**Веб-приложения и фреймворки**

**Немного истории**

Много лет назад веб-сайты целиком генерировались на сервере, и каждое нажатие кнопки или переход по ссылке вызывали перезагрузку всей страницы. Хоть 56k-модемы и скрипели изо всех сил, скорость соединения всё ещё была далека от сегодняшней, поэтому веб-сайты сильно отставали от обычных системных приложений в удобстве и функциональности. Они содержали в основном статичную информацию, а веб-разработчиков многие даже не причисляли к программистам и снисходительно называли веб-мастерами или даже веб-дизайнерами.

В какой-то момент произошёл переворот: появилась технология AJAX, а вслед за ней SPA — веб-приложения, которые могли реагировать на действия пользователя и подгружать с сервера данные динамически, то есть без перезагрузки основной страницы. Такие приложения начали активно конкурировать с системными, а во многом даже опережать их: например, в скорости доставки новых версий. Стало появляться всё больше сложных веб-приложений, которые раньше трудно было себе представить: к примеру, полноценные графические редакторы — такие как «Фигма».

Началась эпоха Web 2.0, а JavaScript начал свой путь к захвату мира.

**Зачем нужны фреймворки**

HTML, CSS и JavaScript — отличные инструменты для разработки простых веб-страниц. Однако, с увеличением сложности проекта обычно требуются более комплексные решения: библиотеки и фреймворки. Фреймворк — это каркас приложения, предоставляющий разработчику универсальную структуру будущего проекта, а также набор готовых решений для типовых задач.

Первая основная задача — разбиение интерфейса на отдельные функциональные блоки (так называемые «компоненты»), которые можно переиспользовать. В качестве примера можно привести аватар с именем пользователя или кнопку лайка, которые могут встречаться в приложении много раз и в разных местах. Часто такие компоненты используют все три технологии: HTML, CSS и JavaScript. В этом случае удобно хранить их код рядом и отделить его от кода других компонентов. Многие фреймворки, включая «Реакт», предоставляют такую возможность.

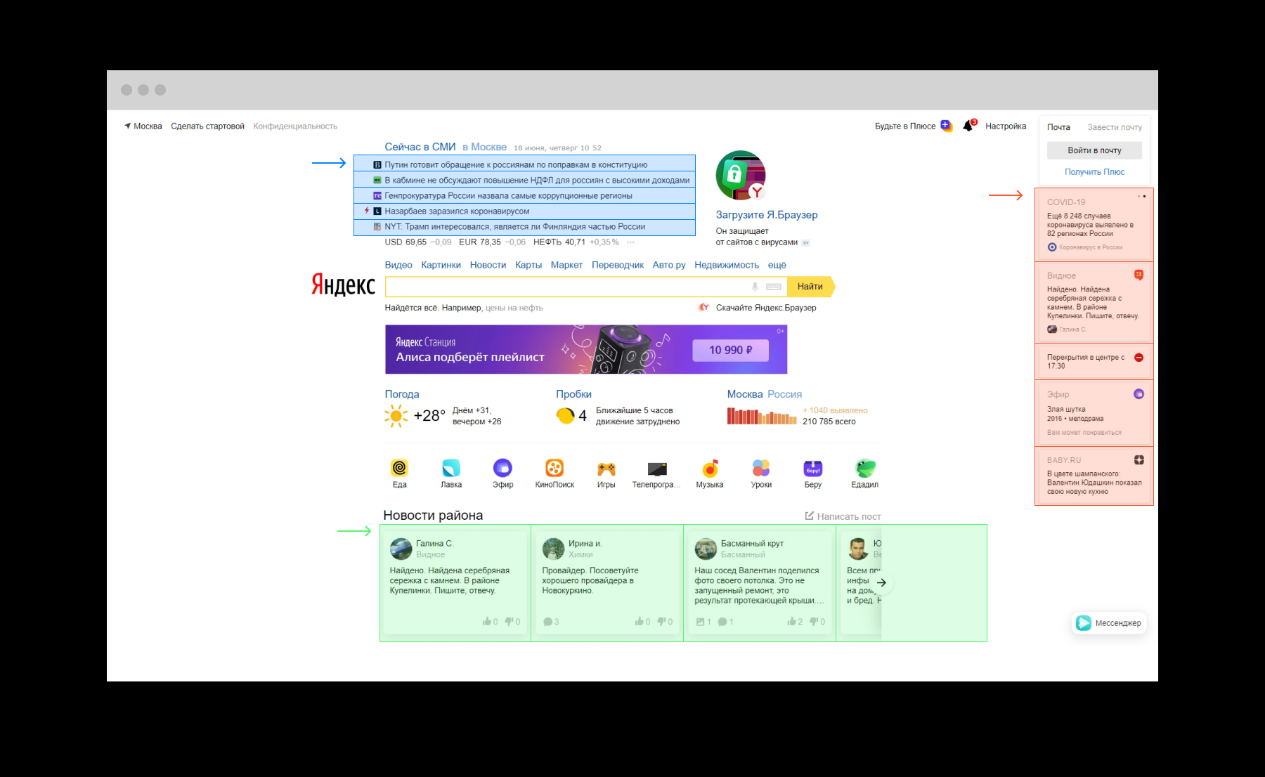
Другая важнейшая задача — динамическое обновление интерфейса. Классический (императивный) подход заключается для разработчика в описании последовательности изменений на HTML-странице при наступлении каких-либо событий (например: «при клике на кнопку сделать её цвет синим»). «Реакт» позволяет использовать декларативный подход, при котором разработчику достаточно описать все возможные состояния интерфейса в любой момент времени (например: «если кнопка нажата, она должна быть синей»). Такой подход показал себя более гибким и надёжным в больших проектах.

«Реакт» предлагает универсальные решения для обеих задач, описанных выше. При этом сама библиотека «Реакт» не является фреймворком в классическом понимании, так как не предоставляет разработчику заранее заданную структуру проекта. Однако, на сегодняшний день вокруг «Реакта» сформировалась целая экосистема, включающая в себя дополнительные библиотеки и плагины, и даже методология, определяющая общепринятые подходы к разработке (паттерны), — в том числе относительно структуры приложений.

**Компоненты**

В прошлом уроке мы обозначили основные типовые задачи, возникающие при разработке крупных веб-проектов. Одна из них — выделение компонентов интерфейса.

