# Создание нового проекта

Откроем командную строку и перейдём в директорию, в которой хотим создать проект. Для создания нового CRA-проекта используют команду npx create-react-app название\_проекта. Название проектов обычно пишут латиницей. Наш проект называется calorie-zen, поэтому введём команду npx create-react-app calorie-zen

Перейдите в директорию проекта командой cd calorie-zen. Находясь в ней, введите ещё одну команду — npm start.

# Структура проекта

└── src/

├── components/

│ ├── App.js

│ ├── App.css

│ ├── App.test.js

│ └── landing/

│ ├── Bed.js

│ ├── Bed.css

│ ├── Header.js

│ ├── Header.css

│ └── <!-- остальные компоненты, файлы для тестирования -->

├── images/

│ └── bed.png

└── <!-- дополнительные файлы src/ -->

└── src/

├── components

├── images

├── fonts/

│ └── Robotocondensed.woff

└── <!-- остальные файлы в директории src -->

# Добавление React Router в проект

Чтобы установить библиотеку React Router в проект на «Реакте», откройте главную директорию проекта и введите npm i --save react-router-dom. Эта версия React Router предназначена для маршрутизации в браузерных приложениях. (npm install react-router-dom)

Чтобы использовать React Router в проекте, нужно импортировать необходимые компоненты. Если хотите добавить Route в приложение, убедитесь, что он обёрнут в компонент BrowserRouter. Компонент Route устанавливает связь между путём, который указан в пропсе path, и URL-адресом, который в данный момент используется браузером. При каждом обращении к этому URL будет отображаться компонент внутри маршрута.

Импорты

import { BrowserRouter } from 'react-router-dom';

import { Route } from 'react-router-dom';

Пример App.js

import React from 'react';

import './App.css';

import Header from './Header';

import Diary from './Diary';

import { BrowserRouter } from 'react-router-dom';

import { Route } from 'react-router-dom';

function App() {

return (

<BrowserRouter>

<Header />

<Route path="/">

<Diary />

</Route>

</BrowserRouter>

);

}

export default App;

# Режим 'use strict'

**В модулях строгий режим включён по умолчанию**

Чтобы его включить, напишите в начале файла 'use strict':

Скопировать кодJAVASCRIPT

'use strict';

Строгий режим задаёт this равным undefined для функций, вызванных по имени.

Если переменная не объявлена, в неё нельзя записать значение.

Строгий режим можно включить для отдельной функции. Для этого следует написать 'use strict'; в начале тела этой функции

Метод call вызывает функцию, явно указывая её контекст.

const user = {

username: 'Peter',

auth() {

console.log(`${this.username} has logged in`);

}

};

const adminAuth = user.auth;

adminAuth.call(user); *// Peter has logged in*

Метод call вызывает функцию, он принимает сколько угодно аргументов:

* первый аргумент — это контекст, то есть объект, который будет записан в this. В примере это объект user;

все аргументы, начиная со второго, это параметры самой функции, которую мы вызываем

Этих аргументов может быть сколько угодно:

Скопировать кодJAVASCRIPT

const user = {

username: 'Peter',

auth() {

console.log(arguments); *// выведем аргументы в консоль*

}

};

const adminAuth = user.auth;

adminAuth.call(user, 1, 2, 3, 4, 5); *// Arguments(5) [1, 2, 3, 4, 5]*

Метод apply делает всё то же самое, что и call, то есть явно указывает значение this, но по-другому принимает аргументы. Методу apply первым аргументом передают контекст, а параметры функции — вторым аргументом в массиве.

displayDetails.apply(car, ['Hello', 'Matt']);

Если в теле функции не используется ключевое слово this, можно не задумываться, делать её стрелочной или нет — делайте, как хотите. Например, удобно использовать стрелочные функции в колбэке, потому что это короче:

Скопировать кодJAVASCRIPT

const numbers = [2, 3, 5];

const doubledNumbers = numbers.map(number => number \* 2);

console.log(doubledNumbers); *// 4, 6, 10*

Но если this внутри функции нужен — стоит задуматься, должна она быть стрелочной или нет.

