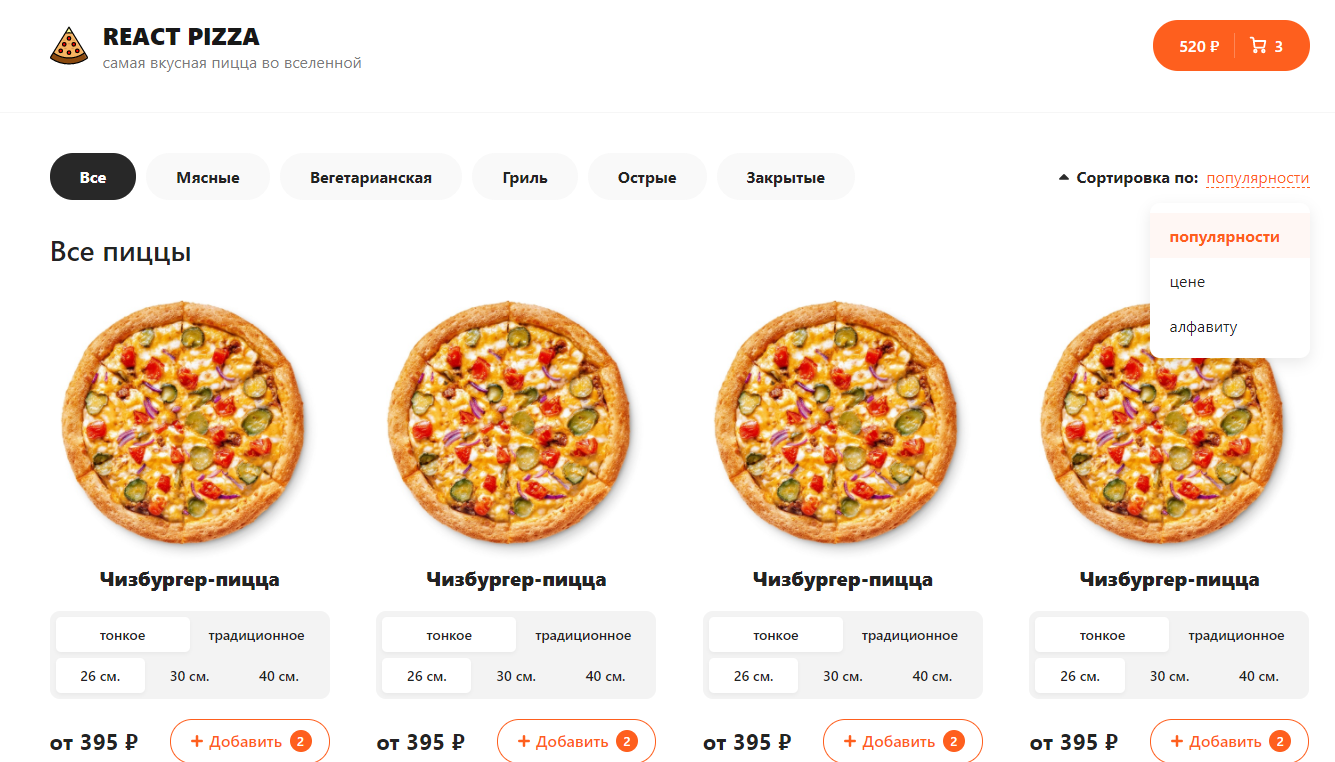
**Курс React Pizza**

<https://www.youtube.com/watch?v=bziVFvq8cLQ&t=7s>

Магазин по продаже пиццы.

На экран выводится перечень пицц.



У каждого элемента есть пункты выбора типа теста («тонкое» или «традиционное») и размера.

Каждый элемент содержит кнопку «Добавить», при клике по которой пицца добавляется в корзину, а на самое кнопке отображается количество заказанных пицц данного типа. На кнопке «Корзина» отображается общая стоимость и количество добавленных в неё пицц. При клике на корзину открывается окно корзины, содержащее список выбранных пицц с возможностью изменения количества, а также кнопка «Оплатить». Если корзина пустая, то информационное сообщение и кнопка «Назад».

Для фильтрации есть меню «Все», «Мясные», «Вегетарианские» и пр. для выбора категории.

Также кнопка сортировки по популярности, цене и алфавиту.

Изображения пицц берем с сайта dodopizza.ru (<https://dodopizza.ru/maikop/pionerskaya268>) – копируем URL картинки

Полный стек:

- ReactJS

- Redux (библиотека для хранения данных на клиентской части)

- redux-thunk (доп. библиотека для Redux. Асинхроные действия).

- React Router (библиотека для реализации переходов без перезагрузки страницы).

- Axios (библиотека для отправки AJAX-запросов)

- json-server (для хранения пицц на фейковом сервере)

Общий flow: Сначала делается верстка и далее html передается javascript-кодеру.

Готовая верстка тут: <https://github.com/Archakov06/react-pizza-html.git>. Клонируем этот репозиторий:   
git clone <https://github.com/Archakov06/react-pizza-html.git>

В нем нас интересует только \*.html

В папке проектов даем команду “npx create-react-app pizza-shop” для создания нового проекта

<React.StrictMode> позволяет детализировать ошибки. с версии 16.8

App.js – компонент <App> - основной компонент, включающий в себя все остальные компоненты приложения.

Компонент – это функция, возвращающая JSX в операторе return(). Её обрабатывает webpack. JSX преобразуется в код js через createElement() и пр. С помощью React.createElement() создается Virtual DOM, в котором React определяет, надо ли перерисовывать элемент. VirtualDOM – это дерево объектов, представляющее дерево html с дополнительными свойствами. Хранится в памяти. При изменении свойства в Virtual DOM React проверяет старое и новое его значение, и если оно изменилось, происходит перегенерация кода html. Это улучшает производительность: объекты, у которых ничего не изменилось, не перерисовываются. createElement() – «дорогостоящая» операция. В отличие от React, jQuery всегда перерисовывает все дерево html.

Чистим: удаляем logo, тесты и пр. Оставляем только app.js и index.js. Убираем стили в App().

В index.html меняем <title>Pizza Shop</title>, meta description, по желанию правим manifest.json, если предполагается использовать на мобилках.

Теперь надо взять верстку из <https://github.com/Archakov06/react-pizza-html.git> и превратить её в React – компоненты. Для этого клонируем или скачиваем zip-файл репозитория git.

Там в каталоге public есть файлы index.html, cart.html, cart-empty.html - их надо преобразовать в компоненты React.

Начальная верстка: часть кода index.html начальной верстки (из react-pizza-html.git) вставляем в оператор return() компонента App

* copy/paste часть кода index.html - <div class=”wrapper”>…</div>
* заменяем </li class=”active”> на </li> - иначе ошибка. Это ошибка верстки и обработки gulp
* заменяем все “class=” на “className=”
* Внутри тегов <svg> преобразуем имена свойств в стиль camelCase. Например: “stroke-width” => “stokeWidth”.  
  и т.д. пока не исчезнут ошибки в консоли.

Проверяем – что-то отображается. (не забываем в терминале дать npm start)

Вместо CSS будем использовать SCSS. Он проще и не требует настройки webpack

npm install node-sass

npm audit fix

Копируем из исходника папку src/scss в папку проекта src/scss.

В index.js пишем: import './scss/app.scss';

Папку fonts из исходника копируем в src\scss\fonts

При ошибках правим в \*.scss ссылки @import '../variables'; (правим пути)

Отображается, но нет **логотипа**.

* создаем папку src/assets, в ней папку img  
  В asssets будут храниться статические файлы, в том числе картинки
* переносим в неё содержимое папки public\img из исходника  
  import logo from './assets/img/pizza-logo.svg'
* Меняем src в логотипе:  
  <img width="38" src={logo} alt="Pizza logo" />

Нельзя в src использовать пути. Сначала надо его импортировать, а потом использовать полученную ссылку. Webpack затем подставит правильный путь к файлу.

Проверяем – все норм. В консоли ошибок не должно быть.

<https://www.youtube.com/watch?v=xvuDzujtxN4>

Итак, у нас есть первоначальная верстка. Теоретически можно оставить и так. Но React использует компонентный подход. Повторное использование компонентов позволяет избежать дублирование кода, а специфичность компонентов обеспечивается передачей им параметров – пропсов.

Проект будет сначала на классовых компонентах, потом на функциональных

**Сначала сделаем компоненты Header и Button**

В папке src создаем файл Header.jsx. Имена файлов компонентов должны быть с большой буквы. Имя функции-компонента также должно начинаться с большой буквы – тогда React понимает, что это не просто функция, а компонент, а также позволяет отличить от html-тегов (напр. <button>)

Для расширения имен файлов компонентов - .js или .jsx. “jsx” визуально дает понять, что это компонент, а не javascript.

В самом начале кода любого компонента необходимо импортировать React:

import React from 'react';

Функцию компонента необходимо экспортировать:

export default function Header() { return (…); }

Там, где используется – import Header from ‘./Header’

При использовании “export default” в импорте можно использовать другое имя.

Если “default” не используется, то следует писать import { Header } from ‘./Header’

Порядок поиска расширений: js -> jsx -> ts -> tsx -> папка с index.js -> папка с index.jsx -> …

В файл Header.jsx в Header(), в return() переносим разметку <div className=”header”>

В App вместо этого div вставляем компонент <Header />

import React from 'react';

import logo from './assets/img/pizza-logo.svg'

// функциональный компонент

export default function Header() {

  return (

    <div className="header">

      <div className="container">

        <div className="header\_\_logo">

          <img width="38" src={logo} alt="Pizza logo" />

...

  );

}

**Классовые компоненты**

* Должны наследоваться (extends) от React.Component
* Содержат метод render(), который возвращает JSX
* если есть конструктор – constructor(props), то в нем должен вызываться super()
* если есть пропсы, он доступны через объект this.props даже без конструктора
* могут содержать другие методы:
  + componentDidMount() – вызывается после создания и внедрения компонента
  + componentDidUpdate() – компонент обновлен. Произошло какое-то действие, которое привело к перерисовке компонента.
  + componentWillUnmount() – вызывается, когда компонент удаляется.

Пример:

class Header extends React.Component {

  render() {

    return (

      <div className="header">

...

      </div>

    );

  }

}

export default Header;

В процессе работы компонент проходит через ряд этапов жизненного цикла. На каждом из этапов вызывается определенная функция, в которой мы можем определить какие-либо действия:

1. **constructor(props)**: конструктор, в котором происходит начальная инициализация компонента
2. **static getDerivedStateFromProps(props, state)**: вызывается непосредственно перед рендерингом компонента. Этот метод не имеет доступа к текущему объекту компонента (то есть обратиться к объкту компоненту через this) и должен возвращать объект для обновления объекта state или значение null, если нечего обновлять.
3. **render()**: рендеринг компонента
4. **componentDidMount()**: вызывается после рендеринга компонента. Здесь можно выполнять запросы к удаленным ресурсам
5. **componentWillUnmount()**: вызывается перед удалением компонента из DOM

Кроме этих основных этапов или событий жизненного цикла, также имеется еще ряд функций, которые вызываются при обновлении состояния после начального рендеринга компонента, если в компоненте происходят обновления:

1. **static getDerivedStateFromProps(props, state)**
2. **shouldComponentUpdate(nextProps, nextState)**: вызывается каждый раз при обновлении объекта props или state. В качестве параметра передаются новый объект props и state. Эта функция должна возвращать true (надо делать обновление) или false (игнорировать обновление). По умолчанию возвращается true. Но если функция будет возвращать false, то тем самым мы отключим обновление компонента, а последующие функции не будут срабатывать.
3. **render()**: рендеринг компонента (если shouldComponentUpdate возвращает true)
4. **getSnapshotBeforeUpdate(prevProps, prevState)**: вызывается непосредственно перед компонента. Он позволяет компоненту получить информацию из DOM перед возможным обновлением. Возвращает в качестве значения какой-то отдельный аспект, который передается в качестве третьего параметра в метод componentDidUpdate() и может учитываться в componentDidUpdate при обновлении. Если нечего возвращать, то возвращается значение null
5. **componentDidUpdate(prevProps, prevState, snapshot)**: вызывается сразу после обновления компонента (если shouldComponentUpdate возвращает true). В качестве параметров передаются старые значения объектов props и state. Третий параметр - значение, которое возвращает метод getSnapshotBeforeUpdate

Основные отличия классовых от функциональных:

ФК – обычная функция, которая что-то возвращает. Классовые содержат свою логику, которая засоряет код. Лучше UI отдельно, логика отдельно. Поэтому функциональные компоненты предпочтительнее.

**Пропсы**

Функция-компонент может принимать аргументы-параметры – пропсы. Это всегда один объект, свойства которого – параметры, переданные при вызове компонента. Для удобства аргумент функции можно деструктурировать.

Например:

<CartButton sum={560} count={4} />

тогда в компоненте CartButton:

export default function CartButton({ sum, count }) {

…

<span>{sum} ₽</span>

…

<span>{count}</span>

}

В классовых компонентах элементы пропсов доступны через this.props.sum и this.props.count.

“this” – это ссылка на текущий класс.

Для получения содержимого между тегами - <Компонент>*внутренности*</Компонент> есть свойство children.

Например:

<Comp>*какой-то текст или разметка*</Comp>

function Comp(props) {

props.children; // *какой-то текст или разметка*

}

Если пропс может принимать значение true/false, то не обязательно указывать <Comp bool={true}/>, можно просто <Comp bool/> - в этом случае bool станет true, а если не указывать bool, оно станет undefined – аналог false.

Для условного вывода JSX в зависимости от какого-то значения:

{bool & <Comp />}

и

{this.props.outline ? <Comp1 /> : <Comp2 />}

и даже так:

<button className={bool ? “outline” : “solid”}>{this.props.children}</button>

Для упрощения работы со стилями устанавливаем библиотеку ‘classnames’:

npm install classnames

npm audit fix

Использование:

если раньше мы писали типа

<button className={`button ${this.props.outline ? “button-outline” : ‘’}}

то с classnames: импортируем библиотеку: import classNames from 'classnames';

внутри {} вызываем classNames(). Функция принимает несколько аргументов в виде строк или объектов. У нас первый аргумент статический – “button”. Вторым аргументом передаем объект. Имя свойства объекта - имя класса CSS, а значение – условие, при котором этот класс применится (если true, то применится, если false/undefined -–то не применится).  
 <button className={classNames(‘button’, {

`button-outline`: this.props.outline

})}

Пример:

import React from 'react'

import classNames from 'classnames';

export default function Button(props) {

return (

<button className={classNames('button', {

'button--outline': props.outline

})}>{props.children}</button>

);

};

<https://youtu.be/SZma4oQN8SY?t=489>

Свойства, состоящие из нескольких слов, надо писать слитно с camelCase.

Чтобы повесить на кнопку onClick:

<button onClick={()=>*обработчик*}/>

или

<button onClick={*обработчик*} /> - если функция *обработчик* объявлена где-то в коде как function *обработчик*().

Если написать такой код:

function clickButton = () => { alert(“hello”); }

и вставить в <Button onClick={ clickButton }/>

то такой код не выполнится. Потому что <Button> != <button> из html. Поэтому в <Button> из пропсов надо взять параметр onClick (а это функция) и использовать его уже в теге OnClick тега <button>:   
<button onClick={props.onClick}>

Теперь надо превратить кнопку «Корзина» в <Button> - в Header.jsx

Если поместить разметку кода кнопки «Корзина» в компонент <Button> внутри Header:

 <div className="header\_\_cart">

          <Button>

            <a href="/cart.html" className="button button--cart">

…

</Button>

Вырезаем внутренность <a> и перемещаем в <Button> - портятся стили, т.к. у тега <button> нет стиля ‘button—cart’. Надо сделать, чтобы у кнопки было какое-то состояние (в данном случае – стиль).

