# Мотивация

## Трудно повторно использовать логику состояний между компонентами

В React нет способа «присоединить» повторно используемое поведение к компоненту (например, подключение к хранилищу). Если вы работали с React какое-то время, то вам могут быть знакомы такие паттерны, как рендер-пропсы и компоненты высшего порядка, которые пытаются решить эту проблему. Но эти паттерны заставляют вас изменять структуру компонентов, что делает код громоздким и трудным в поддержке. Если вы посмотрите на типичное React-приложение в React DevTools, то увидите «ад обёрток» из компонентов, окружённых провайдерами, консьюмерами, компонентами высшего порядка, рендер-пропсами и другими абстракциями. Хоть мы и можем отфильтровать их в DevTools, всё это указывает на более глубокую проблему в React. Нужен более удобный способ повторно использовать логику вокруг состояния.

С помощью хуков вы можете извлечь логику состояния из компонента, чтобы её протестировать или повторно использовать. **Хуки позволяют вам повторно использовать логику состояния, не затрагивая дерево компонентов.**

## Сложные компоненты становятся трудными для понимания

Нам часто приходилось поддерживать компоненты, которые изначально были простыми, но превратились в неуправляемый беспорядок, состоящий из логики состояния и побочных эффектов. Каждый метод жизненного цикла часто содержит смесь несвязанной логики. Например, компоненты могут загружать данные в componentDidMount и componentDidUpdate. Однако тот же метод componentDidMount может содержать несвязанную логику, которая добавляет обработчики события с отменой подписки в componentWillUnmount. Взаимосвязанный код, который изменяется вместе, разделяется, но совершенно несвязанный код в конечном итоге объединяется в один метод.

**Хуки позволяют разбить один компонент на маленькие функции по их назначению (например, подписке или загрузке данных)**, а не на основе методов жизненного цикла. Вы также можете контролировать внутреннее состояние с помощью редюсера, чтобы поведение было более предсказуемым.

## Классы путают как людей, так и машины

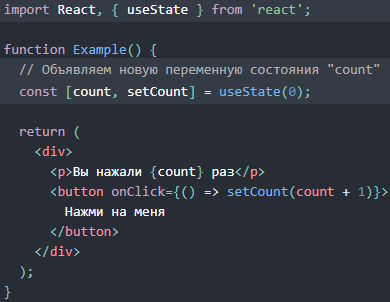
Про this и т.д.

# Что же такое хук?

Хуки — это функции, с помощью которых вы можете «подцепиться» к состоянию и методам жизненного цикла React из функциональных компонентов. Хуки не работают внутри классов — они дают вам возможность использовать React без классов.

React содержит несколько встроенных хуков, таких как useState. Вы также можете создавать собственные хуки, чтобы повторно использовать их в других своих компонентах. Давайте для начала рассмотрим встроенные хуки.

# Хук состояния (useState)

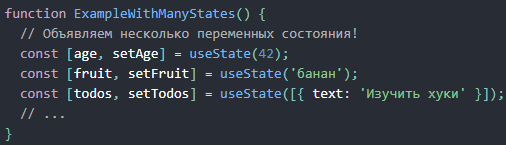


В этом примере, useState — это хук. Мы вызываем его, чтобы наделить наш функциональный компонент внутренним состоянием. React будет хранить это состояние между рендерами. Вызов useState возвращает две вещи (точнее массив, в котором эти две вещи лежат): текущее значение состояния и функцию для его обновления. Эту функцию можно использовать где угодно, например, в обработчике событий. Она схожа с this.setState в классах, но не сливает новое и старое состояние вместе.

Единственный аргумент useState — **это начальное состояние**. В примере выше — это 0. Исходное значение аргумента используется только при первом рендере.

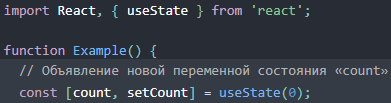
## Объявление нескольких переменных состояния.

Хук состояния можно использовать в компоненте более одного раза:

****

## Объявление переменной состояния

Хук **useState** предоставляет функциональным компонентам доступ к состоянию React. Раньше, если вы писали функциональный компонент и осознавали, что вам нужно наделить его состоянием, вам приходилось превращать этот компонент в класс. Теперь же вы можете использовать хук внутри существующего функционального компонента.



* **useState()** объявляет «переменную состояния» (здесь - count). Таким образом мы можем «сохранить» некоторые значения между вызовами функции. useState — это новый способ использовать те же возможности, что даёт this.state в классах. Обычно переменные «исчезают» при выходе из функции. К переменным состояния это не относится, потому что их сохраняет React.
* Единственный **аргумент** **useState** — это исходное состояние. В отличие от случая с классами, состояние может быть и не объектом, а строкой или числом, если нам так удобно.
* Вызов **useState вернёт** пару значений в массиве: текущее состояние и функцию, обновляющую состояние.

