**Введение**

В этой теме мы будем говорить об информации.

Какими бы потрясающе красивыми современные приложения и веб-сайты ни были; какие бы изощрённые и сложные технологии, библиотеки и фреймворки в них ни использовались; сколько бы многомиллионных бюджетов ни вкладывалось в их разработку и поддержку; — всё это служит одной единственной цели. Эта цель — отображение и передача информации.

Передача информации ни много ни мало является причиной возникновения жизни на Земле. Несколько миллиардов лет назад где-то на планете случайно синтезировалась сложная молекула, которая обладала уникальным свойством: в определённых условиях она могла синтезировать копию самой себя. С каждой новой копией молекулы становились сложнее, но сохраняли уникальную функцию и продолжали множиться. У некоторых из них происходили случайные сбои: функция терялась и молекулы со временем разрушались. Другие же с каждой новой копией, наоборот, всё лучше адаптировались к изменениям в окружающей среде. В конечном счёте всё живое, включая веб-разработчиков, произошло от той самой молекулы и расселилось по всей планете. При этом самое главное, что мы смогли передать друг другу и пронести через миллиарды лет, — это информацию об уникальном строении молекулы, которая умеет воспроизводить саму себя. А ещё о несметном множестве приобретённых за это время усовершенствований.

Мы живём в информационную эру, и это название не содержит в себе ни капли преувеличения. Возможность почти мгновенно передать любую информацию в любую точку земного шара дала огромный скачок в развитии человечества. Ещё никогда люди не могли так быстро узнавать о новых мемах и обмениваться фотографиями смешных котиков.

Приятно осознавать, что в эту эпоху на передовой находимся именно мы — веб-разработчики. В наших руках будущее интернета и во многом именно от нас зависит, каким оно будет.

Можно начинать строить планы о ещё более прекрасной жизни для всего живого на Земле. Но для начала было бы неплохо научиться отображать данные в приложениях, написанных на «Реакте».

# Поднятие стейта

Компонент может содержать стейт с информацией, нужной дочернему компоненту. В таком случае значение стейта просто передаётся в дочерний компонент с помощью пропсов. Это называется «односторонним потоком данных» (англ. “one-way data binding.”).

Иногда может понадобиться передавать значения стейта другим компонентам, например, родительским. Для этого нужно перенести внутреннее состояния компонента и методов, которые им управляют, на один или несколько уровней выше. Эта техника называется «поднятие стейта» (англ. "state lifting").

Представьте, что в вашем приложении есть функция выбора ночной темы. При смене темы нужно добавить соответствующий CSS-класс для самого верхнего DOM-элемента в приложении.

Обычный компонент для выбора дня или ночи может выглядеть так:

Скопировать кодJSX

function ThemeSelect() {

const [theme, setTheme] = React.useState("day");

function handleChange(e) {

setTheme(e.target.value);

}

return (

<select onChange={handleChange} className={theme}>

<option value="day">День</option>

<option value="night">Ночь</option>

</select>

);

}

Значение выбранной темы сохраняется в стейт-переменной theme и используется в качестве CSS-класса для тега select. Однако, чтобы мы могли менять CSS для всех элементов интерфейса приложения, нам нужно задать этот класс самому верхнему DOM-элементу. Обычно он находится в корневом компоненте App:

Скопировать кодJSX

function App() {

return (

<div>

<h2>Добро пожаловать</h2>

<ThemeSelect />

</div>

);

}

Чтобы работать с theme изнутри App, нужно сделать две вещи:

1. Перенести создание стейта theme и функцию handleChange из ThemeSelect в App.
2. Добавить в ThemeSelect пропс onChange, который будет ссылаться на функцию handleChange.

