**Какие темы изучите?**

Вы с успехом преодолели самый первый спринт на React-интенсиве и погрузились в мир декларативного программирования. Наш рейс отправляется дальше — в мир бизнес-логики приложения и её хранения.

**В этом спринте вы изучите:**

* Проблему глубокого проброса пропсов и то, как она влияет на проектную работу
* Глобальное хранилище в React проекте с использованием React Context
* Продвинутую альтернативу встроенному Context — библиотеку Redux
* Понятия экшенов и редьюсеров в Redux
* Структурирование запросов к API при использовании Redux с помощью усилителей
* Расширение для Redux в браузере — Redux Devtools
* Улучшенный пользовательский интерфейс с помощью группы событий Drag and Drop
* Библиотеку react-dnd и её связь с Redux

**Навыки, которые вы получите**

Вы научитесь создавать глобальное хранилище для приложения с помощью Redux, доступ к которому получают все подписанные компоненты приложения. Узнаете, как структурировать всю бизнес-логику приложения и пользоваться ей. Научитесь взаимодействовать с API с помощью усилителей в Redux и обрабатывать несколько исходов выполнения запроса.

Вы сумеете писать Drag and Drop в приложении несколькими способами. В том числе попробуете на практике библиотеку react-dnd.

Всё это позволит вам самостоятельно проектировать приложения малого и среднего размера. Навыки, которые мы перечислили выше, — неотъемлемая часть работы с почти любым React-приложением. Спринт будет сложным, но классным.

Перейдём к первой теме.

**Введение в глобальное состояние**

Одно из главных преимуществ компонентного подхода — разделение ответственности. Компоненты — автономные и самостоятельные блоки, которые слабо связаны друг с другом.

Слабая связность важна, потому что позволяет безопасно вносить изменения в одни компоненты и не нарушает работу других. В React она достигается за счёт того, что компоненты строго определяют набор данных, которые нужны для их работы. В этом им помогает фиксированный список пропсов.

Помимо пропсов компоненты могут содержать внутреннее состояние. Оно может быть вычислено на основе пропсов или получено по запросу к внешнему API. Рассмотрим компонент, который показывает информацию о случайном фильме:

Скопировать кодJSX

function RandomMovie() {

const [movie, setMovie] = useState();

useEffect(() => {

const URL = "https://api.nomoreparties.co/beatfilm-movies/";

fetch(URL + Math.floor(Math.random() \* 100 + 1))

.then((response) => response.json())

.then((data) => setMovie(data));

}, []);

return movie ? (

<section>

<h1>{movie.nameRU}</h1>

<p>Год: {movie.year}</p>

<p>Страна: {movie.country}</p>

<p>Режиссёр: {movie.director}</p>

<p>{movie.description}</p>

</section>

) : (

<p>loading...</p>

);

}

При монтировании этого компонента отправляется запрос к API, и полученные данные записываются в значение состояния movie.

Но кроме внутреннего состояния бывают данные, которые нужны почти всем компонентам. К примеру, в вашем приложении может присутствовать функция выбора языка. В зависимости от выбранного значения каждый компонент должен использовать разные переводы своего текстового содержимого.

Ещё один пример таких данных — информация о текущем пользователе. Эта информация приходит из запроса к API, а использоваться может на любом уровне приложения. Скорее всего, вы захотите отобразить пользовательский блок с аватаром в шапке приложения. Или вам понадобится отобразить счётчик непрочитанных сообщений, чтобы пользователь мог видеть актуальный статус переписок с любой страницы приложения.

В React такие данные принято называть глобальным состоянием, так как информация, которая содержится в таком состоянии, используется во всём приложении.

Существует несколько подходов и библиотек, которые позволяют организовать в приложении глобальное состояние и привязать его к компонентам. В этой теме вы познакомитесь с основными механизмами управления состоянием, которые реализованы в библиотеке React:

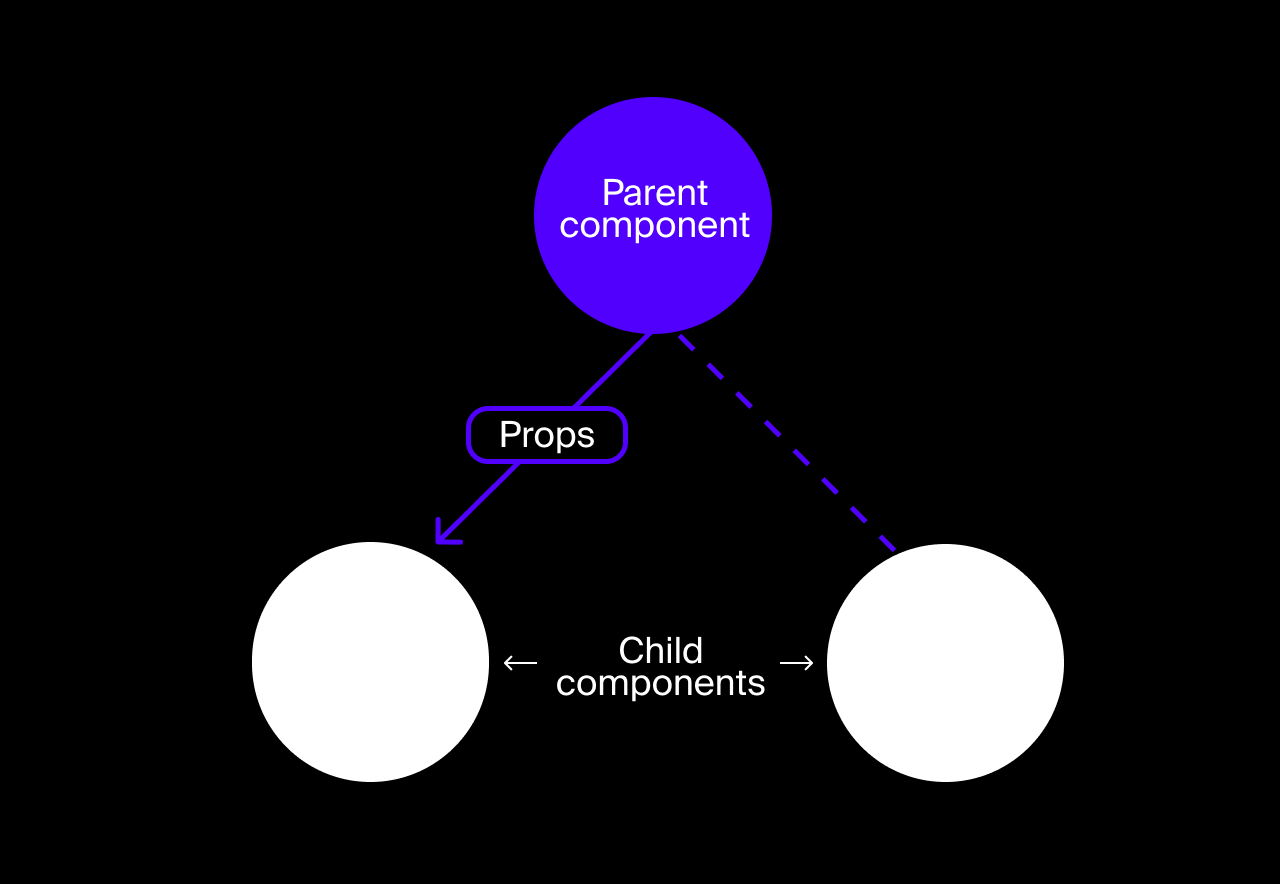
* Глубоким пробросом пропсов (prop drilling) — подходом, с помощью которого можно организовать передачу состояния компонентам посредством пропсов.
* Контекстом (React Context API) — методом, который позволяет передавать состояние дочерним компонентам на любую глубину.

Также вы познакомитесь с хуком useReducer(), который расширяет возможности useState() и позволяет более тонко настроить порядок изменения состояния.

Управление глобальным состоянием — тема не простая, но очень важная. Если вы готовы, то вперёд! Увидимся в следующем уроке.