Компоненты — это функциональные блоки, из которых состоит интерфейс. Они могут существовать в приложении как в единственном экземпляре, так и повторяться: например, с разным содержимым. Крупные компоненты могут состоять из других, более мелких компонентов, которые в конечном счёте состоят из простых HTML-элементов.



На скриншоте выше выделены переиспользуемые компоненты. Блоки одного цвета имеют общую структуру, но разное содержимое. Мы можем описать общий для них HTML, CSS и JavaScript в одном компоненте, присвоить ему имя и впоследствии просто «призывать» его в нужном месте, меняя лишь его содержимое.

Разработчики «Реакта» пошли дальше: а что если сделать компонентами не только повторяющиеся, а вообще любые блоки в интерфейсе, отдельные страницы приложения и даже само приложение целиком?

Такой процесс называется декомпозицией и позволяет более абстрактно и на разных уровнях управлять составными частями приложения. Разделение ответственности — одна из самых важных задач в программировании, и от её успешного решения зависит надёжность и масштабируемость системы. Почти как у Макиавелли: «Разделяй и властвуй».

Сегодня большинство приложений на «Реакте» начинаются с главного компонента под названием App, который в свою очередь включает в себя все остальные компоненты. В итоге получается целая древовидная структура из вложенных друг в друга компонентов.

Но как понять, должен ли тот или иной элемент интерфейса стать компонентом? Когда стоит разбивать один компонент на несколько более мелких? Однозначного ответа на эти вопросы нет, и обычно решение остаётся на усмотрение разработчика. В целом здесь применимы общие правила: когда код компонента начинает разрастаться на несколько экранов в редакторе — время задуматься о декомпозиции.

**Декларативный подход**

В предыдущем уроке мы упомянули об отличиях императивного и декларативного подходов.

При императивном подходе разработчик описывает последовательность изменений, которые должны произойти с каждым элементом интерфейса при наступлении определённых событий. Этот подход мы использовали раньше: брали HTML-элемент, подписывались на какое-нибудь событие (например, click), задавали обработчик-колбэк, внутри которого уже производили манипуляции с этим или другими элементами. Например, задавали содержимое через innerHTML, добавляли CSS-класс через classList, меняли стили через объект style и т.п.:

Скопировать кодJAVASCRIPT

const element = document.querySelector('#myElement');

element.addEventListener('click', () => {

const element2 = document.querySelector('#myAnotherElement');

element2.classList.add('active');

const element3 = document.querySelector('#myText');

element3.innerHTML = 'It was clicked!';

});

Скопировать кодHTML

<div id="myElement">Click me!</div>

<div id="myAnotherElement">

<div id="myText">

Waiting for click...

</div>

</div>

Декларативный подход позволяет взглянуть на эту задачу под другим углом. Что, если при наступлении события мы будем просто менять значение какой-то переменной, а для всех остальных элементов заранее опишем различные состояния при всех возможных значениях этой переменной?

Скопировать кодJAVASCRIPT

const element = document.querySelector('#myElement');

let isClicked = false;

element.addEventListener('click', () => {

isClicked = true;

});

При этом предположим, что наш HTML мог бы зависеть от переменных в JavaScript:

Скопировать кодHTML

<div id="myElement">Click me!</div>

<div id="myAnotherElement" className={isClicked ? 'active' : ''}>

<div id="myText">

{isClicked ? 'It was clicked!' : 'Waiting for click...'}

</div>

</div>

Такой подход позволяет сократить JavaScript-обработчик до одной строки. Но ещё более важно то, что с его помощью удаётся переместить все варианты текстового содержимого и CSS-классов туда, где описаны сами элементы: прямо в HTML-код.

Таким образом мы разумно распределяем ответственность: JavaScript отвечает лишь за изменение текущего состояния приложения (той самой переменной), а HTML-элементы «сами» определяют (декларируют) то, как они будут выглядеть при разных его состояниях.

Однако, в обычном HTML-коде подобные приёмы невозможны.

Браузер не смог бы считать JavaScript-переменные, а фигурные скобки {} в нашем HTML он бы просто проигнорировал. Поэтому выше приведён пример не простого HTML, а так называемого JSX — специального языка шаблонов, проводящего «мост» между HTML и JavaScript.

Декларативный подход оказывается невероятно полезным для поддержания порядка в крупных веб-проектах, а JSX стал одной из главных технологий, используемых в экосистеме «Реакта», и несомненным её преимуществом. В следующем уроке мы наконец подключим «Реакт» и остановимся на нём подробнее.

**«Реакт» и JSX**

**Подключение «Реакта»**

Пришло время подключить «Реакт» и написать первый код с его использованием.

Вот упрощённая версия примера [с сайта «Реакта»](https://ru.reactjs.org/docs/getting-started.html):

Скопировать кодHTML

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<script src="https://unpkg.com/react@16/umd/react.development.js"></script>

<script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>

<script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>

</head>

<body>

<div id="root"></div>

<script type="text/jsx">

ReactDOM.render((

<h1>Hello, world!</h1>

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

</script>

</body>

</html>

Разберём подробно, что здесь происходит.

В первую очередь подключаем два скрипта: react и react-dom. Такое разделение существует, потому что основная библиотека react универсальна и предназначена не только для веб-проектов. Работая с браузерными приложениями, мы всегда будем подключать сразу две библиотеки: основную и ту, что отвечает за работу с DOM. На этапе разработки используются сборки с расширением .development.js: они содержат дополнительные инструменты для отладки кода на «Реакте».

JSX — «синтаксический сахар», поэтому следующим шагом мы подключаем «Бабель». Он занимается тем, что переводит JSX в стандартный синтаксис JavaScript, который может выполняться в любом браузере. В одном из следующих уроков мы разберём, что на самом деле находится «под капотом» JSX, а пока просто укажем «Бабелю», какой код нужно транспилировать, добавив атрибут type="text/jsx" у тега script.

Одна из основных функций в «Реакте» — ReactDOM.render. С неё начинается любой проект. Эта функция обрабатывает JSX и отрисовывает результат в DOM. Первым аргументом она принимает сам JSX-код, а вторым — указатель на DOM-элемент, в который необходимо вставить полученный результат.

Создайте файл index.html с кодом из примера выше и попробуйте запустить его в браузере.