## Чтение состояния

Просто обращаемся напрямую к переменной:



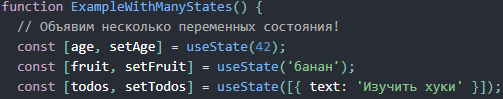
## Обновление состояния

Просто передаем нужное значение в функцию:



## Использование нескольких переменных состояния

Когда нам нужны несколько переменных, мы можем назвать каждую из них собственным именем:

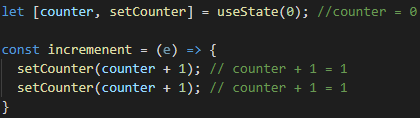


В примере выше мы видим локальные переменные age, fruit и todos, которые можем обновлять независимо друг от друга.

Использовать несколько переменных состояния совсем не обязательно, потому что они могут быть объектами или массивами, которые группируют связанные по смыслу данные. **Обратите внимание**, что, в отличие от this.setState в классах, *обновление переменной состояния всегда замещает её значение, а не осуществляет слияние.*

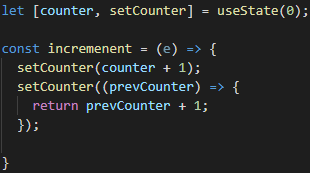
## Асинхронность

Функция изменения состояния хука работает асинхронно:



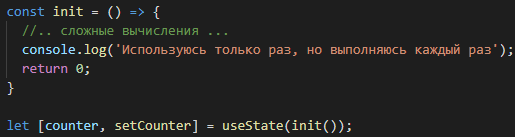
То есть setCounter() сработает уже вне основного потока кода функции increment, а после его выполнения.

Чтобы использовать предыдущее состояние стейта, в setCouner() нужно передавать не значение напрямую, а коллбэк, который в качестве параметра принимает предыдущее значение counter и возвращает новое на его основании:



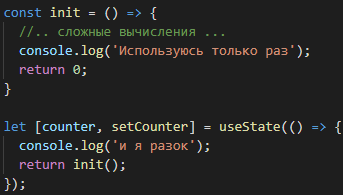
## Функция и инициализация стейта

Инициализационное значение может возвращать функция:



Возвращаемое значение функции будет использовано только в первый раз, а в следующие рендеры она будет работать в холостую, отнимая ресурсы.

Для того, чтобы этого избежать, в useState нужно передать коллбэк, который вернет результат и выполнится только один раз:



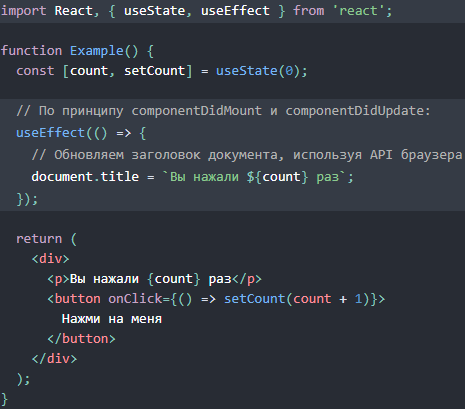
# Хук эффекта

## Вместо componentDidMount, componentDidUpdate и componentWillUnmount

Вам скорее всего доводилось ранее запрашивать данные, делать подписки или вручную менять DOM из React-компонента. Мы расцениваем эти операции как «побочные эффекты» или side effects (или сокращённо «эффекты»), так как они могут влиять на работу других компонентов и их нельзя выполнить во время рендера.

С помощью хука эффекта useEffect вы можете выполнять побочные эффекты из функционального компонента. Он выполняет ту же роль, что и **componentDidMount, componentDidUpdate** и **componentWillUnmount** в React-классах, объединив их в единый API.

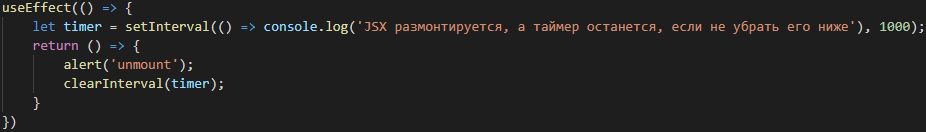
К примеру, этот компонент устанавливает заголовок документа после того, как React обновляет DOM:



Когда вы вызываете useEffect, React получает указание запустить вашу функцию с «эффектом» после того, как он отправил изменения в DOM. Поскольку эффекты объявляются внутри компонента, у них есть доступ к его пропсам и состоянию. По умолчанию, React запускает эффекты после каждого рендера, **включая первый рендер**.