Для этого указываем <Button className=’button—cart’/>, в компоненте Button() получаем props.className и добавляем его в classNames():

<div className="header\_\_cart">

   <Button **className="button--cart"**>

export default function Button(props) {

  return (

    <button className={classNames('button', **props.className**, {

      'button--outline': props.outline

    })}>{props.children}</button>

  );

};

Рекомендуется делать деструктуризацию пропсов в компоненте:

export default function Button({ className, outline, children }) {

  return (

    <button

      className={classNames('button', className, { 'button--outline': outline })}>{children}</button>

  );

};

деструктурировать можно и массив:

[a,b,c] = someArray

**Изменение структуры проекта:**

Создаем папку src/components и переносим в неё Header.jsx, Button.jsx. Правим ссылку на логотип в импорте logo. В дальнейшем будем использовать index.js в папках.

Рекомендуется установит в VSCode сниппет «ES7 React/Redux/GraphQL…» Тогда можно использовать команды emmet с TAB:

imp - import moduleName from ‘module’

exp - export default moduleName

enf - export const functionName = (params) => { }

**imr** - import React from 'react'

clg - console.log(object)

useState - const [state, setState] = useState(initialState);

rfc – сразу создает компонент с propTypes

**Категории и useState()**

Переходим к логике. Задача: при клике на категории (Все, Мясные и т.д.) выделялась бы текущая категория. Для этого используются хуки. Раньше чтобы компонент хранил какие-то данные, приходилось использовать классы.

Создаем Categories.jsx (можно создать пустой файл Categories.jsx и в нем ввести “rfc” и нажать TAB)

Переносим в него <div className=”categories”>

В App.js этот тег заменяем компонентом <Categories/>

Если компонентов много, и их все надо импортировать в App (Header, Categories, Button и пр.), то можно все сократить до одного импорта:

* в папке components создаем файл index.js
  + В нем импортируем нужные компоненты и экспортируем как единый объект экспорта  
      
    import Button from "./Button";  
    import Header from "./Header";  
    import Categories from "./Categories";  
      
    export {Button, Header, Categories}:
* В App через деструктуризацию импорта вытаскиваем только нужные компоненты:  
  import {Header, Categories} from ‘./components’

Можно даже так, объединив импорт с экспортом (весь categories/index.js):

export {default as Button} from './Button'

export {default as Header} from './Header'

export {default as Categories} from './Categories'

**Категории**

Задача: отображать не статический список, а то, что передается пропсами, например:

<Categories items={["Мясные","Вегетарианские","Гриль","Острые","Закрытые"]}/>

Тогда в Categories() получаем этот массив в пропсах: const Categories = ({items})

Для отображения используем map():

<ul>

  <li className="active">Все</li>

  {

    items.map((name, index) => <li key={name}>{name}</li>)

  }

</ul>

Если какой-то элемент отображается через map(), то у него обязательно должен быть уникальный атрибут key. Использование индекса массива key={index} не рекомендуется, т.к. он может измениться, например, при вставке элемента внутрь массива.

Значение key хранится в памяти React. Если он изменился, то соответствующий элемент будет перерисовываться, а если не изменился – не будет. Это оптимизирует производительность.

Задача: при выборе типа активировать соответствующий пункт меню.

Активный пункт меню выделяется атрибутом active.

Попробуем в каждый элемент <li> добавить onClick={()=>alert(name)}. Проверяем – работает.

Для передачи события onClick на верх, через функцию onClick, переданную через пропсы, используется синтаксис:

onClick={()=>onClick(name}. Тогда в App.js:

<Categories items={["Мясные", …]} onClick={(data) => alert(data)}

При клике на пункт меню «Мясные» будет выводиться alert с текстом «мясные».

Т.е. в Categories вызовется событие onClick в котором вызовется функция onClick() из пропсов с аргументом name. Сама же функция вызовется в компоненте App; только имя аргумента не name, а data.

Чтобы переключать категории:

* В App создаем локальный стейт  
  const [activeCategory, setActiveCategory] = useState("Все");
* В <Categories> добавляем пропс onClick, который модифицирует стейт и сохраняет в нем название активного пункта:  
  <Categories   
   items={["Все", "Мясные", "Вегетарианские", "Гриль", "Острые", "Закрытые"]}  
   onClick={(cat) => setActiveCategory(cat)}  
   activeCategory={activeCategory}  
   />
* Внутри Categories добавляем  
  <li … className={activeCategory === name ? "active" : ""}

Переименовываем метод пропсов onClick в onSelectCategory – чтобы не было путаницы.

const Categories = ({ items, **onSelectCategory, activeCategory** }) => {

**let activeCat = activeCategory || "Все";**

  return (

    <div className="categories">

      <ul>

        {

          items.map((name) =>

            <li

              key={name}

              onClick={() => onSelectCategory(name)}

**className={activeCat === name ? "active" : ""}**

            >{name}</li>)

        }

      </ul>

    </div>

  );

}

И в App.js:

function App() {

  const [activeCategory, setActiveCategory] = useState("Все");

...

<Categories

              items={["Все", "Мясные", "Вегетарианские", "Гриль", "Острые", "Закрытые"]}

**onSelectCategory**={(cat) => setActiveCategory(cat)}

**activeCategory={activeCategory}**

            />

*stop at* [*https://youtu.be/SZma4oQN8SY?t=5872*](https://youtu.be/SZma4oQN8SY?t=5872) *– тут хуйня с классовыми компонентами*

Про хуки: <https://ru.reactjs.org/docs/hooks-intro.html>

useState() возвращает массив:

* переменная
* функция, которая меняет значение этой переменной

Аргумент useState() – начальное значение переменной.

При клике на кнопку выбора категории срабатывает onClick(). В нем мы вызываем функцию, которая помещает в состояние activeCategory имя этой категории. Если имя категории в <li> совпадает с активной категорией, то добавляется стиль “active”.

Можно делать так: в состоянии хранить не имя, а индекс массива категорий. По умолчанию 0 (useState(0)), тогда в items.map(item, index) используем onSelectCategory(index). Также локальный стейт можно реализовать прямо внутри Categories(), а не в App().

Для каждой переменной-состояния нужно использовать отдельный useState().

В функцию, которая меняет значение стейта (второй элемент массива, возвращаемого useState()) аргументом можно передавать не новое значение стейта, а функцию. Эта функция (обычно анонимная) аргументом принимает текущий стейт. Пример для счетчика ( [counter, setCounter]=useState(0) ):

setCounter((prev) => prev+1));

Если инкрементом, то надо ++prev, а не prev++ - в втором случае в setCounter() передастся 0

Это аналогично setCounter(counter+1).

Напрямую значение стейта менять нельзя – React не узнает, что соответствующий UI-компонент надо перерисовать. В стейтах надо хранить такие значения, изменение которых влияют на отображение информации на странице.

Приводим наг код к уроку

* Убираем из App() стейт. В параметрах <Categories items={[…]} убираем все; оставляем только items
* из массива items убираем первый элемент “Все”. Будем работать с индексами
* В Categories() реализуем локальный стейт   
  const [activeCategory, setActiveCategory] = useState(null);  
  (null будет означать “Все”)
* Реализуем функцию const onSelectCategory = (index) => { setActiveCategory(index); }  
  Она в стейт помещает индекс выбранного элемента списка категорий
* В теге <ul> первый элемент <li> статический и выглядит как   
  <li onClick={() => onSelectCategory(null)} className={activeCategory === null ? "active" : ""}
* Цикл перебора использует индекс:  
   items.map((name, index) =>  
   <li  
   key={name}  
   onClick={() => onSelectCategory(index)}  
   className={activeCategory === index ? "active" : ""}  
   >{name}</li>)

Теперь в стейте категорий хранится индекс массива выбранной категории (0 …x), а если ничего не выбрано - т.е. «Все» - то null.

import React, { useState } from 'react';

const Categories = ({ items }) => {

  const [activeCategory, setActiveCategory] = useState(null);

  const onSelectCategory = (index) => { setActiveCategory(index);  }

  return (

    <div className="categories">

      <ul>

        <li onClick={() => onSelectCategory(null)}

          className={activeCategory === null ? "active" : ""}

        >Все</li>

        {

          items.map((name, index) =>

            <li

              key={name}

              onClick={() => onSelectCategory(index)}

              className={activeCategory === index ? "active" : ""}

            >{name}</li>)

        }

      </ul>

    </div>

  );

}

export default Categories;

**Проверка на undefined**

Если по каким-то причинам в пропсах Categories() массив item не получен (например, ошибка сервера), то его значение будет undefined. Тогда рендеринг выдаст ошибку о невозможности применить метод map() к undefined.

Чтобы этого избежать, достаточно написать:

{items && items.map(….)}

Если items = undefined|null, то map() не выполнится. Это более короткая форма if(a) { a }

<https://youtu.be/IMBAK-DftVM?t=461>

Итак, на данный момент у нас есть компоненты App, Header, Button, Categories. Можно переключать категории.

Сделаем функционал сортировки и функционал корзины. Пока статичный.

**Сортировка**

Код html пока в App.js в теге <div className="sort">.

* Создаем компонент SortPopup.jsx и переносим этот код в него.
* Добавляем экспорт/импорт в components/index.js
* В App() вместо <div className="sort"> используем <SortPopup />

В SortPopup при нажатии элемент сортировки в «сортировка по», меню сортировки должно показываться или скрываться, т.е. переключаться. А при выборе категории меню должно скрываться.

Реализуем открытие меню при клике на пункт сортировки. Для этого нам нужен флаг переключения видимости меню. Для этого реализуем стейт const [visiblePopup, setVisiblePopup] = useState(false); Изначально оно скрыто (false). Тогда к тегу <span>популярности</span> добавляем onClick, который меняет видимость меню:  
<span onClick={() => setVisiblePopup(!visiblePopup)}>популярности</span>

Однако, этого нам недостаточно - нам надо скрыть блок <div className="sort\_\_popup"> в зависимости от значения visiblePopup. Тогда надо поместить блок <div className="sort\_\_popup"> внутрь блока кода и вызывать его в зависимости от visiblePopup:

{visiblePopup &&

   <div className="sort\_\_popup">

     <ul>

        <li className="active">популярности</li>

        <li>цене</li>

        <li>алфавиту</li>

     </ul>

   </div>

}

Теперь при кликах на <span>популярность</span> меню сортировки будет показываться/скрываться.

**Выделение активного пункта сортировки**

Делаем аналогично выбора категории: пробегаемся по массиву типов сортировок через map(), вешаем onClick(), в котором назначем выбранный индекс массива.

Для начала в «<span onClick={() => setVisiblePopup(!visiblePopup)}>популярности</span>» функцию в onClick() выносим в отдельный обработчик - toggleVisiblePopup().

const togglePopupVisible = () => {

setVisiblePopup(!visiblePopup);

}

Получается

<span onClick toggleVisiblePopup}>популярности</span>

Анонимная функция здесь вредит. Почему? Это вредит производительности. При использовании анонимной функции она создается при каждом рендере компонента как новый экземпляр, выделяя память и пр. При использовании внешней функции – этой проблемы нет.

Чтобы меню сортировки скрывалось при клике за пределами области меню сортировки, используем следующий подход: как только компонент создается, на весь документ вешается обработчик onClick(). Внутри него проверяем – по какой части страницы (какому элементу) произведен клик. Если он произведен вне меню сортировки, то вызываем setVisiblePopup(false), а если внутри меню сортировки – то ничего не делаем.

Для этого используем хук **useEffect()**. <https://ru.reactjs.org/docs/hooks-effect.html>

useEffect(*функция*, [*зависимости*]).

useEffect() проверяет – был ли компонент первый раз смонтирован, обновлен, удаляется и пр. Т.е. следит за эффектом компонента.

* Если массив *зависимости* не указан в вызове, то *функция* будет выполняться при каждом рендере компонента
* Если задан пустой массив *зависимости*, то *функция* будет выполнена один раз при первом рендере компонента. *Функция* вызывается также в том случае, если компонент удалился, а потом снова внедрен.
* Если массив *зависимости* непустой, то *функция* будет вызваться при изменении любого элемента из массива *зависимости*.
* Если нужно выполнить какое-то действие при удалении компонента (например, закрыть соединение с БД), то *функция* должна вернуть другую функцию, в которой происходит «очистка».

В нашем случае: при первом рендере повесить событие onClick на весь документ document.body - document.body.addEventListener(‘click’, …):

useEffect(()=> {

document.body.addEventListener('click', () => {

console.log('CLICK');

})

},[])

Тогда при клике на любом месте страницы в консоли будет “CLICK”.

А если написать useEffect(…., [visiblePopup]), то при каждом изменении значения visiblePopup будет вызываться document.body.addEventListener('click') и при каждом клике обработчик 'click' будет вызываться несколько раз.

Создаем функцию-обработчик handleOutsideClick(e). e – это аргумент типа mouse event. У него есть свойство target, ссылающееся на кликнутый элемент страницы.

const handleOutsideClick = (e) => {

    console.log('CLICKED!', e.target);

  }

useEffect(() => {

    document.body.addEventListener('click', handleOutsideClick)

  }, [])

Теперь в handleOutsideClick() надо определить, что мы кликнули за пределами окна сортировки и скрыть <div className="sort">. Для этого нужно сохранить ссылку на этот элемент div. В обычном js для этого используется document.querySelector(‘.sort’). Но в React так делать нельзя: на момент вызова компонента <div> еще не создан. Хотя можно поместить код document.querySelector(‘.sort’) внутрь useEffect(). Но это все равно неправильно.

Для хранения ссылки на html-элемент в React используется хук **useRef()**.

* 1. Создаем ссылку как const refer = useRef([*начальное значение .current*])  
     Возвращается объект, содержащий поле .current c начальным значением.
  2. В нужный html-элемент добавляем атрибут ref со значением, созданным useRef (например, <div ref={refer}/>

Таким образом, на <div> вешается свойство .current. Причем в current может хранится любое изменяемое значение, но его изменение не приводит к рендеру. Использование объекта для хранения значения переменной в свойстве current использует особенность того, что объекты передаются по ссылке, и это позволяет всегда получить актуальное значение переменной.

Обычно в useRef() аргумент не указывается.

Создаем ссылку sortRef с помощью useRef() и добавляем её в параметр ref в <div className=”sort”>:

import React, { useState, useEffect, useRef } from 'react'

…

const sortRef = useRef();

…

<div className="sort" ref={sortRef}>

Тогда в handleOutsideClick() значение sortRef.current будет указывать на элемент <div className="sort" ref={sortRef}>. Это позволит определить, куда мы кликнули. В const handleOutsideClick (e) параметр “e” (MouseEvent) содержит свойство path – массив html-элементов – начиная от элемента, на котором произошел клик и до самого внешнего (window). Например, если мы щелкнем на логотипе, то получим массив из 10 элементов:  
 [img, div.header\_\_logo, div.container, div.header, div.wrapper, div#root, body, html, document, Window]. А при А А при клике на заголовок сортирвки «сортировка по» получим массив:

[0: b, 1: div.sort\_\_label, 2: div.sort, 3: div.content\_\_top, 4: div.container, 5: div.content, 6: div.wrapper, 7: div#root, 8: body, 9: html, 10: document, 11: Window]

Тогда если в массиве нет “div.sort” – т.е. мы кликнули за пределами div.sort - то скрываем меню сортировки. Поверку делаем с помощью метода Arraty.includes(), скрытие – через стейт setVisiblePopup(). Т.к. массив path – это массив объектов DOM, то в нем ищем объект sortRef.current:

  const handleOutsideClick = (e) => {

    if (!e.path.includes(sortRef.current)) {

      setVisiblePopup(false);

    }

  }

Т.е если кликнутый элемент не содержит div’а, на который мы повесили ref, то скрыть меню вызовом setVisiblePopup(false);

Важно: event.path работает не на всех браузерах – например, в Firefox возвращает undefined. См. <https://stackoverflow.com/questions/39245488/event-path-is-undefined-running-in-firefox> Чтобы это исправить, надо переписать код:

const handleOutsideClick = (event) => {

    const path = event.path || (event.composedPath && event.composedPath());

    if (!path.includes(sortRef.current)) {

      setVisiblePopup(false);

    }

  }

**Выбор пункта меню сортировки**

Аналогично Categories:

* Из App() в SortPopup пропсами передаем массив сортировок:  
  <SortPopup items={["популярности", "цене", "алфавиту"]} />
* Принимаем их пропсами: SortPopup({items}) и через items.map() отображаем теги <li>
* Создаем стейт const [activeItem, setActiveItem] = useState(0);
* Создаем обработчик выбора -   
  const onSelectItem = (index) => {   
   setActiveItem(index);  
   setVisiblePopup(false); // чтобы скрыть меню  
  }
* В <li> onClick = onClick={() => handleActiveItem(index)}, также className={activeItem === index ? "active" : ""}
* В заголовке <span> для отображения текущего элемента:  
  <span onClick={togglePopupVisible}>{items[activeItem]}</span>

const [activeItem, setActiveItem] = useState(0);

…

const onSelectItem = (index) => {

    setActiveItem(index);

    setVisiblePopup(false);

}

...