**Динамический интерфейс**

Самое интересное начинается, когда в приложении появляется интерактивность. Вспомним пример с событием клика из предыдущего урока.

Здесь и далее мы будем приводить содержимое <script type="text/jsx"></script>.

Скопировать кодJAVASCRIPT

*// Переменная, отвечающая за состояние приложения*

let isClicked = false;

*// Функция, отвечающая за отрисовку интерфейса*

*// с учётом текущего состояния isClicked*

function renderAll() {

ReactDOM.render((

<div>

<div id="myElement">Click me!</div>

<div id="myAnotherElement" className={isClicked ? 'active' : ''}>

<div id="myText">

{isClicked ? 'It was clicked!' : 'Waiting for click...'}

</div>

</div>

</div>

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

}

*// Вызов первичной отрисовки при открытии страницы*

renderAll();

*// При наступлении события мы изменяем переменную состояния*

*// и заново вызываем отрисовку*

const element = document.querySelector('#myElement');

element.addEventListener('click', () => {

isClicked = true;

renderAll();

});

В этом примере мы добавили функцию renderAll, чтобы вызывать отрисовку «Реакта» не только при первом открытии страницы, но и повторно при любом изменении состояния приложения: например, после клика. Эта функция содержит в себе вызов функции ReactDOM.render, в которую передаётся JSX, зависящий от состояния переменной isClicked (а также, как и ранее, указатель на корневой HTML-элемент).

**JSX**

Напомним, что JSX представляет из себя «симбиоз» HTML и JavaScript. Чтобы подставить значение из JavaScript в HTML используются фигурные скобки {}. Внутри них можно писать любые выражения на JavaScript, возвращающие строки и числа, которые будут подставлены в HTML. Тернарный оператор ?: используется наиболее часто: он позволяет описать условную логику с двумя и более вариантами.

Поначалу такой код может показаться несколько непривычным: JavaScript переплетается с HTML-подобным JSX. Чтобы лучше организовать код, в следующем уроке мы начнём выделять компоненты с помощью «Реакта».

**Синтаксис JSX: основы**

Синтаксис JSX — декларативный, а значит, он позволяет писать гибкие шаблоны и описывать все возможные состояния интерфейса. Рассмотрим основные конструкции, которые используются в JSX.

**Скобки**

HTML-подобный код перемешивается с «настоящим» JavaScript. Это затрудняет чтение:

Скопировать кодJSX

ReactDOM.render(

<h2>

Что я здесь делаю?

</h2>,

document.querySelector('#root')

);

Чтобы разделить код и вёрстку, рекомендуется обрамлять JSX в круглые скобки:

Скопировать кодJSX

ReactDOM.render((

<h2>

Можно я тут останусь?

</h2>

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

**Условная логика**

Одна из самых частых задач в построении динамического интерфейса (да и вообще в программировании) — реализация условной логики. В прошлом уроке мы уже приводили пример, когда структура интерфейса меняется в зависимости от значения JavaScript-переменной. В JSX внутри фигурных скобок можно писать любые JavaScript-выражения, поэтому мы можем использовать операторы ?: и &&.

**Тернарный оператор** ?: — сокращённый аналог if/else:

Скопировать кодJSX

ReactDOM.render((

<div>

{isDaylight ? (

<h2>Добрый день!</h2>

) : (

<h2>Спокойной ночи!</h2>

)}

</div>

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

**Оператор** && — аналог if, но без else — используется, если нужно отобразить часть JSX разметки только при определённом условии, а в противном случае не отображать ничего вообще:

Скопировать кодJSX

ReactDOM.render((

<div>

{isLunchTime && <h2>Время обеда!</h2>}

</div>

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

Оба оператора могут содержать составные условия:

Скопировать кодJAVASCRIPT

ReactDOM.render((

<div>

{isThursday && wasRaining && <h2>Пора вернуть долг!</h2>}

</div>

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

Скопировать кодJSX

ReactDOM.render((

<div>

{(isFrost && isSun) ? (

<h2>День чудесный</h2>

) : (

<h2>День обычный</h2>

)}

</div>

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

**Классовая борьба**

JSX очень похож на HTML, но в нём есть некоторые отличия, которые придётся запомнить. Начнём с того, что для задания CSS-класса используется атрибут className вместо class:

Скопировать кодJSX

ReactDOM.render((

<div>

<div className="proletariat">

Шаг держи революционный!

Близок враг неугомонный!

Вперёд, вперёд, вперёд,

Рабочий народ!

</div>

<div className="bourgeoisie">

Ешь ананасы, рябчиков жуй,

день твой последний приходит, буржуй.

</div>

</div>

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

**Стили**

Одна из «фич» JSX — возможность задавать значения style в виде JavaScript-объектов. При этом в названиях CSS-свойств вместо kebab-case используется camelCase:

Скопировать кодJSX

const cssRules = {

width: 6792,

height: 6752,

borderRadius: '50%',

background: '#934838',

color: 'black',

};

ReactDOM.render((

<div style={cssRules}>Какая я планета?</div>

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

Значение можно задать сразу (без промежуточной переменной). Для этого используются двойные фигурные скобки {{...}} — внешние означают подстановку значения, а внутренние относятся к объекту, описывающему набор стилей:

Скопировать кодJSX

ReactDOM.render((

<div style={{

width: 3475,

height: 3472,

borderRadius: '50%',

background: '#d0d5d2',

color: '#444444',

}}>

Я тоже хочу быть планетой!

</div>

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

Ещё один плюс: можно не указывать размерность для значений в пикселях — «Реакт» сделает это сам.

**Фрагменты**

Иногда в JSX используется «безымянный» тег <>...</>:

Скопировать кодJSX

ReactDOM.render((

<>

<div id="myElement">Click me!</div>

<div id="myAnotherElement">It was clicked!</div>

</>

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

Это так называемый «фрагмент» в «Реакте».

Дело в том, что блок JSX-кода по правилам должен содержать только один элемент верхнего уровня. В примере выше используются сразу два элемента, находящиеся рядом: #myElement и #myAnotherElement. Чтобы JSX-код правильно работал, мы «оборачиваем» эти два элемента во фрагмент.

Фрагменты не обладают дополнительной функциональностью и могут восприниматься как своеобразные «обрамляющие скобки» в JSX-коде. Вместо них можно было бы использовать обычный <div>, но это создаст лишние элементы в DOM — напрасная трата ресурсов и может вызывать дополнительные сложности с вёрсткой.