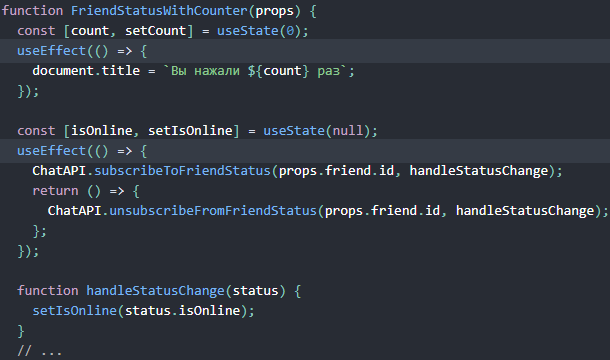
## Эффекты со сбросом

При необходимости вы можете вернуть из эффекта функцию, которая указывает эффекту, как выполнить за собой «сброс». Например, этот компонент использует эффект, чтобы запустить таймер, и выполняет сброс, когда компонента собирается размонтироваться:



Вы можете сделать так, чтобы [React пропускал повторные подписки](https://ru.reactjs.org/docs/hooks-effect.html" \l "tip-optimizing-performance-by-skipping-effects) (ниже) если props.friend.id, который мы передали в ChatAPI, остался без изменений.

Так же как и useState, вы можете использовать более одного эффекта в компоненте:



**Когда именно React будет сбрасывать эффект?** React будет сбрасывать эффект перед тем, как компонент размонтируется. Однако, как мы уже знаем, эффекты выполняются не один раз, а при каждом рендере. Вот почему React также сбрасывает эффект из предыдущего рендера, перед тем, как запустить следующий. *Сброс происходит каждый раз, когда мы заново заходим в коллбэк (эффект), либо когда компонента затирается из DOM.*

## Пояснения

Хуки дают вам возможность организовать побочные эффекты в компоненте по связанным частям (например, добавление или отмена подписки), вместо того, чтобы принуждать вас делить всё согласно методам жизненного цикла.

* **Что же делает useEffect?** Используя этот хук, вы говорите React сделать что-то после рендера. React запомнит функцию (то есть «эффект»), которую вы передали и вызовет её после того, как внесёт все изменения в DOM.
* **Почему же мы вызываем useEffect непосредственно внутри компонента?** Это даёт нам доступ к переменным состояния (или любым другим пропсам) прямиком из эффекта. Нам не нужен специальный API для доступа к этой переменной — она уже находится у нас в области видимости функции, потому что Хуки используют JavaScript-замыкания.
* **useEffect выполняется после каждого рендера**. *По умолчанию* он будет выполняться после каждого рендера и обновления. Вместо того, чтобы воспринимать это с позиции «монтирования» и «обновления», мы советуем просто иметь в виду, что эффекты выполняются после каждого рендера.

Опытные JavaScript-разработчики могут подметить, что функция, которую мы передаём в useEffect, будет меняться (объявляться) при каждом рендере. На самом деле, это было сделано преднамеренно. Это как раз то, что даёт нам возможность получать актуальную версию переменной count изнутри эффекта, не беспокоясь о том, что её значение устареет. Каждый раз при повторном рендере, мы ставим в очередь новый эффект, который заменяет предыдущий. В каком-то смысле, это включает поведение эффектов как часть результата рендера, то есть каждый эффект «принадлежит» определённому рендеру.

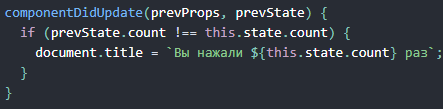
В отличие от componentDidMount или componentDidUpdate, эффекты, запланированные с помощью useEffect, не блокируют браузер при попытке обновить экран. Ваше приложение будет быстрее реагировать на действия пользователя, даже когда эффект ещё не закончился. Большинству эффектов не нужно работать в синхронном режиме. Есть редкие случаи, когда им всё же нужно это делать (например, измерять раскладку), но для этого мы разработали специальный хук useLayoutEffect с точно таким же API, как и у useEffect.

## Совет: используйте разные хуки для разных задач

Один из ключевых моментов, которые мы описали в [мотивации](https://ru.reactjs.org/docs/hooks-intro.html#complex-components-become-hard-to-understand), приводит аргументы о том, что в отличии от хуков, классовые методы жизненного цикла часто содержат логику, которая никак между собой не связана, в то время как связанная логика, разбивается на несколько методов.

## Оптимизация производительности за счёт пропуска эффектов

В некоторых случаях сброс или выполнение эффекта при каждом рендере может вызвать проблему с производительностью. В классовых компонентах, мы можем решить это используя дополнительное сравнение prevProps или prevState внутри componentDidUpdate:

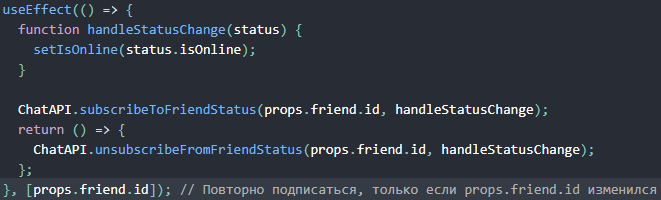


Вы можете сделать так, чтобы React пропускал вызов эффекта, если определённые значения остались без изменений между последующими рендерами. Чтобы сделать это, передайте массив в useEffect вторым необязательным аргументом:



Если count будет равен 5 и наш компонент повторно рендерится с тем же значением count = 5, React сравнит [5] из предыдущего рендера и [5] из следующего рендера. Так как, все элементы массива остались без изменений (5 === 5), React пропустит этот эффект. Если у вас будет несколько элементов в массиве, React будет выполнять наш эффект, в том случае, когда хотя бы один из них будет отличаться.