После этого код App будет выглядеть так:

Скопировать кодJSX

function App() {

const [theme, setTheme] = React.useState("day");

function handleThemeChange(e) {

setTheme(e.target.value);

}

return (

<div className={theme}>

<h2>Добро пожаловать</h2>

<ThemeSelect onChange={handleThemeChange} />

</div>

);

}

Обратите внимание, что при переносе мы переименовали функцию handleChange. Внутри компонента ThemeSelect её предназначение было понятно из названия и контекста самого компонента. В более общем компоненте App предназначение функции с названием handleChange уже не такое очевидное, а значит мы должны называть её более явно: handleThemeChange.

При этом сам компонент ThemeSelect упростится:

Скопировать кодJSX

function ThemeSelect(props) {

return (

<select onChange={props.onChange}>

<option value="day">День</option>

<option value="night">Ночь</option>

</select>

);

}

Имя пропса onChange компонента ThemeSelect совпадает с именем атрибута onChange тега select. Здесь это логично и удобно, но в общем случае это необязательно.

Технику поднятия стейта можно также применять, когда несколько компонентов используют одну и ту же информацию. Например, нам нужно добавить компонент ThemeIcon, который выводит соответствующее изображение:

Скопировать кодJSX

function ThemeIcon(props) {

return (

<div className="icon">

{props.theme === 'day' ? '🔆' : '🌙'}

</div>

);

}

Более того, значение, выбранное в выпадающем меню, может однажды понадобиться и в самом ThemeSelect: например, для отметки нужного пункта.

Скопировать кодJSX

function ThemeSelect(props) {

return (

<select onChange={props.onChange}>

<option value="day">День {props.theme === "day" && "✅"}</option>

<option value="night">Ночь {props.theme === "night" && "✅"}</option>

</select>

);

}

Значение пропсов theme для обоих компонентов должно передаваться им из стейта их общего родительского компонента App:

Скопировать кодJSX

function App() {

const [theme, setTheme] = React.useState("day");

function handleThemeChange(e) {

setTheme(e.target.value);

}

return (

*// Первое использование theme*

<div className={theme}>

<h2>Добро пожаловать</h2>

{/\* Второе использование theme \*/}

<ThemeIcon theme={theme} />

{/\* Третье использование theme \*/}

<ThemeSelect theme={theme} onChange={handleThemeChange} />

</div>

);

}

И снова имена пропсов theme компонентов ThemeIcon и ThemeSelect совпадают с именем стейт-переменной theme компонента App. Это не обязательно, но очень удобно.

Рассмотрев все примеры, мы выяснили, что значение выбранной темы может понадобиться многим компонентам в приложении. Особенно, если простой CSS-стилизации для них будет недостаточно.

Такое случается часто, но добавлять ко всем компонентам пропс theme и прокидывать его на более низкие уровни через промежуточные компоненты не очень удобно. В следующих уроках мы научимся работать с данными, которые доступны в приложении глобально.

Пример

function App() {

const [size, setSize] = React.useState(0);

function handleSizeChange(e) {

setSize(e.target.value);

}

return (

<>

<Slider onChange={handleSizeChange} size={size} />

<Bubbles size={size}/>

</>

);

}

function Slider(props) {

return (

<div id="slider-container">

<input type="range" min="0" max="100" value={props.size} onChange={props.onChange} />

<div className="counter">{props.size}</div>

</div>

);

}

function Bubbles(props) {

// Наиболее простой способ отрисовать RandomBubble 20 раз подряд:

return Array.from(Array(20), (\_, i) => (

<RandomBubble size={props.size} key={i} />

));

}

function RandomBubble(props) {

// Используем стейт, чтобы запомнить исходные параметры при первом рендере

const [initialStyle] = React.useState({

top: getRandomInt(0, document.body.offsetHeight),

left: getRandomInt(0, document.body.offsetWidth),

backgroundColor: `#${getRandomInt(0, (256 \*\* 3) - 1).toString(16)}`,

});

const [initialScale] = React.useState(getRandomInt(1, 10) / 20);

return (

<div className="bubble" style={{

...initialStyle,

// Исходный масштаб домножается на значение props.size или на 0, если props.size не задан

transform: `scale(${initialScale \* (props.size || 0)})`,

}} />

);

