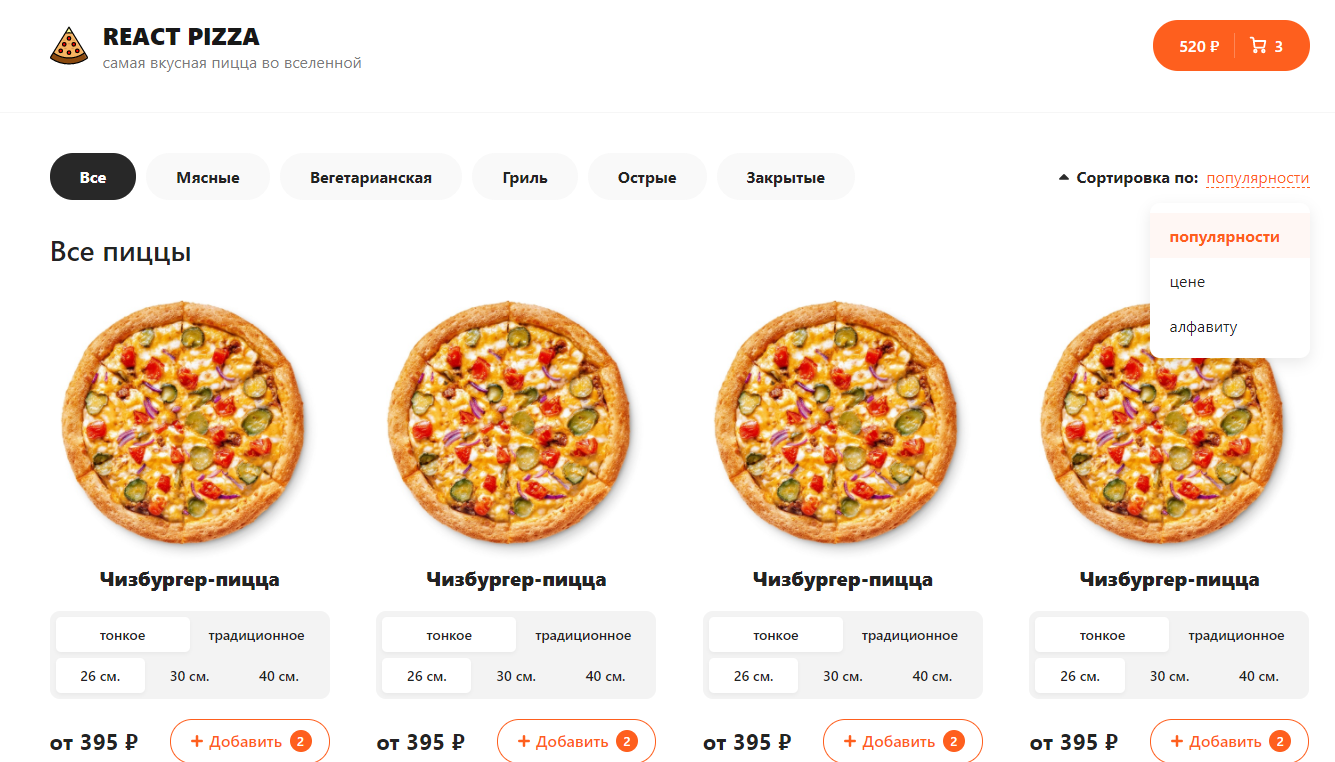
**Курс React Pizza**

<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.

stop at <https://youtu.be/IMBAK-DftVM?t=3086>

хуйня какая-то. Я устал.