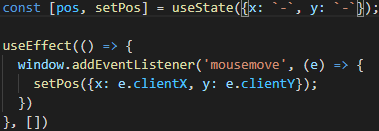
Это также работает для эффектов с этапом сброса:



**Примечание**

Если вы хотите использовать эту оптимизацию, обратите внимание на то, чтобы массив включал в себя **все значения из области видимости компонента (такие как пропсы и состояние), которые могут изменяться с течением времени, и которые будут использоваться эффектом**. В противном случае, ваш код будет ссылаться на устаревшее значение из предыдущих рендеров. Отдельные страницы документации рассказывают о том, [как поступить с функциями](https://ru.reactjs.org/docs/hooks-faq.html#is-it-safe-to-omit-functions-from-the-list-of-dependencies) и [что делать с часто изменяющимися массивами](https://ru.reactjs.org/docs/hooks-faq.html#what-can-i-do-if-my-effect-dependencies-change-too-often).

Если вы хотите запустить эффект и сбросить его только один раз (при монтировании и размонтировании), вы можете передать пустой массив ([]) вторым аргументом. React посчитает, что ваш эффект не зависит от каких-либо значений из пропсов или состояния и поэтому не будет выполнять повторных рендеров:



(*тут нам нужно только один раз поставить слушателя*)

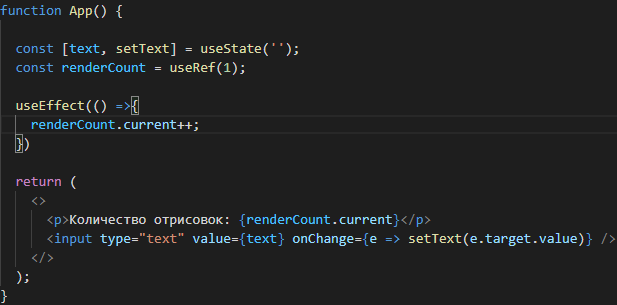
Это не обрабатывается как особый случай — он напрямую следует из логики работы массивов зависимостей.

Если вы передадите пустой массив ([]), пропсы и состояние внутри эффекта всегда будут иметь значения, присвоенные им изначально. Хотя передача [] ближе по модели мышления к знакомым componentDidMount и componentWillUnmount, обычно есть [более хорошие](https://ru.reactjs.org/docs/hooks-faq.html#is-it-safe-to-omit-functions-from-the-list-of-dependencies) [способы](https://ru.reactjs.org/docs/hooks-faq.html#what-can-i-do-if-my-effect-dependencies-change-too-often) избежать частых повторных рендеров. Не забывайте, что React откладывает выполнение useEffect, пока браузер не отрисует все изменения, поэтому выполнение дополнительной работы не является существенной проблемой.

# Хук useRef

## Применение как хранение значения

Если нам нужно хранить какое-то состояние, как это делает useState, но при этом, когда что-то в него пишем не хотим вызывать ререндер компоненты – поможет useRef:

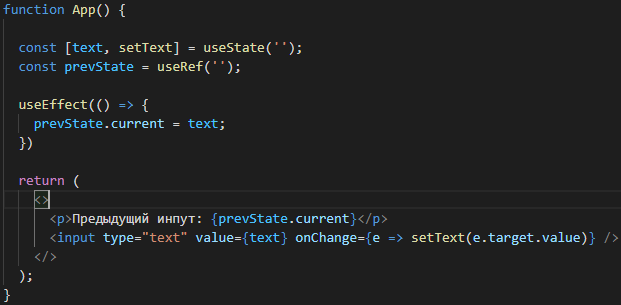


useRef() возвращает объект со свойством current, в котором мы можем хранить наши значения.

## Применение как ссылка на DOM элемент

Также может работать и как React.createRef(), но есть отличие. Разница в том, что createRef всегда будет создавать новую ссылку. В компоненте на основе классов вы обычно помещаете ref в свойство экземпляра во время построения (например, this.input = createRef()). У вас нет этой опции в функциональном компоненте. useRef заботится о том, чтобы каждый раз возвращать один и тот же ref, что и при первоначальном рендеринге.

