**Хуки** — это функции, с помощью которых можно «подцепиться» к состоянию и методам жизненного цикла функциональных React-компонентов.

Правила хуков:

1. Можно вызывать только на верхнем уровне;
2. Можно вызывать только из React-функций;
3. Название начинается с use…

Существует несколько типов Хуков:

1. Нативные хуки React:
   1. useState
   2. useEffect
   3. useContext
   4. useReducer
   5. useCallback
   6. useMemo
   7. useRef
2. Хуки React Redux:
   1. useSelector
   2. useDispatch
3. Хуки React Router DOM
   1. useNavigate
   2. useParams
   3. useHistory
4. **Нативные хуки React**
   1. **useState**

**useState** - это хук, который позволяет добавить переменную состояния к компоненту. Мы вызываем его, чтобы наделить функциональный компонент внутренним состоянием. React будет хранить это состояние между рендерами.

**const [state, setState] = useState(initialState)**

**useState** принимает в себя один аргумент - это изначальное состояние (**initialState**).

**useState** возвращает массив с двумя значениями:

1) Переменной состояния (**state**). Во время первого рендеринга она будет соответствовать переданному вами изначальному состоянию (**initialState**);

2) Функцей изменения переменной состояния **set**, которая позволяет обновить состояние до другого значения и запустить повторный рендеринг.

Она может принимать в себя либо новое значение setValue(newState), либо callback-функцию с предыдущим значением в качестве параметра setValue((prevState) => {})

* 1. **useEffect**

**useEffect** — это хук, который позволяет синхронизировать компонент с внешней системой.

**useEffect** принимает в качестве параметров **callback-функцию** (с логикой эффекта) и может принимать **массив зависимостей**.

**useEffect ( ( ) =>{**

*/ \* логика эффекта \*/*

**} , [** */\* массив зависимостей (необязательный параметр) \*/* **];**

Если массив зависимостей отсутствует: эффект запускается при каждом рендере.

Если массив зависимостей пустой, то эффект сработает при первом рендере и при размонтировании компонента.

Если массив зависимостей содержит ссылки на переменные, то эффект запуститься при первом рендере и при изменении переменных, от которых он зависит

Хук **useEffect** может возвращать функцию очистки (clean-up), которая будет вызываться при удалении компонента из DOM-дерева (размонтировании) или при изменении зависимостей.

**useEffect(setup, dependencies?)**

Опционально функция setup может возвращать функцию очистки. (Когда компонент будет добавлен в DOM, React запустит функцию настройки. После каждого повторного рендеринга с измененными зависимостями React сначала запустит функцию очистки (если ее обозначили) со старыми значениями, а затем - функцию настройки с новыми значениями. После того, как компонент будет удален из DOM, React запустит функцию очистки.)

* 1. **useContext**

**useContext** - это хук, позволяющий получать значения из компонента-провайдера на любом уровне вложенности.

**useContext** позволяет передавать данные в дочерние компоненты без указывания их в props-ах.

**useContext** *возвращает* значение контекста для вызывающего его компонента.

**const value = useContext(** /\* *Мой контекст* \*/ **);**

Контекст (*Мой контекст*) создаётся с помощью createContext.

Он определяется как значение, переданное ближайшему SomeContext.Provider над вызывающим компонентом в дереве. (Чтобы определить значение контекста, React просматривает дерево компонентов и находит ближайшего поставщика контекста выше по дереву).

Если такого провайдера нет, возвращаемое значение будет значением defaultValue, которое вы передали в createContext для этого контекста. Возвращаемое значение всегда актуально. React автоматически перерисовывает компоненты, которые считывают некоторый контекст, если он изменяется.

Мы должны воспользоваться функцией createContext (импортируется из реакта), применив ее к начальному состоянию контекста initialContext (его создаем и типизируем).

Затем создаем текущий контекст const stateContext = createContext(initialContext).

Оборачиваем родительский компонент в <stateContext.Provider value={{state: ‘x’}}>]</stateContext.Provider>.

Затем где необходимо, достаём из контекста то, что нам надо через деструктуризацию: const {state} = useContext(stateContext).

