```

# useRef: управление состоянием анимации и DOM элементами

## 1. Создание рефа:

Мы создаем реф `modalRef`, чтобы получить доступ к DOM элементу модального окна. Это необходимо для управления классами, которые запускают анимацию.

## 2. Открытие модального окна:

При открытии модального окна мы устанавливаем состояние `isVisible` в `true`, что отображает модальное окно. Затем в `useEffect` добавляем класс `opening`, который запускает анимацию открытия.

## 3. Закрытие модального окна:

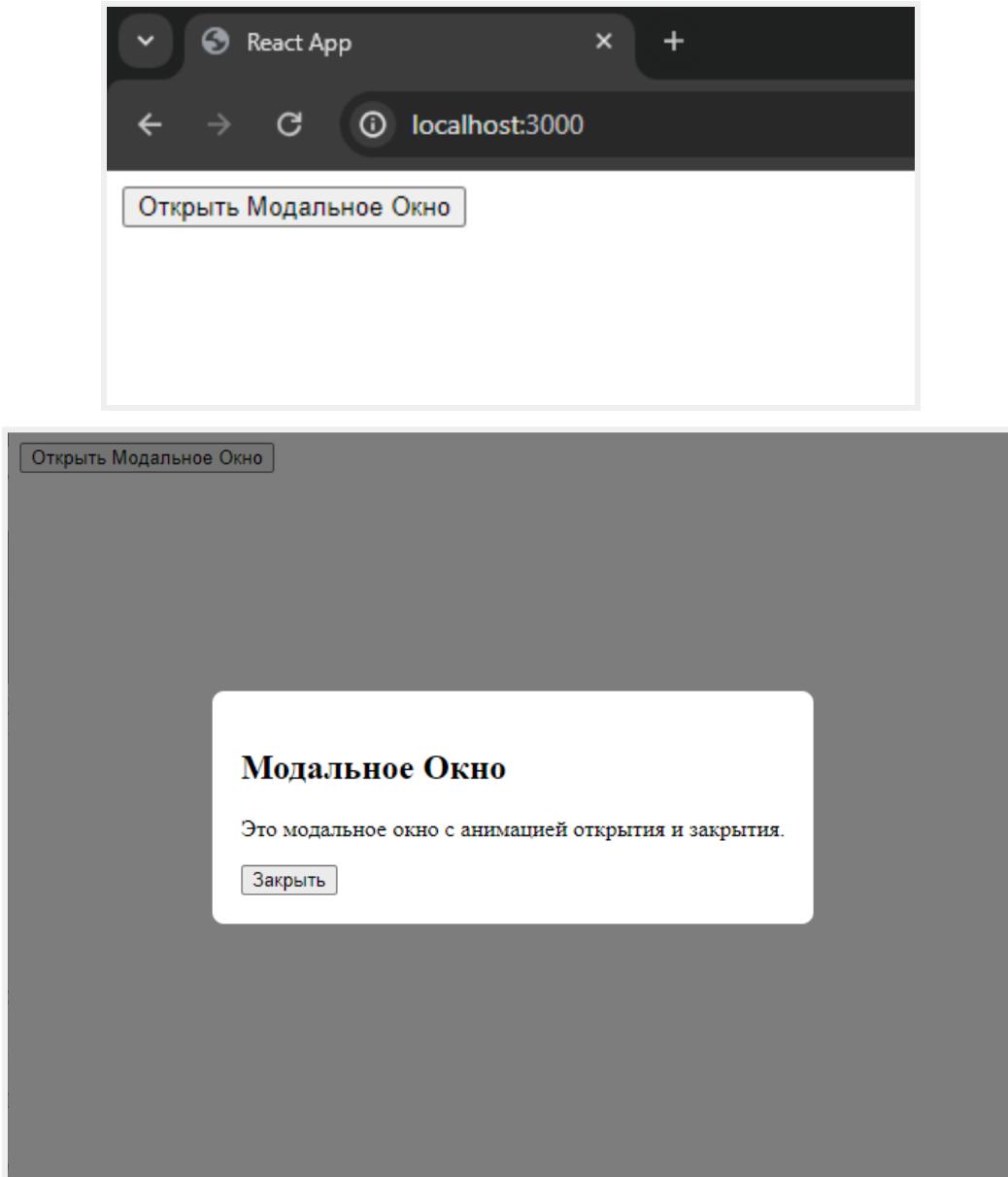
При закрытии модального окна мы добавляем класс `closing`, который запускает анимацию закрытия. Используем `setTimeout`, чтобы дождаться завершения анимации перед скрытием модального окна.

## 4. Использование `useRef`:

`useRef` позволяет нам напрямую манипулировать DOM элементом, не вызывая повторного рендеринга компонента. Это особенно полезно для управления анимациями, где изменения стиля и классов должны происходить напрямую на DOM элементе.

## 5. Управление анимацией через CSS:

Мы определяем анимации открытия и закрытия в CSS с помощью классов `opening` и `closing`, которые изменяют `transform` и `opacity` модального окна.



Оптимизация работы компонентов с использованием useRef и useEffect:

Управление анимациями: использование CSS для анимаций позволяет оптимизировать их выполнение с помощью аппаратного ускорения, что делает их плавными и эффективными.

Следуя этим практикам, вы сможете создавать продвинутые UI компоненты, которые работают эффективно и предоставляют пользователю приятный опыт взаимодействия.

## ☆ Задание для закрепления

Создание модального окна с анимацией открытия и закрытия:

1. Создайте компонент Modal

В папке `src` создайте новый файл `Modal.js`.

Импортируйте необходимые библиотеки: `React`, `useState`, `useRef`, `useEffect`.

2. Определите состояние и реф для модального окна

Создайте состояние `isVisible` с начальным значением `false`, чтобы управлять видимостью модального окна.

Создайте реф `modalRef`, чтобы получить доступ к DOM элементу модального окна.

3. Реализуйте функции открытия и закрытия модального окна

Создайте функцию `openModal`, которая устанавливает `isVisible` в `true`.

Создайте функцию `closeModal`, которая добавляет класс `closing` к модальному окну, ждет завершения анимации закрытия и затем устанавливает `isVisible` в `false`.

4. Управляйте классами анимации с помощью `useEffect`

Добавьте `useEffect`, который будет отслеживать изменения в `isVisible`.

Когда `isVisible` становится `true`, добавьте класс `opening` к модальному окну.

Когда `isVisible` становится `false`, убедитесь, что класс `closing` добавляется правильно для запуска анимации закрытия.

5. Создайте разметку модального окна

Внутри компонента `Modal` добавьте кнопку для открытия модального окна.

Когда `isVisible` равно `true`, отобразите модальное окно и оверлей.

Внутри модального окна добавьте кнопку для его закрытия.

6. Определите стили для анимации в CSS

Создайте файл `Modal.css` и импортируйте его в `Modal.js`.

Определите стили для оверлея и модального окна.

Создайте классы `opening` и `closing`, которые будут управлять анимацией открытия и закрытия модального окна.

## 7. Свяжите стили и компоненты

Примените стили из `Modal.css` к компоненту `Modal`.

Убедитесь, что классы `opening` и `closing` добавляются и удаляются в зависимости от состояния и рефа.

## Продвинутые концепции useRef

Использование useRef для хранения состояния, которое не требует рендеринга:

В React есть два основных способа хранения данных: `useState` и `useRef`. Оба они сохраняют данные между рендерами компонента, но есть важные различия:

`useState` вызывает повторный рендеринг компонента, когда состояние изменяется.  
`useRef` не вызывает повторный рендеринг при изменении своего значения.

Иногда нам нужно хранить данные, которые не требуют рендеринга компонента при их изменении. Это можно сделать с помощью `useRef`.

Пример: Управление внешними библиотеками

Представим, что у нас есть компонент, который использует внешнюю библиотеку для анимации, и нам нужно сохранить ссылку на объект этой библиотеки. В этом случае изменение состояния объекта библиотеки не должно вызывать повторный рендеринг компонента, потому что это может повлиять на производительность или привести к ненужным перерисовкам.

Вот пример, как использовать `useRef` для управления библиотекой анимации:



```
import React, { useEffect, useRef } from 'react';
import SomeAnimationLibrary from 'some-animation-library';

const AnimatedComponent = () => {
    // Создаем ref для хранения экземпляра анимационной библиотеки
    const animationInstanceRef = useRef(null);

    useEffect(() => {
        // Инициализируем анимацию при первом рендре компонента
        animationInstanceRef.current = new SomeAnimationLibrary.Animation({
            element: document.getElementById('animatedElement'),
            animationType: 'bounce',
        });

        // Запускаем анимацию
        animationInstanceRef.current.start();

        // Очищаем анимацию при размонтировании компонента
        return () => {
            if (animationInstanceRef.current) {
                animationInstanceRef.current.stop();
            }
        };
    }, []);

    return <div id="animatedElement">Анимированный элемент</div>;
};

export default AnimatedComponent;
```

В этом примере мы делаем следующее:

1. Создаем ref: `const animationInstanceRef = useRef(null);` - создаем ref, который будет хранить экземпляр анимационной библиотеки.
2. Инициализация анимации: В `useEffect` мы инициализируем анимацию и сохраним экземпляр библиотеки в `animationInstanceRef.current`.
3. Запуск анимации: Используем методы библиотеки для запуска анимации.
4. Очистка: В возвращаемой функции из `useEffect` останавливаем анимацию при размонтировании компонента.

Почему используем useRef:

1. Безrerендера: Изменение `animationInstanceRef.current` не вызывает рендеринг компонента, в отличие от `useState`.

2. Сохранение ссылки: `useRef` позволяет хранить и изменять значение между рендерами компонента без его повторного рендера.
3. Управление сторонними библиотеками: `useRef` удобно использовать для интеграции со сторонними библиотеками, которые управляют DOM элементами напрямую.

Таким образом, `useRef` предоставляет удобный способ работы с внешними библиотеками и управления состоянием, которое не должно приводить к рендерингу компонента. Это особенно полезно для улучшения производительности и предотвращения ненужных обновлений.