# Props drilling

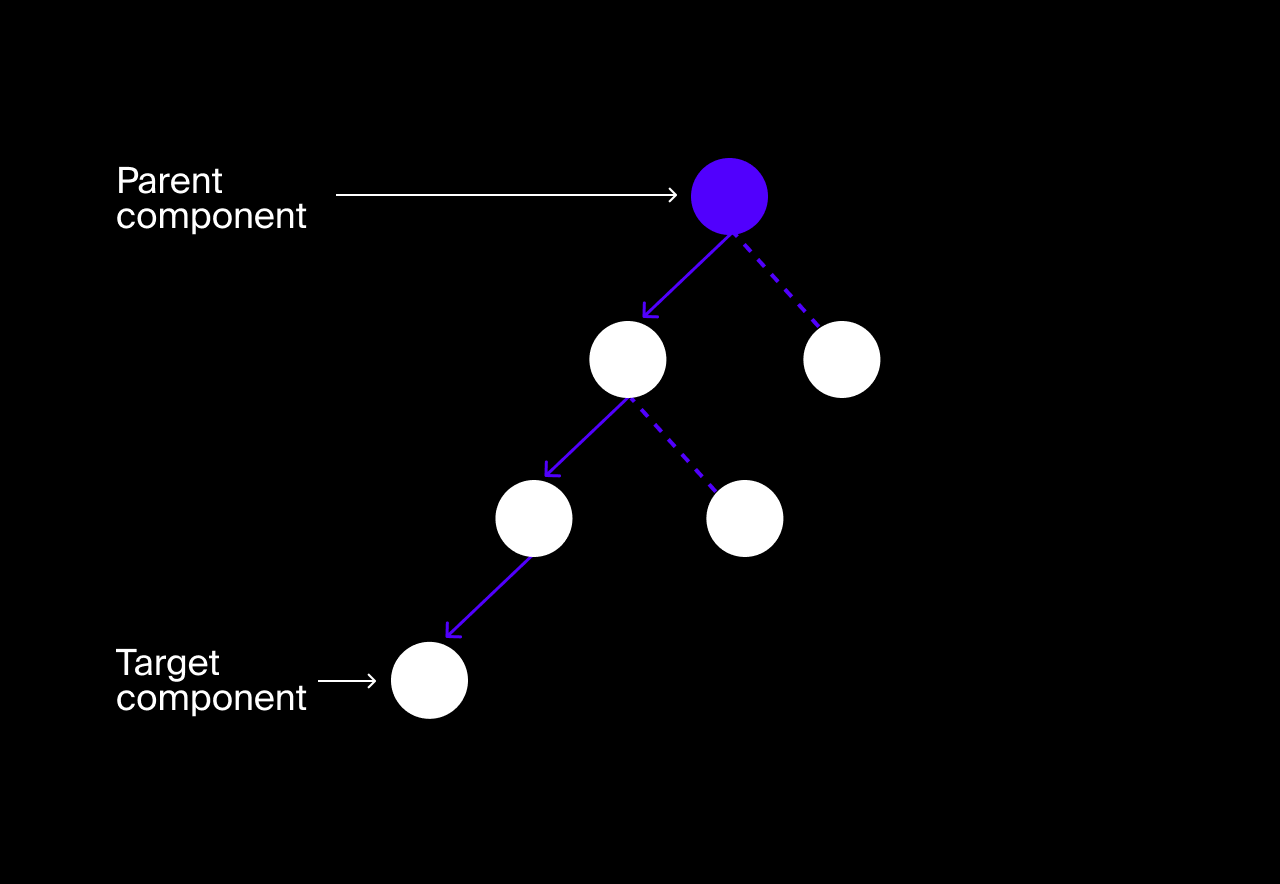
Компонент может содержать стейт с информацией, нужной дочернему компоненту. В таком случае значение стейта просто передаётся в дочерний компонент с помощью пропсов. Это называется «однонаправленным потоком данных» (англ. “one-way data binding.”). Мы уже сталкивались [с этим понятием в первом спринте](https://practicum.yandex.ru/learn/react/courses/8bb9f1d2-104b-4854-a4d5-d5d8766421ec/sprints/9582/topics/7e2a703f-0df6-4ca7-9d20-a3e67e1ed501/lessons/b7fced44-c88c-4353-a804-054fe338bd35/).



При однонаправленном потоке данных информация просто передаётся из родительского компонента в дочерний

Иногда нужно передать значения стейта другим компонентам, например родительским. Для этого необходимо перенести внутреннее состояние компонента и методов, которые им управляют, на один или несколько уровней выше. Эта техника называется «поднятие стейта» (англ. ”state lifting“).

Поднятие стейта можно использовать и для управления глобальным состоянием приложения. Такой подход принято называть глубоким пробросом пропсов (англ. ”props drilling“).



