React-Pizza

Оглавление

Начало. JS

https://www.youtube.com/watch?v=okqptJNYbXg&list=PL0FGkDGJQjJG9eI85xM1_iLIf6BcEdaNl&index=2

https://peaceful-dove-11d.notion.site/React-Pizza-v2-REMASTERED-655fa3a5ea4f4bce8faeee2f28a8fb22

Технологии:

- ReactJS 18
- TypeScript
- Redux Toolkit (хранение данных / пицц)
- React Router v6 (навигация)
- Axios + Fetch (отправка запроса на бэкенд)
- React Hooks (хуки)
- Prettier (форматирование кода)
- CSS-Modules / SCSS (стилизация)
- React Content Loader (скелетон)
- React Pagination (пагинация)
- Lodash.Debounce
- Code Splitting, React Loadable, useWhyDidYouUpdate

Начало. Заметки

hot-reload - изменения в коде моментально отображаются в браузере.

Ctrl - Shift-P - Preferences - Open Setting(JSON) - открываем settings.json - главный файл настроек

Для **компонентов** две следующие записи равнозначны:

```
<Categories />   и
{Categories()}
```

Для статичных файлов в React принято создавать папку src/assets. Например, картинки положим в src/assets/img. Правим после этого пути в картинках:

```
import logoSvg from '../assets/img/pizza-logo.svg';
...
<img width="38" src={logoSvg} alt="Pizza logo" />
```

Пример рендер-списка:

```
 return (
  <div className="categories">
   <ul>
    {categories.map((item, index) => (
     <li
      onClick={() => {
       onClickCategory(index);
      }}
      className={activeIndex === index ? 'active' : ''}>
      {item}
     </li>
    ))}
   </ul>
  </div>
 );
}
```

В пропсах, если передается **объект**, можно сократить код. Вместо

```
<PizzaBlock
  title={obj.title}
  price={obj.price}
   image={obj.imageUrl}
   sizes={obj.sizes}
  types={obj.types}
/>
```

передать:

```
<PizzaBlock {...obj} />
```

Фрагменты

В React можно рендерить только один родительский тег. Если тег не нужен, то можно использовать **фрагмент**:

<> ... </>

Фрагменты позволяют формировать список дочерних элементов, не создавая лишних узлов в DOM. Полная запись фрагмента:

```
<React.Fragment>
  <ChildA />
  <ChildB />
  <ChildC />
</React.Fragment>
```

Фрагменты, объявленные с помощью <React.Fragment>, могут иметь **ключи**. key — это единственный атрибут, допустимый у Fragment на данный момент.

Вывод списков и Reconciliation алгоритм

При выводе списков и других итерируемых элементов, к каждому такому элементу, необходимо добавить постоянное уникальное значение для атрибута key. Передавать в key **индекс массива** считается плохой практикой. Однако, если список **статичен** и не будет меняться, то это допутимо.

В качестве key нельзя передавать **Math.random()**

popup окно с сотировкой

Используем условный рендеринг

```
const [open, setOpen] = React.useState(false);
...
    {open && (
      <div className="sort__popup">
        <ul>
          <li className="active">популярности</li>
          <li>цене</li>
          <li>алфавиту</li>
        </ul>
      </div>
    )}
```

Другой способ условного рендеринга:

```
    {open ? (
      <div className="sort__popup">
        <ul>
          <li className="active">популярности</li>
          <li>цене</li>
          <li>алфавиту</li>
        </ul>
      </div>
    ) : ( 'Popup none' )}
```

Бэкенд. Mockapi

https://mockapi.io/

Авторизация через Google или Github. У меня через Github.

Создаем новый проект react-pizza. Создаем новый ресурс items. Далее ничего не выбираем, а нажать **CREATE.** Получим пустой массив. **Адрес сервера** выдается в mockapi. Ресурсы будут загружены по адресу:

*https://62d162dcdccad0cf176680f0.mockapi.io/**items***

Копируем свои данные из файла json в mockapi по кнопке **data**

Скелетон

Есть специальная библиотека https://skeletonreact.com/

Там можем сами наросовать скелетон карточки. Измеряем на своем сайте размеры

Устанавливаем библиотеку:
npm i react-content-loader

Далее создаем src/PizzaBlock/**Skeleton.jsx**. Переносим туда код из библиотеки и меняем название компонента, добавим класс:

```
import React from 'react';
import ContentLoader from 'react-content-loader';
const Skeleton = () => (
  <ContentLoader
    className="pizza-block"
    speed={2}
    width={280}
    height={466}
    viewBox="0 0 280 466"
    backgroundColor="#f1f5f2"
    foregroundColor="#ecebeb">
    <circle cx="135" cy="126" r="126" />
    <rect x="0" y="275" rx="10" ry="10" width="280" height="27" />
    <rect x="137" y="296" rx="0" ry="0" width="0" height="1" />
    <rect x="0" y="332" rx="10" ry="10" width="280" height="52" />
    <rect x="123" y="412" rx="25" ry="25" width="156" height="48" />
    <rect x="4" y="425" rx="5" ry="5" width="100" height="22" />
  </ContentLoader>
);
export default Skeleton;
```

Далее в **App.js**:

```
function App() {
  const [items, setItems] = React.useState([]);
  const [isLoading, setIsLoading] = React.useState(true);  // флаг для скелетона

  React.useEffect(() => {
    fetch('https://62d162dcdccad0cf176680f0.mockapi.io/items')
      .then((res) => res.json())
      .then((data) => {
        setItems(data);
        setIsLoading(false); // меняем флаг
      });
  }, []);

  return (
    <div className="wrapper">
      <Header />
      {isLoading && 'Загрузка...'}
      <div className="content">
        <div className="container">
...
          <h2 className="content__title">Все пиццы</h2>
          <div className="content__items">
            {isLoading  // условный рендеринг
              ? [...new Array(9)].map((_, i) => <Skeleton key={i} />) // для скелетона сгенерируем пустой массив
              : items.map((obj) => <PizzaBlock key={obj.id} {...obj} />)}
          </div>
        </div>
      </div>
    </div>
  );
}
```

Роутер React Router 6

https://reactrouter.com/

Позволяет делать навигацию без перезагрузки страницы.