}

// Возвращает случайное целое число в заданном диапазоне

function getRandomInt(min, max) {

return Math.floor(Math.random() \* (max - min + 1)) + min;

}

ReactDOM.render((

<App />

), document.querySelector('#root'));

# Глобальный стейт

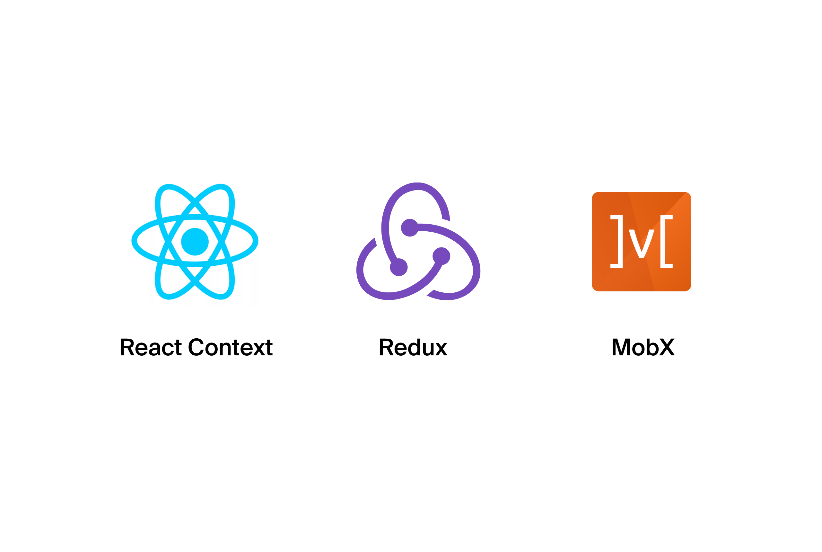
Одно из главных преимуществ компонентного подхода — разделение ответственности. Компоненты — автономные и самостоятельные блоки, слабо связанные друг с другом.

Слабая связность важна, потому что она позволяет безопасно вносить изменения в одни компоненты, не боясь нарушить работу других. В «Реакте» она достигается за счёт того, что компоненты строго определяют набор нужных для своей работы данных. И в этом им помогает фиксированный список пропсов.

Однако, бывают данные, нужные почти всем компонентам. В прошлом уроке мы рассмотрели пример с ночной темой. Подобным образом в вашем приложении может присутствовать функция выбора языка. В зависимости от выбранного значения каждый компонент должен использовать разные переводы для своего текстового содержимого.

Ещё один пример таких данных — информация о текущем пользователе. Эта информация приходит из запроса к API, а использоваться она может на любом уровне приложения. Скорее всего вы захотите отобразить пользовательский блок с аватаром в шапке приложения. Возможно, вам понадобится скрыть кнопку «Отправить сообщение» рядом с именем пользователя, который находится в середине списка всех активных пользователей. Так вы убережёте их от порочного круга экзистенциальных бесед с самими собой.

Существует несколько подходов и библиотек, которые позволяют организовать в приложении глобальный стейт и привязать его к компонентам. Так, React Context встроен в библиотеку React.js. А Redux — сторонняя библиотека.



Далее мы подробно поговорим про React Context, а в конце темы дадим вводную информацию об устройстве и предназначении Redux.

# Создание и подключение контекста

Конечно, можно было бы использовать поднятие стейта: хранить выбранную настройку в стейте компонента App и добавить каждому компоненту пропс translations, который будет задаваться родителем и пробрасываться дальше всем дочерним элементам. Но такой подход потребует много дополнительного кода.

Чтобы не писать это вручную, разработчики «Реакта» предлагают использовать контекст — специальный способ передачи данных одновременно во все компоненты поддерева или даже всего приложения.