* 1. **useReducer**

**useReducer** - хук для работы с централизованным состоянием приложения, архитектурно схож с библиотекой Redux.

**const [state, dispatch] = useReducer(reducer, initialArg, init?)**

**useReducer** *возвращает* массив, первым элементом которого является **state** (переменную, которая хранит текущее состояние приложения, формирующееся на основе исходного состояния), а вторым - **dispatch** (функцию, взаимодействующую с функцией-редьюсером, принимает в качестве параметра объект с действием и полезной нагрузкой. Только dispatch может менять состояние state). Функция dispatch позволяет изменить состояние и вызвать ре-рендер.

Параметры:

**reducer**: функция-редуктор, которая указывает на то, как следует обновить состояние. Это должна быть чистая функция: принимать в качестве аргументов состояние (**state**) и действие (**action**) и возвращать следующее состояние. Состояние и действие могут быть любого типа.

**initialArg**: значение, из которого вычисляется начальное состояние. Это может быть значение любого типа. То, как из него вычисляется начальное состояние, зависит от следующего аргумента **init**.

(опционально) **init:** функция-инициализации, которая должна возвращать начальное состояние. Если этот аргумент не указан, начальное состояние устанавливается в InitialArg. В противном случае начальное состояние устанавливается в результате вызова **init(initialArg)**.

**Dispatch** только обновляет переменную состояния для следующего рендера. Если вы прочитаете переменную состояния после вызова функции dispatch, вы получите старое значение.

В useReducer нельзя напрямую изменять то, что пришло в аргументах функции (state, action), выполнять какие-либо сайр-эффекты (запросы к api), вызывать не чистые функции (Date.now(), Math.random()), игнорировать копирование состояния.

* 1. **useCallback**

**useCallback** — это хук, который позволяет кэшировать определение функции между повторными рендерами.

**const cachedFn = useCallback(fn, dependencies)**

**useCallback** принимает встроенный коллбэк и массив зависимостей.

**useCallback** возвращает мемоизированную версию коллбэка, который изменяется, только если изменяются значения одной из зависимостей.

Параметры

**fn:** значение функции, которое вы хотите кэшировать. Она может принимать любые аргументы и возвращать любые значения. React вернет (не вызовет!) функцию во время первоначального рендеринга. При следующем рендеринге React снова предоставит вам ту же функцию, если зависимости не изменились с момента последнего рендеринга. В противном случае он даст вам функцию, которую вы передали во время текущего рендеринга, и сохранит ее на случай, если ее можно будет повторно использовать позже. React не будет вызывать вашу функцию. Функция возвращается вам, чтобы вы могли решить, когда и нужно ли ее вызывать.

**зависимости:** список всех переменных значений, на которые ссылается fn. Они включают реквизиты, состояние и все переменные и функции, объявленные непосредственно внутри тела компонента. Список зависимостей должен иметь постоянное количество элементов и быть встроенным, например [dep1, dep2, dep3]. React будет сравнивать каждую зависимость с ее предыдущим значением.

**useCallback** возвращает:

При начальном рендере useCallback возвращает переданную функцию fn.

При последующих рендерах он либо вернет уже сохраненную функцию fn из последнего рендера (если зависимости не изменились), либо вернет функцию fn, которую вы передаёте во время этого рендера.

**useMemo + useCallback** позволяют организовать точечный рендеринг только тех компонентов, которые поменялись, даже если используется useContext.

**UseCallback** работает только с функциями, а useMemo со всем. И передаётсямасив зависимостей

* 1. **useMemo**

**useMemo** — это хук, который позволяет кэшировать результат вычислений между повторными рендерами.

При начальном рендере **useMemo** возвращает результат вызова calculateValue без аргументов.

Во время следующих рендеров он либо вернет уже сохраненное значение из последнего рендера (если зависимости не изменились), либо снова вызовет calculateValue и вернет результат, который вернул calculateValue.

**const cachedValue = useMemo(calculateValue, dependencies)**

**useMemo** нужен для того что бы не рендерилоь что то лишнее, например сорт. Нужен для оптимизации приложени, что бы не производились сложные операции без надобности. Зачастую где есть массив.

Параметры:

**calculateValue**: функция, вычисляющая значение, которое вы хотите кэшировать. Она должна быть чистой, не принимать аргументов и возвращать значение любого типа. React вызовет функцию во время первоначального рендера. При следующем рендере React снова вернет то же значение, если зависимости не изменились с момента последнего рендера. В противном случае он вызовет calculateValue, вернет результат и сохранит его, чтобы его можно было повторно использовать позже.