1. Если функция вызвана с оператором new, используется только что созданный объект.
2. Если функция вызвана одним из методов call, apply или после привязки методом bind, в this оказывается переданный в метод объект.
3. Если вызван метод объекта, тогда this — это объект, на котором этот метод вызван
4. Если ни одно правило не сработало, то идёт в ход привязка по умолчанию: в строгом режиме — undefined, иначе — глобальный объект window.

# Добавление изображений

Изображение bed.png нужно импортировать в Bed.js. Сделаем это внутри компонента:

Скопировать кодJSX

import React from 'react';

import bedPath from '../../images/bed.png'; *// Путь к изображению внутри сборки*

function Bed() {

return (

<img alt='A comfortable bed' src={bedPath}/>

);

}

export default Bed;

# Шрифты

Для этого удобно использовать библиотеки шрифтов, к примеру, [библиотеку Google Fonts](https://fonts.google.com/)а

@font-face {

font-family: 'Roboto Condensed';

src: url(../../fonts/Robotocondensed.woff) format('woff');

}

# Внутреннее состояние: хук useState

*// Функциональный компонент User*

function User(props) {

*// Хук, управляющий внутренним состоянием.*

const [rating, setRating] = React.useState(0);

*/\**

*\* Обработчики событий: изменяют внутреннее состояние*

*\*/*

function handleLike() {

setRating(1);

}

function handleDislike() {

setRating(-1);

}

return (

<div className="user">

<img src={`img/${props.id}.png`} width="75" />

{props.name}

<div className="rating">

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

{rating}

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

</div>

</div>

);

}

**Эффекты: хук useEffect**

function NeonCursor() {

const [position, setPosition] = React.useState({ top: 0, left: 0 });

React.useEffect(() => {

function handleMouseMove(event) {

setPosition({

top: event.pageY,

left: event.pageX,

});

}

*// Список действий внутри одного хука*

document.addEventListener('mousemove', handleMouseMove);

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

*// Возвращаем функцию, которая удаляет эффекты*

return () => {

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

document.removeEventListener('mousemove', handleMouseMove);

};

});

return (

<img

src="./cursor.png"

width={30}

style={{

position: 'absolute',

top: position.top,

left: position.left,

pointerEvents: 'none',

}}

/>

);

}

# Зависимости

Зависимости передаются вторым аргументом в функцию React.useEffect — в виде массива с переменными. Это могут быть как пропсы, так и переменные внутреннего состояния текущего компонента. В нашем случае нас интересует пропс selectedId, поэтому нужный нам код будет выглядеть так:

Скопировать кодJAVASCRIPT

React.useEffect(() => {

fetchMagicWandById(selectedId);

}, [selectedId]);

Теперь это эффект с зависимостью, который запоминает последнее значение selectedId и вызывает переданный в него колбэк, только когда это значение поменяется/

Из концепции массива зависимостей вытекает интересное свойство: если передать пустой массив, то такой эффект будет вызван всего один раз.

# Установка React Router

Чтобы установить библиотеку React Router в проект на «Реакте», откройте главную директорию проекта и введите npm i --save react-router-dom.

В файле index.js, который расположен внутри каталога src/, обернём основной компонент App в компонент BrowserRouter:

*// index.js*

import React from 'react';

import ReactDOM from 'react-dom';

import { BrowserRouter } from 'react-router-dom'; *// импортируем BrowserRouter*

import App from './components/App';

import './index.css';

*// теперь обернём компонент App в BrowserRouter*

ReactDOM.render(

<React.StrictMode>

<BrowserRouter>

<App />

</BrowserRouter>

</React.StrictMode>,

document.getElementById('root')

);

Внутрь Route вложим компонент, который будет отрисовываться при каждом обращении к URL-пути, передаваемому пропсу path:

*// App.js*

import React from 'react';

import { Route } from 'react-router-dom';

import Dashboard from './Dashboard';

import Header from './Header';

import './App.css';

function App() {

return (

<div className="App">

<Header />

<Route path="/">

<Dashboard />

</Route>

</div>

);

}

export default App;

Компонент Route устанавливает связь между путём, который указан в пропсе path, и URL-адресом, который в данный момент используется браузером.