Установка
npm install react-router-dom@6

В **index.js** в корне проекта:
```
import { BrowserRouter } from 'react-router-dom';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <BrowserRouter>
    <App />
  </BrowserRouter>,
);
```

Создадим src/**pages/Home.jsx**, src/**pages/Cart.jsx,** src/**pages/NotFound.jsx**.
Замечаем, что header повторяется, а содержимое остальное разное на разных страницах.

В **App.js** в корне проекта:
```
import { Routes, Route } from 'react-router-dom';

function App() {
  return (
    <div className="wrapper">
      <Header />
      <div className="content">
        <div className="container">
          <Routes>
            <Route path="/" element={<Home />} />
            <Route path="/cart" element={<Cart />} />
            <Route path="*" element={<NotFound />} />
          </Routes>
        </div>
      </div>
    </div>
  );
}
```

Ссылки см. другую документацию.
При возврате со страниц на главную, можно **убрать прокрутку страницы**, которая сохраняется в браузере. Для этого при первом рендере (файл **Home.jsx**):

```
  React.useEffect(() => {
    fetch ...
    window.scrollTo(0, 0);
  }, []);
```

Модули SCSS

Создаем файл components/NotFoundBlock/**NotFoundBlock.module.scss.** В названии обязательно **module**

```
.root {
  padding: 100px;
  max-width: 750px;
  margin: 0 auto;
  text-align: center;

  span {
    font-size: 64px;
  }
}
```

Применение в src/ components/NotFoundBlock/**index.jsx**:
```
import styles from './NotFoundBlock.module.scss';

const NotFoundBlock = () => {
  console.log(styles);
  return (
    <h1 className={styles.root}>
      <span>☹</span>
      <br />
      Ничего не найдено :(
```

```
  </h1>
 );
};
```

Будет создан класс с уникальным именем **root**: "NotFoundBlock_root__cXSMD"

Адаптив. Grid

Для grid адаптив без media:

```
 &__items {
  display: grid;
  //grid-template-columns: repeat(4, 1fr);
  // grid-template-rows: repeat(2, 1fr);
  grid-column-gap: 11px;
  justify-items: center;
  grid-template-columns: repeat(auto-fit, minmax(280px, 1fr));
 }
```

Поиск по пиццам

Создаем components/**Search/index.jsx** и components/**Search/Search.module.scss**

Иконку возьмем с сайта https://www.iconfinder.com/ в формате **svg**
Скопируем код svg в компонент и уберем оттуда пару строк. Делаем для нее класс. Файл components/**Search/index.jsx:**

```
import React from 'react';
import styles from './Search.module.scss';

const Search = () => {
 return (
  <div className={styles.root}>
   <svg
    className={styles.icon}
    enableBackground="new 0 0 32 32"
    id="EditableLine"
    version="1.1"
    viewBox="0 0 32 32"
    xmlns="http://www.w3.org/2000/svg">
    <circle
...
    />
   </svg>
   <input className={styles.input} placeholder="Поиск Пиццы..." />;
  </div>
 );
};
```

Состояние для поиска сделаем на уровне App.js и опрокидываем его в Search. В **Search/index.jsx** делаем контролируемый инпут. Также сделаем кнопку для очистки данных и для ее вывода используем условный рендеринг:

```
const Search = ({ searchValue, setSearchValue }) => { // состояние из App
 return (
  <div className={styles.root}>
   <svg
    className={styles.icon} ...
   </svg>
   <input
    onChange={(event) => setSearchValue(event.target.value)} // обработка значения инпута
    className={styles.input}
    placeholder="Поиск Пиццы..."
    value={searchValue}  // контролируемый инпут
   />
   {searchValue && (   // условный вывод кнопки для очистки. Отображать только, когда есть ввод
    <svg
     onClick={() => setSearchValue('')}
     className={styles.clearIcon}
     viewBox="0 0 20 20"
     xmlns="http://www.w3.org/2000/svg">
```

```
      <path d="M10 8.586L2.929 1.515 1.515 2.929 8.586 10l-7.071 7.071 1.414 1.414L10 11.414l7.071 7.071 1.414-1.414L11.414 10l7.071-7.071-1.414-1.414L10 8.586z" />
    </svg>
   )}
  </div>
 );
};
```

Простой вариант поиска без бэкенда в файле src/**components/Home.jsx**:

```
 const pizzas = items
  .filter((obj) => obj.title.toLowerCase().includes(searchValue.toLowerCase()))
  .map((obj) => <PizzaBlock key={obj.id} {...obj} />);
 const skeletons = [...new Array(9)].map((_, i) => <Skeleton key={i} />);

 return (
  <>
   <div className="content__top">
    <Categories value={categoryId} onChangeCategory={(index) => setCategoryId(index)} />
    <Sort value={sortType} onChangeSort={(sortProperty) => setSortType(sortProperty)} />
   </div>
   <h2 className="content__title">Все пиццы</h2>
   <div className="content__items">{isLoading ? skeletons : pizzas}</div>
  </>
 );
```