## Применение: получить предыдущий стейт

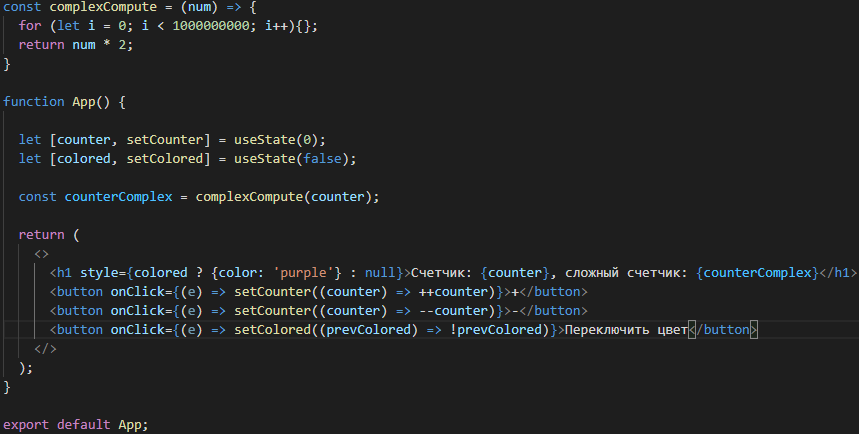


После того, как DOM отрисован, срабатывает useEffect, в котором сохраняется значение стейта. Далее, когда стейт изменился, сначала вставилось в HTML значение предыдущего стейта, а текущее сохраняется в prevState только после отрисовки.

# Хук useMemo

## Мотивация

Имеем компонент:



В функции complexCompute производятся сложные вычисления, которые записываются в переменную и используются для вывода в разметке.

Кнопка «Переключить цвет» переключает тумблер и вызывает изменения стейта, что приводит к ререндеру компоненты. Но т.к. функция App начинает выполняться заново, то и complexCompute также снова начнет пересчитывать значение, хотя нам это не нужно (counter все равно не менялся). Таким образом переключение цвета и потенциально другие возможности компонента будут работать с задержкой.

## Решение с помощью useMemo

Можно увидеть, что результат, который возвращает complexCompute можно закешировать, если значение, на основании которого он рассчитывается – не изменился. В данном случае результат рассчитывается на основании counter:



## Еще одно применение

Допустим, мы хотим, чтобы useEffect запускал свой коллбек, если изменится style, который мы указали вторым аргументом в качестве заисимости:



Но прикол в том, что эффект (коллбек) все равно вызовется, даже если мы нажали на кнопку прибавления, т.к. при ререндере в style создается новый объект, хоть и с теми же свойствами (другая ссылка). Поэтому наш коллбэк в useEffect все равно выполнится.

В данной ситуации объект style также можно закешировать:



# Хук useCallback

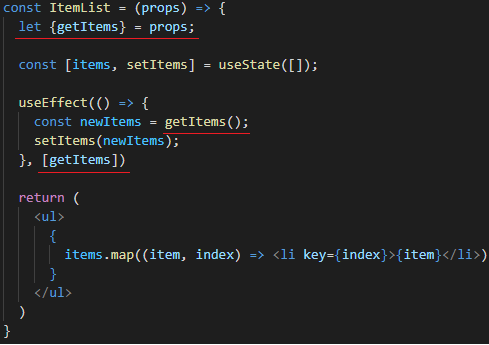
## Мотивация

Имеем ситуацию: в компоненте App объявлена функция, которая может генерировать массив на основании каких-то данных, полученных с сервера.

Эту функцию прокинули в пропсы в дочернюю компоненту, которая использует ее чтобы получить массив и отрисовать что-то на его основании:



В дочерней компоненте объявлен useEffect, эффект которого обновляет свой стейт новым массивом, если в пропс getItems пришла новая функция:

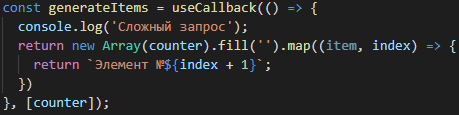


Но когда мы нажмем на кнопку «Переключить цвет», то произойдет изменение стейта, а значит компонента будет перерисована. Следовательно, будет перерисован и дочерний ItemList, в пропс getItems которой придет уже новая ссылка на функцию, а значит ограничение useEffect’а не сработает и в эффекте снова будет вызвана getItems.

Значит нам нужно закешировать generateItems, чтобы в эту ссылку приходила новая функция при ререндере, только если изменится counter, а не что-то другое.