# Компонент Switch

Если разместить несколько компонентов Route внутри Switch, отрисуется только один из них.

*// App.js*

import React from 'react';

import { Route, Switch } from 'react-router-dom'; *// импортируем Switch*

import Dashboard from './Dashboard';

import Header from './Header';

import Reviews from './Reviews';

import AboutMe from './AboutMe';

import './App.css';

function App() {

return (

<div className="App">

<Header />

<Switch>

<Route exact path="/"">

<Dashboard />

</Route>

<Route path="/reviews">

<Reviews />

</Route>

<Route path="/about-me">

<AboutMe />

</Route>

</Switch>

</div>

);

}

export default App;

# Компонент Link

*// Dashboard.js*

import React from 'react';

import './Dashboard.css';

import { Link } from 'react-router-dom';

function Dashboard () {

return (

<div className="dashboard">

<h2>Emoji Critic — всё об эмодзи</h2>

<p>

#1 среди авторов обзоров на эмодзи в 2020 году!

</p>

<Link to="/reviews">

Нажмите, чтобы увидеть самые свежие ревью!

</Link>

</div>

)

}

export default Dashboard;

Мы добавили Link внутри JSX под тегом <p>. У компонента Link есть пропс to для описания URL, по которому следует перейти.

# Компонент NavLink

<NavLink to="/" activeClassName="nav\_\_link\_active">Dashboard</NavLink>

Этот компонент отличается от Link тем, что у него есть пропс activeClassName. Он принимает в качестве значения селектор CSS-класса. Указанный в activeClassName CSS-класс будет добавлен к тому элементу NavLink, маршрут которого совпадает с текущим URL в браузере.

*// NavBar.js*

import React from 'react';

import { NavLink } from 'react-router-dom';

import './NavBar.css';

function NavBar () {

return (

<nav className="menu">

<NavLink exact to="/" className="menu\_\_link">Домой</NavLink>

<NavLink to="/reviews" className="menu\_\_link">Обзоры эмодзи</NavLink>

<NavLink to="/about-me" className="menu\_\_link">Обо мне</NavLink>

</nav>

)

}

export default NavBar;

Осталось описать CSS-правило для класса, указанного в activeClassName. Сделаем активные ссылки внутри App.css зелёными.

Скопировать кодCSS

*/\* Добавим новый селектор в NavBar.css \*/*

.menu\_\_link\_active {

color: #4AEA37;

}

# «Реакт-хук»  useRouteMatch

Хук useRouteMatch позволяет узнать текущее состояние маршрута. Импортируем его из react-router-dom — так же, как другие компоненты, предоставляемые библиотекой. С помощью хука useRouteMatch можно легко создавать вложенные маршруты:

Скопировать кодJSX

import React from "react";

import { Route, Link, useRouteMatch } from "react-router-dom";

import MyStory from "./MyStory";

import Hobbies from "./Hobbies";

import Contact from "./Contact";

import "./AboutMe.css";

function AboutMe() {

const { path, url } = useRouteMatch();

return (

<div>

<ul className="links">

<li>

<Link to={`${url}/my-story`}>Моя история</Link>

</li>

<li>

<Link to={`${url}/hobbies`}>Хобби</Link>

</li>

<li>

<Link to={`${url}/contact`}>Контактная информация</Link>

</li>

</ul>

<Route path={`${path}/my-story`}>

<MyStory />

</Route>

<Route path={`${path}/hobbies`}>

<Hobbies />

</Route>

<Route path={`${path}/contact`}>

<Contact />

</Route>

</div>

);

}

export default AboutMe;

Благодаря useRouteMatch мы можем получить две переменные: url и path. Переменной url мы можем создать относительную ссылку внутри компонента Link. А переменной path — относительный путь для компонентов Route.