<span onClick={togglePopupVisible}>{items[activeItem]}</span

…

 {visiblePopup &&

        <div className="sort\_\_popup">

          <ul>

            {

              items && items.map((name, index) =>

                <li

                  key={name}

                  onClick={() => onSelectItem (index)}

                  className={activeItem === index ? "active" : ""}

                >{name}</li>)

            }

          </ul>

        </div>

  }

Кнопку порядка сортировки (стрелка) сделаем потом. Общий алгоритм: создать стейт для текущего порядка сортировки const [sortOrder, setSortOrder] = useState(false), для изменения значка добавить стиль «перевернутого» значка .rotate с transform: rotate(180deg): (в sass надо использовать “&”)

svg {

margin-right: 8px;

transform: rotate(180deg);

&.rotated {

transform: rotate(0deg);

}

}

В SortPopup в svg добавляем className с условным оператором: className = {sortOrder ? "rotated" : “”}, добавляем onClick={()=>setSortOrder(!sortOrder)}

**Роутер в React**

<https://youtu.be/IMBAK-DftVM?t=5675>

Переход на страницу «корзина» будем делать через библиотеку react-router. При переходе на страницу корзины в адресной строке должен смениться URL, причем без перезагрузки.

Установка:

npm install react-router-dom

Использовать его будем в index.js:

* импортируем BrowserRouter и Route из ‘react-router-dom:  
  import { BrowserRouter, Route } from 'react-router-dom'
* Оборачиваем все приложение App компонентом <BrowserRouter>

Теперь можно анализировать адресную строку браузера и в зависимости от URL вызывать тот или иной компонент с помощью Route.

Внутри <BrowserRouter> прописываем несколько компонентов Route с параметрами:

* path – путь в адресной строке. Причем:
  + “/” –
* exact – строгое соответствие. Например, если для path=”/” не указать exact, то компонент, указанный в component будет отображаться при любом URL, напр. http://localhost:3000/qwe
* component – Компонент, который надо отобразить по этому пути

Например:

 <Route exact path="/" component={App} />

означает, что по адресу [http://localhost:3000**/**](http://localhost:3000/) будет отображаться компонент App, т.е. основное приложение. Причем если указать <http://localhost:3000/qwe>, то будет пустой экран.

Можно роуты вынести в index.js:

ReactDOM.render(

  <React.StrictMode>

    <BrowserRouter>

      <Route exact path="/" component={App} />

      <Route exact path="/header" component={Header} /> // test

    </BrowserRouter>

  </React.StrictMode>,

  document.getElementById('root')

);

Но маршруты (компоненты Route) мы будем создавать в App, потому что в нем есть общие элементы (заголовок, общие стили), а контент разный.

Поэтому index.js выглядит так:

ReactDOM.render(

  <React.StrictMode>

    <BrowserRouter>

      <App />

    </BrowserRouter>

  </React.StrictMode>,

  document.getElementById('root')

);

Теперь в App.js оставляем Header, но div.content будет храниться в разных компонентах. Это будут основные страницы приложения – поэтому создаем папку src/pages. В ней – файлы Home.jsx с главной страницей, и Cart.jsx с компонентом «Корзина»

**Домашняя страница – Home.jsx**

Переносим в него <div className=”container”>

Не забываем импорт компонентов: import { Categories, SortPopup } from '../components'

App.js теперь выглядит так:

function App() {

return (

<div className="wrapper">

<Header />

<div className="content">

<Home />

</div>

</div>

);

}

В Cart.jsx пока сделаем просто <h1>Корзина</h1>

Теперь надо сделать, чтобы по ссылке “/” выводился Home, а по ссылке “/cart” – Cart.

function App() {

  return (

    <div className="wrapper">

      <Header />

      <div className="content">

        <Route exact path="/" component={Home} />

        <Route exact path="/cart" component={Cart} />

      </div>

    </div>

  );

}

Т.к. App у нас обернут в index.js компонентом BrowserRouter, то можем использовать Route.

Теперь по ссылке <http://localhost:3000> рендерится Home, а по ссылке <http://localhost:3000/cart> - корзина (Cart.jsx)

Чтобы переходить на другие страницы по ссылке:

* В Header добавляем новый компонент Link из ‘react-router-dom’: import { Link } from 'react-router-dom'
* В Header у нас есть кнопка корзины – компонент <Button>. Оборачиваем его компонентом Link:  
  <Link><Button …> … </Button></Link>
* В <Link> добавляем параметр “to”, значением которого – маршрут (один из тех, которые указаны в <Route path=”..”/>. Для корзины это <Link to=”/cart”>

<Link> преобразует компонент в ссылку типа <a href={to}> и вешает на неё обработчик клика. Причем перерисовки всей страницы не происходит.

Чтобы вернуться из корзины на основную страницу по клику на заголовок, верхнюю часть компонента Header оборачиваем компонентом <Link to=”/”> - <div className="header\_\_logo">

function Header() {

  return (

    <div className="header">

      <div className="container">

        <Link to="/">

          <div className="header\_\_logo"> … </div>

        </Link>

        <div className="header\_\_cart">

          <Link to="/cart">

            <Button className="button--cart"> …

Один <Link> не должен содержать другой вложенный <Link>

*Лайфхак: если написать <Route path=”/” exact> <h1>Заголовок</h1> </Route>, то заголовок в h1 будет отображаться только если страница находится по роуту “/”.  
Например, если в Header написать  
<Route exact path="/"><p>самая вкусная пицца в Майкопе!</p></Route>  
то на странице Корзины строка «самая вкусная пицца в Майкопе!» не будет отображаться.  
Это позволяет обойтись без if .. else*

#5: <https://www.youtube.com/watch?v=5XKs7dvVQD8>

**Асинхронные запросы и fetch()**

Для нашего магазина нужна СУБД, с которой можно общаться по http, но без backend.

Для можно просто скопировать файл db.json в папку public, тогда он будет доступен по URL <http://localhostL3000/db.json>

Тестовый db.json: <https://github.com/Archakov06/react-pizza/blob/master/public/db.json>

Структура: объект с полем pizzas, представляющим собой массив объектов-пицц:

{

"pizzas": [

{

"id": 0,

"imageUrl": "https://dodopizza.azureedge.net/static/.... ",

"name": "Пепперони Фреш с перцем",

"types": [0,1],

"sizes": [26, 30, 40],

"price": 803,

"category": 0,

"rating": 4

}, …

]

}

Где каждый элемент массива:

* id: уникальный идентификатор
* imageUrl: ссылка на картинку
* name: наименование пиццы
* types: массив типов. 0-тонкое, 1-традиционное
* sizes: массив размеров в см.
* price: цена
* category: номер категории (мясная, вегетарианская, гриль, острая, закрытая)
* rating: рейтинг. Число 0-5

Обращаться к <http://localhost:3000/db.json> можно через стандартный fetch() и axios.

Мы пока будем получать данные с db.json, сохранять в App, а затем через пропсы передавать в другие компоненты. Потом отрефакторим код.

1. Данные с сервера надо получить один раз при первом рендере компонента. Для этого нужно использовать хук useEffect().
2. в useEffect() вызываем fetch(). Это асинхронная функция, которая возвращает Promise.

Общий синтаксис:  
fetch('http://localhost:3000/db.json')

.then(resp => resp.json())

.then(data => console.log(data))

Проверяем:

* в App создаем локальный стейт для массива пицц (изначально пустой)
* в useEffect() через fetch() получаем содержимое db.json, помещаем массив в стейт
* Через map() выводим элементы массива

function App() {

**const [pizzas, setPizzas] = useState([]);**

**useEffect(() => {**

**fetch('http://localhost:3000/db.json')**

**.then(resp => resp.json())**

**.then(data => setPizzas(data.pizzas));**

**}, []);**

return (

<div className="wrapper">

<Header />

<div>

**<ul>**

**{**

**pizzas.map(p => <li key={p.id}>{p.name}</li>)**

**}**

**</ul>**

</div>

<div className="content">

<Route exact path="/" component={Home} />

<Route exact path="/cart" component={Cart} />

</div>

</div>

);

}

<https://youtu.be/5XKs7dvVQD8?t=2145>

Для отображения одной пиццы на главной странице создаем компонент PizzaBlock.jsx, выносим в него один из <div className="pizza-block"> из Home.jsx. Добавляем импорт/экспорт в components/index.js

**Получение перечня пицц в App**. Список хранится в стейте pizzas. Передаем его пропсом pizzas в компонент Home. Но при записи <Route component={Home}> пропсы указать нельзя, Поэтому вместо “component” надо указывать “render” со значением – функцией, которая рендерит компонент, у которого можно указать пропсы:

<Route exact path="/" render={()=><Home pizzas={pizzas}/>} />

(если не нужно передавать пропсы, то проще использовать “component”)

Далее в Home.jsx принимаем пропс: function Home({pizzas}) { ..} и через map() выводим компоненты PizzaBlock:

{pizzas && pizzas.map(pizza => <PizzaBlock key={pizza.id} pizza={pizza}/>)}

В компоненте PizzaBlock принимаем пропс pizza – это объект одной пиццы – и выводим его поля в jsx. Можно можно сразу сделать деструктурирование пропса на компоненты {name, imageUrl, price,…}, еще вариант – сразу передать все пропсы через деструктурированный объект:

* В Home: <PizzaBlock key={pizza.id} {…pizza}/>
* В PizzaBlock: function PizzaBlock({ name, imageUrl, price, id, types, sizes })

Этот подход следует использовать, когда заранее точно известно, какие пропсы должен получить компонент.

{pizzas && pizzas.map(pizza => <PizzaBlock key={pizza.id} {...pizza} />)}

export default function PizzaBlock({ imageUrl, name, types, sizes, price }) {

Тут есть засада: рендер PizzaBlock может произойти еще до того, как получен массив пицц с сервера - тогда в PizzaBlock передается undefined. Это надо отловить, используя if(!name) return {<div></div>). На всякий случай

* создаем локальный массив типов const testo = ['тонкое', 'традиционное'], т.к. у нас в JSON для них только 0 и 1.
* в качестве размеров и типов в пропсах (sizes, types) у нас массивы, так что выводим их через map(). Не забываем про параметр key
* Активные пункты массивов sizes и types пусть будут первые – индекс 0. Для этого создаем 2 стейта для типов и размеров - useState(0). на <li> вешаем onClick, в которых меняем стейт
* Для подсветки активного пункта в файле src\scss\components\\_pizza-block.scss после “&.active” добавляем блок &.inactive { opacity: 0.3; }
* Выводим типы:  
  {types.map((type, index) => <li key={index} className={index === currentType ? "active" : "inactive"} onClick={() => setCurrentType(index)} >{testo[type]}</li>)}
* Выводим размеры:  
  {sizes.map((size, index) => <li key={size} className={size === sizes[currentSize] ? "active" : "inactive"} onClick={() => setCurrentSize(index)} >{sizes[index]} см.</li>)}
* Вместо анонимных функций в onClick создаем обработчики onSelectType() и onSelectSize() и используем их в onClick={()=>onSelectType(index)}. Это потребуется дальше, при использовании Redux

import React from 'react'

export default function PizzaBlock({ imageUrl, name, types, sizes, price }) {

  const testo = ['тонкое', 'традиционное'];

  const [currentType, setCurrentType] = React.useState(0); // index of array testo

  const [currentSize, setCurrentSize] = React.useState(0); // index of array sizes

  const onSelectType = (index) => {

    setCurrentType(index);

  }

  const onSelectSize = (index) => {

    setCurrentSize(index);

  }

  if (!name) { return <div></div>; }

  return (

    <div className="pizza-block">

      <img className="pizza-block\_\_image" src={imageUrl} alt="Pizza"/>

      <h4 className="pizza-block\_\_title">{name}</h4>

      <div className="pizza-block\_\_selector">

        <ul>

          {types.map((type, index) => <li

            key={index}

            className={index === currentType ? "active" : "inactive"}

            onClick={() => onSelectType(index)}

          >{testo[type]}</li>)}

        </ul>

        <ul>

          {sizes.map((size, index) => <li

            key={size}

            className={size === sizes[currentSize] ? "active" : "inactive"}

            onClick={() => onSelectSize(index)}

          >{sizes[index]} см.</li>)}

        </ul>

      </div>

      <div className="pizza-block\_\_bottom">

        <div className="pizza-block\_\_price">от {price} ₽</div>

        <div className="button button--outline button--add">

          <svg …  </svg>

          <span>Добавить</span>

          <i>2</i>

        </div>

      </div>

    </div>

  )

}

Вместо

className={size === sizes[currentSize] ? "active" : "inactive"}

можно использовать библиотеку “classnames” и написать так:

className={classNames({

'active': index === currentType, // класс 'active' будет применен только при заданном условии – если индекс совпадает

'inactive': index !== currentType

})}

*В видео используется массив availableSizes = [26, 30, 40] и актуальные размеры, а не индексы массива. IMHO это неправильно. Там используется подход «вывести все, но подсветить доступные». У меня – вывести только доступные, а подсветить активный выбранный.*

**axios**

Установка: npm install axios

axios позволяет выполнять запросы к серверу более удобным способом.

Импортируем его в App.js:

import axios from 'axios'

<https://youtu.be/5XKs7dvVQD8?t=5584>

Вместо fetch() пишем:

axios.get('http://localhost:3000/db.json').then((resp) => console.log(resp));

видно, что resp – это объект. У него есть свойство data. data – объект, содержащий массив pizzas:

Object

* 1. config: {url: "http://localhost:3000/db.json", method: "get", headers: {…}, transformRequest: Array(1), transformResponse: Array(1), …}
  2. data:
     1. pizzas: (10) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}]
     2. \_\_proto\_\_: Object
  3. headers: {accept-ranges: "bytes", cache-control: "public, max-age=0", content-encoding: "gzip", content-type: "application/json; charset=UTF-8", date: "Tue, 20 Jul 2021 08:26:10 GMT", …}
  4. request: XMLHttpRequest {readyState: 4, timeout: 0, withCredentials: false, upload: XMLHttpRequestUpload, onreadystatechange: *ƒ*, …}
  5. status: 200
  6. statusText: "OK"

Т.е. с axios не надо заморачиваться с декодированием ответа с сервера и JSON.

Тогда вместо fetch() будет такой код:

useEffect(() => {

    axios.get('http://localhost:3000/db.json')

      .then(({data}) => setPizzas(data.pizzas)); // resp.data.pizzas

  }, []);

В видео только один метод – axios.get(), т.к. пиццы добавлять не будем и другие методы (post, put и пр.) использовать не будем.

**PropTypes**

Используется для проверки типов данных переданных пропсов. Лучше использовать TypeScript, но это за рамками данного видео.