## Решение с помощью useCallback

Код нашей функции мы обернем в useCallback:



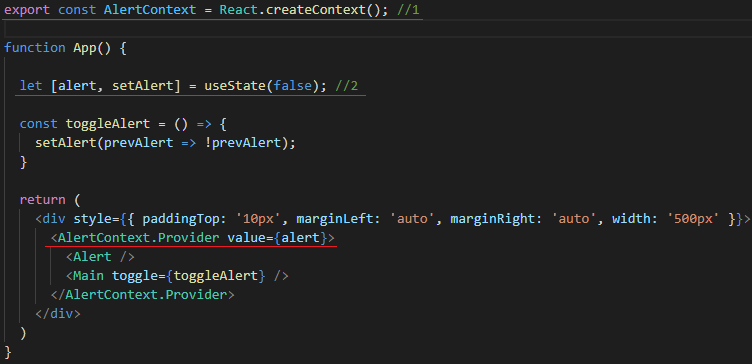
Это первым параметром передается именно код функции, **у нее может быть параметр. То есть useCallback вернет функцию.**

Теперь, если counter не поменялся, то в generateItems при ререндере будет приходить одна и та же ссылка на функцию. Если counter изменится, то ссылка на функцию обновится, это будет считаться уже новой функцией. А до этого при любом ререндере она считалась новой функцией, даже если counter не менялся.

# Хук useContext

Позволяет работать с React.createContext(), не оборачивая потребитель контекста в Consumer.

## Простой пример



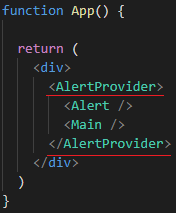
Тут задача состоит в том, чтобы потомки <App>, а конкретно <Alert> имела доступ у переменной состояния alert, на основании которой она будет либо показывать свою разметку, либо ничего не показывать:



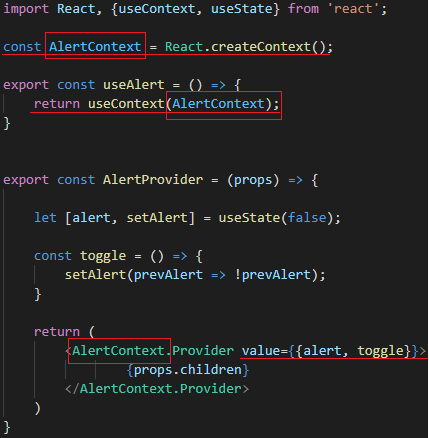
Кроме импорта хука useContext нам еще нужно импортировать контекст который мы создали в App.

## Паттерн использования

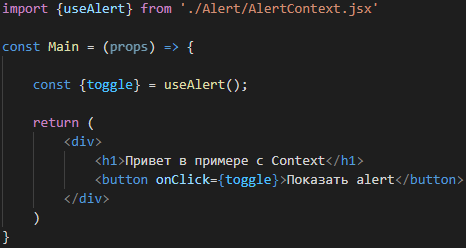
Суть в том, что мы создадим компонент-обертку, в котором создадим какое-то состояние, которое будет доступно всем его потомкам, которых он рендерит.



AlertProvider – это обертка:



Хук useContext, возвращает то, что положили в проп value компонента AlertContext.Provider. useContext вызывается в потомках, которым нужны эти данные. Чтобы в дочерних компонентах не нужно было импортировать AlertContext и useContext вместе – обернем получение контекста в функцию useAlert, которая обеспечит получение в дочерних компонентах данных и импортировать нужно будет в них только её:



В Main, с помощью useAlert, которая вызывает хук useContext, получили из контекста функцию переключения значения из стейта.

А в Alert получаем само значение:



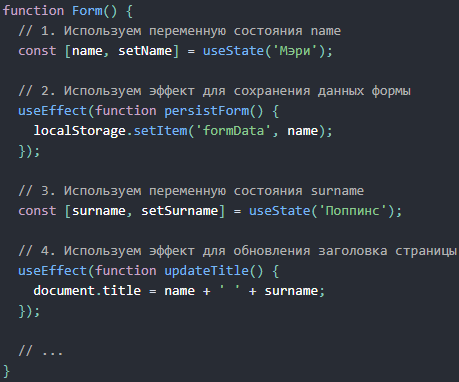
# Хук useReducer

# Правила хуков

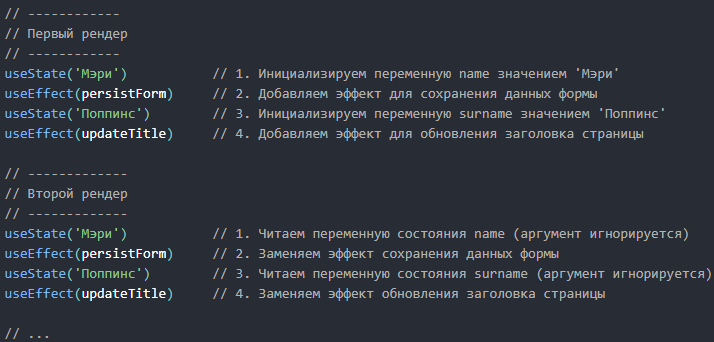
* Хуки следует вызывать только **на верхнем уровне**. Не вызывайте хуки внутри циклов, условий или вложенных функций.
* Хуки следует вызывать только **из функциональных компонентов React**. Не вызывайте хуки из обычных JavaScript-функций. Есть только одно исключение, откуда можно вызывать хуки — это ваши пользовательские хуки. Мы расскажем о них далее.