# Динамические маршруты

<Route path='/friends/0'>

<Friend serverData={serverData} />

</Route>

<Route path='/friends/1'>

<Friend serverData={serverData} />

</Route>

*// файл App.js*

import React from "react";

import "./App.css";

import { BrowserRouter, NavLink, Route, Switch } from "react-router-dom";

*// импортируем нужные компоненты*

import Friends from "./Friends";

import Friend from "./Friend";

import Dashboard from "./Dashboard";

import serverData from "../serverData";

function App() {

return (

<BrowserRouter>

<div className="App">

<header className="header">

<NavLink to="/" className="header\_\_logo">

Parrot Friendship Society

</NavLink>

<nav className="menu">

<ul className="menu\_\_list">

<li className="menu\_\_list-item">

<NavLink className="menu\_\_link" to="/friends">

Друзья

</NavLink>

</li>

</ul>

</nav>

</header>

<Switch>

<Route exact path="/">

<Dashboard />

</Route>

<Route exact path="/friends">

<Friends serverData={serverData} />

</Route>

<Route path="/friends/:id">

<Friend serverData={serverData} />

</Route>

</Switch>

</div>

</BrowserRouter>

);

}

export default App;

Внутри пути компонента Route стоит двоеточие : перед id. Так мы можем получить доступ к любому значению, которое находится на этом же месте в URL

Скопировать кодJSX

<Route path='/friends/:id'>

<Friend serverData={serverData} />

</Route>

Затем можно получить доступ к значению параметра внутри компонента:

Скопировать кодJSX

<p>{favoriteParrot}</p>

# «Реакт-хук» useParams

Чтобы это работало, нужно использовать «Реакт-хук» useParams:

*// компонент Friend*

import React from 'react';

import { useParams } from 'react-router-dom'; *// импортируем хук*

function Friend() {

let { id } = useParams(); *// получаем значение id*

return (

<div>

<h2>{id}</h2>

</div>

);

}

export default Friend;

Мы импортировали хук useParams из react-router-dom и с помощью деструктуризации извлекли переменную id. Эта переменная будет соответствовать любому значению, которое находится на месте :id в URL.

Посмотрим ещё раз на маршруты:

Скопировать кодJSX

<Switch>

<Route exact path='/friends'>

<Friends serverData={serverData} />

</Route>

<Route path='/friends/:id'>

<Friend serverData={serverData} />

</Route>

</Switch>

*// компонент Friend*

import React from 'react';

import { useParams } from 'react-router-dom'; *// импортируем хук*

import './Friend.css';

function Friend(props) {

let { id } = useParams(); *// получаем доступ к параметру URL*

let { friends } = props.serverData; *// достаём данные, используя деструктуризацию*

const friend = friends.find(f => f.id === id);

return (

<div className="friend">

<div className="friend\_\_card">

<img className="friend\_\_userpic" src={friend.profilePicLight} alt={friend.name}/>

<div className="friend\_\_details">

<h3 className="friend\_\_name">{friend.name}</h3>

<p className="friend\_\_location">Местоположение: {friend.location}</p>

<p className="friend\_\_quantity">Количество домашних попугаев: {friend.parrotsOwned.length}</p>

<p className="friend\_\_fav-quote">Любимое высказывание о птицах: "{friend.favBirdQuote}"</p>

</div>

</div>

</div>

);

}

export default Friend;

# «Реакт-хук» useHistory Объект history

import { useParams, useHistory } from 'react-router-dom';

Теперь вызовем хук внутри функционального компонента. Для этого присвоим его константе history:

const history = useHistory();

Внутри JSX добавим кнопку <button>. Нажатие на неё вызовет метод history.goBack, который отправит пользователя на предыдущую страницу:

</div>

<button className="button button\_type\_back" onClick={() => history.goBack()}></button>

Ещё у объекта history бывают такие методы:

* history.goForward — перемещает пользователя «вперёд» по навигации истории;
* history.push — создаёт запись в навигации истории.