**Самозакрывающиеся теги**

Любой открывающий тег должен иметь свою пару — закрывающий тег. Если у элемента нет внутреннего содержимого, то тег должен быть самозакрывающимся. Для этого ставят слеш / перед его закрывающей скобкой:

Скопировать кодJSX

<img src="logo/png" />

**Синтаксис JSX: списки и события**

**Списки**

Существует ещё одна частая задача — описание разметки для элементов списка. Под списком в данном случае подразумеваются любые однотипные данные, повторяющиеся более одного раза: меню сайта, список пользователей, галерея изображений и т.п.

Например, в ленте сообщений чата, написанного на «Реакте», нужен только массив с данными и код JSX, который описывает разметку одного сообщения, а затем использует цикл, чтобы вывести полный результат.

Предположим, что с сервера мы получаем такой массив с данными о комментариях:

Скопировать кодJAVASCRIPT

const comments = [{

id: 1,

author: 'Лиза',

text: 'Все уже сдали деньги на субботник?',

}, {

id: 2,

author: 'Джеймс',

text: 'Никто не видел мой сэндвич?',

}, {

id: 3,

author: 'Грег',

text: 'Продам мопед',

}];

Опишем разметку стандартного сообщения:

Скопировать кодJSX

<div>

<h3>{message.author}</h3>

<div>{message.text}</div>

</div>

Но как передать нужные значения определённого объекта массива comments в разметку? Для этого и нужен цикл.

Циклы в JSX реализуются методом map массива. При этом каждому внешнему JSX-элементу, создаваемому в цикле, необходимо задать уникальный ключ, для этого используют атрибут key.

Скопировать кодJSX

ReactDOM.render((

<div>

<h2>Сообщения</h2>

{comments.map((message, i) => (

// Важный атрибут: key

<div key={message.id}>

<h3>{message.author}</h3>

<div>{message.text}</div>

</div>

))}

</div>

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

Правила «Реакта» всегда требуют задавать ключи при работе со списками. Если этого не сделать, в консоли появится соответствующая ошибка. Они нужны для корректной работы движка, и уже в следующем спринте мы расскажем почему — когда будем разбирать внутреннее устройство «Реакта».

Здесь мы используем уже имеющийся у нас уникальный идентификатор — message.id. Использовать идентификаторы в качестве ключа очень удобно, так как обычно они есть у всех данных, которые возвращает API. Если такого идентификатора в данных нет, можно использовать индексы элементов (второй аргумент метода map), но только если есть абсолютная уверенность в том, что порядок элементов никогда не меняется:

Скопировать кодJSX

ReactDOM.render((

<div>

<h2>Сообщения</h2>

{comments.map((message, i) => (

// Важный атрибут: key

<div key={i}>

<h3>{message.author}</h3>

<div>{message.text}</div>

</div>

))}

</div>

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

Подробнее изучим списки уже в следующем спринте, но если вы хотите скорее ознакомиться с принципами их работы, вот [статья из документации](https://ru.reactjs.org/docs/lists-and-keys.html).

**Обработчики событий**

Вы уже умеете добавлять обработчики событий с помощью addEventListener. Кроме того, в обычном HTML есть специальные атрибуты событий onclick, onmouseenter и другие.

В JSX тоже есть такие атрибуты, только выглядят они немного по-другому:

Скопировать кодJSX

function handleClick() {

console.log('Не дави на меня!');

}

function handleMouseEnter() {

console.log('Ты мне солнце заслонил!');

}

function handleMouseLeave() {

console.log('Ну ты чего, нормально же общались!');

}

ReactDOM.render((

<button

onClick={handleClick}

onMouseEnter={handleMouseEnter}

onMouseLeave={handleMouseLeave}

>

Поиграй со мной!

</button>

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

В примере выше значения атрибутов onClick, onMouseEnter и onMouseLeave указывают на JavaScript-функции, которые должны находиться в области видимости.

Полный список поддерживаемых событий можно посмотреть в [документации «Реакта»](https://ru.reactjs.org/docs/events.html#supported-events).

const list = [{

name: 'Хлеб',

quantity: '1 батон',

}, {

name: 'Ром',

quantity: '3 бутылки',

}, {

name: 'Кока-кола',

quantity: '3 бутылки',

}, {

name: 'Туалетная бумага',

quantity: 'Вся, что есть',

}];

ReactDOM.render((

<ul>

{list.map((item, i) =>

<li key={i}>

<b>{item.name}</b>

<i>{item.quantity}</i>

</li>)}

</ul>

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

**Пример с кнопками**

function handleClick() {

function handleKeyUp(e) {

document.getElementById('title').innerText = e.target.value;

}

function handleClick(e) {

document.getElementById('title').innerText += ' ' + e.target.textContent;

}

function handleMouseEnter(e) {

e.target.classList.add('hover');

}

function handleMouseLeave(e) {

e.target.classList.remove('hover');

}

function handleMouseDown(e) {

e.target.classList.add('active');

}

function handleMouseUp(e) {

e.target.classList.remove('active');

}

ReactDOM.render((

<div>

<input type="text" onKeyUp={handleKeyUp} />

<button

onClick={handleClick}

onMouseEnter={handleMouseEnter}

onMouseLeave={handleMouseLeave}

onMouseDown={handleMouseDown}

onMouseUp={handleMouseUp}>🤩</button>

<button

onClick={handleClick}

onMouseEnter={handleMouseEnter}

onMouseLeave={handleMouseLeave}

onMouseDown={handleMouseDown}

onMouseUp={handleMouseUp}>🤯</button>

<button

onClick={handleClick}

onMouseEnter={handleMouseEnter}

onMouseLeave={handleMouseLeave}

onMouseDown={handleMouseDown}

onMouseUp={handleMouseUp}>🤪</button>

</div>

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

# Функциональные компоненты

### Компоненты в JSX

Разделение кода на компоненты — одна из главных возможностей, которую предоставляет «Реакт».

Ранее мы приводили пример компонента: аватар с именем пользователя. Такой блок может встречаться в приложении много раз и отображать информацию о разных пользователях.

У каждого компонента должно быть своё имя, и в React принято называть компоненты с заглавной буквы. Предположим, что мы дали нашему первому компоненту имя User.