Фактически глубокий проброс пропсов ничем не отличается от поднятия стейта, разница только в «глубине» передачи пропсов

Например, вы делаете своё приложение для написания рецензий, в котором наряду с гениальными фичами есть и обычный личный кабинет с изменяемыми данными пользователя:

Скопировать кодJSX

function UserPage() {

const [user, setUser] = React.useState({ name: "Виссарион" });

function handleSubmit(e) {

e.preventDefault();

setUser({ name: e.target.name.value });

}

return (

<section>

<h2>Привет, {user.name}! Это твой личный кабинет.</h2>

<form onSubmit={handleSubmit}>

<div>

<label htmlFor="name">Имя: </label>

<input id="name" type="text" placeholder={user.name} />

</div>

<div>

<button type="submit">Сохранить</button>

</div>

</form>

</section>

);

}

Пока всё просто: данные о пользователе инициализируются, хранятся и изменяются внутри компонента. Но что, если потребуется отображать информацию о пользователе в других местах приложения: главной странице, шапке или чате? Для реализации такого функционала можно воспользоваться пропсами.

Перенесите состояние в компонент на уровень выше. Чтобы понять, на сколько уровней следует поднимать стейт, нужно разобраться, в каких компонентах предстоит работать с информацией из этого стейта. После этого важно найти общего родителя, к которому относятся эти компоненты. Обычно именно в нём и хранится глобальное состояние:

Скопировать кодJSX

function App() {

*// перенесли состояние в родительский компонент*

const [user, setUser] = React.useState({ name: "Андрей" });

return <UserPage />;

}

function UserPage() {

*// ...*

}

Теперь стейт user определён в компоненте верхнего уровня и не зависит от компонента UserPage. Чтобы воспользоваться информацией из глобального стейта и изменять его состояние внутри компонента UserPage, нужно передать стейт user и функцию-сеттер setUser в качестве пропсов:

Скопировать кодJSX

function App() {

const [user, setUser] = React.useState({ name: "Андрей" });

return <UserPage user={user} updateUserInfo={setUser} />;

}

function UserPage(props) {

function handleSubmit(e) {

e.preventDefault();

props.updateUserInfo({ name: e.target.name.value });

}

return (

<section>

<h2>Личный кабинет.</h2>

<form onSubmit={handleSubmit}>

<div>

<label htmlFor="name">Имя: </label>

<input id="name" type="text" placeholder={props.user.name} />

</div>

<div>

<button type="submit">Сохранить</button>

</div>

</form>

</section>

);

}

Аналогично можно передать стейт любому другому компоненту:

Скопировать кодJSX

function App() {

const [user, setUser] = React.useState({ name: "Андрей" });

return (

<Header user={user} />

<Reviews user={user} />

<UserPage user={user} updateUserInfo={setUser} />

);

}

function Header(props) {

return <header>{props.user.name}</header>;

}

function Reviews(props) {

*// ...*

}

function UserPage(props) {

*// ...*

}

С этим разобрались. Теперь представьте, что вы решили немного увеличить вложенность компонентов. Например, в процессе рефакторинга выделить форму редактирования данных о пользователе в отдельный компонент:

Скопировать кодJSX

function App() {

const [user, setUser] = React.useState({ name: "Андрей" });

return (

*// ...*

<UserPage user={user} setUser={setUser} />

);

}

*// перенесите форму из UserPage в отдельный компонент UserEditForm*

function UserPage(props) {

return (

<section>

<h2>Личный кабинет.</h2>

<UserEditForm user={props.user} updateUserInfo={props.setUser} />

</section>

);

}

function UserEditForm(props) {

function handleSubmit(e) {

e.preventDefault();

props.updateUserInfo({ name: e.target.name.value });

}

return (

<form onSubmit={handleSubmit}>

<div>

<label htmlFor="name">Имя: </label>

<input id="name" type="text" placeholder={props.user.name} />

</div>

<div>

<button type="submit">Сохранить</button>

</div>

</form>

);

}

В таком случае придётся передавать пропсы через всё дерево компонентов: от родительского компонента, в котором определён стейт, до самого последнего компонента, в котором этот стейт используется.