Так же как метод history.goBack похож на нажатие кнопки «Назад» в браузере, а history.goForward соответствует нажатию кнопки «Вперёд», history.push аналогичен открытию новой страницы

# Компонент Redirect Переадресация

Компонент Redirect заменяет текущую запись в навигации истории и перенаправляет пользователя на другую страницу. Его часто используют в веб-приложениях после авторизации: например, чтобы авторизованный пользователь не мог вернуться на страницу входа

Чтобы проверить значение loggedIn и определить, вошёл ли пользователь в систему. Если да, пользователь увидит компонент своего профиля UserProfile. В противном случае Redirect отправит пользователя на экран авторизации:

Скопировать кодJSX

<Route path="/my-profile">

{!loggedIn ? <Redirect to="/log-in" /> : <UserProfile />}

</Route>

# Создание страницы 404

Хороший тон –– проинформировать пользователя, что он перешёл на неопределённый маршрут, если пользователь перейдёт по ссылке, которой нет в приложении

Создадим новый компонент PageNotFound и поместим его в директорию components/:

import React from 'react';

import { Link } from 'react-router-dom';

import './PageNotFound.css';

import Bye from '../images/404.svg';

function PageNotFound () {

return (

<div className="not-found">

<h3 className="not-found\_\_title">

<span>404</span> - Страница не найдена

</h3>

<img className="not-found\_\_image" src={Bye} alt=""/>

<p className="not-found\_\_text">

Ой, здесь ничего нет

</p>

<Link className="button button\_type\_to-main" to="/">Назад</Link>

</div>

)

}

export default PageNotFound;

Важен порядок компонентов Route внутри Switch: только первый совпадающий маршрут будет отрисован. Когда Switch обнаружит, что URL браузера не совпадает с /, /reviews или /about-me, то отрисует компонент PageNotFound. Для этого в конце Switch добавим ещё один компонент Route со значением path="\*". Символ \* обозначает «любой».

Скопировать кодJSX

// маршруты внутри файла App.js

<Switch>

<Route exact path="/">

<Dashboard />

</Route>

<Route path="/reviews">

<Reviews />

</Route>

<Route path="/about-me">

<AboutMe />

</Route>

<Route path="/about-us">

<AboutUs />

</Route>

<Route path="\*">

<PageNotFound />

</Route>

</Switch>

Теперь, если запустить проект на локальном сервере и перейти на несуществующий маршрут, например, localhost:3000/zdesnichegonet, приложение отрисует страницу 404.

# Поднятие стейта

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").

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>

);

}

При этом сам компонент ThemeSelect упростится:

function ThemeSelect(props) {

return (

<select onChange={props.onChange}>

<option value="day">День</option>

<option value="night">Ночь</option>

</select>

);

}

function ThemeIcon(props) {

return (

<div className="icon">

{props.theme === 'day' ? '🔆' : '🌙'}

</div>

);

}

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.