Этот способ подходит при **небольшом и статичном** объеме данных. Иначе лучше использовать бэкенд.
Другой способ с помощью mockAPI:

```
 React.useEffect(() => {
  setIsLoading(true);
  const order = sortType.sortProperty.includes('-') ? 'asc' : 'desc';
  const sortBy = sortType.sortProperty.replace('-', '');
  const category = categoryId > 0 ? `category=${categoryId}` : '';
  const search = searchValue ? searchValue.toLowerCase() : '';
  fetch(
   `https://62d162dcdccad0cf176680f0.mockapi.io/items?search=${search}&${category}&sortBy=${sortBy}&order=${order}`,
  )
   .then((res) => res.json())
   .then((data) => {
    setItems(data);
    setIsLoading(false);
   });
  window.scrollTo(0, 0);
 }, [categoryId, sortType, searchValue]);
```

При тестировании видим, сто Mockapi некорректно работает при поиске со строкой поиска и категориями.

Пагинация

Создаем components/**Pagination/index.jsx** и components/**Pagination/Pagination.module.scss**

Есть специальные библиотеки для пагинации, н-р, react-paginate. Установка:
npm i react-paginate

Копируем код из документации в наш компонент и делаем свою стилизацию. В стилизации, когда мы стилизуем "чужой" класс, используем :global:

```
 :global {
  .selected {
   a {
    background-color: #fe5f1e;
    color: #fff;
   }
  }
 }
```

В **Home.jsx** добавим в адресную строку лимиты для страниц (см. документацию Mockapi):

```
  fetch(
   `https://62d162dcdccad0cf176680f0.mockapi.io/items?page=1&limit=4&search=${search}&${category}&sortBy=${sortBy}&order=${order}`,
```

```
  )
```

Здесь **page** - номер первой страницы, **limit** - число элементов на странице

Нормальный бэкенд должен для пагинации возвратить массив со страницами, но Mockapi этого не делает. Поэтому зададим параметры вручную.

Создадим новое состояние. В **Home.jsx**

```
const Home = ({ searchValue, setSearchValue }) => {
...
 const [currentPage, setCurrentPage] = React.useState(1); // создаем состояние

 React.useEffect(() => {
...
   fetch(
    `https://62d162dcdccad0cf176680f0.mockapi.io/items?page=${currentPage}&limit=4&search=${search}&${category}&sortBy=${sortBy}&orde
r=${order}`, // задаем номер текущей страницы и количество товаров на странице
   )
...
 }, [categoryId, sortType, searchValue, currentPage]);
...
 return (
  <>
...
    <Pagination onChangePage={(number) => setCurrentPage(number)} /> // передаем текущую страницу в компонент
  </>
 );
};
```

В **Pagination.jsx**

```
import React from 'react';
import ReactPaginate from 'react-paginate';
import styles from './Pagination.module.scss';

const Pagination = ({ onChangePage }) => {
 return (
  <ReactPaginate
   className={styles.root}
   breakLabel="..."
   nextLabel=">"
   onPageChange={(event) => onChangePage(event.selected + 1)}  // текущая страница
   pageRangeDisplayed={4} // количество элементов на странице
   pageCount={3} // всего страниц
   previousLabel="<"
   renderOnZeroPageCount={null}
  />
 );
};
```

Redux

Стейтменеджеры - **Redux, MobX**. Помогают создавать глобальное хранилище данных. **Redux** - отдельная js-библиотека. Может работать с разными технологиями, не только React.

Почему **Redux** лучше **контекста**? При использовании контекста, если происходит изменение в контексте в одной переменной, то будут рендериться все компоненты, подписанные на этот контекст, даже если они не используют эту переменную. В Redux перерендерится только тот компонент, который использует эту переменную. Для больших приложений это имеет большое значение.

Тем не менее, нужно запомнить, что Redux "под капотом" **использует контекст**.

React-Redux - библиотека, которая "приспосабливает" React к Redux.

Redux-Toolkit - это пакет, облегчающий работу с Redux.

Пробуем Redux в своем приложении: сделаем так, чтобы **фильтрация** хранилась в Redux.

Установка

npm install @reduxjs/toolkit react-redux

1. Создадим **src/redux/store.js**

```
import { configureStore } from '@reduxjs/toolkit';
import   filter from './slices/filterSlice';

export const store = configureStore({
 reducer: { filter },   //  что равнозначно {filter: filter}, название из filterSlice
});
```

2. Соединим его с нашим приложением **в index.js**:

```
import { store } from './redux/store';
import { Provider } from 'react-redux';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
 <BrowserRouter>   // неважно, в какой последовательности вкладывать BrowserRouter и Provider
  <Provider store={store}>
   <App />
  </Provider>
 </BrowserRouter>,   ...
```