Например, нам нужно, чтобы prop.name был string, prop.types и props.sizes– массивы целых, props.price – float и т.п. Для этого нужна библиотека ‘prop-types’

<https://ru.reactjs.org/docs/typechecking-with-proptypes.html>

<https://www.npmjs.com/package/prop-types>

<https://habr.com/ru/post/319358/> (чуток устарело)

Установка: npm install --save prop-types

В PizzaBlock.jsx надо импортировать (import PropTypes from 'prop-types') и в конце файла, перед export, добавить

PizzaBlock.propTypes = {

  imageUrl: PropTypes.string,

  name: PropTypes.number,

  types: PropTypes.array,

  sizes: PropTypes.array,

  price: PropTypes.number,

}

Тогда, например, если вместо строки для пропса “name” будет передано число, то в консоли будет предупреждение и можно определить, в чем ошибка. Это помогает в отладке приложения.

Если типов может быть несколько, то надо использовать типа такого:

optionalUnion**:** PropTypes.oneOfType([

    PropTypes.string,

    PropTypes.number,

    PropTypes.instanceOf(Message)

  ]),

Для перечислений (enum) - PropTypes.oneOf(['News', 'Photos'])

Для уточнения типов элементов массива лучше использовать PropTypes.**arrayOf**(PropTypes.number)

Если нужно указать, что параметр обязателен, то добавить в конце типа .isRequired. Например: name: PropTypes.string.isRequired

В итоге:

PizzaBlock.propTypes = {

  imageUrl: PropTypes.string,

  name: PropTypes.string.isRequired,

  types: PropTypes.arrayOf(PropTypes.number).isRequired,

  sizes: PropTypes.arrayOf(PropTypes.number).isRequired,

  price: PropTypes.number,

}

PizzaBlock.defaultProps = {

name: '-',

imageUrl: '',

  types: [0],

  sizes: [26],

  price: 0

}

Также можно создать объект PizzaBlock.defaultProps по аналогии с .propTypes. См. <https://ru.reactjs.org/docs/typechecking-with-proptypes.html>

Если какие-то пропсы не будут переданы в компонент, то они возьмутся из defaultProps.

**REDUX**

<https://youtu.be/jby4ePnSqo4?t=319>

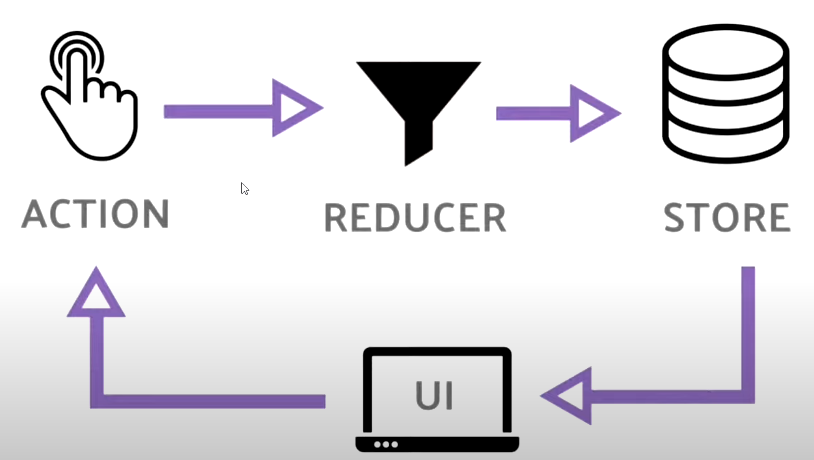
Redux – это библиотека, которая позволяет хранить данные. React и Redux – разные вещи. React отвечает за UI, Redux – за хранение данных.

Наша задача – хранить данные не в локальных в стейтах компонентов, а глобально, в централизованном хранилище. Затем эти данные будем передавать в те компоненты, которым они нужны.

В любом сайте есть данные, которые должны быть доступны на разных страницах. Причем при изменении данных на одной станице они должны сразу отображаться на другой странице. Данные пока у нас хранятся в локальных стейтах useState() и в другие компоненты мы передаем их через пропсы.

В Redux все данные хранятся в централизованном хранилище и передаются из единого центра в разные компоненты. Причем при изменении данных соответствующие компоненты должны перерисоваться, чтобы отобразить измененные данные.

Схема работы React и Redux:



Основные понятия:

* **action**: это действие «сделай что-то». Это объект, содержащий «что надо сделать» и дополнительные параметры «это «что-то» надо сделать вот «с этим»»
* Действие (action) передается в **редюсер**. Это «чистая функция», т.е. не меняет внешние данные. Редюсер аргументами получает текущее состояние (state) и action, в зависимости от типа действия модифицирует state и возвращает *новый* state. Новое значение необходимо, чтобы React отловил изменение данных.
* **хранилище** (store) – это и есть набор данных. При его изменении оно оповещает React об этом, а тот перерисовывает компонент.

Допустим, у нас есть компонент, который следит за изменениями в хранилище. При добавлении пиццы мы вызываем событие onClick(). Оно передает действие «добавить пиццу» вместе с дополнительными данными о новой пицце в редюсер. Редюсер анализирует тип действия - «добавить пиццу» - и помещает новую пиццу в хранилище. Изменение в хранилище передается в React и он в компоненте отображает новую пиццу. Причем изменения произойдут только в тех компонентах, которых мы сами укажем.

Объект action (действие) содержит обязательное поле **type**, идентифицирующее само действие, и, если нужно, свойство **payload** – объект с дополнительными данными. Например:

{

type: ‘ADD\_PIZZA’,

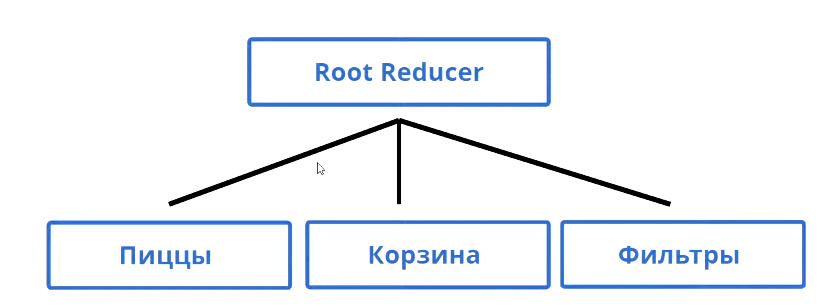
payload: { id: 2, name: ‘Пепперони’, size: 26, type: 0 }

}

Этот объект передается в редюсер. Тот проверяет: если type равен ‘ADD\_PIZZA’, то добавить объект, указанный в payload, в некоторый массив.

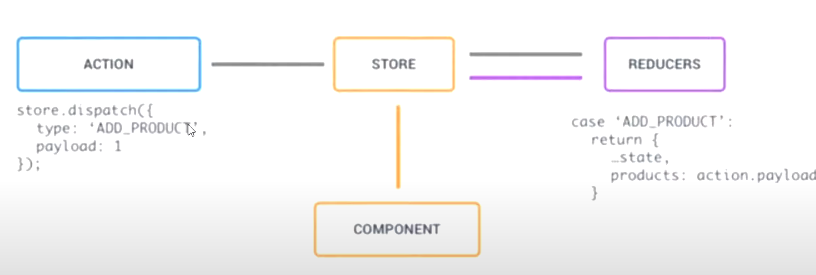
**RootReducer**

Мы можем создавать несколько редюсеров – для пицц, для корзины, для фильтров (м.б. еще клиенты, сотрудники, повара, заказы, доставка и пр.)



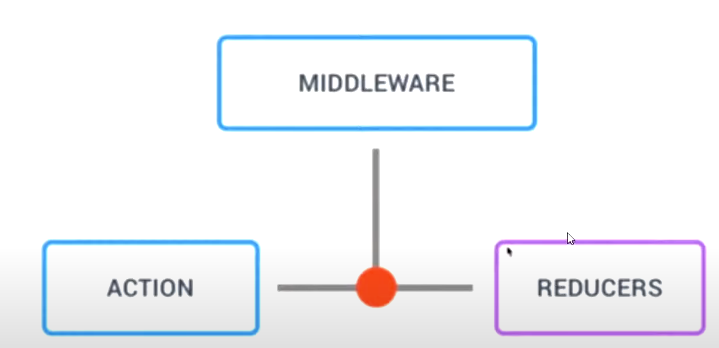
Из их всех создается т.н. «корневой редюсер». Для простых приложений его можно не создавать, но в случае сложных хранилищ из модификация будет очень сложной.

В нашем случае будет корневой редюсер, и он уже будет оповещать другие редюсеры. Action передается сначала в корневой редюсер, а затем по остальным редюсерам, и где-то обработается.



Допустим, мы даем команду «добавить продукт». Редюсер проверяет тип действия, и если в action type=’ADD\_PRODUCT’, то добавляет. Измененные данные возвращаются в store, а затем в компонент.

Помимо этой схемы есть понятие **middleware**. Это функции-посредники между action и reducer.



Если action говорит «сделай это», а редюсер говорит «я сделал» и возвращает данные, то middleware может перехватить эти данные и что-то сделать. Причем перехват может быть как до передачи действия в редюсер, так и при возврате данных из редюсера.

Например, можно реализовать логику «при добавлении пиццы отправь POST на сервер». Для этого можно написать функцию-посредник, которая еще до изменения в хранилище отправит запрос на сервер.

Чаще всего используется thunk. Его и будем использовать.

**Использование Redux в нашем приложении**

<https://youtu.be/jby4ePnSqo4?t=2322>

<https://redux.js.org/>

Установка:

npm install redux @reduxjs/toolkit react-redux

Общий принцип: <https://redux.js.org/tutorials/quick-start>

1. Создаем хранилище –store - с помощью configureStore()
2. Наше приложение оборачиваем компонентом <Provider store={store}> (react-redux)
3. Создаем «отрезки» хранилища – слайсы с редюсерами.
4. Добавляем слайсы в store
5. Для изменения стейта в компонентах:
   1. данные из стейта получаем через useSelector()
   2. получаем dispatch через useDispatch()
   3. отправляем команды на изменение стейта с помощью dispatch(*action*())

По видео будет чуток иначе, ~~я буду использовать reduxToolkit~~

Хранилище будет в папке src/redux.

Там создаем файл **store.js**. Там будет создаваться хранилище

Шаблон:

import { createStore } from 'redux'

const *reducer* = (state, action) => {

return state;

}

const store = createStore(*reducer*);

export default store;

И в index.js импортируем store и Provider (из react-redux) и помещаем App внутрь Provider:

<Provider store={store}>

<App />

</Provider>

Благодаря провайдеру не нужно передавать значения из хранилища в компоненты через пропсы.

Разделим логику redux на отдельные части: actions, reducers, store. Для этого создаем подпапки:

* reducers – редюсеры. Потом их соединим в один редюсер
* actions – там будут храниться действия

Задача: вынести логику переключения категорий пицц и сортировку в redux. Для этого создаем редюсер filters.js. Начальный стейт – initialState – будет содержать 2 свойства: category:null и sortBy: ‘popular’ – значения по умолчанию для индекса массива категорий, а также имя параметра сортировки. Хотя в Home пока для сортировки используется индекс массива, позднее будем передавать объекты.

сonst initialState = {

category: null, // по умолчанию «Все»

sortBy: 'popular' // один из ['popupar','alphabet','price']

}

Меняем в Home:

<SortPopup items={[{name: "популярности", type: 'popular'}, {name:"цене", type: 'price'}, {name: "алфавиту", type: 'alphabet'}]} />

И в SortPopup вывод меню сортировки:

* const [activeItem, setActiveItem] = useState('alphabet');
* <span onClick={togglePopupVisible}>{items[activeItem].name}</span>
* { items && items.map((obj, index) => <li key={obj.type} onClick={() => onSelectItem(index)} className={activeItem === index ? "active" : ""} >{obj.name}</li>) }

В редюсере filters реализуем логику переключения сортировки

const filters = (state = initialState, action) => {

switch (action.type) {

case 'SORT\_BY':

return { ...state, sortBy: action.payload };

default:

return state;

}

}

Это означает: При самом первом вызове в объекте state будут значения из intialState. Когда компонент вызовет dispatch() с action, у которого поле type = 'SORT\_BY', то нужно заменить в объекте state значение поля sortBy значением, переданное в action.payload (значение поля ‘type’ объекта сортировки - ‘popular’,’price’,’alphabet’). Например, можно передать объект action {type: ‘SORT\_BY’, payload:’price’}

Если же совпадения по action.type не найдено, редюсер обязательно должен возвращать текущий стейт без изменения – поэтому в switch/case должен быть оператор default с return текущего стейта.

Теперь нужно создать action creator’ы – это функции, которые возвращают объекты action {type, payload). Если есть payload, то креатор должен получать его аргументом.

Создаем файл src/redux/actions/filters.js, в котором будем создавать и экспортировать функции:

export const setSortBy = (sortBy) => ({ type: 'SORT\_BY', payload: sortBy });

export const setCategory = (catIndex) => ({ type: 'SET\_CATEGORY', payload: catIndex });

export const setSortOrder = (order) => ({ type: 'SET\_SORT\_ORDER', payload: order });

Теперь в редюсере filters хранятся только действия, связанные с фильтрацией.

<https://youtu.be/jby4ePnSqo4?t=4964>

Теперь редюсер filters надо поместить в хранилище store.

**Редюсер pizzas**

Выносим логику [pizzas]=useState() из App. Для этого:

1. создаем редюсер pizzas.js, в котором initialState={items: [], isLoaded: false}  
   isLoaded означает, что процесс загрузки с сервера завершен
2. В pizzas(state, action) если action.type=’SET\_PIZZAS’, то вернуть {…state, items:action.payload}
3. В src/redux/actions/pizzas.js создаем action creator setPizzas(pizzas), которая возвращает объект actions {type: ‘SET\_PIZZAS’, payload: pizzas):  
   export const setPizzas = (pizzas) => ({ type: 'SET\_PIZZAS', payload: pizzas });
4. Для обработки процесса загрузки пицц с сервера в initialState добавляем свойство isLoading: true

**Объединение редюсеров**

Для помещения редюсеров в хранилище store нужно создать корневой редюсер и прикрепить к нему редюсеры pizzas и filters. Для этого в store.js:

1. импортируем редюсеры pizzas и filters
2. с помощью combineRedicers() из ‘redux’ создаем корневой редюсер. Аргумент – объект; свойства – имена редюсеров, значения – импортированные значения редюсеров.

import { createStore, combineReducers } from 'redux'

import pizzas from './reducers/pizzas'

import filters from './reducers/filters'

const rootReducer = combineReducers({

  filters: filters,

  pizzas // т.к. имена свойств и значений совпадают, запись можно упростить

});

const store = createStore(rootReducer);

export default store;

Выносим блок импорта редюсеров и создания rootReducer в отдельный файл reducers/index.js

import { combineReducers } from 'redux'

import pizzas from './pizzas'

import filters from '.filters'

const rootReducer = combineReducers({

  filters: filters,

  pizzas // т.к. имена свойств и значений совпадают, запись можно упростить

});

export default rootReducer

store.js:

import { createStore } from 'redux'

import rootReducer from './reducers';

export default createStore(rootReducer);

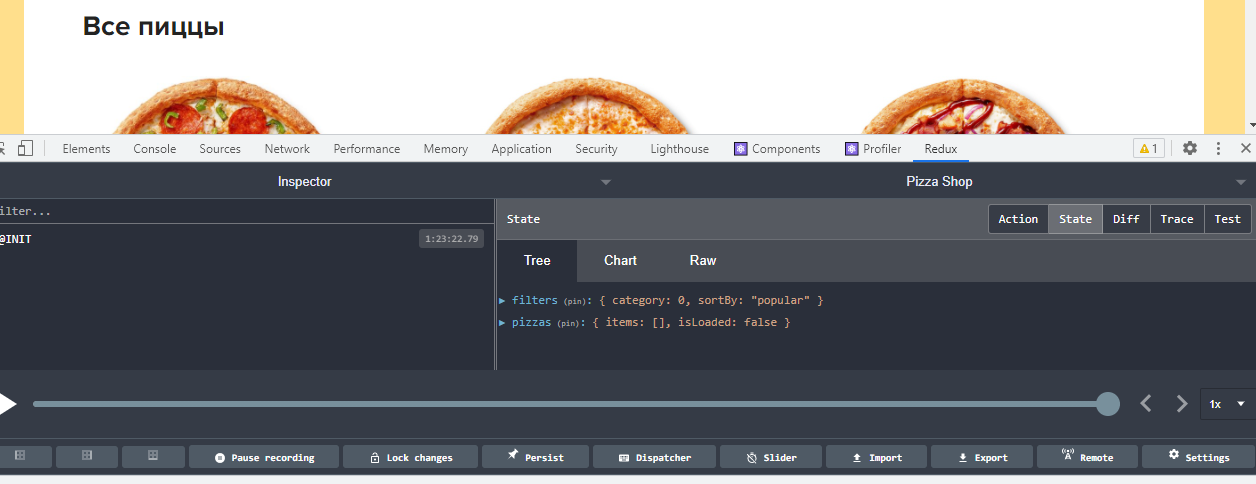
Для проверки можно поместить store в глобальный объект window и вызвать store.getState(). Видно, что там 2 стейта – filters и pizzas. Для изменения стейта – store.dispatch({type: SET\_SORT, payload: ‘price’}).