**зависимости:** список всех переменных значений, на которые есть ссылки в коде calculateValue. Они включают реквизиты, состояние и все переменные и функции, объявленные непосредственно внутри тела вашего компонента. Список зависимостей должен иметь постоянное количество элементов и быть встроенным, например [dep1, dep2, dep3]. React будет сравнивать каждую зависимость с ее предыдущим значением.

**useMemo** принимает создающую функцию и массив зависимостей. Он будет повторно вычислять мемоизированное значение только тогда, когда значение какой-либо из зависимостей изменилось. Это помогает избежать дорогостоящих вычислений при каждом рендере.

Функция, переданная в useMemo запускается во время рендеринга.

**useMemo** всегда возвращает мемоизированное значение.Если не передать массив, то новое значение будет вычисляться при каждом рендере

* 1. **useRef**

**useRef** принимает: **initialValue**, это значение, которое вы хотите, чтобы текущее свойство объекта ref было изначально. Это может быть значение любого типа. Этот аргумент игнорируется после первоначального рендеринга.

**useRef** возвращает объект с одним свойством **current**.

**const refContainer = useRef(initialValue)**

По сути, **useRef** похож на «коробку», которая может содержать изменяемое значение в своём свойстве .current.

В current изначально установлено значение initialValue, которое вы передали. Позже вы можете установить его на что-то другое. Если вы передадите объект ref в React в качестве атрибута ref для узла JSX, React установит свое текущее свойство. При следующем рендеринге useRef вернет тот же объект.

Пояснения:

* Вы можете изменить свойство ref.current. В отличие от состояния, это свойство можно изменять. Однако, если он содержит объект, который используется для рендеринга (например, часть вашего состояния), вам не следует изменять этот объект.
* Когда вы изменяете свойство ref.current, React не выполняет повторный рендеринг вашего компонента. React не знает, когда вы меняете его, потому что **ref — это простой объект JavaScript**.
* Не записывайте и не читайте ref.current во время рендеринга, за исключением инициализации. Это делает поведение вашего компонента непредсказуемым.

В строгом режиме React будет дважды вызывать функцию вашего компонента, чтобы помочь вам найти случайные примеси. Это поведение предназначено только для разработки и не влияет на сборку для production. Каждый объект ref будет создан дважды, но одна из версий будет отброшена. Если функция вашего компонента чистая (как и должно быть), это не должно влиять на поведение.

При следующем рендеринге useRef вернет тот же объект. Вы можете изменить его свойство current, чтобы сохранить информацию и прочитать ее позже. Это может наапомнить состояние т.е. первое возвращаемое значение из массива хука useState, но есть важное отличие.

Изменение ref не вызывает повторный рендеринг.

Это означает, что ссылки идеально подходят для хранения информации, которая не влияет на визуальный вывод вашего компонента. Например, если вам нужно сохранить идентификатор интервала (имеется ввиду setInterval) и получить его позже, вы можете поместить его в ref. Чтобы обновить значение внутри ref, вам нужно вручную изменять его свойство current.

При использовании ref вы будете уверены в том что:

* Вы можете **хранить информацию** между повторными рендерами (в отличие от обычных переменных, которые сбрасываются при каждом рендеринге).
* Его изменение **не вызывает повторный рендеринг** (в отличие от переменных состояния, которые вызывают повторный рендеринг).
* Информация является **локальной** для каждой копии вашего компонента (в отличие от внешних переменных, которые являются общими).

Изменение ссылки не запускает повторный рендеринг, поэтому ref’ы не подходят для хранения информации, которую вы хотите отобразить на экране.

Но хук **useRef** полезен не только установкой атрибута с рефом. Он [удобен для сохранения любого мутируемого значения](https://ru.legacy.reactjs.org/docs/hooks-faq.html" \l "is-there-something-like-instance-variables).

Это возможно, поскольку useRef( ) **создаёт обычный JavaScript-объект**. Единственная разница между useRef( ) и просто созданием самого объекта {current: ...} — это то, что хук useRef даст один и тот же объект с рефом при каждом рендере.

Имейте в виду, что **useRef** не уведомляет вас, когда изменяется его содержимое. **Мутирование свойства .current не вызывает повторный рендер**. Если вы хотите запустить некоторый код, когда React присоединяет или отсоединяет реф к узлу DOM, вы можете использовать [колбэк-реф](https://ru.legacy.reactjs.org/docs/hooks-faq.html" \l "how-can-i-measure-a-dom-node) вместо этого.