Вот как будет выглядеть JSX-код, использующий этот компонент:

Скопировать кодJSX

ReactDOM.render((

<>

<h2>Мои воображаемые друзья:</h2>

<User id="1" name="Gregory" />

<User id="2" name="James" />

<User id="3" name="Allison" />

</>

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

Компоненты, как и обычные HTML-элементы, могут иметь атрибуты. Атрибуты компонентов в React называются «пропсы» (англ. props, сокращение от properties, «свойства»).

### Создание функционального компонента

Как же создать компонент в «Реакте»? Для этого существует несколько способов, и один из них… обыкновенная функция. Действительно, компоненты в «Реакте» можно создавать в виде обычной JavaScript-функции. Такая функция-компонент принимает единственный аргумент — объект props — и возвращает структуру компонента в JSX:

Скопировать кодJSX

*// Компонент User*

function User(props) {

return (

<p>

<img src={`https://code.s3.yandex.net/web-code/react/${props.id}.png`} width="75" />

<br /><b>{props.name}</b>

</p>

);

}

*// Основной код приложения*

ReactDOM.render((

<>

<h2>Мои воображаемые друзья:</h2>

<User id="1" name="Gregory" />

<User id="2" name="James" />

<User id="3" name="Allison" />

</>

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

Этот пример уже работает — его можно запустить в браузере.

При отрисовке страницы компонент User будет вызван трижды — с разными пропсами, заданными в основном коде приложения.

Каждый раз, когда отрисовка вызывается повторно (в примере из предыдущего урока это происходило после клика), каждая функция-компонент будет вызвана снова. В примере выше это не принципиально, так как функция User всегда возвращает одно и то же (в зависимости от пропсов), однако более сложные компоненты могут менять своё содержимое между отрисовками, а также могут меняться передаваемые им «сверху» пропсы.

**Классовые компоненты**

В прошлом уроке мы научились создавать функциональные компоненты. Есть в «Реакте» и альтернативный способ создания компонентов: с помощью ООП-классов. Исторически он появился раньше, и большинство существующих проектов используют именно его.

**Создание классового компонента**

Для этого используется синтаксис class MyComponent extends React.Component, где MyComponent — имя создаваемого компонента. Это наследование от встроенного в «Реакт» класса Component, который предоставляет дополнительную функциональность для компонентов.

**Перепишем** компонент User с использованием класса:

Скопировать кодJSX

*// Функциональный компонент User*

function User(props) {

return (

<p>

<img src={`https://code.s3.yandex.net/web-code/react/${props.id}.png`} width="75" />

<br /><b>{props.name}</b>

</p>

);

}

Скопировать кодJSX

*// Классовый компонент User*

class User extends React.Component {

render() {

return (

<p>

<img src={`https://code.s3.yandex.net/web-code/react/${this.props.id}.png`} width="75" />

<br /><b>{this.props.name}</b>

</p>

);

}

}

Как видно, изменилось не так много:

1. Появился класс User, который наследуется от встроенного в «Реакт» класса React.Component.
2. Содержимое компонента-функции (function User) переместилось внутрь метода render, принадлежащего этому новому классу.
3. props — это теперь не аргумент функции, а поле экземпляра класса, доступ к которому осуществляется с помощью ключевого слова this.

Что же даёт подобное изменение? Дело в том, что у встроенного класса React.Component есть свои свойства и методы для решения специфичных задач — подобно встроенным методам в обычном JavaScript. С помощью наследования разработчик получает к ним доступ.

**Внутреннее состояние компонента**

Часто внутри компонента необходимо хранить данные, которые меняются в течение «жизни» этого компонента. Эти данные не приходят извне в виде пропсов, а создаются и меняются прямо внутри самого компонента.

Пример таких данных для нашего компонента — счётчик рейтинга пользователя. Допустим, мы хотим, чтобы у наших пользователей была возможность голосовать за репутацию, ставя лайки или дизлайки друг другу.

Для этого будем использовать поле state и метод setState экземпляра классового компонента.

Скопировать кодJSX

*// Классовый компонент User*

class User extends React.Component {

constructor(props) {

super(props);

*// Начальные значения внутреннего состояния*

this.state = {

rating: 0,

};

}

*/\**

*\* Обработчики событий: изменяют внутреннее состояние*

*\*/*

handleLike = () => {

this.setState({ rating: 1 });

};

handleDislike = () => {

this.setState({ rating: -1 });

};

*// JSX-структура компонента*

render() {

return (

<p>

<img src={`https://code.s3.yandex.net/web-code/react/${this.props.id}.png`} width="75" />

<br /><b>{this.props.name}</b>

<div className="rating">

<button onClick={this.handleLike}>👍</button>

{this.state.rating}

<button onClick={this.handleDislike}>👎</button>

</div>

</p>

);

}

}

*// Основной код приложения*

ReactDOM.render((

<>

<h2>Мои воображаемые друзья:</h2>

<User id="1" name="Gregory" />

<User id="2" name="James" />

<User id="3" name="Allison" />

</>

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

Попробуйте запустить этот код в браузере, и разберём по порядку, что здесь происходит.

**Во-первых,** мы добавили метод constructor. Именно этот метод вызывается при самой первой отрисовке, когда «Реакт» создаёт экземпляр нашего компонента:

Скопировать кодJSX

constructor(props) {

super(props);

*// Начальные значения внутреннего состояния*

this.state = {

rating: 0,

};

}

В этот метод попадают первые значения пропсов, а наша задача прокинуть их выше — в конструктор родительского класса — с помощью ключевого слова super. Именно там из них будет создан объект this.props.

Далее мы объявляем объект внутреннего состояния this.state, а также задаём для него начальные значения. Единственное поле, которое мы в него добавили — rating — отвечает за текущий рейтинг пользователя. Его начальное значение (при открытии страницы) равно 0.

**Во-вторых,** мы объявили два метода-обработчика событий кликов для кнопок 👍 и 👎. Внутри себя они вызывают метод this.setState из родительского класса, которой выполняет две функции: изменяет значение состояния this.state и оповещает движок «Реакта» о том, что внутреннее состояние компонента изменилось, а следовательно его необходимо перерисовать:

Скопировать кодJSX

*/\**

*\* Обработчики событий: изменяют внутреннее состояние*

*\*/*

handleLike = () => {

this.setState({ rating: 1 });

};

handleDislike = () => {

this.setState({ rating: -1 });

};

В результате при клике на одну из этих кнопок «Реакт» повторно вызовет метод render (уже с обновлённым значением this.state) и, определив изменения в JSX, внесёт необходимые обновления в DOM. Помните, раньше мы вручную вызывали renderAll при клике, чтобы вызвать перерисовку «Реактом»? Метод setState делает это за нас.