Для отладки в Chrome можно установить “**Redux Dev Tools**” из магазина. После установки включается по Ctrl+Shift+E.

Для его подключения в код надо в createStore() добавить второй аргумент:  
window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_\_ && window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_\_()

export default createStore(rootReducer,   
 window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_\_ && window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_\_()); // store

В браузере в консоли F12 появляется вкладка «Redux»



Теперь в store у нас два стейта (редюсера) – filters и pizzas.

Задача: брать список пицц из хранилища. Причем в Home пропсы передавать не будем.

В App() убираем useState() для [pizzas], но useEffect() оставляем.

Теперь надо передать в redux массив data.pizzas, полученный из axios.

В классовых компонентах мы бы использовали в методе класса componentDidMount() метод connect(mapStateToProps, mapDispatchToProps). Они передают dispatch и store в пропсы компонента и далее они доступны через this.props.xxxx – типа this.props.setPizzas(data.pizzas)

Но в функциональных компонентах код проще и производительнее. ФК не хранит методы жизненного цикла, которые могут не пригодиться. Желательно при разработке использовать ФК.

В функциональных компонентах с использованием хуков:

1. импортируем action setPizzas из файла redux/actions/pizzas
2. получаем dispatch из useDispatch().   
   useDispatch()() возвращает ссылку на функцию dispatch() текущего хранилища (store). С его помощью мы будем передавать action для взаимодействия с хранилищем.
3. После вызова axios вызываем dispatch(setPizzas(data.pizzas))
4. Для получения части хранилища используем хук useSelector(). Он возвращает часть хранилища, нужного нам (редюсер). Аргументом передается анонимная функция, которая принимает всё хранилище целиком и возвращает его часть. У нас в store 2 стейта – pizzas и filters - а нужен только pizzas, то используем useSelector(store => store.pizzas). В возвращенном объекте – стейте pizzas – есть нужный массив нам items[], которые можно передать пропсом в компонент <Home>.

import React, { useEffect } from 'react'

import { useDispatch, useSelector } from 'react-redux'

import axios from 'axios'

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

import { Header } from './components'

import Cart from './pages/Cart';

import Home from './pages/Home';

import { setPizzas } from './redux/actions/pizzas'

function App() {

//const [pizzas, setPizzas] = useState([]);

const **dispatch** = **useDispatch**();

const **pizzas** = **useSelector**(store => store.pizzas);

useEffect(() => {

axios.get('http://localhost:3000/db.json')

.then(({ data }) => **dispatch(setPizzas(data.pizzas))**);

}, [dispatch]);

return (

<div className="wrapper">

<Header />

<div className="content">

<Route exact path="/" render={() => <Home pizzas={**pizzas.items**} />} />

<Route exact path="/cart" component={Cart} />

</div>

</div>

);

}

export default App;

Т.е. если у нас rootReducer = combineReducers({ filters, pizzas })

то селектором можно получить отдельные части rootReducer, например, через useSelector((store) => store.filters).

Рекомендуется в useSelector() вытаскивать минимально необходимые для функционирования компонента данные, чтобы избежать лишней его перерисовки.

Например:

const pizzasItems = useSelector(store => store.pizzas.items)

…

<Route exact path="/" render={() => <Home pizzas={pizzasItems} />} />

Рендер App() происходит 2 раза:

1. при первом создании компонента
2. после получения данных с сервера после dispatch()

Общее правило:

* если компонент использует локальные данные, которые никуда больше не передаются и не влияют на другие компоненты, то лучше использовать локальный useState(), иначе – redux.
* Иногда лучше взять значение стейта в родительском компоненте и прокинуть в дочерний компонент пропсом. useSelector() должно быть по минимуму; лучше пробросить значения пропсами.
* В нашем простом приложении нет ничего плохого, если внутри компонентов <Categories> и <SortPopup> использовать useSelector(). Но все же лучше пропсами, т.к. может произойти ненужная перерисовка компонента.

**Рефакторинг Home**

* Из App не передаем в Home пропсы. Пишем просто  
  <Route exact path="/" component={Home} />
* Из Home убираем прием пропсов
* Переносим код useSelector() в Home:  
  const pizzas = useSelector(store => store.pizzas.items);
* Выводим в Home список пицц:  
  { pizzas.map(pizza => <PizzaBlock key={pizza.id} {...pizza} />) }

Почему useEffect() с setPizzas() в App, а не в Home? Потому что если бы он был в Home, то при каждом рендере Home отправлялся бы запрос на сервер (получение списка пицц). Например, при возврате из корзины в Home тот перерисовывается и снова запрос на сервер. Поэтому получение начальных надо делать в основном компоненте – App. Этот основной компонент должен получать данные с сервера, сохранять их в хранилище, а уже потом вызывать дочерние компоненты.

Поэтому в App.js получение данных осуществляется один раз – в useEffect(…,[])

<https://youtu.be/X3cNlZcaD9I?t=4028>

Но переключение категорий нет смысла хранить в Redux. Это локальное состояние, влияющее только на отображение списка пицц только внутри компонента Home. В Redux можно сохранить только активную категорию.

Задача: при выборе категории выбранная категория сохранялась в redux.

1. В Categories через пропсы передаем callback setActiveItem
2. В onSelectCategory() добавляем вызов setActiveItem()  
   Т.е. сначала сохраняем в локальном стейте, а затем передаем через callback. Помним, что null = “Все»
3. В Home импортируем action setCategory, создаем еще один dispatch = useDispatch()
4. В Home в вызов <Categories> добавляем пропс setActiveItem={(index) => dispatch(setCategory(index))}

Теперь в store значение ‘category’ меняется при кликах на категориях. Такой подход позволит в дальнйшем получать с сервера отфильтрованные данные.

Но при каждом изменении в Home, то всякий раз будет производится перерисовка компонента Categories и вызывать новый долгий запрос на сервер. Реакт смотрит на изменение ссылок в пропсах. Но анонимная функция в setActiveItem() меняется всякий раз при рендере, и компонент Category получает новые пропсы (хотя внутреннее содержание может остаться тем же) – это вызывает его перерисовку. Поэтому в пропсах нежелательно передавать анонимные функции, а создавать для этого отдельную функцию.

Правильнее так (в Home):

const onSelectCategory = (cat) => {

   dispatch(setCategory(cat));

}

<Categories items={[ …]} setActiveItem={onSelectCategory} />

В Categories, однако, onClick оставляем анонимную функцию – это не пропс, а обработчик события ‘click’ и на рендер не влияет.

Проблема: при получении списка пицц с сервера меняется стейт и это приводит к перерисовке Categories. Это надо исправить. Для этого:

* Перечень категорий, передаваемый пропсом items в Categories, выносим в константу-массив categoryNames перед функцией Home()
* Указываем <Categories items={categoryNames} setActiveItem={onSelectCategory} />

Теперь в items – константа. Даже если Home перерисуется, ссылка на categoryNames останется прежней.

Однако, onSelectCategory() находится внутри Home(), и после перерисовки Home создается новая ссылка, попадающая пропсом в Categories, что вызывает его перерисовку. Можно указать, что ссылка onSelectCategory не должна меняться. Для этого надо её «запомнить» с помощью хука React.useCallback(): помещаем код onSelectCategory() внутрь вызова useCallback():

 const onSelectCategory = React.useCallback((cat) => {

    dispatch(setCategory(cat));

  }, []);

См. <https://habr.com/ru/post/529950/>

*использование useCallback() выгодно только в случае передачи функции как props*

Мы уже добились того, что пропсы items и setActiveItem не меняются.

Чтобы закончить, код функции Categories() надо поместить внутрь **React.memo**():

const Categories = React.memo(({ items, setActiveItem }) => { … });

Если у нас есть много динамических данных, необходимо использовать мемоизацию с испольованием useCallback() и memo()

См. <https://habr.com/ru/company/plarium/blog/442116/>

<https://michael-pautov.medium.com/%D0%BA%D0%B0%D0%BA-%D0%B8%D1%81%D0%BF%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D1%82%D1%8C-react-memo-%D0%B8-usecallback-d6c2f134821>

Аналогично делаем для меню сортировки:

* Выносим массив объектов сортировок за пределы Home() в константу sortItems
* <SortPopup items={sortItems} />
* Помещаем код SortPopup внутрь React.memo():  
  export default React.memo(function SortPopup({ items }) { … } );

Теперь изменение сортировки не приводит к перерисовке компонентов Home и Categories.

React.memo() делает поверхностное сравнение пропсов – т.е. сравнивает ссылки. sortItems у нас константа за пределами компонентов, и ссылка на неё не меняется. Это аналог метода shouldComponentUpdate() в классовых компонентах.

**json-server**

<https://github.com/typicode/json-server>

json-server – тестовый backend без написания кода. Работа через файл json

Установка:

npm install –D json-server

(устанавливаем в devDependencies)

Файл db.json у нас уже есть в каталоге public – используем его

В package.json в секции scripts добавляем строку:

"server": "json-server --watch public/db.json --port 5000"

Запуск:

npm run server

или c командной строки:

npx json-server --watch public/db.json --port 5000"

Тогда получение пицц с сервера будет происходить по URL <http://localhost:5000/pizzas>, плюс получаем возможность фильтрации и сортировки прямо на сервере. Причем в консоли VSCode отображаются запросы.

Запуск приложения: сначала “npm run server”, а затем “npm start”

Теперь в Home меняем URL на <http://localhost:5000/pizzas>. Учитываем, что в data уже массив pizzas:

useEffect(() => {

    //axios.get('http://localhost:3000/db.json')

    axios.get('http://localhost:5000/pizzas')

      .then(({ data }) => {

        console.log('data =', data);

        dispatch(setPizzas(data))

      });

  }, [dispatch]);

Благодаря тому, что в redux хранится в стейте значение category, которое устанавливается в компоненте Categories, в компоненте App мы можем получить это значение и использовать для фильтрации списка пицц по этой категории (а также параметр сортировки).

<https://www.youtube.com/watch?v=Ix9iV72GMKg>

**Перенос обращения к бизнес-логики в Redux и Thunk.**

Для работы с сервером нам нужны 2 action: первый отправляет запрос на сервер и по завершении вызывает второй action, который меняет стейт. Поэтому в actions/pizzas.js добавляем новый action creator – fetchPizzas(). (обычно названия типа fetchXXX означают методы, которые получают данные с сервера).

Можно перенести код с axios прямо в fetchPizzas():

export const fetchPizzas = () => {

axios.get('http://localhost:5000/pizzas')

.then(({ data }) => setPizzas(data));

};

Но такой код не работает, т.к. такой action creator не возвращает объект типа action, а dispatch() и useDispatch() недоступен вне компонента.

Для решения этой проблемы необходима библиотека ‘**redux-thunk**’.

Вкратце – это middleware (функция-посредник), которая позволяет реализовать асинхронные actions.

<https://github.com/reduxjs/redux-thunk>

thunk нужен, чтобы из одного action creator можно было выполнить dispatch другого action creator. Это посредник (middleware) между store и action. Это функция; она проверяет: если action - простой объект, то возвращает этот объект. Если же action - функция (action creator), выполнит её и передаст аргумент dispatch; затем этот dispatch можно использовать для вызова другого action creator:

export const fetchPizzas = () => (**dispatch**) => {

axios.get('http://localhost:5000/pizzas')

.then(({ data }) => {

**dispatch**(setPizzas(data))

});

};

Зато action creator setPizzas() будет работать как обычно. Обычные action creator возвращают обычные объекты (с полями type, payload), а асинхронные – функции.

Чтобы этот код заработал, надо установить библиотеку redux-thunk

Установка:

npm install redux-thunk

Подключение (в store.js):

* импортируем thunk: import thunk from 'redux-thunk';
* в createStore() вторым параметром указываем applyMiddleware(thunk)

Проблема в том, что у нас есть еще один middleware - \_\_REDUX\_DEVTOOLS\_EXTENSION\_\_(). Они оба следят за нашими action.

Решения см. <https://github.com/zalmoxisus/redux-devtools-extension#installation> - п.1.2

import { applyMiddleware, compose, createStore } from 'redux'

import thunk from 'redux-thunk';

import rootReducer from './reducers';

const composeEnhancers = window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_COMPOSE\_\_ || compose;

export default createStore(

  rootReducer,

  composeEnhancers(applyMiddleware(thunk))

); // store

Здесь:

1. если в браузере установлен Redux dev tools, то в качестве «сборщика расширений» используем его, а иначе – функцию compose() из библиотеки redux.
2. С помощью compose() объединяем несколько middleware в одну функцию
3. С помощью applyMiddleware(thunk) – мы применяем посредника «thunk» к хранилищу – он будет использоваться на каждом action

Теперь в fetchPizzas() нужно сделать так, чтобы дождаться ответа от сервера и передать данные в redux.

Для этого её нужно превратить в асинхронный action creator

import axios from 'axios'

export const fetchPizzas = () => (dispatch) => {

  axios.get('http://localhost:5000/pizzas')

    .then(({ data }) => {

      dispatch(setPizzas(data))

    });

};

export const setPizzas = (pizzas) => ({ type: 'SET\_PIZZAS', payload: pizzas });

Таким образом мы указываем, что fetchPizzas() - асинхронный action creator. Это функция, которая возвращает другую функцию, к которой добавляет параметр dispatch.

В App.js:

function App() {

  const dispatch = useDispatch();

  useEffect(() => {

**dispatch(fetchPizzas());**

  }, []);

  return (

    <div className="wrapper">

      <Header />

      <div className="content">

        <Route exact path="/" component={Home} />

        <Route path="/cart" component={Cart} />

      </div>

    </div>

  );

}

Здесь в App мы используем fetchPizzas() как обычный action. Но т.к. она возвращает функцию, redux-thunk применяет к ней свою обработку.

В идеале создается файл api.js, содержащие «чистые» функции только взаимодействия с сервером, а в actions прописать их вызов. Это позволит легко менять URL, базы данных и пр.

**Фильтрация по категориям**

Задача: при кликам по категориям отправлять запрос на сервер и получать с него отфильтрованный список пицц (Все, Мясные и пр.). Этот запрос будет асинхронным. Причем пока запрос обрабатывается, мы должны что-то отобразить пользователю («Идет загрузка…» и т.п.)

Для перерисовки списка пицц при выборе категории в useEffect() можно указать массив зависимостей: при изменении каких значений будет происходить перерисовка компонента Home со списком пицц. Это filters.category и filters.sortBy.