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>

);

}

# Глобальный стейт

## React.createContext Чтобы создать контекст, нужно использовать встроенный метод

Этот метод возвращает новый объект контекста. Важное свойство этого объекта — Provider

*// 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>

);

}

# 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

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

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>

);

}

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>

);

}

# Списки и ключи

В JSX работа со списками осуществляется с помощью метода map:

Чтобы помочь алгоритму понять, что элементы просто поменялись местами, нужно применить специальный атрибут key (англ. «ключ»)

Теперь, когда «Реакт» будет сравнивать новые списки чатов, он сможет опираться на этот атрибут, чтобы понять, что элементы изменили свой порядок внутри списка

ReactDOM.render(

<>

<h2>Чаты</h2>

{chats.map((chat) => (

<Chat

key={chat.id}

id={chat.id}

name={chat.name}

lastMessageAt={chat.lastMessageAt}

/>

))}

</>,

document.querySelector("#root")

);

После выбора чата может понадобиться заменить весь список сообщений, за который отвечает компонент MessageList. Иногда самое простое решение — дать команду «Реакту» полностью демонтировать старый компонент MessageList и смонтировать на его месте новый.

Чтобы это сделать, компоненту можно задать новый ключ, соответствующий ID выбранного чата:

ReactDOM.render(

<>

<h2>Чаты</h2>

{chats.map((chat) => (

<Chat

key={chat.id}

id={chat.id}

name={chat.name}

lastMessageAt={chat.lastMessageAt}

/>

))}

<h2>Сообщения</h2>

<MessageList key={selectedChatId} />

</>,

document.querySelector("#root")

);

# Работа с формами

В «Реакте» принято использовать «управляемые компоненты». Это элементы управления в формах, значения которых синхронизируются со стейтом компонента, где они находятся.

function Input() {

*// Стейт, в котором содержится значение инпута*

const [value, setValue] = React.useState('');

*// Обработчик изменения инпута обновляет стейт*

function handleChange(e) {

setValue(e.target.value);

}

function handleShow() {

alert(value);

}

function handleClear() {

setValue('');

}

return (

<>

<input type="text" value={value} onChange={handleChange} />

<button onClick={handleShow}>Показать</button>

<button onClick={handleClear}>Очистить</button>

</>

);

}

ReactDOM.render((

<Input />

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

В React <textarea> использует атрибут value. Таким образом, форму с <textarea> можно написать почти тем же способом, что и форму с однострочным <input>:

<textarea value={state.value} onChange={handleChange} />

В HTML <select> создаёт выпадающий список. HTML-код в этом примере создаёт выпадающий список вкусов:

<select>

<option value="grapefruit">Грейпфрут</option>

<option value="lime">Лайм</option>

<option selected value="coconut">Кокос</option>

<option value="mango">Манго</option>

</select>

Пункт списка «Кокос» выбран по умолчанию из-за установленного атрибута selected. React вместо этого атрибута использует value в корневом теге select. В управляемом компоненте так удобнее, потому что обновлять значение нужно только в одном месте (state). Пример:

handleChange(event) {setState({value: event.target.value}); }

Выберите ваш любимый вкус:

<select value={state.value} onChange={handleChange}> <option value="grapefruit">Грейпфрут</option>

<option value="lime">Лайм</option>

<option value="coconut">Кокос</option>

<option value="mango">Манго</option>

</select>

В атрибут value можно передать массив, что позволит выбрать несколько опций в теге select:

<select multiple={true} value={['Б', 'В']}>

# Рефы

Это означает, что обычно разработчику не нужно взаимодействовать с реальными DOM-элементами и обращаться к ним напрямую. Всё необходимое можно делать в JSX: устанавливать CSS-классы, задавать стили или создавать подписку на события.

Иногда наличие прямого указателя на реальный DOM-элемент необходимо.

* управление фокусом, выделение текста или воспроизведение медиа;
* императивный вызов анимаций;

«Реакт» предоставляет такой механизм — рефы.