3. Slice - аналог склада (раньше это называлось **редюсером**). Н-р, склад с пиццами **PizzaSlice**, склад с корзиной **cartSlice,** фильтр **filterSlice**
Создадим **src/redux/slices/filterSlice.js** (rxslice-Tab):

```
import { createSlice } from '@reduxjs/toolkit';

const initialState = {  // начальное состояние
 categoryId: 0,
 sort: { name: 'популярности', sortProperty: 'rating' },
};

const filterSlice = createSlice({  // создаем "склад"
 name: 'filters', // даем имя, которое будем потом использовать в actions.type: "filters/setCategoryId"
 initialState,
 reducers: { // создаем методы - actions
  setCategoryId(state, action) {  //  под "капотом" redux-toolkit состояние state и actions: {type: "filters/setCategoryId", payload: 0}
   state.categoryId = action.payload;
  },
 },
});

export const { setCategoryId } = filterSlice.actions;  // экспорт actions = reducers
export default filterSlice.reducer;  // всегда делаем экспорт по умолчанию
```

4. В **Home.js**

```
import { useSelector, useDispatch } from 'react-redux';
import { setCategoryId } from '../redux/slices/filterSlice';

const Home = () => {
...
 const categoryId = useSelector((state) => state.filter.categoryId);  // Вытаскиваем состояние - слушатель этого состояния. Если состояние меняется, компонент перерисовывается
 const dispatch = useDispatch(); // создаем "рупор" для выполнения действий, по сути это метод в store

 const onChangeCategory = (id) => {
  dispatch(setCategoryId(id));
// Используем действие
// Функция setCategoryId(id) возвращает объект {type: 'filters/setCategoryId', payload: 2} - toolkit делает этот возврат автоматически
 };

 return (
  <>
   <div className="content__top">
    <Categories value={categoryId} onChangeCategory={onChangeCategory} />
...
```

Axios

Axios — это облегченный клиент HTTP на базе сервиса $http с Angular.js v1.x, похожего на собственный JavaScript Fetch API. Но он использует меньше кода, больше возможностей и проще fetch

https://www.digitalocean.com/community/tutorials/react-axios-react-ru

Установка
npm i axios

В **Home.js** меняем fetch:

```
import axios from 'axios';

  axios.get(
`https://62d162dcdccad0cf176680f0.mockapi.io/items?page=${currentPage}&limit=4&search=${search}&${category}&sortBy=${sortBy}&order=${order}`,
    )
    .then((response) => {
     setItems(response.data);  // можем взять данные сразу с помощью свойства data без метода json
      ...
    });
```

Использование UseRef для наведения фокуса на инпут при очистке поиска

Возвращает объект со свойством **current**, в который записано текущее значение **ref** ({current: текущий_элемент}). Чтобы связать его с dom-элементом, в элементе нужно добавить **атрибут ref** со значением - этим объектом.

В **Search/index.jsx**

```
 const inputRef = React.useRef();
 const onClickClear = () => {
  setSearchValue('');
  inputRef.current.focus();
 };
...
    <input ...      ref={inputRef}   />
    {searchValue && (
     <svg      onClick={() => onClickClear()} ...      </svg>
    )}
```

Lodash. Декоратор Debounce. UseCallback. Задержка при вводе строки поиска

Теперь проблема - запрос на бэкенд при вводе поиска делается при вводе каждого символа.

Декоратор – это обёртка вокруг функции, которая изменяет поведение последней. Основная работа по-прежнему выполняется функцией. Результатом декоратора debounce(f, ms) должна быть обёртка, которая передаёт вызов f не более одного раза в ms миллисекунд. Другими словами, когда мы вызываем debounce, это гарантирует, что все остальные вызовы будут игнорироваться в течение ms.
https://learn.javascript.ru/call-apply-decorators

Свой debounce делать не будем. Будем использовать библиотеку **Lodash**. Можем устанавливать не всю библиотеку, а только нужную функци. через npm:
npm i lodash.debounce

Использование: _.debounce(func, [wait=0], [options={}])

В **Search/index.jsx**

Чтобы не было лишнего считывания функции при клике, используем **useCallback** с зависимостью - []. Используем два состояния для инпута - локальное и для передачи наружу.

```
import { SearchContext } from '../../App';
import debounce from 'lodash.debounce';

const Search = () => {
 const [value, setValue] = React.useState('');  // локальное состояние
 const { setSearchValue } = React.useContext(SearchContext);  // состояние для передачи наружу
 const inputRef = React.useRef();
```

```
  const onClickClear = () => {  // очистка инпута
    setSearchValue('');
    setValue('');
    inputRef.current.focus();
  };

  const updateSearchValue = React.useCallback( // используем задержку debounce в 300 мс, функция читается только 1 раз
    debounce((str) => {
      setSearchValue(str);
    }, 300),
    [],
  );

  const onChangeInput = (event) => {  // обработка инпута
    setValue(event.target.value);
    updateSearchValue(event.target.value);
  };

  return (
    <div className={styles.root}>
...
      <input
        onChange={onChangeInput}  // клик
        className={styles.input}
        placeholder="Поиск Пиццы..."
        value={value}  // выводим локальное состояние
        ref={inputRef}
      />
      {value && (
        <svg
          onClick={() => onClickClear()} ...  // очистка
        </svg>
      )}
    </div>
  );
};
```

Сохраняем параметры фильтрации в URL

Будем использовать window.location.search (строка поиска, начиная с ?). Пример:

```
// ?sortProperty=rating&categoryId=0&currentPage=1
```

Можно вместо этого useSearchParams **из react-router-dom**.

