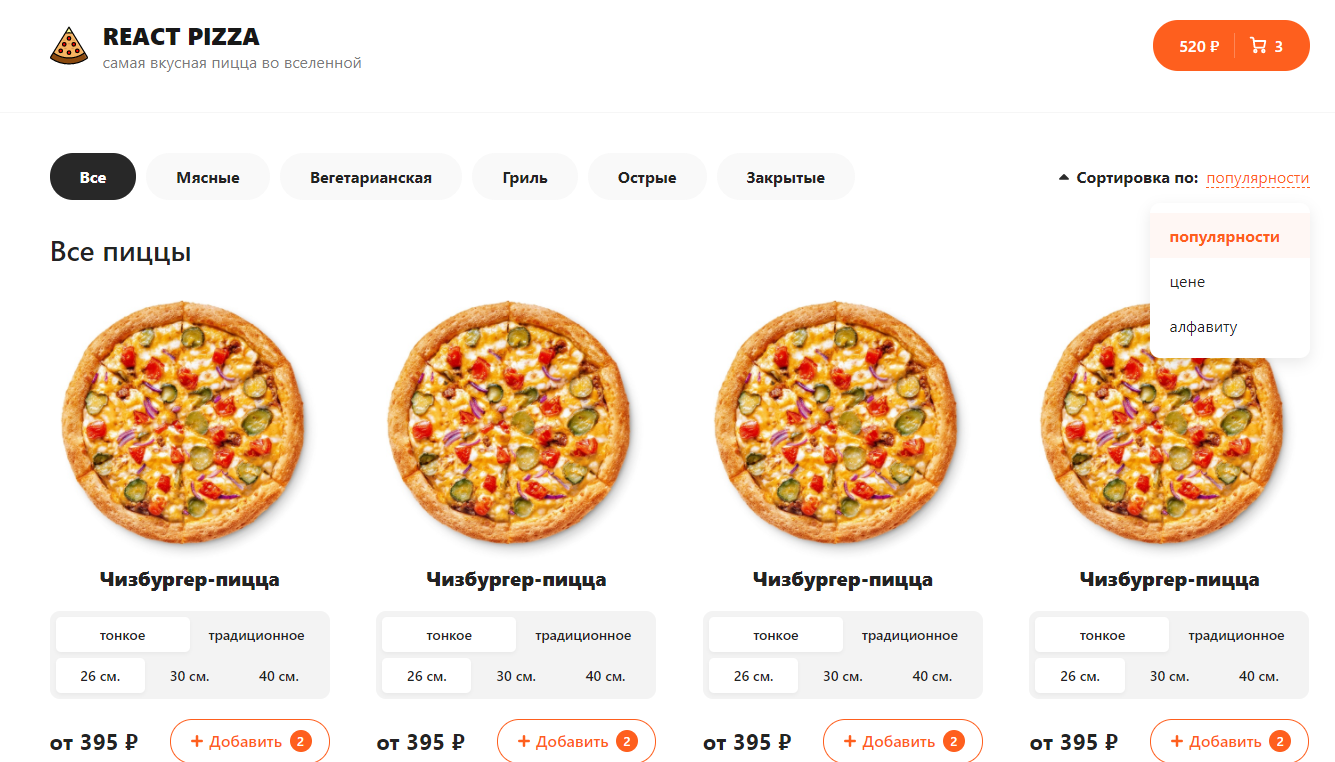
**Курс React Pizza**

<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);

**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>

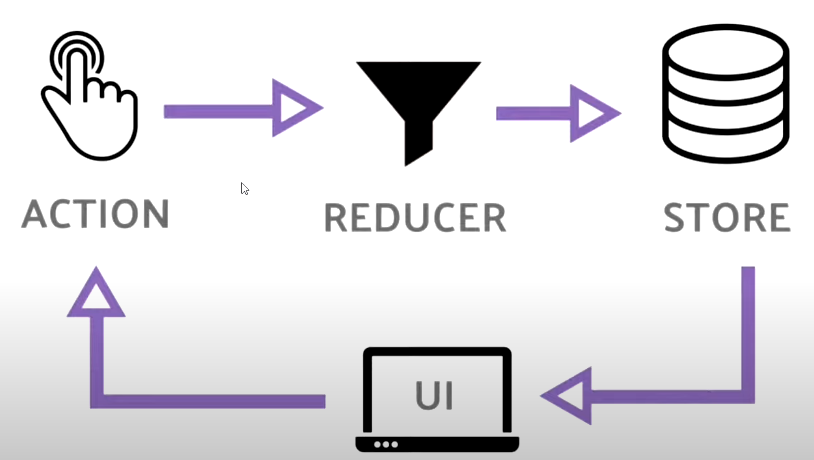
Redux – это библиотека, которая позволяет хранить данные. React и Redux – разные вещи. React отвечает за UI, Redux – за хранение данных.