**Для функциональных компонентов** существует специальный хук useRef

Он возвращает объект, который с помощью JSX-атрибута ref можно присвоить любому элементу, чтобы получить доступ к нему.

function VideoPlayer() {

const videoRef = React.useRef(); *// записываем объект, возвращаемый хуком, в переменную*

function handleClick() {

videoRef.current.play(); *// вызываем нужный метод на поле current объекта*

}

return (

<>

<video ref={videoRef} src="./clip.mp4" /> // указали элементу атрибут ref => получили прямой доступ к DOM-элементу

<button onClick={handleClick}>▶️</button> /\* привязали обработчик к кнопке \*/

</>

);

}

**Для классовых компонентов** всё почти так же, только для создания рефа используется функция React.createRef, а сам реф обычно записывается в this:

this.videoRef = React.createRef(); *// создали реф и записали его в переменную –– она будет свойством у this*

}

# Чистые компоненты

Добавим внешний компонент ChatWithTools, который содержит Chat внутри себя и реагирует на события мыши, чтобы отображать дополнительные элементы.

Скопировать кодJSX

const ChatWithTools = (props) => {

const [shouldShowTools, setShouldShowTools] = React.useState(false);

function handleMouseEnter() {

setShouldShowTools(true);

}

function handleMouseLeave() {

setShouldShowTools(false);

}

return (

<div className="chat-with-tools" onMouseEnter={handleMouseEnter} onMouseLeave={handleMouseLeave}>

<Chat {...props} /> *//<Chat {...props} />. С помощью такой конструкции можно копировать в дочерний компонент сразу все собственные пропсы.*

{shouldShowTools && (

<div className="tools">

<button>Не беспокоить</button>

<button>Удалить</button>

</div>

)}

</div>

);

};

const chats = */\* ... \*/*

ReactDOM.render((

<>

<h2>Чаты</h2>

{chats.map((chat) => (

<ChatWithTools id={chat.id} name={chat.name} lastMessageAt={chat.lastMessageAt} />

))}

</>

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

Чтобы сделать функциональный компонент чистым, достаточно обернуть его в специальную функцию React.memo:

const Chat = React.memo((props) => {

return (

<div className="chat">

<img src={`img/${props.id}.png`} width="75" />

<h2>{Math.random()}</h2>

<div className="date">{props.lastMessageAt}</div>

</div>

);

});

Теперь при наведении мыши не происходит цепной реакции — она сразу «обрывается» на компоненте Chat, так как его пропсы не меняются:

Так мы используем ресурсы более разумно.

Сделать чистым классовый компонент ещё проще: надо лишь наследоваться не от React.Component, а от React.PureComponent:

Скопировать кодJSX

class Chat extends React.PureComponent {

React.memo и React.PureComponent определяют изменения в пропсах по алгоритму «поверхностного сравнения» Поэтому нужно избегать ситуаций, когда значения пропсов со сложными типами создаются в момент передачи, например:

Скопировать кодJSX

<MyPureComponent

userNames={['Gregory', 'James', 'Allison']}

onClick={() => console.log(1)}

/>

# Компоненты более высокого порядка

В этой теме мы уже рассмотрели функцию React.memo. Эта функция — пример паттерна проектирования, который называется HOC (от англ. higher-order component, «компонент более высокого порядка»).

Функция React.memo запоминает последние пропсы и кэширует результат рендера своего дочернего компонента. При следующем рендере, если пропсы не изменились, она возвращает закэшированный результат. Такое поведение универсально и применимо почти к любому компоненту.

HOC-компоненты часто имеют слово "with" в названии, указывая на то, какая именно функциональность добавляется.

# Немного о компонентах более высокого порядка

НОС-компоненты позволяют добавить функциональность одному или нескольким компонентам. Если какие-то компоненты нуждаются в общей функциональности, можно обернуть их в HOC-компонент и описать в нём всё, что нужно. В больших проектах компоненты более высокого порядка позволяют лучше структурировать код и сделать его понятнее.