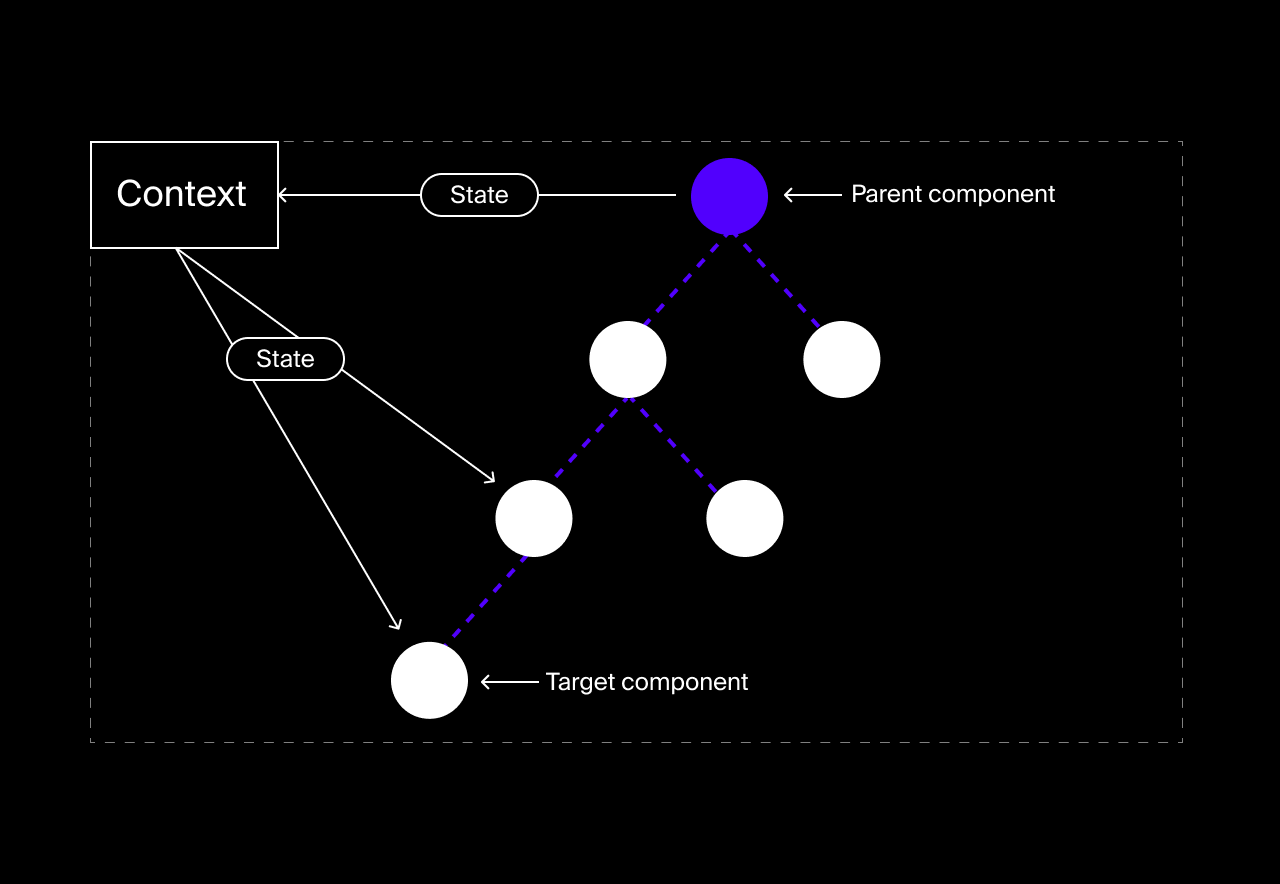
В этом и кроется основной недостаток глубокого проброса пропсов: из-за необходимости явно передавать стейт через всё дерево компонентов между ними увеличивается связность, что значительно усложняет разработку и поддержку приложения.

Применение механизма props drilling уместнее в случае, когда нужно объявить глобальное состояние для небольшого количества компонентов с малой вложенностью. При этом данные в стейте имеют простую структуру.

В ином случае стоит воспользоваться другим методом, например контекстом. С ним вы познакомитесь в следующем уроке.

# React.Context

Для передачи данных, которые могут понадобиться большому количеству компонентов, разработчики React предлагают использовать встроенный метод контекста — React context API. Контекст позволяет передавать данные сразу во все компоненты поддерева или даже всего приложения.



С помощью контекста вы создаёте «область видимости». Внутри неё каждый компонент может получить данные

При этом контекст никак не влияет на компоненты, которые расположены в поддереве и не используют состояние, находящееся внутри контекста. Это позволяет избежать проблемы увеличения связности, которая возникает при глубоком пробросе пропсов.

## Создание React.Context

Чтобы создать контекст, нужно воспользоваться встроенным методом React createContext(). Этот метод возвращает новый объект контекста.