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’,

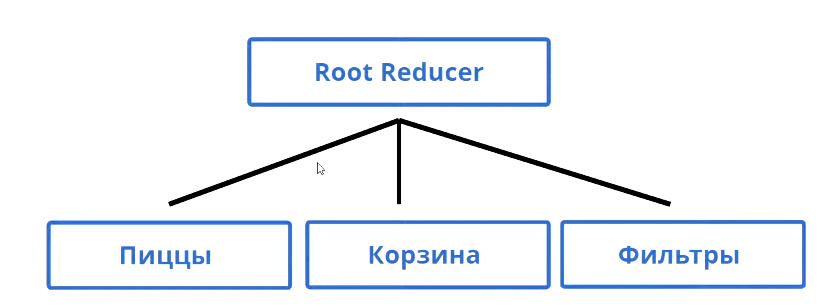
payload: { id: 2, name: ‘Пепперони’, size: 26, type: 0 }

}

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

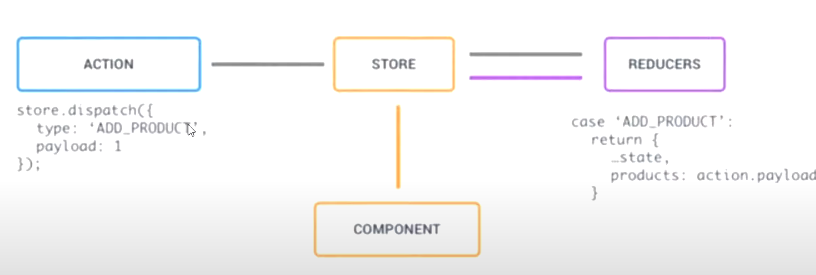
**RootReducer**

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



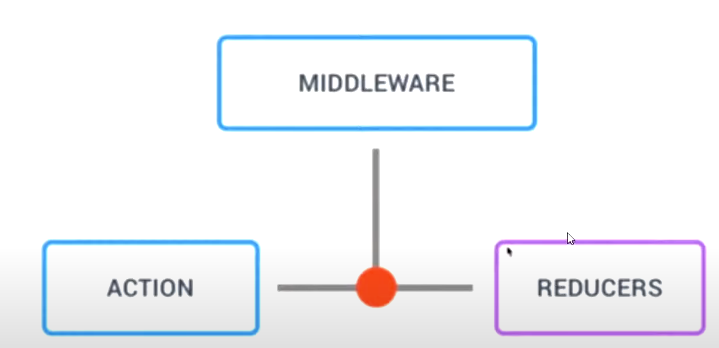
Из их всех создается т.н. «корневой редюсер». Для простых приложений его можно не создавать, но в случае сложных хранилищ из модификация будет очень сложной.

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:0 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: 0,

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);

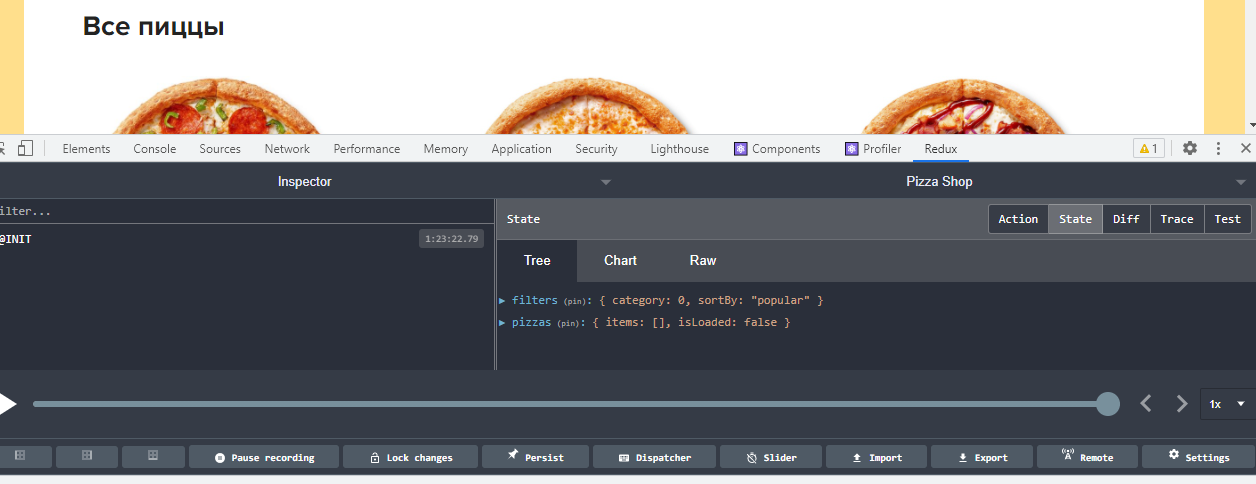
Для проверки можно поместить store в глобальный объект window и вызвать store.getState(). Видно, что там 2 стейта – filters и pizzas. Для изменения стейта – store.dispatch({type: SET\_SORT, payload: ‘price’}).

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()