Для этого сначала переносим код useEffect() с fetchPizzas() из App.js в Home.js. Код App.js сильно сокращается – только компоненты. Однако, здесь засада: при возврате из корзины в Home опять подгружаются пиццы с сервера. Т.к. при переходе Home -> Cart компонент Home удаляется, а при возврате создается заново. Для частичной решения этой проблемы в Home/useEffect() можно вставить проверку: загрузить список пицц сервера только тогда, когда их в store нет (количество = 0):

React.useEffect(() => {

    if (pizzas.length === 0) {

      dispatch(fetchPizzas());

    }

  }, []);

(хотя есть засада с сохранением фильтра и сортировки)

**Индикатор загрузки**

Задача: пока данные загружаются с сервера, показать «часики», а по завершении загрузки – скрыть их и отобразить нужный компонент. Для этого в стейте есть pizza.isLoaded. В action setPizzas() мы устанавливаем его в true, чтобы дать понять, что пиццы загружены.

Во время загрузки с сервера вместо блока { pizzas.map() } надо показать индикатор, используя isLoaded.

[*https://youtu.be/Ix9iV72GMKg?t=3394*](https://youtu.be/Ix9iV72GMKg?t=3394)

*Индикатор можно сделать с помощью* [*https://skeletonreact.com/*](https://skeletonreact.com/) *- в стиле facebook/Instagram. Но надо рисовать самим. Я не делал – использовал* [*https://www.npmjs.com/package/react-js-loader*](https://www.npmjs.com/package/react-js-loader)

const pizzas = useSelector(store => store.pizzas.items);

const isLoaded = useSelector(store => store.pizzas.isLoaded);

<div className="content\_\_items">

        {isLoaded

          ? pizzas.map(pizza => <PizzaBlock key={pizza.id} {...pizza} />)

          : <Loader type="spinner-circle" color="#000000" bgColor="#FFFFFF" />

        }

      </div>

В Home через useSelector() получаем переменные для категорий и сортировки:

const { category, sortBy } = useSelector(store => store.filters);

Передаем их в зависимости useState():

React.useEffect(() => {

    dispatch(fetchPizzas());

}, [category, sortBy]);

Тогда при изменении категории или сортировки будет вызываться fetchPizzas().

* В actions/pizzas меняем export const setLoaded = (val) => ({ type: 'SET\_LOADED', payload: val }) – чтобы устанавливалось значение isLoaded.
* В reducers/pizzas для ‘SET\_LOADED’: return { ...state, isLoaded: action.payload };

Этот action – setLoaded(false) – применяем в fetchPizzas() перед вызовом axios. А по завершении вызываем функцию setPizzas(), внутри которой устанавливаем isLoading=true

export const fetchPizzas = () => (dispatch) => {

  dispatch(setLoaded(false));

  axios.get('http://localhost:5000/pizzas')

    .then(({ data }) => {

      dispatch(setPizzas(data))

    });

};

Важно не просто вызывать action creator типа setLoaded() – надо её вызывать через dispatch(). Внутрь dispatch передается объект action.

Рефактор Categories:

* В пропсы добавляем activeCategory  
  const Categories = React.memo(({ **activeCategory**, items, setActiveItem }) => {
* удаляем стейт const [activeCategory, setActiveCategory] = useState(null);
* В onSelectCategory() удаляем вызов setActiveCategory(index)
* Из Home в <Categories> передаем пропс activeCategory={category}  
  *Это значение из селектора const { category, sortBy } = useSelector(store => store.filters)*
* Удаляем обработчик onSelectItem() и в onClick указываем onClick={() => setActiveItem(null)} (setActiveItem передается в пропсах).  
  Можно переименовать setActiveItem -> onClickCategory

В Categories можно дописать PropTypes:

Categories.propTypes = {

activeCategory: PropTypes.number,

items: PropTypes.arrayOf(PropTypes.string).isRequired,

setActiveItem: PropTypes.func

}

Categories.defaultProps = {

activeCategory: null,

items: []

}

Теперь смена категории управляется redux’ом. Причем при каждом клике список пицц перезагружается.

**Сортировка (SortPopup)**

* убираем стейт activeItem. Это индекс массива объектов items – {name, type}, передаваемых в пропсах из Home
* Из Home в SortPopup передаем пропс activeSortType={sortBy}. Причем sortBy – это не индекс массива sortItems (как было ранее), а значение поля type одного из объектов этого массива
* В Home создаем метод onSelectSortType () и передаем пропсом в SortPopup как onClickSortType={onSelectSortType}.  
  Сама функция:  
   const onSelectSortType = React.useCallback((sortName) => {  
   dispatch(setSortBy(sortName));  
   }, []);
* В SortPopup:
  + т.к. activeSortType – это строка, то для активной метки ищем в массиве по type:  
    <span onClick={togglePopupVisible}>{items.find(e => e.type === activeSortType).name}</span>  
    (лучше вынести во внешнюю переменную activeLabel)
  + В map() вывода меню сортировки меняем  
    className={obj.type === activeSortType ? "active" : ""}  
    onClick={() => onSelectItem(obj.type)}
  + В defaultProps добавляем activeSortType: 'popular' (на всякий случай – если

Теперь надо получать список пицц с сервера в зависимости от категории и сортировки.

json-server предоставляет возможность указывать параметры отбора и сортировки в URL. Например для запросов GET на адрес сервера http://localhost:5000:

* /pizzas – получить все объекты
* /pizzas/1 – получить объект с id=1
* /pizzas?category=1 – фильтр по полю category (для строк – учитывается регистр)
* /pizzas?\_sort=price&\_order=asc – сортировка по полю price в возрастающем порядке (\_order=desc – в убывающем)
* /pizzas?q=*строка\_поиска* – полнотекстовый поиск по заданной строке

Подробнее см. <https://www.npmjs.com/package/json-server>

**Сортировка и фильтрация на сервере**

Задача: при выбора пункта сортировки должен отправиться запрос на сервер json, тот должен отсортировать данные в соответствии с нашим выбором, и вернуть ответ.

В общем виде запрос на сервер выглядит как http://localhost:5000/pizzas?category=*категория&*\_sort=*ти&order=desc*.   
Где *категория* – индекс массива categoryNames[] из Home, *тип* – одно из ‘popular’, ‘alphabet’, ‘price’,’ rating’. Причем если *категория* = null, то вывести *все* пиццы.

Для этого в fetchPizzas() надо передать аргументы :

React.useEffect(() => {

    dispatch(fetchPizzas(category, sortBy));

  }, [category, sortBy]);

В redux/actions/filters.js, в fetchPizzas() принимаем эти аргументы:

export const fetchPizzas = (category, sortBy) => (dispatch) => { … }

Теперь fetchPizzas() при каждом изменении категории или пункта сортировки будет принимать аргументы типа category=1 (или null) и sortBy=’alphabet’

Теперь надо изменить URL обращения к серверу и указать наши параметры. Для категорий это выглядит так:

if(category != null) {

options += `?category=${category}`;

}

axios.get(`http://localhost:5000/pizzas?${options}`)

В категории может быть null или число, в sortBy – строка, в sortOrder = asc|desc.

Для порядка сортировки:

* в пропсы activeSortOrder и onClickSortOrder
* В SortPopup создаем обработчик const onSelectOrder = () => { onClickSortOrder(); } и используем его в onClick на иконке: onClick={onSelectOrder}
* В Home создаем функцию const onSelectSortOrder = React.useCallback(() => { dispatch(setSortOrder()); }, []); и передаем пропсом как onClickSortOrder
* В actions/filters - export const setSortOrder = () => ({ type: 'SWITCH\_SORT\_ORDER' });
* В редюсере для 'SWITCH\_SORT\_ORDER' : return { ...state, sortOrder: state.sortOrder === 'asc' ? 'desc' : 'asc' };

fetchPizzas() выглядит так:

export const fetchPizzas = (category, sortBy, sortOrder = "desc") => (dispatch) => {

  console.log(` fetchPizzas(): cat=${category} sortBy=${sortBy}`);

  dispatch(setLoaded(false));

  let options = "";

  if (category != null) {

    options += `category=${category}`;

    options += "&";

  }

  if (sortBy === 'price') {

    options += "\_sort=price";

  } else if (sortBy === 'rating') {

    options += '\_sort=rating';

  } else if (sortBy === 'alphabet') {

    options += "\_sort=name"

  }

  options += `&\_order=${sortOrder}`;

  axios.get(`http://localhost:5000/pizzas?${options}`)

    .then(({ data }) => {

      dispatch(setPizzas(data))

    });

};

Код можно упростить, если в массиве Home/sortItems[] привести названия полей type к именам полей в БД (именам свойств объектов pizzas):

const sortItems = [

  { name: "популярности", type: 'rating' },

  { name: "цене", type: 'price' },

  { name: "алфавиту", type: 'name' }

];

Также указать в начальных значениях стейта ‘rating’ вместо ‘popular’.

Тогда в fetchPizzas() можно сильно сократить:

export const fetchPizzas = (category, sortBy, sortOrder = "desc") => (dispatch) => {

  console.log(` fetchPizzas(): cat=${category} sortBy=${sortBy}`);

  dispatch(setLoaded(false));

  let options = "";

  if (category != null) {

    options += `category=${category}`;

    options += "&";

  }

  options += `\_sort=${sortBy}&\_order=${sortOrder}`;

  axios.get(`http://localhost:5000/pizzas?${options}`)

    .then(({ data }) => {

      dispatch(setPizzas(data))

    });

};

Также можно использовать setSortBy(type, order) или (лучше) sortBy({type, order}), чтобы использовать единый action.payload

Т.к. при кликах на категории и сортировку у нас меняются значения [category, sortBy, sortOrder] – зависимости useEffect() в Home – то это приведет к вызову fetchPizzas() и перезапрос данных с сервера.

<https://youtu.be/Ix9iV72GMKg?t=8559>

Все работает – и выбор категорий, и сортировка.

*Для подсветки компонентов, которые перерисовываются, в консоли разработчка Chrome F12, вкладка Components, кнопка «настройки», включить “Highlight updates when component render”*

Выбор толщины теста и размера у нас через локальные стейты, а не через redux

<https://www.youtube.com/watch?v=IZVNNc8FHYg>

Можно подискутировать, нужно ли каждый раз обращаться на backend за данными или загрузить их сначала все, а потом сортировать на фронтенде. С backend мы получаем всегда актуальные данные, но страдает производительность. Если же обработка на фронтенде, то можем в случае магазина столкнуться с тем, что заказанный товар уже закончился в СУБД.

Итак, на данный момент реализовано: загрузка списка пицц с сервера, реализована фильтрация и сортировка. Осталось реализовать корзину.

**Корзина**

В файл src/pages/Cart.jsx переносим в него часть кода верстки из cart.html. Она должна отображаться внутри App вместо Home, внутри div.container. Поэтому в Cart.jsx переносим содержимое <div class="container container--cart">, затем меняем “class” на ‘className’, а также имена свойств с дефисами преобразовываем camelCase (stroke-width => strokeWidth, stroke-line-join => strokeLinejoin и т.п.). Пока не исчезнут ошибки.

В redux мы должны хранить массив заказов – заказанные пиццы, их количество и общая сумма заказа. Сумма и количество должны отображаться на кнопке «Корзина»:



Причем не стоит каждый раз вычислять эти значения при рендере Header. Например, нельзя для отображения суммы каждый раз вызывать Array.reduce(). Правильно хранить и рассчитывать эти значения в store (точнее, в редюсере).

Мы сделаем так: в стейте корзины будет объект items {}, у которого свойства – id пиццы, а значение – массив объектов пицц (*нахуя?*)

Создаем в src/reducers/cart.js редюсер для корзины. Начальный стейт будет:

const initalState = {

items: {},

totalPrice: 0,

totalCount: 0

}

*Для названия actions (значения acton.type – типа, ‘SET\_PIZZAS’) рекомендуется использовать константы.*

Реализуем в редюсере cart ‘SET\_TOTAL\_PRICE’ и ‘SET\_TOTAL\_COUNT’ (*на самом деле не нужно – см. ниже*):

const initalState = {

items: {},

totalPrice: 0,

totalCount: 0

}

const cart = (state = initalState, action) => {

switch (action.type) {

case 'SET\_TOTAL\_PRICE':

return { ...state, totalPrice: action.payload };

case 'SET\_TOTAL\_COUNT':

return { ...state, totalCount: action.payload };

default:

return state;

}

}

export default cart;

В src/redux/actions/cart.js создаем функции action creator (*тоже не нужно*):

export const setTotalPrice = (price) => ({ type: 'SET\_TOTAL\_PRICE', payload: price });

export const setTotalCount = (count) => ({ type: 'SET\_TOTAL\_COUNT', payload: count });

Импортируем reducers/cart.js в reducers/index.js и добавляем cart в вызов combineReducers():

import pizzas from './pizzas.js'

import filters from './filters.js'

import cart from './cart'

const rootReducer = combineReducers({ filters, pizzas,  cart });

Теперь на кнопке «Корзина» надо отображать данные стейта totalPrice и totalCount. Эта кнопка находится в компоненте Header.

Для этого в Header():

1. получаем totalPrice и totalCount через useSelector():  
   const {totalPrice, totalCount} = useSelector(store => store.cart);
2. В <span>, выводящих количество и сумму, выводим <span>{totalPrice} ₽</span> и <span>{totalCount}</span>

Мы указали, что из всего стейта cart нам надо следить за свойствами totalPrice и totalCount. Только при их изменении произойдет перерисовка Header.

Через useSelector() следует отбирать только нужные данные.

<https://youtu.be/IZVNNc8FHYg?t=2598>

На самом деле actions SET\_TOTAL\_PRICE и SET\_TOTAL\_COUNT не нужны, т.к. totalPrice и totalCount будут пересчитываться при добавлении каждой пиццы в корзину (в объект items стейта cart).

**Реализация добавления пиццы в корзину**

В src/actions/cart.js добавляем action добавления пиццы в корзину:

export const addPizzaToCart = (pizza) => ({ type: 'ADD\_TO\_CART', payload: pizza });

В редюсере для ‘ADD\_TO\_CART’:

    case 'ADD\_TO\_CART':

      return {

         ...state,

         items: { [action.payload.id]: [...state.items[action.payload.id], action.payload] }

   }

Здесь:

* через action.payload передается объект добавляемой в корзину пиццы {name, price, catgory,…
* В стейте cart, в объекте items его ключи = id пицц, а значения – массивы пицц с этим id.
* Если в объекте items нет свойства с id пиццы, то оно будет добавлено.
* В массив items[] добавляется объект пиццы

В объекте items в массивах будут храниться модифицированные версии объекта pizzas с json-сервера. Состав каждого объекта массива:

* id
* name
* imageUrl
* type – тут вместо массива types - текстовое представление толщины теста («традиционное», «тонкое»)
* size – выбранный размер в см. Число.
* price

Например, если у нас есть пицца {id: 1, name: “Сырная”}, и мы в корзину добавили 2 штуки, то стейт cart будет выглядеть приблизительно так:

state = {

{

1: [{id: 1, name: “Сырная”, …}, {id:1, name: “Сырная”, …}],

3: [{id: 3, name: “Пепперони с перцем”, price:…}]

},

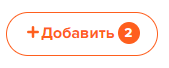
totalPrice: 1250,

totalCount: 3

}

(из-за такой сложности модификации сложного стейта лучше использовать библиотеку immer <https://immerjs.github.io/immer/> или вместо redux – MobX.

**Реализация кнопки добавления в корзину**



Она находится в PizzaBlock.jsx в элементе <div className="button button--outline button--add">

Преобразуем его в компонент Button, созданный ранее в Button.jsx:

<Button className="button--add" outline>

(классы button и button-outline применяются в Button, причем ‘button-outline’ применяется если пропс outline=true)

Можно в PizzaBlock использовать useSelector() и useDispatch(), но при этом в Home на каждый блок PizzaBlock создается новый dispatch. Т.е. если на экране отображается 10 пицц, то будет 10 dispatch. Поэтому лучше создать dispatch в Home и передавать его в PizzaBlock через пропсы. Тем более что в Home уже есть dispatch. Однако, если есть много вложенных компонентов, то придется пробрасывать пропс через все дерево компонентов.