Обычно контекст создаётся в отдельном файле и экспортируется из него:

Скопировать кодJSX

*// userContext.js*

export const UserContext = React.createContext();

Важное свойство объекта контекста — Provider. В провайдере содержится специальный React-компонент, который позволяет внедрить контекст в определённый участок существующего JSX-дерева.

Чтобы добавить контекст в приложение, достаточно обернуть компоненты в провайдер контекста и присвоить параметру value значение состояния:

Скопировать кодJSX

import { UserContext } from './userContext.js';

function App() {

const user = { name: "Виссарион" };

return (

<UserContext.Provider value={user}>

<UserPage />

</UserContext.Provider>

);

}

*// ...*

После этого состояние, которое хранится в контексте, станет доступно всем компонентам внутри <UserContext.Provider> на любом уровне вложенности.

Получить значение контекста в функциональном компоненте можно с помощью хука useContext():

Скопировать кодJSX

function UserPage() {

const user = useContext(UserContext);

return (

<section>

<h2>Привет, {user.name}! Это твой личный кабинет.</h2>

/\* ... \*/

</section>

);

}

Но в этом случае компонент способен только получать данные из контекста. Чтобы дочерние компоненты могли изменять значение контекста, нужно вместе со значением передать функцию-сеттер. Например, с помощью хука useState() — он как раз возвращает данные и функцию, с помощью которой можно изменить эти данные:

Скопировать кодJSX

const UserContext = React.createContext();

function App() {

*// создайте стейт с помощью хука useState()*

const userState = useState({ name: "Виссарион" });

return (

*// положите стейт в провайдер контекста*

<UserContext.Provider value={userState}>

<UserPage />

</UserContext.Provider>

);

}

function UserPage() {

*// значение состояния можно получить через деструктуризацию*

*// прямо как получение состояния в хуке useState()*

const [user] = useContext(UserContext);

return (

<section>

<h2>Привет, {user.name}! Это твой личный кабинет.</h2>

<UserEditForm />

</section>

);

}

function UserEditForm() {

*// точно так же, через деструктуризацию, можно получить и функцию-сеттер*

const [user, setUser] = useContext(UserContext);

function handleSubmit(e) {

e.preventDefault();

setUser({ name: e.target.name.value });

}

return (

<form onSubmit={handleSubmit}>

<div>

<label htmlFor="name">Имя: </label>

<input id="name" type="text" placeholder={user.name} />

</div>

<div>

<button type="submit">Сохранить</button>

</div>

</form>

);

}

## Особенность React.Context

С помощью контекста можно передавать данные компонентам любой вложенности и при этом связность между компонентами не будет увеличиваться. Это полностью решает проблему применения prop drilling, с которой мы столкнулись в предыдущем уроке.

Но не всё так гладко. У контекста есть важная особенность, о которой нужно помнить: при изменении значения контекста произойдёт перерисовка всех компонентов, которые подписаны на этот контекст.

Таким образом, использование одного контекста для хранения всех данных грозит проблемами с производительностью приложения. Представьте, что в контекст записаны статус аутентификации, информация о пользователе, тема, языки и сообщения. В таком случае при изменении имени пользователя React перерисует все компоненты, которые подписаны на глобальный контекст, даже если они никак не зависят от информации о пользователе. Поэтому не стоит хранить всю информацию о приложении в одном контексте.

Чтобы обойти эту проблему, можно создать множества атомарных стейтов вместо одного глобального. Но и у этого подхода есть недостаток — из-за синтаксиса контекста можно усложнить компонент большим количеством обёрток:

Скопировать кодJSX

function App() {

*// ...*

return (

<ThemeContext.Provider value={/\*...\*/}>

<I18nContext.Provider value={/\*...\*/}>

<AuthContext.Provider value={/\*...\*/}>

<UserContext.Provider value={/\*...\*/}>

/\* ... \*/

</UserContext.Provider>

</AuthContext.Provider>

</I18nContext.Provider>

</ThemeContext.Provider>

);

}

Контекст — мощный инструмент для передачи данных между компонентами и управления простым состоянием. Но из-за своих особенностей он не подходит для серьёзного стейт-менеджмента. Для решения задач хранения, передачи и управления множеством сущностей разработчики создали библиотеки, которые по сложности иногда сравнимы с самим React. В следующей теме вы познакомитесь с одной из них — Redux.