Чтобы создать контекст, нужно использовать встроенный метод React.createContext. Этот метод возвращает новый объект контекста. Важное свойство этого объекта — Provider. В нём содержится специальный «Реакт-компонент», который позволяет внедрить контекст в определённый участок существующего JSX-дерева.

Обычно контекст создаётся в отдельном файле и экспортируется из него. Иногда в этом файле находится и само содержимое вариантов контекста:

Скопировать кодJSX

*// translationContext.js*

export const TranslationContext = React.createContext();

export const translations = {

en: {

greeting: 'Hello World',

},

ru: {

greeting: 'Привет, мир!',

},

};

Компонент Provider имеет пропс value со значением, которое нужно распространить всем дочерним элементам.

Скопировать кодJSX

*// App.js*

*// Импортируем объект контекста*

import { TranslationContext, translations } from './translationContext';

function App() {

*// Стейт, отвечающий за текущий язык*

const [lang, setLang] = React.useState('en');

return (

*// «Внедряем» данные из translations[lang] с помощью провайдера контекста*

<TranslationContext.Provider value={translations[lang]}>

{/\* Поддерево, в котором будет доступен контекст \*/}

<Main />

</TranslationContext.Provider>

);

}

Хоть провайдер и распространяет контекст автоматически по всему поддереву, дополнительно нужно подключить каждый из дочерних компонентов, в которых на самом деле будут использоваться эти данные.

Например, у нас могут быть компоненты, у которых вообще нет текстового содержимого, поэтому не нужно, чтобы они тоже перерисовывались, когда пользователь меняет язык в приложении.

Классовые и функциональные компоненты по-разному подключаются к контексту внутри поддерева. В следующем уроке вы научитесь это делать.

# Подписка на контекст

Вы подключили дерево компонентов к провайдеру контекста. Теперь, чтобы использовать его значение в любом из этих компонентов, нужно «подписать» компонент на контекст. Благодаря этому компонент будет перерисовываться каждый раз, когда меняется значение контекста.

**В функциональных компонентах** для этого используют хук React.useContext. Этот хук возвращает значение контекста, которое было передано в пропс value провайдера:

Скопировать кодJSX

function Header() {

*// Подписываемся на контекст TranslationContext*

const translation = React.useContext(TranslationContext);

*// Выводим содержимое translations.en.greeting,*

*// которое теперь доступно через translation.greeting*

return (

<h1>

{translation.greeting}

</h1>

);

}

**В классовых компонентах** для этой же задачи используют статическое поле (его ещё называют поле класса): contextType. Этому полю нужно присвоить объект контекста, после чего в компоненте становится доступно специальное свойство — this.context. Именно оно будет содержать значение контекста:

Скопировать кодJSX

import { TranslationContext } from './tranlationContext';

class Header extends React.Component {

static contextType = TranslationContext;

*// Выводим содержимое translations.en.greeting,*

*// которое теперь доступно через this.context.greeting*

render() {

return (

<h1>

{this.context.greeting}!

</h1>

);

}

}

# Использование нескольких провайдеров

Ничто не мешает нам добавить к дереву несколько провайдеров:

Скопировать кодJSX

*/\* Импортируем оба объекта контекста \*/*

import { TranslationContext, translations } from './tranlationContext';

import { CurrentUserContext } from './currentUserContext';

function App() {

*// Стейт, отвечающий за текущий язык*

const [lang, setLang] = React.useState('en');

*// Стейт, отвечающий за данные текущего пользователя*

const [currentUser, setCurrentUser] = React.useState({ name: 'Алёша' });

return (

*// «Внедряем» данные из translations[lang] с помощью провайдера первого контекста*

<TranslationContext.Provider value={translations[lang]}>

{/\* «Внедряем» данные из currentUser с помощью провайдера второго контекста \*/}