1. **Хуки React Redux**

**2.1. useSelector**

**useSelector** - хук для доступа к состоянию и его ключам.

**const contacts = useSelector(contactsSelector)**

Позволяет извлекать данные из состояния(state) хранилища(store) Redux с помощью функции селектора.

**useSelector** принимает один обязательный аргумент - функцию-селектор, которая будет использоваться для выбора необходимого куска состояния. Эта функция должна принимать текущее состояние хранилища и возвращать нужную часть состояния. Функция-селектор может быть создана с помощью библиотеки Reselect, что позволяет кэшировать результат и оптимизировать производительность.

**useSelector** возвращает выбранную часть состояния хранилища, а также добавляет компонент в список слушателей на изменение этой части состояния. Если состояние изменилось, компонент будет обновлен

useSelector() по умолчанию использует строгую проверку равенства ссылок ===, а не с приведением типов.

Вы можете вызывать useSelector() несколько раз в пределах одного функционального компонента. Каждый вызов useSelector() создает отдельную подписку на Redux хранилище (store).

Вторым **необязательным** агрументом для useSelector можно передавать функцию shallowEqual из пакета react-redux, которая выполняет поверхносное сравнение двух объектов, либо же можно написать отдельную функцию для строго сравнения.

Пример с функцией shallowEqual:

**2.2. useDispatch**

Хук для доступа к методу dispatch.

**const dispatch = useDispatch()**

Он возвращает ссылку на функцию dispatch из Redux store.

Хук **useDispatch** ничего в себя не принимает.

!!!Не путать с методом (функцией) **dispatch.** А метод (функция) **dispatch** принимает в себя **type** и **payload**

При вызове хука **useDispatch** возвращается функция **dispatch**, которая является методом объекта Store и позволяет отправлять действия (**actions**) в хранилище **Redux**

1. **Хуки React Router DOM**

React Router поставляется с несколькими хуками, которые позволяют получить доступ к состоянию маршрутизатора и выполнять навигацию внутри компонентов.

**3.1. useNavigate**

Хук **useNavigate()** из пакета react-router-dom предоставляется для навигации между страницами в приложении, используя роутинг, который определен в компоненте.

**const navigate = useNavigate();**

Хук useNavigate(), а точнее функция, полученная в результате его вызова **принимает** два параметра:

1. **Обязательный.** Путь, по которому нужно перейти (может быть строкой или объектом;
2. **Необязательный**. Объект с дополнительными параметрами, которые будут переданы в URL: state, search, hash, key.
3. Либо можно передать только дельта-число, чтобы переходить по стэку истории. Например, навигация (-1) по своей сути аналогична нажатию кнопки «Назад»

Хук useNavigate **возвращает** функцию, которую можно использовать для программной навигации.

Если в вторым параметром передать replace со значением true, то в таком случае при переходе на другую страницу запись в стеке истории будет перезаписана, а не добавлена новая.

Возвращает функцию, которая позволяет осуществлять программные перемещения (история, прямые ссылки).

const navigate = useNavigate();

navigate(‘’/users);

navigate(-1);

**3.2. useParams**

Хук **useParams не принимает** никаких параметров.

**useParams** возвращает объект пар ключ/значение параметров URL. Используйте его для доступа к match.params текущего <Route>.

//App.tsx

<Route path=‘/users/:id’ element={<UserCard />} />

//UserCard.tsx

const {id} = useParams();

Return <h1>{id}</h1>;

Хук **useParams** **возвращает** объект,  свойства которого соответствуют динамическим частям маршрута. Пример работы.

Изначальный путь был следующий: posts/:category/:title

В адресной строке было: posts/:1344456/:132

Объект, из useParams будет следующего вида: { category: ‘1344456’, title: ‘132’ }

Дочерние маршруты наследуют все параметры от своих родительских маршрутов.

Таким образом, хук **useParams()** очень удобен для работы с динамическими параметрами маршрута в React-приложении, и позволяет избежать передачи параметров через props.

* 1. **useHistory**

Хук **useHistory** дает доступ к истории, которой вы можете пользоваться для навигации.

import { useHistory } from "react-router-dom";

function HomeButton() {

**let history = useHistory();**

function handleClick() {

history.push("/home");

}

return (

<button type="button" onClick={handleClick}>

Go home

</button>

);

}