Установим библиотеку **qs**:

```
npm i qs
```

Из нее используем **qs.stringify** для формирования строки с параметрами:

```
  const queryString = qs.stringify({
    sortProperty: sort.sortProperty,
    categoryId,
    currentPage,
  });
// queryString:  sortProperty=rating&categoryId=0&currentPage=1
```

а также **qs.parse**:

```
const params = qs.parse(window.location.search.substring(1));
// {
  "sortProperty": "rating",
  "categoryId": "0",
  "currentPage": "1"
}
```

Формируем объект из строки, вырезаем символ **?**

Также используем хук useNavigate из **react-router-dom**

В **Home.js:**

```
import qs from 'qs';
import { useNavigate } from 'react-router-dom';
```

```
const Home = () => {
 const navigate = useNavigate();
...
React.useEffect(() => {
  if (window.location.search) {
   const params = qs.parse(window.location.search.substring(1));  // при первой загрузке парсим параметры
  }
 }, []);

 React.useEffect(() => { // запрос к серверу
...
  axios...
 }, [categoryId, sort.sortProperty, searchValue, currentPage]);

 React.useEffect(() => {
  const queryString = qs.stringify({ // формируем строку из параметров
   sortProperty: sort.sortProperty,
   categoryId,
   currentPage,
  });

  navigate(`?${queryString}`);// переходим на нужную страницу
 }, [categoryId, sort.sortProperty, currentPage]);
```

Передадим параметры в redux. В **filterSlice.js** добавим:

```
 reducers: {
...
  setFilters(state, action) {
   state.sort = action.payload.sort;
   state.currentPage = Number(action.payload.currentPage);
   state.categoryId = Number(action.payload.categoryId);
  },
 },
```

В **Home.js** при парсинге адресной строки изменим состояние в redux:

```
 React.useEffect(() => {
  if (window.location.search) {
   const params = qs.parse(window.location.search.substring(1));
   const sort = sortList.find((obj) => obj.sortProperty === params.sortProperty);  //sortList импортируем предварительно из компонента Sort
   dispatch(
    setFilters({
     ...params,
     sort,
    }),
   );
  }
 }, []);
```

Далее, у нас все работает, но при первом рендеринге делается два запроса - первый по начальному состоянию, а второй уже по параметрам из строки. Исправим это. Для этого добавим флаг с помощью **useRef**:

```
const Home = () => {
...
 const isSearch = React.useRef(false);
...

 const fetchPizzas = () => {
  setIsLoading(true);
  const order = sort.sortProperty.includes('-') ? 'asc' : 'desc';
  const sortBy = sort.sortProperty.replace('-', '');
  const category = categoryId > 0 ? `category=${categoryId}` : '';
  const search = searchValue ? searchValue.toLowerCase() : '';

  axios ...
 };

// если рендер первый, то проверяем есть ли URL параметры и сохраняем в redux
 React.useEffect(() => {
```

```
  if (window.location.search) {
    const params = qs.parse(window.location.search.substring(1));

    const sort = sortList.find((obj) => obj.sortProperty === params.sortProperty);

    dispatch(
      setFilters({
        ...params,
        sort,
      }),
    );

    isSearch.current = true;  // если в параметрах что-то есть, не надо делать первый лишний запрос
  }
 }, []);

// делаем первый запрос, только если ничего нет в параметрах
 React.useEffect(() => {
  window.scrollTo(0, 0);
  if (!isSearch.current) {
    fetchPizzas();
  }

  isSearch.current = false; // чтобы сделать один запрос, если есть параметры
 }, [categoryId, sort.sortProperty, searchValue, currentPage]);
```

Остается еще одна проблема - при первом рендере приложения без параметров в строку добавляются параметры и делается **navigate**, хотя пользователь еще ничего не менял. Для этого делаем еще один флаг:

```
const isMounted = React.useRef(false);
...
// не добавлять URL параметры при первом рендере, при всех остальных добавлять
 React.useEffect(() => {
  if (isMounted.current) {
    const queryString = qs.stringify({
      sortProperty: sort.sortProperty,
      categoryId,
      currentPage,
    });
    navigate(`?${queryString}`);
  }
  isMounted.current = true;
 }, [categoryId, sort.sortProperty, currentPage]);
```

Закрытие popup окна сортировки при клике вне области popup

Сделаем так, чтобы при клике вне компонента sort popup закрывался. Сделаем ссылку на DOM-элемент с классом sort. Сделаем внутри sort обработчик клика по всему документу body. Также позаботимся о том, чтобы обработчик клика удалялся при размонтировании компонента (при переходе на другую страницу)

В **Sort.jsx:**

```
const Sort = () => {
 const sortRef = React.useRef();
...
 React.useEffect(() => {
  const handleClickOutSide = (event) => {
    if (!event.path.includes(sortRef.current)) {  // закрываем popup
      setOpen(false);
    }
  };

  document.body.addEventListener('click', handleClickOutSide);

  return () => {
    document.body.removeEventListener('click', handleClickOutSide);
  };
 }, []);
...
```

```
return (
  <div ref={sortRef} className="sort">
    <div className="sort__label">
...
```

Корзина с товарами

Хранение товаров в корзине обычно делается на бэкенде. Мы попробуем и на фронтенде. Будем в корзине пока делать упрощенно группировку товаров по **id**.