Следует обратить внимание на то, что методы-обработчики handleLike и handleDislike объявлены не в виде обычных методов класса (как, например, метод constructor), а в виде полей, которым присваиваются стрелочные функции. Это позволяет им не потерять контекст this при использовании в качестве обработчиков событий.

**В-третьих,** мы добавили новый элемент .rating в JSX, содержащий текущее значение счётчика и две кнопки для его изменения:

Скопировать кодJSX

<div className="rating">

<button onClick={this.handleLike}>👍</button>

{this.state.rating}

<button onClick={this.handleDislike}>👎</button>

</div>

**В качестве бонуса** можно воспользоваться изящностью JSX и слегка улучшить наш компонент: запретить нажимать на одну и ту же кнопку несколько раз подряд. В нашем приложении это ни к чему не приводит, поэтому с точки зрения интерфейса лучше, если после клика кнопка будет становиться неактивной. Это довольно легко сделать с помощью динамического значения для атрибута disabled:

Скопировать кодJSX

<div className="rating">

<button onClick={this.handleLike} disabled={this.state.rating > 0}>👍</button>

{this.state.rating}

<button onClick={this.handleDislike} disabled={this.state.rating < 0}>👎</button>

</div>

Пример из тренажера

class Switch extends React.Component {

constructor(props) {

super(props);

}

render() {

const className = `switch ${this.props.color} ${this.props.isActive ? 'on' : 'off'}`;

return (

<div className={className}>

<button className="img" />

<h3>{this.props.title}</h3>

</div>

);

}

}

ReactDOM.render((

<>

<Switch title="Счастье" color="blue" isActive={false} />

</>

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

ДАЛЕЕ, добавим возможность изменять значение isActive

class Switch extends React.Component {

constructor(props) {

super(props)

this.state = {

isActive: this.props.isActive,

}

handleClick = () => {

this.setState({isActive: !this.state.isActive })

};

render() {

// Используем JavaScript-шаблон для склейки значения атрибута

const className = `switch ${this.props.color} ${this.state.isActive ? 'on' : 'off'}`;

return (

<div className={className}>

<button className="img" onClick={this.handleClick} />

<h3>{this.props.title}</h3>

</div>

);

}

}

ReactDOM.render((

<Switch title="Счастье" color="blue" isActive={false} />

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

# Жизненный цикл классовых компонентов

Продолжим изучать встроенные методы класса React.Component. В этом уроке расскажем о методах, которые позволяют управлять «жизненным циклом» (англ. lifecycle) классовых компонентов и тем самым лучше контролировать их поведение. Например, выполнить определённый код в заранее заданный момент.

Каждый компонент в течение своего жизненного цикла может быть монтирован (англ. mount), обновлён (англ. update) и размонтирован (англ. unmount).

**Монтирование** происходит, когда HTML-дерево компонента вставляется движком «Реакта» в DOM.

**Обновление** происходит в одном из трёх случаев:

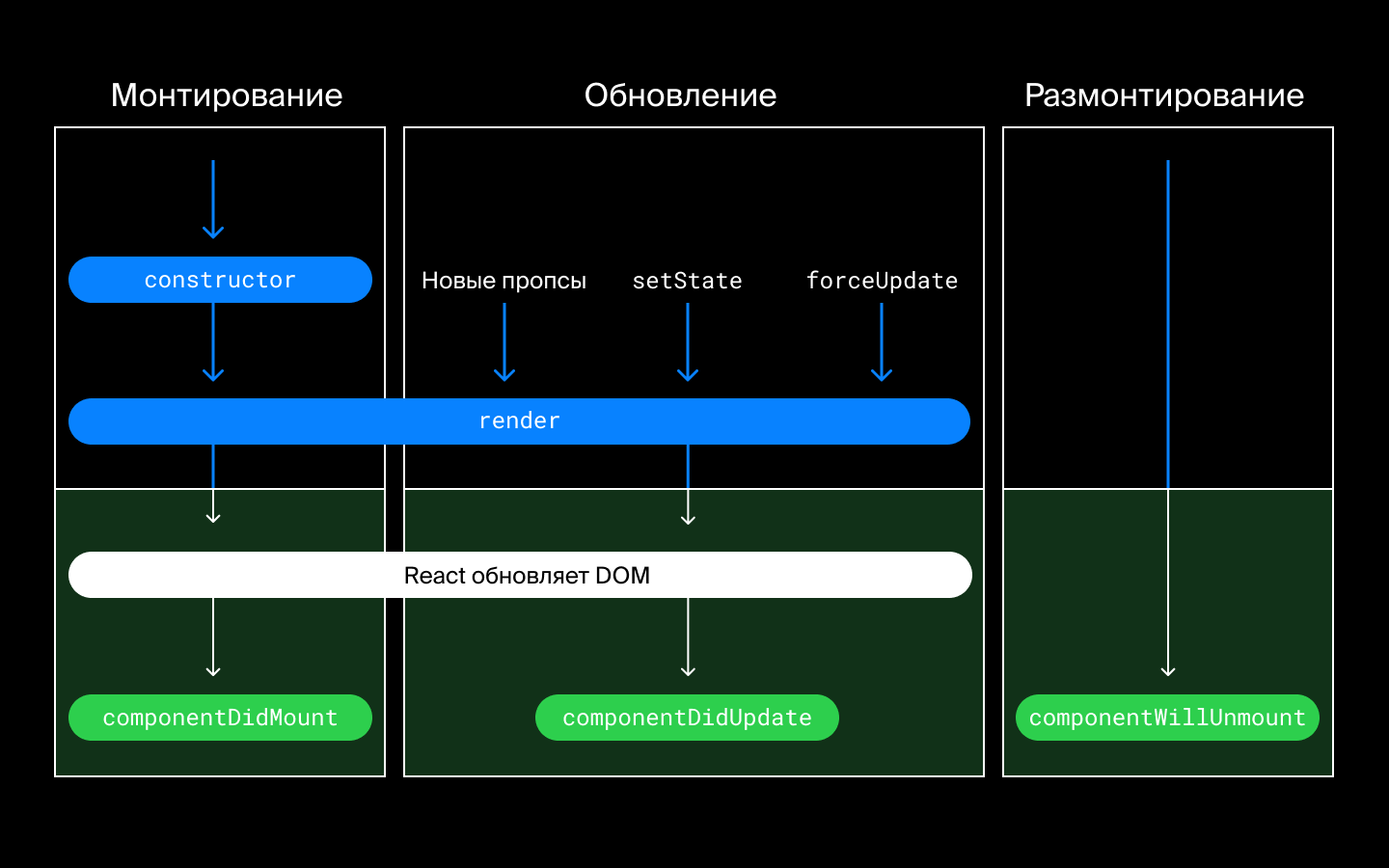
* если произошёл рендер родительского компонента,
* если изменилось внутреннее состояние в результате вызова this.setState,
* если обновление было инициировано вызовом встроенного метода this.forceUpdate.