<CurrentUserContext.Provider value={currentUser}>

{/\* Поддерево, в котором будут доступны оба контекста \*/}

<Main />

</CurrentUserContext.Provider>

</TranslationContext.Provider>

);

}

Чтобы подписать функциональные компоненты, например, на 2 контекста, нужно просто 2 раза использовать хук с разными переменными контекста:

Скопировать кодJSX

import { TranslationContext } from './translationContext';

import { CurrentUserContext } from './currentUserContext';

function Header() {

*// Подписываемся на контекст TranslationContext*

const translation = React.useContext(TranslationContext);

*// Подписываемся на контекст CurrentUserContext*

const currentUser = React.useContext(CurrentUserContext);

return (

<h1>

{translation.greeting}, {currentUser.name}!

</h1>

);

}

С классовым подходом этот номер не пройдёт. Статическому полю contextType можно присвоить только одно значение, а значит компонент может быть подписан только на один контекст.

Чтобы обойти это ограничение в «Реакте» существует довольно сложная концепция — Context.Consumer. Мы не изучаем её на курсе подробно, потому что в последнее время на практике концепцию Context.Consumer используют редко. Вы можете почитать о ней [в официальной документации](https://ru.reactjs.org/docs/context.html#consuming-multiple-contexts).

Кстати, именно это ограничение стало одной из основных предпосылок к появлению в движке React.js более универсального механизма хуков.

# Введение в Redux

Redux — небольшая библиотека, которая позволяет организовать управление данными в JavaScript-приложениях. Библиотеку Redux создал российский разработчик Дэн Абрамов в 2015 году, когда ему было 23 года. С тех пор Дэн работает в основной команде React в Facebook и является одной из самых заметных фигур в современной веб-разработке.

С помощью Redux можно создать нечто похожее на базу данных прямо в JavaScript-приложении. Такая база представляет из себя обычный JavaScript-объект и называется хранилищем (англ. “store”). Во многом этот способ организации данных похож на React Context, однако Redux представляет из себя более сложную и функциональную концепцию.

# Основная идея Redux

В хранилище могут содержаться абсолютно все данные приложения и все переменные состояние, влияющие на интерфейс. Например, в нём можно хранить любые API-данные, идентификатор текущей страницы или состояние диалоговых окон (открыто/закрыто).

# Преимущества

При таком подходе у приложения появляется единый «источник правды» для его полного состояния в отдельно взятый момент времени.

Например, JS-объект можно легко сохранить в постоянное хранилище браузера (LocalStorage) и быстро восстановить после обновления страницы. Так пользователь увидит тот же самый интерфейс, который он видел, когда посещал страницу в последний раз.

С другой стороны это может быть полезно при отладке ошибок. Если возникает ошибка, можно легко восстановить полное состояние хранилища и воспроизвести её. Это сложнее сделать, если информация «размазана» по внутренним состояниям отдельных компонентов приложения. По этой же причине в Redux-приложениях легче писать юнит-тесты.

### «Иммутабельность»

Важнейшая концепция Redux — неизменяемость или иммутабельность (англ. “immutability”) объекта хранилища. Это означает, что при любом изменении данных в хранилище вместо внесения изменений в существующий объект, на его основе создаётся новая, изменённая копия. Чаще всего для этого используется оператор spread ... и методы массивов типа map. Сравните два этих примера:

Скопировать кодJSX

const store = { title: 'Hello', name: 'Gregory' };

*// Прямое изменение исходного объекта:*

store.name = 'James';