Для начала создадим HOC-компонент ProtectedRoute в директории /components:

Скопировать кодJSX

*// ProtectedRoute.js*

import React from "react";

import { Route, Redirect } from "react-router-dom";

*// этот компонент принимает другой компонент в качестве пропса*

*// он также может взять неограниченное число пропсов и передать их новому компоненту*

const ProtectedRoute = ({ component: Component, ...props }) => {

return (

<Route>

{() =>

props.loggedIn ? <Component {...props} /> : <Redirect to="./login" />

}

</Route>

);

};

export default ProtectedRoute;

Чтобы понять, что происходит внутри нового HOC-компонента, посмотрим на строку ({ component: Component, ...props }). Для получения компонентов, которые переданы через пропс component, HOC-компонент ProtectedRoute использует деструктуризацию. Пропсы пробрасываются в него также при помощи деструктуризации. Это значит, что данные могут поступать от родителя к HOC-компоненту, который потом используется повторно для создания любых новых компонентов.

Привет. При реализации в этой статье дана рекомендация использовать пустой массив. Но тогда появляется ворнинг о том, что функции должны быть указаны в массиве зависимостей. Если же их там указываю, то теряется эффект пустого массива и выводит рекомендацию использовать useEffect callback. Помоги пожалуйста разобраться, как этого избежать, и нужно ли функции вносить в зависимости, или оставить пустой массив?

2 files

https://ca.slack-edge.com/TPV9DP0N4-U01EVLJ5XJM-7d9fb5e82d6d-48

[**Михаил Зятьков**](https://app.slack.com/team/U01EVLJ5XJM)  [7 minutes ago](https://yandex-students.slack.com/archives/G01ENBXA9BR/p1621935087088200?thread_ts=1621748186.082700&cid=G01ENBXA9BR)

[@Александр Катрюхин](https://yandex-students.slack.com/team/U01F0MCBYVB) Привет, давай сначала разберемся с последним вопросом: нужно ли функции вносить в зависимости, т.к. он здесь самый важный.  
В useEffect в зависимости мы вносим то, что будет изменяться. В данном случае насколько я понимаю не предполагается изменение наших функций handleEscClose и handleOverlayClose.Теперь разберемся, что делать с варнингами.

1. мы можем просто игнорировать его, если понимаем почему и для чего мы это делаем. Для этого просто ставим комментарий // eslint-disable-next-line react-hooks/exhaustive-deps
2. Мы можем перенести тело наших функций в сам эффект, если они нигде больше в компоненте не используются
3. Мы можем вынести эти две функции за рамки нашего компонента, если они не используют никакие данные из самого компонента
4. Если перечисленное выше не наш случай, то добавляем их в массив зависимостей в useEffect, но при этом сами функции оборачиваем в хук [useCallback](https://ru.reactjs.org/docs/hooks-reference.html" \l "usecallback" \t "_blank)

Привет. [@Михаил Зятьков](https://yandex-students.slack.com/team/U01EVLJ5XJM) При редактировании профиля все отрабатывает, но в консоли такой warning. Это ошибка? Запутался немного с управляемыми компонентами...

[**Михаил Зятьков**](https://app.slack.com/team/U01EVLJ5XJM)  [26 days ago](https://yandex-students.slack.com/archives/G01ENBXA9BR/p1621396112015600?thread_ts=1621342918.007800&cid=G01ENBXA9BR)

[@Александр Катрюхин](https://yandex-students.slack.com/team/U01F0MCBYVB) чтобы избежать этой ошибки, тебе нужно в компоненте EditProfile в эффекте добавить небольшую проверку и в зависимости от результатов вызывать  setName и setDescription.  
Смотри, что у тебя сейчас происходит.

1. currentUser по-умолчанию равен пустой строке
2. currentUser мы получаем асинхронно, следовательно, компонент EditProfile создается раньше, чем мы получаем значение с сервера
3. т.к. currentUser - пустая строка, то ‘’.name и ‘’.about равны undefined
4. поэтому мы видим ошибку в консоле

 React.useEffect((currentUser) => {

    if (currentUser) {

      setName(currentUser.name);

      setDescription(currentUser.about);

  }}, [currentUser]);