Создаем **redux/slices/cartSlice.js.** Внесем изменения в **store.js**

В **Header.jsx** добавим общую цену и количество товаров (возьмем состояние из redux):

```
import { useSelector } from 'react-redux';

function Header() {
 const { items, totalPrice } = useSelector((state = state.cart));
...
```

В components/**PizzaBlock/index.jsx** сделаем добавление товара:

```
 ...
 const dispatch = useDispatch();
 const cartItem = useSelector((state) => state.cart.items.find((item) => item.id === id)); // находим соотв. элемент в state
 const addedCount = cartItem ? cartItem.count : 0;  // берем количество

  const onClickAdd = () => {
    const item = {    id, title, price, imageUrl,  type: typeNames[activeType],  size: activeSize,   };
    dispatch(addItem(item));
  };
...
  <button className="button button--outline button--add" onClick={onClickAdd}>
...
 <span>Добавить</span>
 {addedCount > 0 && <i>{addedCount}</i>}
```

Асинхронное программирование

Асинхронные задачи ставятся в очередь event loop

http://latentflip.com/loupe/ - можно наглядно посмотреть

Используем для запроса к серверу синтаксис **async/await** и отлавливаем ошибки. В **Home.js:**

```
 const fetchPizzas = async () => {
...
   try {
     const res = await axios.get(  `https://62d162dcdccad0cf176680f0.mockapi.io/...}`, );
     setItems(res.data);
   } catch (error) {
     console.log('error: ', error);
   } finally {
     setIsLoading(false);
   }
 };
```

Бизнес-логика

В контексте фронтенда бизнес-логика - это имеется ввиду работа с бэкендом - какие действия делаются при этом. Вопрос - где бизнес-логика? Ответ - в **UI** или в **отдельном коде**. Мы эту логику вынесем в **redux**, и для этого будем использовать асинхронные экшены с помощью redux-toolkit.

Создадим **redux/slices/ pizzaSlice.js** и сделаем там сохранение информации с сервера

Теперь наша задача - разделить логику работы с сервером на 2 части:

- дай информацию с сервера и сохрани ее
- сделай при этом что-то еще

Сайд-эффекты в **redux** - это в экшенах всякие консоль-логи, алерты и все, что не относится к изменениям в данных. Нужно их избежать.

Асинхронность. createAsyncThunk, ThunkAPI
https://redux-toolkit.js.org/api/createAsyncThunk

createAsyncThunk - для создания асинхронного экшена. Из документации - createAsyncThunk(): принимает тип операции и функцию, возвращающую промис, и генерирует **thunk**, отправляющий типы операции **pending/fulfilled/rejected** на основе промиса

В redux/slices/**pizzaSlice.js:**

```
import { createAsyncThunk, createSlice } from '@reduxjs/toolkit';
import axios from 'axios';

// Создаем асинхронный экшен: сделать запрос к серверу, вернуть ответ - бизнес-логика
// задаем type - 'pizza/fetchPizzasStatus'
export const fetchPizzas = createAsyncThunk('pizza/fetchPizzasStatus', async (params,thunkAPI) => {
 const { order, sortBy, category, search, currentPage } = params;
 const { data } = await axios.get( `https://62d162dcdccad0cf176680f0.mockapi.io/items?page=${currentPage... }` );
 return data;  // возвращаем ответ с сервера
});

const initialState = {
 items: [],
 status: 'loading', // будет 3 статуса: loading | success | error
};

const pizzaSlice = createSlice({
 name: 'pizza',
 initialState,
 reducers: {
  setItems(state, action) {
   state.items = action.payload;
  },
 },
 extraReducers: {  // логика для различных вариантов ответов  с сервера (pending, fulfilled, rejected)
  [fetchPizzas.pending]: (state) => {
   state.status = 'loading';
   state.items = [];
  },
  [fetchPizzas.fulfilled]: (state, action) => {
   state.status = 'success';
   state.items = action.payload;
  },
  [fetchPizzas.rejected]: (state) => {
   state.status = 'error';
   state.items = [];
  },
 },
});

export const { setItems } = pizzaSlice.actions;
export default pizzaSlice.reducer;
```

Делаем правки в **Home.js:**

```
...
 const { items, status } = useSelector((state) => state.pizza);
...
 const getPizzas = async () => {
  const order = sort.sortProperty.includes('-') ? 'asc' : 'desc';
...
  dispatch(fetchPizzas({ order, sortBy, category, search, currentPage }));  // убираем try catch
  window.scrollTo(0, 0);
 };

...
 // делаем первый запрос, только если ничего нет в параметрах
 React.useEffect(() => {
  if (!isSearch.current) {
   getPizzas();
  }
```

```
    isSearch.current = false;
  }, [categoryId, sort.sortProperty, searchValue, currentPage]);
...
  return (
    <>
...
      <h2 className="content__title">Все пиццы</h2>
      {status === 'error' ? (
        <div className="content__error-info">
          <h2>Произошла ошибка ☹</h2>
          <p>К сожалению, не удалось получить данные с сервера. Попробуйте повторить позже.</p>
        </div>
      ) : (
        <div className="content__items">{status === 'loading' ? skeletons : pizzas}</div>
      )}
...
```

ThunkAPI - дополнительная утилита к **createAsyncThunk.** объект (в котором есть :

- dispatch - нужен, если в момент отправки запроса хотим,н-р, сбросить фильтр,

- getState - если хотим узнать текущий state. Мы можем,н-р, вместо параметров **params** вытащить sortBy, category и др. отсюда,