Но не спешите закрывать эту тему: вас ждёт последний урок, в котором мы расскажем о хуке useReducer(). Его ещё можно смело назвать «старшим братом» хука useState().

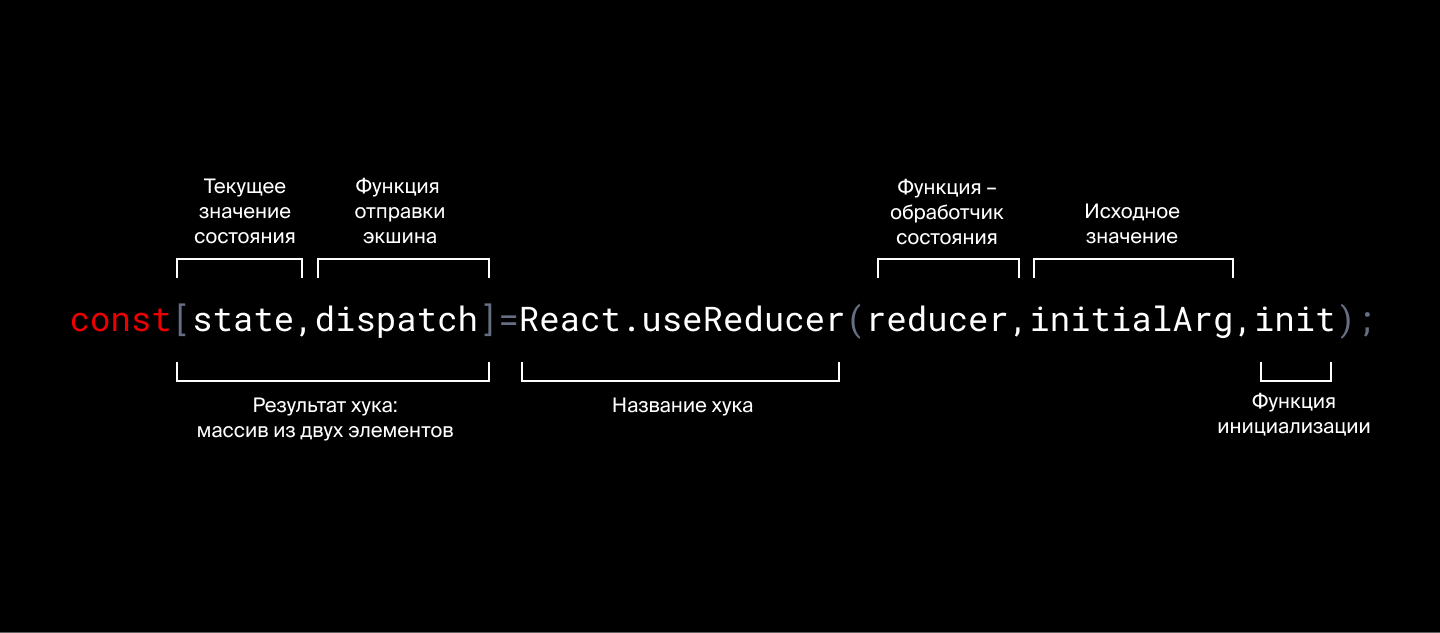
**useReducer()**

Вы уже [познакомились с хуком useState()](https://practicum.yandex.ru/learn/react/courses/8bb9f1d2-104b-4854-a4d5-d5d8766421ec/sprints/9582/topics/6673a3cf-f58b-4824-a5f0-32616f61ff80/lessons/641af03f-5ab9-4661-a656-df1122a83025/), с помощью которого можно сохранять и изменять состояние компонента, когда потребуется. useState() удобно применять при работе с простыми данными, а логика их изменения сводится к обычной перезаписи.

Но иногда возникает необходимость хранить сложные объекты, новое состояние которых зависит от предыдущего значения, а методы обновления этих объектов содержат дополнительную логику. Для решения подобных задач разработчики React придумали хук useReducer(), который можно назвать «старшим братом» хука useState().

**Интерфейс хука useReducer()**

useReducer() состоит из следующих компонентов:



*Интерфейс хука useReducer().*

Хук принимает до трёх аргументов:

* функцию reducer(), которая отвечает за дополнительную логику обработки состояния;
* начальное значение состояние initialArg;
* необязательный аргумент: функцию инициализации init(), с помощью которой можно вычислить начальное значение состояния.