## Объяснение

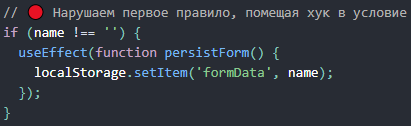
Как мы ранее узнали, хуки состояния или эффектов в одном и том же компоненте можно использовать многократно:



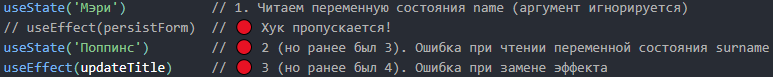
Итак, как же React сопоставляет переменные состояния с вызовами useState? Ответ таков: React полагается на порядок вызова хуков. Наш пример работает, потому что порядок вызова хуков одинаков при каждом рендере:



До тех пор пока порядок вызова хуков одинаков в каждом рендере, React может сопоставить некое внутреннее состояние с каждым из них. Но что случится, если мы поместим вызов хука (например, эффект persistForm) внутрь условного оператора?

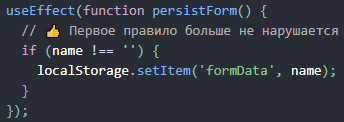


Условие name !== '' равняется true при первом рендере, поэтому хук выполняется. Тем не менее, при следующем рендере пользователь может обратить это условие в false, очистив поля формы. Теперь во время рендера хук будет пропущен и порядок вызовов хуков изменится:



React не будет знать, что вернуть для второго вызова хука useState. React ожидал, что второй вызов хука в этом компоненте соответствует эффекту persistForm, так же как при предыдущем рендере, но это больше не так. Начиная с этого момента, вызов каждого хука, следующего за пропущенным, также будет сдвинут на один назад, что приведёт к ошибкам.

**Вот почему хуки должны вызываться на верхнем уровне компонента.** Если мы хотим запускать эффект по условию, то можем поместить это условие внутрь хука:



Реакт запоминает порядок вызовов хуков в компоненте, и благодаря этому может сопаставить, например, какое значение должен вернуть хук useState в очередной раз. Таким образом обновленное значение попадет в нужную нам переменную

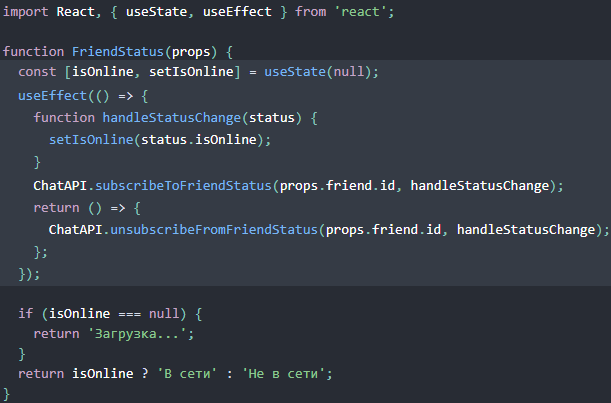
# Создание собственных хуков

## Основы

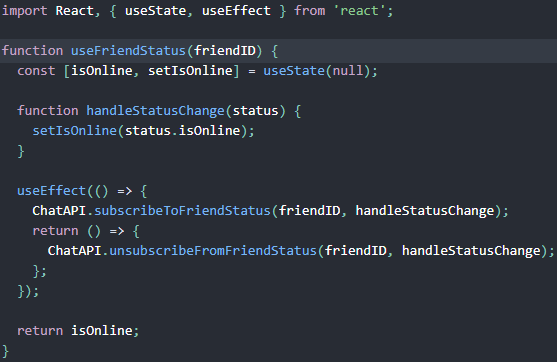
Иногда нужно повторно использовать одинаковую логику состояния в нескольких компонентах. Традиционно использовались два подхода: [компоненты высшего порядка](https://ru.reactjs.org/docs/higher-order-components.html) и [рендер-пропсы](https://ru.reactjs.org/docs/render-props.html). С помощью пользовательских хуков эта задача решается без добавления ненужных компонентов в ваше дерево.

**Пользовательские хуки** — это больше соглашение, чем дополнение. Если имя функции начинается с ”use” и она вызывает другие хуки, мы расцениваем это как пользовательский хук.