**Размонтирование** происходит, когда HTML-дерево компонента удаляется из DOM.

Каждому из этих событий соответствует один из трёх методов жизненного цикла: componentDidMount, componentDidUpdate и componentWillUnmount.

Как и метод render, это методы обратного вызова: в отличие от setState вы никогда не будете вызывать их вручную, однако вы будете описывать их внутри своих компонентов, чтобы в нужный момент они были вызваны изнутри самого движка «Реакта» — как колбэки.

Наглядная диаграмма жизненного цикла ([интерактивный вариант тут](https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/)):



Чаще всего методы жизненного цикла используются для описания так называемых «побочных эффектов» (англ. side effects). В процессе «жизнедеятельности» компонент помимо непосредственно отрисовки может взаимодействовать с различными API, подписываться на глобальные события, обращаться к серверу за данными и т.д. Именно это принято называть его побочными эффектами или просто эффектами.

Предположим, что мы хотим сделать приложение, в котором будет доступен режим кастомизированного курсора: с помощью чекбокса на страницу будет добавляться компонент с картинкой, следующей за движениями мыши.

Нам понадобятся два компонента: корневой компонент App с чекбоксом и находящийся внутри него компонент NeonCursor, который будет монтироваться, только когда чекбокс отмечен.

Сперва опишем компонент App.

Скопировать кодJSX

*// Корневой компонент приложения*

class App extends React.Component {

constructor(props) {

super(props);

this.state = { isCustomCursor: false };

}

handleChange = () => {

this.setState({

isCustomCursor: !this.state.isCustomCursor,

});

};

render() {

return (

<>

<label>

<input type="checkbox" onChange={this.handleChange} />

— Включить неоновый курсор

</label>

{this.state.isCustomCursor && <NeonCursor />}

</>

);

}

}

Самое интересное происходит в этой строчке:

Скопировать кодJSX

{this.state.isCustomCursor && <NeonCursor />}

Здесь мы используем оператор && для создания условной логики: компонент NeonCursor будет включён в структуру JSX только тогда, когда условие перед оператором (this.state.isCustomCursor) истинно. Изначально значение этого поля равно false, но когда пользователь установит галочку чекбокса, будет вызван метод this.setState, который его изменит. Это приведёт к монтированию компонента NeonCursor. Если пользователь снимет галочку, условие снова изменится и компонент будет удалён из дерева JSX и, соответственно, из DOM.

Теперь опишем сам компонент NeonCursor.

Скопировать кодJSX

*// Компонент, отвечающий за кастомизированный курсор*

class NeonCursor extends React.Component {

constructor(props) {

super(props);

this.state = { top: 0, left: 0 };

}

*// Метод будет вызван сразу после монтирования: создаём эффекты*

componentDidMount() {

document.addEventListener('mousemove', this.handleMouseMove);

document.documentElement.classList.add('no-cursor');

}

*// Метод будет вызван непосредственно перед размонтированием: удаляем эффекты*

componentWillUnmount() {

document.documentElement.classList.remove('no-cursor');

document.removeEventListener('mousemove', this.handleMouseMove);

}

handleMouseMove = (event) => {

this.setState({

top: event.pageY,

left: event.pageX,

});

};

render() {

return (

<img

src="./cursor.png"

width="30"

style={{

position: 'absolute',

top: this.state.top,

left: this.state.left,

pointerEvents: 'none',

}}

/>

);

}

}

В этом компоненте описаны два метода его жизненного цикла. Метод componentDidMount будет вызван как только компонент монтируется в DOM: в нём мы подписываемся на событие mousemove документа, а также добавляем документу CSS-класс, отключающий стандартный курсор на всех элементах страницы. Кстати, этот класс выглядит так:

Скопировать кодCSS

*/\* Элемент с классом no-cursor и все вложенные в него элементы \*/*

.no-cursor, .no-cursor \* { cursor: none; }

Когда пользователь снимет галочку чекбокса, наш компонент размонтируется, и нам очень важно «подчистить» (англ. cleanup) произведённые им эффекты: удалить слушатель mousemove и снять с документа CSS-класс, чтобы снова включился стандартный курсор. Эти действия мы производим в методе componentWillUnmount.

В этом методе команды вызываются в обратном порядке по отношению к методу componentDidMount — это стандартная практика называется принцип «шнурков и ботинок». Мы всегда сначала надеваем ботинки, а затем завязываем шнурки, а когда собираемся их снять, всё происходит наоборот.

Благодаря подписке на событие mousemove каждое движение курсора вызывает метод handleMouseMove, который обновляет внутреннее состояние компонента с помощью this.setState, что влечёт за собой вызов render с новыми значениями и приводит к обновлению DOM: элемент <img> получает новые координаты и перемещается по экрану.

Вся логика готова. Добавим компонент приложения на страницу:

Скопировать кодJSX

ReactDOM.render(<App />, document.querySelector('#root'));

Неоновый курсор к вашим услугам.