И возвращает массив из:

* текущего значения состоянияstate;
* функции dispatch(), которая вызывает reducer() и передаёт ему необходимые аргументы.

**Как применять useReducer()**

На первый взгляд хук useReducer() может показаться сложным и запутанным, поэтому дополнительно разберём его функциональность на примере компонента Counter:

Скопировать кодJSX

*// начальное значение стейта*

const initialState = { count: 0 };

*// функция-редьюсер*

*// изменяет состояния в зависимости от типа переданного action*

function reducer(state, action) {

switch (action.type) {

case "increment":

return { count: state.count + 1 };

case "decrement":

return { count: state.count - 1 };

default:

throw new Error(`Wrong type of action: ${action.type}`);

}

}

function Counter() {

const [state, dispatch] = useReducer(reducer, initialState);

const handleIncrementClick = () => {

*// при вызове dispatch достаточно указать тип действия*

dispatch({ type: "increment" });

};

const handleDecrementClick = () => {

dispatch({ type: "decrement" });

};

return (

<>

Count: {state.count}

<button onClick={handleDecrementClick}>-</button>

<button onClick={handleIncrementClick}>+</button>

</>

);

}

В целом порядок использования хука useReducer() схож с применением useState(): вы описываете начальное значение состояния, а на выходе получаете стейт и функцию, с помощью которой можно изменить этот стейт. Но обратите внимание на функцию reducer(): в ней заранее заданы все доступные способы модификации стейта. В этом и заключается главное отличие хука useReducer() от useState(). В useState() для изменения состояния нужно производить вычисления внутри компонента. Даже если стейт задан в родительском компоненте, а вычисления происходят в дочернем, независимом от стейта, это состояние всё равно нужно пробросить в дочерний компонент. А чтобы обновить состояние с применением useReducer(), достаточно лишь вызвать функцию dispatch(), передать ей тип необходимых изменений, и тогда состояние будет пересчитано.

**Когда применять useReducer()**

Чаще всего взаимодействие с состоянием компонента можно без проблем описать с помощью хука useState(): он лаконичен, почти не уступает возможностям useReducer() и более прост в применении. Но в процессе развития приложения структура стейта может усложниться, а функции для его обновления — обрасти повторяющимися условиями и стать сильно зависимыми от предыдущего значения состояния. В этот момент стоит провести рефакторинг стейта и переписать взаимодействие с ним с помощью хука useReducer().

Также стоит применить useReducer(), если при проектировании компонента вы понимаете, что при работе с состоянием потребуется описать сложную логику, которую удобнее всего вынести в отдельную функцию reducer().

ХукuseReducer() — мощный инструмент в арсенале любого React-разработчика. Умение использовать этот хук по назначению и понимание принципов его работы позволит вам упростить работу со стейтом. А ещё это может пригодиться в изучении более сложных библиотек для управления состоянием, например Redux.

# Заключение

В этой теме вы изучили базовые механизмы управления глобальным состоянием: глубокий проброс пропсов и контекст. С помощью этих подходов вы можете определять состояние приложения и передавать его по всему поддереву компонентов.

Подход глубокого проброса пропсов удобно применять, если вам нужно передать простое состояние через небольшое количество компонентов.

Контекст будет полезен, когда вам необходимо создать «область видимости», внутри которой любой компонент сможет получить состояние, хранящееся в контексте.

Также вы познакомились с хуком useReducer(), который позволяет упростить работу с состоянием компонента, если логика обработки состояния выходит за рамки простой перезаписи данных.

К сожалению, все эти функции библиотеки React не полностью покрывают потребности управления состоянием в серьёзном приложении. Но сообщество разработчиков придумало множество своих решений, которые позволяют более продуктивно работать с состоянием. С одним из таких решений — библиотекой Redux — вы познакомитесь уже в следующей теме.

## Что можно сделать в проекте

Пройденная вами тема содержит информацию об антипаттерне глубокого проброса пропсов. Ещё и очень важную информацию про хук useReducer(). Этой информации достаточно, чтобы выполнить промежуточную [часть проектной работы](https://practicum.yandex.ru/trainer/web-plus/lesson/d0695643-15f1-4d52-a2da-31402609e15b/). Так вы сможете не только применить уже известные понятия (шаги 2 и 4), но и закрепить новые знания по хуку useReducer() и React.Context (все остальные шаги).

Как только завершите проектную работу — отправляйте её на проверку и приступайте к изучению следующей темы.