*// Создание нового объекта на основе исходного:*

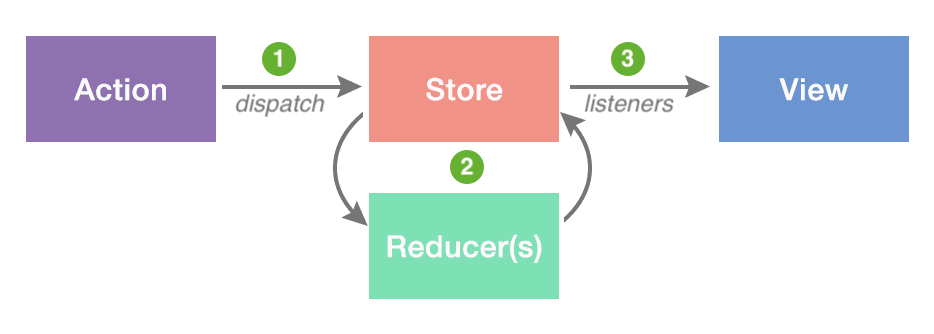
const newStore = { ...store, name: 'James' };

При таком подходе можно легко определить, изменились ли данные: достаточно сравнить newStore === store. Также можно вести историю изменений и если нужно, «откатиться» к предыдущей версии. Это бывает полезно при отладке приложения.

# Принцип работы Redux

Библиотека Redux работает по такому алгоритму:

1. Хранилище никогда не изменяется напрямую. Чтобы произвести изменения в хранилище, нужно отправить (англ. “dispatch”) в него «экшн» (англ. “action”, событие). Экшн — это JavaScript-объект, который всегда должен иметь ключ type. Он также может содержать дополнительные данные. Экшены отправляются в хранилище методом store.dispatch.
2. Внутри этого метода все новые экшены передаются на обработку «редьюсеру» (от англ. “reduce”, свёртывать) — функции, которую должен описать сам разработчик. Каждый раз редьюсер получает на вход новый экшн вместе с актуальным значением хранилища. В зависимости от типа экшена и данных в нём редьюсер создаёт новый объект хранилища на основе предыдущего (как в примере выше) и возвращает его. Основной редьюсер обычно состоит из нескольких дочерних редьюсеров, каждый из которых отвечает за обработку своего «участка» общего хранилища.
3. После окончания работы редьюсера новое значение, которое он вернул, заменяет текущее значение хранилища и вызываются функции-обработчики. Предварительно эти функции нужно установить с помощью метода store.subscribe. Таким образом разработчик получает доступ к новому значению хранилища и может произвести изменения в DOM.



# Redux и React

Redux — универсальная библиотека и используется не только в «Реакте». Например, многие используют её вместе с фреймворком Angular. Чтобы подключить Redux именно к «Реакту» применяют «соединительную» библиотеку [react-redux](https://react-redux.js.org/" \t "_blank).

Она предоставляет разработчику «Реакт-компонент» Provider (почти как тот, что был в React Context), а также функцию connect. Эта функция позволяет подключить хранилище к отдельным компонентам и использовать его данные, превращая их в пропсы.

# Недостатки и альтернативы

Некоторые разработчики не любят Redux из-за его «многословности». Для его использования обычно приходится писать много однотипного кода. В проекте быстро разрастается число экшенов и их типов, а также самих редьюсеров и других вспомогательных сущностей.

В то же время принципы Redux полезны для крупных проектов с большим количеством изменяющихся данных. Поэтому периодически появляются альтернативные, упрощённые реализации. Например, [библиотека ReactN](https://www.npmjs.com/package/reactn).

# Заключение

В этой теме вы изучили различные подходы к работе с данными в проектах на «Реакте». Большинство приложений существует ради данных –– их передачи и отображения. Поэтому так важно уметь с ними обращаться.

В начале темы вы узнали о том, как использовать стейт сразу в нескольких компонентах с помощью его «поднятия» вверх по иерархии компонентов.

Затем вы познакомились с понятием «глобальный стейт» и узнали об основных механизмах для работы с ним: React Context и Redux.

Какой из них использовать –– выберете вы и ваша команда. Возможно, в будущем именно вам предстоит разработать революционную технологию, которая совместит в себе все преимущества существующих подходов.

И это станет новым витком эволюции.