Хук useState() отвечает за работу с состоянием внутри компонента. В отличие от классовых компонентов, хук занимается сразу всем: инициализацией, обновлением и предоставляет доступ к состоянию.

const Example = () => {

// Пример хука для работы с состоянием

const [count, setCount] = React.useState(0);

return (

<div>

<p>Вы нажали {count} раз(а)</p>

<button onClick={() => setCount(count + 1)}>

Нажми меня

</button>

</div>

);

};

Задача на useState

**const** BtnGroup **=** () **=>** {

*// BEGIN*

**const** [activeButton, setActiveButton] **=** useState(**null**);

**const** getClassName **=** (position) **=>** cn('btn btn-secondary', position, { active: activeButton **===** position });

**return** (

<div className="btn-group" role="group">

<button type="button" className={getClassName('left')} onClick={() **=>** setActiveButton('left')}>Left</button>

<button type="button" className={getClassName('right')} onClick={() **=>** setActiveButton('right')}>Right</button>

</div>

);

*// END*

};

Задача на UseEffect

useEffect(() => {

const fetchData = async () => setProducts(await getProducts());

fetchData();

}, []);

**Итог**

Мы узнали, что с помощью useEffect, мы можем вызывать разные побочные эффекты после того, как компонент отрендерится. Некоторым эффектам нужен сброс, поэтому они возвращают соответствующую функцию.

useEffect(() => {

function handleStatusChange(status) {

setIsOnline(status.isOnline);

}

ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);

return () => {

ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);

};

});

В некоторых эффектах нет этапа сброса, поэтому они не возвращают ничего.

useEffect(() => {

document.title = `Вы нажали ${count} раз`;

});

Хук эффекта покрывает оба сценария единым API.

Здесь нет никакого особого кода для решения проблем с обновлениями, так как useEffect решает их *по умолчанию*. Он сбрасывает предыдущие эффекты прежде чем выполнить новые. Чтобы показать это на практике, давайте рассмотрим последовательность подписок и отписок, которые этот компонент может выполнить в течение некоторого времени.

// Монтируем с пропсами { friend: { id: 100 } }

ChatAPI.subscribeToFriendStatus(100, handleStatusChange); // Выполняем первый эффект

// Обновляем с пропсами { friend: { id: 200 } }

ChatAPI.unsubscribeFromFriendStatus(100, handleStatusChange); // Сбрасываем предыдущий эффект

ChatAPI.subscribeToFriendStatus(200, handleStatusChange); // Выполняем следующий эффект

// Обновляем с пропсами { friend: { id: 300 } }

ChatAPI.unsubscribeFromFriendStatus(200, handleStatusChange); // Сбрасываем предыдущий эффект

ChatAPI.subscribeToFriendStatus(300, handleStatusChange); // Выполняем следующий эффект

// Размонтируем

ChatAPI.unsubscribeFromFriendStatus(300, handleStatusChange); // Сбрасываем последний эффект

Эта логика по умолчанию гарантирует согласованность выполняемых нами действий и исключает баги, распространённые в классовых компонентах из-за упущенной логики обновления