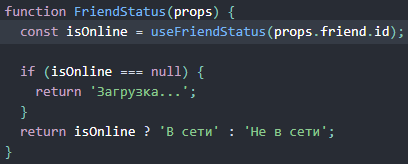
Ранее рассматривали компонент FriendStatus, который вызывал хуки useState и useEffect, чтобы подписаться на статус друга в сети:

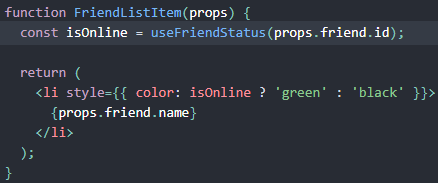


Мы хотим ещё раз использовать эту логику с подпиской, но уже в другом компоненте. Извлечём эту логику в пользовательский хук useFriendStatus:



Хук принимает friendID в качестве аргумента и возвращает переменную, которая показывает, в сети наш друг или нет. Теперь мы можем использовать этот хук в обоих наших компонентах:





Состояния каждого компонента никаким образом не зависят друг от друга. Хуки — это способ использовать повторно логику состояния, а не само состояние. Более того, каждое обращение к хуку обеспечивает совершенно изолированное состояние. Вы даже можете использовать один и тот же хук несколько раз в одном компоненте.

Просто нужная логика оборачивается в функцию, а не пишется такой же код хуков в другой компоненте. Здесь нет ничего нового, логика просто скопирована из компонентов выше.

Так же как и в компонентах, убедитесь, что вы не используете другие хуки внутри условных операторов и вызываете их на верхнем уровне вашего хука.

В отличие от React-компонента, пользовательский хук не обязательно должен иметь конкретную сигнатуру. Мы можем решить, что он принимает в качестве аргументов, и должен ли он что-либо возвращать. Другими словами, всё как в обычных функциях. Её имя всегда следует начинать с use, чтобы вы могли сразу увидеть, что к ней применяются [правила хуков](https://ru.reactjs.org/docs/hooks-rules.html).

Цель нашего хука useFriendStatus — подписать нас на статус друга. Поэтому он принимает в качестве аргумента friendID и возвращает статус друга в сети.

**Как пользовательский хук получает изолированное состояние?** Каждый вызов хука получает изолированное состояние. Поскольку мы вызываем useFriendStatus напрямую, с точки зрения React наш компонент просто вызывает useState и useEffect. И как мы узнали ранее, мы можем вызывать useState и useEffect много раз в одном компоненте, и они будут полностью независимы.

## Передача информации между хуками

Поскольку хуки являются функциями, мы можем передавать информацию между ними.

Продемонстрируем это, используя другой компонент из нашего гипотетического примера чата. Это средство выбора получателей сообщений чата, которое показывает, находится ли выбранный в данный момент друг в сети:



Мы сохраняем выбранный в настоящее время идентификатор друга в переменной состояния recipientID и обновляем его, если пользователь выбирает другого друга в <select>.

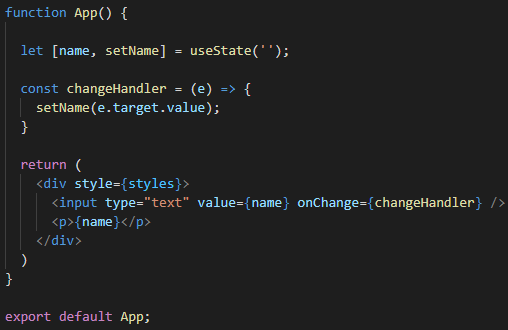
Поскольку вызов хука useState даёт нам последнее значение переменной состояния recipientID, мы можем передать его в наш пользовательский хук useFriendStatus в качестве аргумента:



Это позволяет нам узнать, находится ли выбранный друг в сети. Если мы выберем другого друга и обновим переменную состояния recipientID, наш хук useFriendStatus **отменит подписку на ранее выбранного друга и подпишется на статус вновь выбранного.**

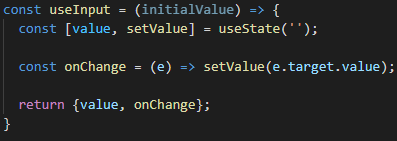
## Еще пример

Имеется компонент формы с полем ввода:



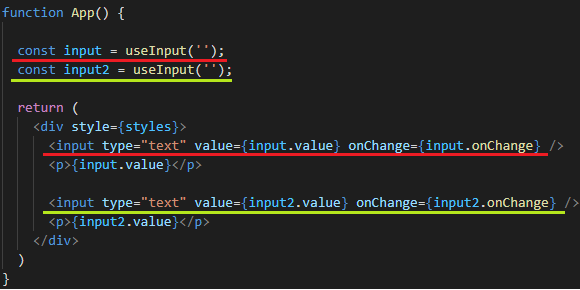
Допустим, нам нужно доабвить еще несколько полей ввода и у каждого должно быть свое состояние. Не дублировать же нам одну и ту же логику чтения из стейта значения и его изменения.

Вынесем логику стейта инпута в отдельный Хук:



Наш хук принимает инициализационное значение для стейта, создает хук стейта и возвращает его текущее значение и функцию для его изменения.

Теперь, с помощью нашего хука, может создать состояния для скольких угодно инпутов:



Состояние для каждого инпута независимо!

# shouldComponentUpdate на Хуках

## Скрин



Документация React.memo - <https://ru.reactjs.org/docs/react-api.html>