- signal - если хотим останавливать запрос,

- rejectWithValue(value, [meta]) - если хотим, н-р, пробросить свою ошибку. Н-р, при пустом массиве пицц. Н-р в redux/slices/**pizzaSlice.js:**

```
export const fetchPizzas = createAsyncThunk('pizza/fetchPizzasStatus', async (params, thunkAPI) => {
  const { order, sortBy, category, search, currentPage } = params;
  const { data } = await axios.get(
`https://62d162dcdccad0cf176680f0.mockapi.io/items?page=${currentPage}&limit=4&search=${search}&${category}&sortBy=${sortBy}&order=${order}`,
  );
  if (data.length === 0) {
    return thunkAPI.rejectWithValue('Нет пицц');
  }
  return thunkAPI.fulfillWithValue(data);
});
...
  extraReducers: {
...
    [fetchPizzas.rejected]: (state, action) => {
      console.log('action: ', action);  //{type: 'pizza/fetchPizzasStatus/rejected', payload: 'Нет пицц', meta: {...}, error: {...}}
      state.status = 'error';
      state.items = [];
    },
...
```

- и др., см. https://redux-toolkit.js.org/api/createAsyncThunk)

Рефакторинг. Использование UseSelector в redux

У нас есть повторяющийся код с UseSelector. Исправляем:

В redux/slices/**cartSlice.js**

```
...
export const selectCart = (state) => state.cart;
export const selectCartItemById = (id) => (state) => state.cart.items.find((item) => item.id === id); // случай с параметром
...
```

В **Cart.js**

```
const { items, totalPrice } = useSelector(selectCart);
```

В **PizzaBlock/index.jsx:**

```
 const cartItem = useSelector(selectCartItemById(id));
```

Роутер

useLocation - частично дублирует **window.location**. Однако в react для перерисовки компонентов нужно использовать именно его хуки. Иначе при изменении адреса компонент этого не узнает.

Сделаем скрытие кнопки корзины, если перешли в корзину. В **Header.js:**

```
 const location = useLocation();
 return (
...
      <div className="header__cart">  // чтобы не поломать верстку, оставим упаковку
       {location.pathname !== '/cart' && (
         <Link to="/cart" className="button button--cart">
         </Link>
       )}
...
```

useParams

Сделаем тренировочно так, чтобы при клике на пиццу отображалась информация об этой пицце на отдельной странице. Создадим новый компонент - **pages/FullPizza.jsx.** Как-нибудь его заполним.

В **App.js:**

```
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/cart" element={<Cart />} />
        <Route path="/pizza/:id" element={<FullPizza />} />  // id - переменная, можно назвать по-другому
        <Route path="*" element={<NotFound />} />
      </Routes>
```

В **FullPizza.jsx:**

```
const FullPizza = () => {
 const { id } = useParams();
```

useParams и useLocation **делают перерисовку**, если в адресной строке что-то меняется.

В бэкенде должна быть возможность выбрать элемент по адресу, н-р:
https://62d162dcdccad0cf176680f0.mockapi.io/**items/4**

Для **Mochapi** нужно иметь ввиду, что **id** в базе данных должен быть **строкой**.

Outlet

Используется в больших проектах в родительских маршрутах для отображения дочерних маршрутов. Есть некий шаблон, н-р, Dashboard, где есть неизменяемые заголовок, боковое меню и изменяемая основная часть. Тогда в компоненте, где описывается шаблон, изменяемую часть заменить на <Outlet />, а в основном роутере внутри компонента-шаблона вписать дочерние элементы.

Сделаем такой шаблон - header - неизменяемая часть, нижняя часть - изменяемая. Создадим **src/layouts/ MainLayout.jsx:**

```
import React from 'react';
import { Outlet } from 'react-router-dom';
import { Header } from '../components';

const MainLayout = () => {
 return (
  <div className="wrapper">
   <Header />
   <div className="content">
    <Outlet />
   </div>
  </div>
 );
};

export default MainLayout;
```

В **App.jsx:**

```
function App() {
 return (
  <Routes>
   <Route path="/" element={<MainLayout />}>
    <Route path="" element={<Home />} />
    <Route path="cart" element={<Cart />} />
    <Route path="pizza/:id" element={<FullPizza />} />
    <Route path="*" element={<NotFound />} />
```

```
    </Route>
   </Routes>
  );
}
```

useNavigate

В **FullPizzaBlock/index.jsx:**

```
const navigate = useNavigate();

 React.useEffect(() => {
  async function fetchPizza() {
   try {
...
    } catch (error) {
     alert('Ошибка при получении пиццы');
     navigate('/');
    }
  }
  fetchPizza();
 }, []);
```

Typescript

Установка

https://create-react-app.dev/docs/adding-typescript/

npm install --save typescript @types/node @types/react @types/react-dom @types/jest

если приложение сразу создается с Typescript:

npx create-react-app my-app --template typescript

Настройка eslint

npm install --save-dev @typescript-eslint/parser @typescript-eslint/eslint-plugin eslint

Создаем файл **.eslintrc.json**:

```
{
 "env": {
  "browser": true,
  "es6": true,
  "es2021": true
 },
 "extends": [
  "eslint:recommended",
  // Отключаем правила из базового набора
  "plugin:@typescript-eslint/eslint-recommended",
  // Базовые правила для TypeScript
  "plugin:@typescript-eslint/recommended",
  // Правила TS, требующие инфо о типах
  "plugin:@typescript-eslint/recommended-requiring-type-checking"
 ],
 // Движок парсинга
 "parser": "@typescript-eslint/parser",
 "parserOptions": {
  // Движку нужен проект TS для правил с типами
  "project": "tsconfig.json",
  "tsconfigRootDir": "."
 },
 // Плагин с наборами правил для TypeScript
 "plugins": ["@typescript-eslint"],
 "rules": {}
}
```