**Пример с размером окна**

кодJSX

class App extends React.Component {

constructor(props) {

super(props);

this.state = { shouldShowScale: false };

}

handleChange = () => {

this.setState({

shouldShowScale: !this.state.shouldShowScale,

});

};

render() {

return (

<>

<label>

<input type="checkbox" onChange={this.handleChange} />

Показать измеритель экрана

</label>

{this.state.shouldShowScale && <Scale />}

</>

);

}

}

class Scale extends React.Component {

constructor() {

super();

this.state = {

windowSize: [window.innerWidth, window.innerHeight],

};

}

componentDidMount() {

window.addEventListener('resize', this.handleWindowResize);

}

componentWillUnmount() {

window.removeEventListener('resize', this.handleWindowResize);

}

handleWindowResize = () => {

this.setState({

windowSize: [window.innerWidth, window.innerHeight],

});

};

render() {

return (

<div className="overlay">

<div className="modal">

<h3>Размер окна:</h3>

<h2>{this.state.windowSize[0]} x {this.state.windowSize[1]}</h2>

</div>

</div>

);

}

}

ReactDOM.render((

<App />

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

# Virtual DOM

В предыдущих примерах мы всё время приводили JSX в качестве основного «строительного материала» для «Реакта». На самом деле JSX — лишь «синтаксический сахар», за которым скрываются вызовы одной единственной, но очень важной функции в «Реакте». Это функция React.createElement, и с помощью неё в «Реакте» создаются виртуальные узлы (англ. virtual nodes).

В самом начале мы подключали «Бабель» — как раз, чтобы трансформировать JSX во множество вызовов React.createElement. Кстати, в [онлайн-версии «Бабеля»](https://babeljs.io/en/repl#?browsers=defaults%2C%20not%20ie%2011%2C%20not%20ie_mob%2011&build=&builtIns=false&spec=false&loose=false&code_lz=DwEwlgbgUABDDGAbAhgZ1QOWQWwKYF4AiABzADsBrGAKwFdsAjAe0NhibIGFEx4L8A3gAoAlDHwA-BB1RNEuAHSImAcyEByJLwoBCdSIC-UCW0ACIID4QQOwg5wEwggaRBAnCAxgDCYEEQQNwg5wLwgwAPSuUP7gECZAA&debug=false&forceAllTransforms=false&shippedProposals=false&circleciRepo=&evaluate=false&fileSize=false&timeTravel=false&sourceType=script&lineWrap=true&presets=env%2Creact%2Cenv&prettier=false&targets=&version=7.9.6&externalPlugins=) можно наглядно увидеть, как происходят такие трансформации — просто введите исходный код, и результат сразу появится в правом окне.

Скопировать кодJSX

// JSX-код

<div

className="pink jumbo"

onClick={() => console.log('click!')}

>

розовый <b>слон</b>

</div>

Скопировать кодJSX

*// Код после транспиляции*

React.createElement(

*// Имя тега или компонента*

'div',

*// Атрибуты или пропсы*

{

className: 'pink jumbo',

onClick: () => console.log('click!')

},

*// Первый дочерний узел (строка)*

'розовый ',

*// Второй дочерний узел (ещё один элемент)*

React.createElement(

'b',

null,

'слон'

)

);

Каждый элемент в JSX трансформируется в один вызов React.createElement. Эта функция принимает два основных аргумента: имя тега или указатель на компонент, а также объект, содержащий все атрибуты и пропсы. Последующие аргументы (их число не ограничено) перечисляют все непосредственные дочерние узлы.

Чаще всего среди дочерних узлов находятся:

* другие вызовы React.createElement;
* строки текста;
* JavaScript-выражения, возвращающие что-то из вышеперечисленного или «пустой» узел (это может быть null, undefined или false).

В результате вызова функции React.createElement внутри движка «Реакта» создаётся «Реакт»-элемент — главная молекула «организма „Реакта“»: JavaScript-объект, содержащий данные, которые были переданы в функцию. В упрощённом виде структура этого элемента выглядит так:

Скопировать кодJAVASCRIPT

{

type: 'div',

props: {

className: 'pink jumbo',

onClick: () => console.log('click!'),

children: [

'Розовый',

{

type: 'b',

children: ['слон']

}

],

},

}

Обратите внимание, что объект children — часть объекта props. В дальнейшем нам это пригодится, когда мы будем работать с компонентами, в которых часть содержимого не только описывается внутри самих компонентов, но и может быть задана извне.

В конечном итоге формируется древовидная структура объектов, где один корневой объект ссылается на несколько дочерних через своё поле children и так далее. Именно эта структура затем передаётся в функцию ReactDOM.render, где происходит самое интересное.

«Реакт» использует эту структуру данных для двух целей.

**Во-первых,** на её основе строятся и вставляются на страницу реальные DOM-элементы (с помощью document.createElement и appendChild), а также вызывается рендер компонентов. В результате рендера компонентов эта структура, а с ней и реальный DOM, «обрастают» дополнительными ветвями, структура которых была описана в компонентах.

**Во-вторых,** «Реакт» использует эти данные для хранения виртуального DOM (англ. Virtual DOM) — виртуальной копии текущей структуры DOM. При последующих рендерах обновлённая структура поэлементно сравнивается с последней сохранённой копией, что позволяет «Реакту» определить минимальное необходимое количество изменений и применить эти изменения в реальном DOM.

Это необходимо, потому что почти любые операции с DOM — узкое место в JavaScript. Алгоритм «Реакта», который называется “reconciliation” (англ. «согласование»), сводит количество этих операций к минимуму. Сравнение двух даже сложных JavaScript-объектов во много раз быстрее, чем аналогичные операции над DOM-деревом. Это позволяет «Реакту» сохранить высокую скорость работы в приложениях с большим количеством меняющихся данных.

# Заключение

Вы закончили первую тему про «Реакт», поздравляем!

В этой теме вы узнали о двух главных преимуществах «Реакта»: компонентах и декларативном подходе; научились работать с новым языком JSX; узнали, как устроены и чем отличаются функциональные и классовые компоненты и начали знакомиться с Virtual DOM — тем, что находится в «Реакте» под капотом.

Многие боятся изучать «Реакт», потому что заранее считают его чем-то очень сложным. На самом же деле цель «Реакта» — наоборот, упростить разработку, сделать её более удобной и быстрой, а код более надежным. Многие вещи, которые потребовали бы сложных программных решений и большого количества кода, в «Реакте» доступны «из коробки». В проектной работе вы научитесь их применять.

Впереди вас ждёт ещё много интересного, желаем успеха!

В следующей теме вы познакомитесь с удобными инструментами и приёмами разработки и узнаете, как с их помощью быстро собрать полноценный проект на «Реакте».