Итак:

* в PizzaBlock добавляем пропсы id и onAddPizza:  
   function PizzaBlock({ id, imageUrl, name, types, sizes, price, **onAddPizza** }) {
* Передаем пропсом:  
  pizzas.map(pizza => <PizzaBlock key={pizza.id} {...pizza} onAddPizza={onAddPizza} />)
* В компоненте Button добавляем (если нет) в пропсы onClick:  
   export default function Button({ onClick, className, outline, children }) {  
   ...  
   <button onClick={onClick} … >
* В Button onClick вызываем onAddPizza() и аргументом передаем объект, состоящий из нескольких свойств. Причем вместо sizes и types указываем выбранные типы из локальных стейтов:  
   <Button onClick={() => onAddPizza({  
   id, name, imageUrl, type: testo[currentType], size: currentSize, price  
   })}
* В PizzaBlock для отладки выводим этот объект:   
  <PizzaBlock onAddPizza={(obj) => console.log(obj)} …/>

В консоли должно быть что-то подобное:

id: 0

imageUrl: "https://dodopizza.azureedge.net/static/Img/Products/f035.... "

name: "Пепперони Фреш с перцем"

price: 803

size: 2

type: "традиционное"

Теперь мы получили пиццу по клику, и созданный объект надо передать в redux.

Использование анонимных функций типа <Button onClick={() => onAddPizza(…)> плохо. Всегда, если есть возможность, следует использовать ссылки.

* Пишем <Button onClick={handleAddPizza}
* Создаем handleAddPizza:

const handleAddPizza = () => {

    onAddPizza({

      id, name, imageUrl, type: testo[currentType], size: currentSize, price

    })

  }

...

<Button onClick={handleAddPizza} .../>

При использовании анонимных функций происходит лишняя перерисовка компонента. Однако, это иногда необходимо – например, в случаях onSelectType() и onSelectSize() – потому что они внутри map().

В корзине для каждой пиццы отображаются:



* наименование (price)
* количество
* общая стоимость текущего вида пиццы с учетом количества (вместо цены)
* тип (толщина теста) - строка
* размер (число)
* Кнопка удаления всего вида текущей пиццы

Поэтому в onAddPizza() передаем соответствующий объект (см. выше), он попадет в стейт cart.items

В Home:

const onAddPizza = React.useCallback((pizza) => {

    dispatch(addPizzaToCart(pizza));

    // eslint-disable-next-line react-hooks/exhaustive-deps

  }, []);

…

<div className="content\_\_items">

        {isLoaded

          ? pizzas.map(pizza => <PizzaBlock onAddPizza={onAddPizza} key={pizza.id} {...pizza} />)

          : <Loader type="spinner-circle" color="#000000" bgColor="#FFFFFF" />

        }

Однако, это выдаст ошибку, что “state.items[action.payload.id] is not iterable” – потому что объект items изначально пустой и js не знает, что значение его свойства – массив. Т.е. мы пытаемся сделать […undefined].

Для решения этой проблемы:

* проверяем – есть ли в state.items указанный id (state.items[action.payload.id].
  + Если нет, создаем новый массив.
  + Если есть – добавляем объект в этот массив.
  + Не забываем, что надо сохранить предыдущие значения state.items, поэтому производим также его деструктурирование

<https://youtu.be/IZVNNc8FHYg?t=5279>

    case 'ADD\_TO\_CART':

      const id = action.payload.id;

      return ({

        ...state,

        items: {

          ...state.items,

          [id]: !state.items[id]  ? [action.payload] : [...state.items[id], action.payload]

        }

      })

Теперь в state.items хранятся массивы пицц с разными параметрами: для одного и того же id могут быть разные size и type.

**Подсчет количества и суммы.**

Если добавить к редюсеру расчет количества пицц в виде

return { ...state, items: { … }, totalCount: state.items.length}

то это не будет работать, т.к. state.items.length обращается к старому стейту, по состоянию до добавления пиццы в стейт.

Для подсчета количества пицц надо подсчитать количество их id – через Object.keys(state.items).length

Поэтому надо сначала создать *новый* объект items и уже из него рассчитывать totalCount:

const newItems = {

...state.items,

[id]: !state.items[id] ? [action.payload] : [...state.items[id], action.payload]

}

Вычисление totalCount:

* получаем массив массивов объектов пицц в стейте items с помощью Object.values(newItems)
* Превращаем двумерный массив в одномерный с помощью flat() или через Array.concat()

Тогда return выглядит так:

return { ...state,

items: newItems,

**totalCount: [].concat.apply([], Object.values(newItems).length**

}

Для расчета цены totalPrice:

* аналогично расчету totalCount «сворачиваем» state.items в одномерный массив
* Через reduce() суммируем поля price каждого объекта:  
  totalPrice: [].concat.apply([], Object.values(newItems).reduce((sum, v) => sum+v.price,0)

|  |
| --- |
| *Я сделал проще:*  *case 'ADD\_TO\_CART':*  *const id = action.payload.id;*  *return {*  *...state,*  *items: {*  *...state.items,*  *[id]: !state.items[id] ? [action.payload] : [...state.items[id], action.payload]*  *},*  ***totalCount: state.totalCount + 1,***  ***totalPrice: state.totalPrice + action.payload.price***  *}* |

Итак, редюсер для ‘ADD\_CART’ выглядит так:

case 'ADD\_TO\_CART':

      const id = action.payload.id;

      const newItems = {

        ...state.items,

        [id]: !state.items[id] ? [action.payload] : [...state.items[id], action.payload]

      };

      // вычисление общего количества и цены

      const itemsValues = Object.values(newItems).flat(); // массив массивов объектов пицц

      const totalCount = itemsValues.length;

      const totalPrice = itemsValues.reduce((sum, item) => sum + item.price, 0);

      return {

        ...state,

        items: newItems,

        // totalCount: state.totalCount + 1,

        // totalPrice: state.totalPrice + action.payload.price

        totalCount, totalPrice

      }

Т.к. мы state.items – объект, то это облегчает формирование значения на кнопке добавления:



В Home:

* получаем cart.items с помощью useSelector():  
  const cartItems = useSelector(store => store.cart.items);
* В PizzaBlock пропсом передаем не весь объект, а только количество данного вида пиццы, находящегося в корзине (по id). Для этого в PizzaBlock передаем пропс countInCart и отображаем в теге <i>: <i>{countInCart}</i>
* В Home передаем пропс  
  countInCart={cartItems[pizza.id] ? cartItems[pizza.id].length : 0}  
  т.е. если в корзине есть пиццы с таким id, взять их количество (длину массива), а если нет – 0
* Также добавляем в PizzaBlock propTypes: countInCart: PropTypes.number

**Корзина – Cart.jsx**

Необходимо реализовать функционал:

* отображение заказанных пицц из корзины
* Кнопки изменения количества, причем с пересчетом цены
* Кнопка удаления
* Отображение общего количество и общей суммы заказа.

*Про immer см.* [*https://www.youtube.com/watch?v=\_gqdE6pjhWU*](https://www.youtube.com/watch?v=_gqdE6pjhWU)

На кнопке добавления отображаемое число = число пицц выбранного вида без группировки по категории и тесту. Но в корзине – с группировкой по категории и тесту, т.к. они отображаются под названием пицц.

<https://youtu.be/hCMnDgOmURQ?t=89>

**Отображение списка пицц в Корзине**

В Cart.jsx пока пиццы выводятся статически в <div className="cart\_\_item">. Выносим его в отдельный компонент CartItem.jsx, в Cart.jsx вместо нескольких <div className="cart\_\_item"> указываем <CartItem />.

Добавляем экcпорт/импорт в components/index.js

Из Cart в CartItem передаем пропсы: id, name, type, size, imageUrl

В CartItem принимаем их и вставляем в разметку где нужно.

Теперь в Cart надо взять значение стейта cart.items:

const { totalCount, totalPrice, items } = useSelector(store => store.cart);

Вставляем {totalCount} и {totalPrice} в соответствующие <span> в Cart.jsx

Также надо подправить в <div className="cart\_\_bottom-buttons"> ссылку <a> - при клике на неё страница “/” перезагружается и стейт обнуляется.

Для этого делаем по аналогии с Header:

* импортируем в Cart import { Link, Route } from 'react-router-dom'
* Вместо тега <a> используем <div> с такими же стилями, а внутри - <Link to=”/”>:

<div className="cart\_\_bottom-buttons">

  {/\* <a href="/" className="button button--outline button--add go-back-btn"> \*/}

  <Link to="/">

   <div className="button button--outline button--add go-back-btn">

    <svg width="8" …  </svg>

      <span>Вернуться назад</span>

   {/\* </a> \*/}

  </div>

 </Link>  
</div>

Отображение списка:

В блоке {} выводим Object.values(items}.forEach( ({ … деструктуриуем объект items …}) => <CartItem … />

|  |
| --- |
| Я сделал с группировкой:  const groupedItems = [];  Object.values(items).flat().forEach(item => {  let outObj = groupedItems.find(obj => obj.id === item.id && obj.type === item.type && obj.size === item.size);  if (!outObj) {  groupedItems.push({ ...item, count: 1, price: item.price });  } else {  outObj.count++;  }  })  И далее  { groupedItems.map(({ id, name, type, size, imageUrl, count, price }) =>  <CartItem key={id} id={id}  *(наверное, для key лучше использовать значение index из map)*  Тогда в корзине отображается количество пицц каждого вида. |

Тормознулся на <https://youtu.be/hCMnDgOmURQ?t=2156>

Мой вариант:

1. В пропсы в CartItem передаем:  
   export default function CartItem({ id, name, type, size, imageUrl, count, price, onAddPizza, onRemovePizza, onDelete })
2. Для изменения количества пицц: В Cart создаем обработчик const handleRemovePizza = React.useCallback((pizza) => { dispatch(removePizzaFromCart(pizza)); }, []), обработчик handleRemovePizza() и обработчик удаления всей строки в корзине handleDelete().
3. В редюсере cart реализуем actions для уменьшения количества пицц и для удаления всей строки:

case 'REMOVE\_FROM\_CART': // decrease count of pizzas in cart

const { id, type, size, price } = action.payload;

const removeIndex = state.items[id].findIndex(item => item.type === type && item.size === size);

if (removeIndex < 0) {

return state;

}

const newArray = [...state.items[id].slice(0, removeIndex), ...state.items[id].slice(removeIndex + 1)];

const newItems = {

...state.items,

[id]: newArray

};

return { ...state, items: newItems, totalCount: state.totalCount - 1, totalPrice: state.totalPrice - price };

case 'DELETE\_ORDER': // delete a row from cart (same pizza with id, type, size)

{

const { id, type, size } = action.payload;

const newArray = state.items[id].filter(order => !(order.id === id && order.type === type && order.size === size))

const newItems = { ...state.items, [id]: newArray };

if (newItems[id].length === 0) {

delete newItems[id];

}

// пересчитать totalCount

const totalCount = Object.values(newItems).flat().length;

const totalPrice = Object.values(newItems).flat().reduce((sum, item) => sum + item.price, 0);

return { ...state.items, items: newItems, totalCount, totalPrice };

}

В CartItem общая стоимость выводится как {count\*price}.

К нужным div добавляем onClick:

* (+):onClick={() => onAddPizza({ id, name, type, size, imageUrl, count, price })}>
* (-):onClick={() => onRemovePizza({ id, name, type, size, price })}>
* (x): onClick={() => onDelete({ id, type, size, price })}
* корзина: onClearCart() – работает только в Cart

В редюсер добавляем обработку action ‘CLEAR\_CART’ как return {…initialState}.

Cart.jsx:

import React from 'react'

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

import { useSelector, useDispatch } from 'react-redux'

import CartItem from '../components/CartItem'

import { addPizzaToCart, removePizzaFromCart, deleteOrder, clearCart } from '../redux/actions/cart'

export default function Cart() {

  const { totalCount, totalPrice, items } = useSelector(store => store.cart);

  // группировка списка по type и size

  const groupedItems = [];

  Object.values(items).flat().forEach(item => {

    let outObj = groupedItems.find(obj => obj.id === item.id && obj.type === item.type && obj.size === item.size);

    if (!outObj) {

      groupedItems.push({ ...item, count: 1, price: item.price });

    } else {

      outObj.count++;

      outObj.totalPrice += item.price;

    }

  });

  // обработчики

  const dispatch = useDispatch();

  // увеличение количества

  const handleAddPizza = React.useCallback((pizza) => {

    dispatch(addPizzaToCart(pizza));

    // eslint-disable-next-line react-hooks/exhaustive-deps

  }, []

  );

  // уменьшение количества

  const handleRemovePizza = React.useCallback((pizza) => {

    dispatch(removePizzaFromCart(pizza));

    // eslint-disable-next-line react-hooks/exhaustive-deps

  }, []);

  // обработчик кнопки "удалить всю строку"

  const handleDelete = React.useCallback((pizza) => {

    dispatch(deleteOrder(pizza));

    // eslint-disable-next-line react-hooks/exhaustive-deps

  }, []);

  // Очистка всей корзины

  const handleClearCart = React.useCallback((pizza) => {

    dispatch(clearCart());

    // eslint-disable-next-line react-hooks/exhaustive-deps

  }, []);

  return (

    <div className="container container--cart">

      <div className="cart">

        <div className="cart\_\_top">

          <h2 className="content\_\_title"><svg width="18" height="18" viewBox="0 0 18 18" fill="none" xmlns="http://www.w3.org/2000/svg">

            <path d="M6.33333 16.3333C7.06971 16.3333 7.66667 15.7364 7.66667 15C7.66667 14.2636 7.06971 13.6667 6.33333 13.6667C5.59695 13.6667 5 14.2636 5 15C5 15.7364 5.59695 16.3333 6.33333 16.3333Z" stroke="white" strokeWidth="1.8" strokeLinecap="round" strokeLinejoin="round" />

            <path d="M14.3333 16.3333C15.0697 16.3333 15.6667 15.7364 15.6667 15C15.6667 14.2636 15.0697 13.6667 14.3333 13.6667C13.597 13.6667 13 14.2636 13 15C13 15.7364 13.597 16.3333 14.3333 16.3333Z" stroke="white" strokeWidth="1.8" strokeLinecap="round" strokeLinejoin="round" />

            <path d="M4.78002 4.99999H16.3334L15.2134 10.5933C15.1524 10.9003 14.9854 11.176 14.7417 11.3722C14.4979 11.5684 14.1929 11.6727 13.88 11.6667H6.83335C6.50781 11.6694 6.1925 11.553 5.94689 11.3393C5.70128 11.1256 5.54233 10.8295 5.50002 10.5067L4.48669 2.82666C4.44466 2.50615 4.28764 2.21182 4.04482 1.99844C3.80201 1.78505 3.48994 1.66715 3.16669 1.66666H1.66669" stroke="white" strokeWidth="1.8" strokeLinecap="round" strokeLinejoin="round" />

          </svg>

            Корзина</h2>

          <div className="cart\_\_clear" onClick={handleClearCart}>

            <svg width="20" height="20" viewBox="0 0 20 20" fill="none" xmlns="http://www.w3.org/2000/svg">

              <path d="M2.5 5H4.16667H17.5" stroke="#B6B6B6" strokeWidth="1.2" strokeLinecap="round" strokeLinejoin="round" />

              <path d="M6.66663 5.00001V3.33334C6.66663 2.89131 6.84222 2.46739 7.15478 2.15483C7.46734 1.84227 7.89127 1.66667 8.33329 1.66667H11.6666C12.1087 1.66667 12.5326 1.84227 12.8451 2.15483C13.1577 2.46739 13.3333 2.89131 13.3333 3.33334V5.00001M15.8333 5.00001V16.6667C15.8333 17.1087 15.6577 17.5326 15.3451 17.8452C15.0326 18.1577 14.6087 18.3333 14.1666 18.3333H5.83329C5.39127 18.3333 4.96734 18.1577 4.65478 17.8452C4.34222 17.5326 4.16663 17.1087 4.16663 16.6667V5.00001H15.8333Z" stroke="#B6B6B6" strokeWidth="1.2" strokeLinecap="round" strokeLinejoin="round" />

              <path d="M8.33337 9.16667V14.1667" stroke="#B6B6B6" strokeWidth="1.2" strokeLinecap="round" strokeLinejoin="round" />

              <path d="M11.6666 9.16667V14.1667" stroke="#B6B6B6" strokeWidth="1.2" strokeLinecap="round" strokeLinejoin="round" />

            </svg>

            <span>Очистить корзину</span>

          </div>

        </div>

        <div className="content\_\_items">

          {groupedItems.map(({ id, name, type, size, imageUrl, count, price }, index) =>

            <CartItem

              key={index}

              id={id}

              name={name}

              type={type}

              size={size}

              imageUrl={imageUrl}

              count={count}

              price={price}

              onAddPizza={handleAddPizza}

              onRemovePizza={handleRemovePizza}

              onDelete={handleDelete}

            />

          )}

        </div>

        <div className="cart\_\_bottom">

          <div className="cart\_\_bottom-details">

            <span> Всего пицц: <b>{totalCount} шт.</b> </span>

            <span> Сумма заказа: <b>{totalPrice} ₽</b> </span>

          </div>

          <div className="cart\_\_bottom-buttons">

            {/\* <a href="/" className="button button--outline button--add go-back-btn"> \*/}

            <Link to="/">

              <div className="button button--outline button--add go-back-btn">

                <svg width="8" height="14" viewBox="0 0 8 14" fill="none" xmlns="http://www.w3.org/2000/svg">

                  <path d="M7 13L1 6.93015L6.86175 1" stroke="#D3D3D3" strokeWidth="1.5" strokeLinecap="round" strokeLinejoin="round" />

                </svg>

                <span>Вернуться назад</span>

                {/\* </a> \*/}

              </div>

            </Link>

            <div className="button pay-btn">

              <span>Оплатить сейчас</span>

            </div>

          </div>

        </div>

      </div>

    </div>

  )

}

Редюсер reducers/cart:

const initalState = {

  items: {},

  totalPrice: 0,

  totalCount: 0

}

const cart = (state = initalState, action) => {

  switch (action.type) {

    case 'ADD\_TO\_CART':

      {

        const id = action.payload.id;

        const newItems = {

          ...state.items,

          [id]: !state.items[id] ? [action.payload] : [...state.items[id], action.payload]

        };

        // вычисление общего количества и цены

        const itemsValues = Object.values(newItems).flat(); // массив массивов объектов пицц

        const totalCount = itemsValues.length;

        const totalPrice = itemsValues.reduce((sum, item) => sum + item.price, 0);

        return { ...state, items: newItems, totalCount, totalPrice };

      }

    case 'REMOVE\_FROM\_CART': // уменьшить количество пицц

      {

        const { id, type, size, price } = action.payload;

        const removeIndex = state.items[id].findIndex(item => item.type === type && item.size === size);

        if (removeIndex < 0) {

          return state;

        }

        const newArray = [...state.items[id].slice(0, removeIndex), ...state.items[id].slice(removeIndex + 1)];

        const newItems = { ...state.items, [id]: newArray };

        return { ...state, items: newItems, totalCount: state.totalCount - 1, totalPrice: state.totalPrice - price };

      }

    case 'DELETE\_ORDER': // удалить всю строку из  Корзины (пицца с id, type, size)

      {

        const { id, type, size } = action.payload;

        const newArray = state.items[id].filter(order => !(order.id === id && order.type === type && order.size === size))

        const newItems = { ...state.items, [id]: newArray };

        // если массив полностью пустой, то удалить items.id

        if (newItems[id].length === 0) {

          delete newItems[id];

        }

        // пересчитать totalCount и totalPrice

        const totalCount = Object.values(newItems).flat().length;

        const totalPrice = Object.values(newItems).flat().reduce((sum, item) => sum + item.price, 0);

        return { ...state.items, items: newItems, totalCount, totalPrice };

      }

    case 'CLEAR\_CART': // очистка корзины

      return { ...initalState };

    default:

      return state;

  }

}

export default cart;

(в блоках case блоки кода в фигурных скобках {…}, чтобы не было ошибок на счет переопределения констант)

В блоке ‘REMOVE\_FROM\_CART’ нам надо уменьшить количество пицц конкретного вида – т.е. с заданным id, толщиной теста и размером – т.е. то, что отображается в одной строке заказа. Поэму в action.payload мы передаем не id, а «образец» - объект, содержащий {id, type, size}. Далее в массиве items[id] ищем такую пиццу (первую попавшуюся, у которой id, type и size равны искомой) и удаляем из массива.

Для подтверждения очистки всей корзины можно использовать:

const handleClearCart = React.useCallback(() => {

if (window.confirm("Очистить корзину?")) {

dispatch(clearCart());

}

// eslint-disable-next-line react-hooks/exhaustive-deps

}, []);

А чтобы кнопка «очистка корзины» вообще не отображалась в Cart.jsx если корзина пуста, то выводим <div className="cart\_\_clear"> по условному выражению

{totalCount && <div className="cart\_\_clear" onClick={handleClearCart}> … </div> }

Другой подход: если totalCount === 0, то вместо содержимого корзины (<div className="cart"> ) выводим блок <div class=”cart-empty”). Последний взять из верстки cart-empty.html, <div class=”cart-empty”>.

Тогда

return (

    <div className="container container--cart">

      {totalCount

? <div className="cart">…

: <div className="cart cart--empty">

В блоке <div class=”cart-empty”> правим ссылку – вместо <a> используем Link:

<div className="cart cart--empty">

  <h2>Корзина пустая <icon>😕</icon></h2>

  <p>

    Вероятней всего, вы не заказывали ещё пиццу.<br />

    Для того, чтобы заказать пиццу, перейди на главную страницу.

  </p>

  <img src={emptyCartImg} alt="Empty cart" />

  <Link to="/" className="button button--black">

     <span>Вернуться назад</span>

   </Link>

</div>

(Link из библиотеки ‘react-router-dom’)

*Засада: у меня при возврате в Home все равно его перерисовка и повторная загрузка пицц с сервера.*

Редюсер cart можно упростить, создав функции расчета количества и суммы – getTotalCount() и getTotalSum():

const getTotalCount = (items) => Object.values(items).flat().length;

const getTotalSum = (items) => Object.values(items).flat().reduce((sum, v) => sum + v.price, 0);

const cart = (state = initalState, action) => {

  switch (action.type) {

    case 'ADD\_TO\_CART':

      {

        const id = action.payload.id;

        const newItems = {

          ...state.items,

          [id]: !state.items[id] ? [action.payload] : [...state.items[id], action.payload]

        };

        return { ...state, items: newItems, totalCount: getTotalCount(newItems), totalPrice: getTotalSum(newItems) }

      }

    case 'REMOVE\_FROM\_CART': // уменьшить количество пицц

      {

        const { id, type, size, price } = action.payload;

        const removeIndex = state.items[id].findIndex(item => item.type === type && item.size === size);

        if (removeIndex < 0) {

          return state;

        }

        const newArray = [...state.items[id].slice(0, removeIndex), ...state.items[id].slice(removeIndex + 1)];

        const newItems = { ...state.items, [id]: newArray };

        return { ...state, items: newItems, totalCount: getTotalCount(newItems), totalPrice: getTotalSum(newItems) }

      }

    case 'DELETE\_ORDER': // удалить всю строку из  Корзины (пицца с id, type, size)

      {

        const { id, type, size } = action.payload;

        const newArray = state.items[id].filter(order => !(order.id === id && order.type === type && order.size === size))

        const newItems = { ...state.items, [id]: newArray };

        if (newItems[id].length === 0) {

          delete newItems[id];

        }

        return { ...state, items: newItems, totalCount: getTotalCount(newItems), totalPrice: getTotalSum(newItems) }

      }

    case 'CLEAR\_CART': // очистка корзины

      return { ...initalState };

    default:

      return state;

  }

}

**Кнопка оплатить**

* В Cart.jsx создаем обработчик handleOrder(), в котором выводим alert(). Можно даже с dispatch(crearCart())
* прописать <div className="button pay-btn" onClick={handleOrder}> <span>Оплатить сейчас</span> </div>

DONE

**Развертывание приложения**

<https://www.youtube.com/watch?v=-pJN9faoa8E>

См. также <https://create-react-app.dev/docs/deployment/>

Развертывание React-приложения на Vercel, Netlify, Heroku, GitHub Pages, Surge (это все бесплатные хостинги)

Наш проект должен лежать на github.com. Например, проект <https://github.com/LLPeterX/pizza-shop.git>

***Локальный сервер***

*npm install -g serve*

*serve -s build*

*Сайт будет доступен по адресу, например, http://192.168.1.12:5000*

**Vercel -** <https://vercel.com/>

1. Регистрируемся. Выбираем “Continue with GitHub”
2. “Import git repository”, выбираем наш проект, copy/paste ссылку на наш проект – напр. <https://github.com/LLPeterX/pizza-shop.git>
3. Ждем, пока приложение импортируется на Vercel.
4. По завершении жмем кнопку “View”
5. Приложение будет доступно по ссылке pizza-shop.vercel.app

Полученной ссылкой теперь можно делиться.

При внесении изменений на github, Vercel автоматически их обрабатывает. Нужно просто подождать.

**Netify** - <https://www.netlify.com/>

1. Регистрируемся: SignUp -> GitHub
2. В панели управления (dashboard) жмем кнопку “New site from Git”
3. Подтверждаем, разрешаем доступы, выбираем из списка нужный проект – “pizza-shop”
4. Жмем кнопку “Deploy site”
5. По завершении обработки выдается ссылка на проект, типа <https://festive-jang-4f0ba3.netify.app>

Аналогично Vercel, Netify следит за изменениями в репозитории Git.

**Surge** - <https://surge.sh/>

Тут все через командную строку. По ссылке <https://surge.sh/help/getting-started-with-surge> - обучающее видео.

В отличии от Vercel и Netify, Surge не использует git и требует собранный проект

1. Устанавливаем глобально пакет «surge»:  
   npm install -g surge
2. Выполняем сборку проекта:  
   npm run build  
   По завершении сборки появляется папка build
3. Переходим в папку build и даем команду “surge”:  
   cd build  
   surge  
   (*Команда surge --help показывает доступные команды*)
   * Вводим имя и пароль. Если еще нет аккаунта на surge, он будет создан. Для проверки можно дать команду ‘surge whoami’
   * Подтверждаем каталог build.
   * Вместо предложенного имени домена указываем свой. Имя домена должно заканчиваться на ‘.surge.sh’ – например “react-pizza.surge.sh”.

По завершении сайт будет доступен по ссылке <https://react-pizza.surge.sh>

**GitHub Pages**

Наш репозиторий на github будет открываться как сайт, т.е. внутри github.

1. На github.com в нашем репозитории жмем кнопку “Settings” и переходим в раздел “Github Pages”.
2. Выбираем “master” (или main), нажимаем “Save”
3. Теперь в разделе “Github Pages” появляется уникальная ссылка. Копируем эту ссылку.
4. В проекте устанавливаем утилиту “gh-pages” (локально) – она будет заливать проект на Github Pages:  
   npm install gh-pages
5. В нашем проекте открываем файл package.json и после свойства “private” вставляем поле “homepage” со значением этой ссылки, например:  
   “homepage”: “<https://LLPeterX.github.io/pizza-shop/>”  
   С помощью этого поля мы указываем, что у нашего репозитория есть web-страница.
6. В package.json в секции “scripts” добавляем 2 поля – “deploy” и “predeploy”. Команда deploy загружает собранное приложение (папку build) на Github Pages с помощью утилиты gh-pages, а “predeploy” вызывается из “deploy” и выполняет сборку проекта, её самим запускать не надо.  
   “deploy”: “gh-pages -d build”,  
   “predeploy: “npm run build”

Теперь для развертывания приложения надо дать команду:

npm run deploy

Запускается “npm run build”, приложение соберется в папке build и будет опубликовано на сайте (требуется вреся). Но по ссылке отображается только readme.md

В репозитории появляется новая ветка “gh-pages”. Чтобы сделать так, чтобы отображался сайт:

1. В репозитории переходим в Settings
2. В разделе “Github Pages” в поле Source вместо “master” выбираем “gh-pages” и жмем “Save”

Обновление занимает пару минут. Теперь сайт доступен по ссылке, указанной в разделе “Github Pages” (например, <https://llpeterx.github.io/pizza-shop/>)

При каждом изменении в проекте надо давать команду “npm run deploy”.

Важно: если приложение работает на NodeJS, то на Github Pages оно работать не будет, т.к. там отображаются только статические файлы – html, css, js. Самописные серверы на NodeJS работать не будут. Такие серверы лучше заливать на Heroku.

**Heroku -** [**https://www.heroku.com/**](https://www.heroku.com/)

Heroku полезен тем, что поддерживает серверы – а у нас есть ‘json-server’.

1. Регистрируемся (кнопка “Sign up for free”). Вводим свои данные и затем заходим в панель управления, где отображаются созданные в Heroku сайты.
2. Жмем кнопку “New” -> Create new app. Вводим название – например, ‘react-pizza-test’
3. Далее есть 2 способа залива приложения на Heroku:
   1. с использованием утилиты “heroku” (Heroku CLI)
   2. с использованием github (будем использовать этот подход – он удобнее)
4. Жмем кнопку “Github”, выбираем нужный репозиторий. Жмем кнопку “Connect”  
   Этим самым мы к проекту на Heroku подключаем проект из Github.
5. Включаем “Enable automatic deploys”. Тогда Heroku будет следить за нашим репозиторием, и при изменениях пересоберёт приложение.
6. Убеждаемся, что выбрана ветка “master” и жмем кнопку “Deploy branch”  
   Ниже появляется консоль с протоколом.
7. По завершении жмем кнопку “View” и… наблюдаем ошибку “Application error”. Для анализа нужно установить приложение Heroku CLI и просмотреть лог.

В нашем приложении используется json-server и Heroku не знает, что сначала нужно запустить его, а после этого – react.

В package.json у нас есть скрипты запуска приложения:

"scripts": {

**"start": "react-scripts start"**,

"build": "react-scripts build",

"test": "react-scripts test",

"eject": "react-scripts eject",

**"server": "json-server --watch public/db.json --port 5000"**

},

Приложение должно взаимодействовать с json-server (а может быть Express и т.п.).

Есть 2 варианта (см. комменты под видео <https://www.youtube.com/watch?v=-pJN9faoa8E>):

1) В корне проекта создаем файл Procfile, в котором указываем, что при запуске приложения также надо запускать json-server (будем считать, что код сервера в файле server.js):  
 web: node server.js

*Причем библиотеку ‘json-server’ надо перенести из “ devDependencies” в “dependencies” и снова push на github*  
После сборки приложения Heroku читает файл Procfile и видит, что надо запустить файл server.js

2) Установить библиотеку ‘concurrently’ и в package.json прописать:  
"start": "concurrently \"react-scripts start\" \"json-server -w public/db.json -p 5000\""

Заливаем все на github. В панели управления Heroku, в консоли, можно видеть процесс сборки (2-3 минуты) и после этого сайт доступен по ссылке типа <https://react-pizza.heroku.app>

Итого:

* Vercel и Netify подходят для приложений React, у которых нет логики, связанной NodeJS.
* surge.sh и Github Pages подходят для публикации лендингов, написанных на React
* Heroku – для использования связки сервер + клиент, для использования СУБД