В **settings.json**:

```
"eslint.validate": ["javascript", "javascriptreact", "typescriptreact", "typescript"],
```

tsconfig.json:

```
{
 "compilerOptions": {
```

```
    "target": "es5",
    "lib": ["dom", "dom.iterable", "esnext"],
    "allowJs": true,
    "skipLibCheck": true,
    "esModuleInterop": true,
    "allowSyntheticDefaultImports": true,
    "strict": true,
    "forceConsistentCasingInFileNames": true,
    "noFallthroughCasesInSwitch": true,
    "module": "esnext",
    "moduleResolution": "node",
    "resolveJsonModule": true,
    "isolatedModules": true,
    "noEmit": true,
    "jsx": "react-jsx"
  },
  "include": ["src", "src/@types"]
}
```

Переименовываем на **src/index.tsx** и на **src/App.tsx**

В **src/index.tsx:**

```
import App from './App.tsx';
```

Перепишем **src/index.tsx:**

```
import ReactDOM from 'react-dom/client';
import App from './App';
import { BrowserRouter } from 'react-router-dom';
import { store } from './redux/store';
import { Provider } from 'react-redux';

const rootElem = document.getElementById('root');

if (rootElem) {
  const root = ReactDOM.createRoot(rootElem);

  root.render(
    <BrowserRouter>
      <Provider store={store}>
        <App />
      </Provider>
    </BrowserRouter>,
  );
}
```

Другой вариант из шаблона, если сразу создать приложение с typescript:

```
...
const root = ReactDOM.createRoot(
  document.getElementById('root') as HTMLElement
);
root.render(
  <React.StrictMode>
    <App />
        1.    </React.StrictMode>
);

// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();
```

Типизируем проект

1. **Типизация компонентов.** Во всех компонентах нужно установить **тип компонента** React.FC:

```
const NotFoundBlock: React.FC
```

Также в компонентах вместо

```
function Category() {}
```

лучше использовать синтаксис со стрелочными функциями:
```
const Categories: React.FC = () => {}
```

2. Типизация хуков useState:
```
 const [pizza, setPizza] = React.useState<{
   imageUrl: string;
   title: string;
   price: number;
 }>();
```

В данном случае ничего не передали в начальное состояние (в круглых скобках), может быть **undefined**. Если хотим, чтобы был **строго объект**, то в качестве начального состояния передать объект с заданными свойствами

Для useParams, useNavigate, useEffect не нужно писать типы, не имеет смысла.

3. Создадим глобальные типы для импорта модулей с картинками и др.
Для установки связи с внешними файлами скриптов javascript в TS служат декларативные или заголовочные файлы. Это файлы с расширением .d.ts, они описывают синтаксис и структуру функций и свойств, которые могут использоваться в программе, не предоставляя при этом конкретной реализации.

Создадим **src/@types/assets.d.ts:**

```
declare module '*.svg' {
  const content: any;
  export default content;
}

declare module '*.png' {
  const content: any;
  export default content;
}

declare module '*.scss' {
  const content: any;
  export default content;
}
```

Также в **tsconfig.json** добавим строку:
```
{
  "compilerOptions": {
    "target": "es5",
...
  },
  "include": ["src", "src/@types"]
}
```

4. Мы можем создавать свои типы данных. Н-р:
```
type SortItem = {
  name: string;
  sortProperty: string;
};

export const sortList: SortItem[] = [
  { name: 'популярности (DESC)', sortProperty: 'rating' },
  { name: 'популярности (ASC)', sortProperty: '-rating' },
...
];
```

5. Типизируем useRef
```
const sortRef = React.useRef(null);
```
В случае React.useRef() элемент по умолчанию будет **undefined**, и свойства **current** у него нет. typescript требует по умолчанию для current **null** или **html-элемент**.

Можем сделать так:
```
const sortRef = React.useRef<HTMLDivElement>(null);
```

Тип элемента можно посмотреть в коде на элементе, где атрибут ref

Может быть проблема, н-р в:
```
inputRef.current.focus();
```
Здесь при **null** нельзя применять метод **focus.** Есть 2 варианта решения:
```
  if (inputRef.current) {
    inputRef.current.focus();
  }
```
Второй вариант (предпочтительнее):
```
inputRef.current?.focus();
```

6. Типизируем пропсы

Например, в **Categories.tsx:**
```
type CategoriesProps = {  // создаем свой тип
 value: number;
 onChangeCategory: any;
};

const Categories: React.FC<CategoriesProps> = ({ value, onChangeCategory }) => {...}
```
или
```
const Categories: React.FC= ({ value, onChangeCategory }: CategoriesProps) => {
```

7. Типизация сторонней библиотеки

Установка (см. информацию об ошибке в VS Code, что предлагается сделать)

npm i @types/lodash.debounce

Может понадобиться перезагрузка приложения

8. Если не хотим пока типизировать, то временно можно добавить:

// @ts-ignore

9. Типизация onClick, onChange и Event (TypeScript)

Описываем функцию в дочернем компоненте:
```
type CategoriesProps = {
 value: number;
 onChangeCategory: (idx: number) => void; // описываем параметры и возвращаемое значение
};
```

Чтобы указать переменной тип функции, нужно в круглых скобках перечислить параметры и их типы, а после стрелки => указать тип возвращаемого значения.

В родительском компоненте:
```
 const onChangeCategory = (id: number) => {
   dispatch(setCategoryId(id));
 };
...
 return (
...
      <Categories value={categoryId} onChangeCategory={onChangeCategory} />
...
```

В пропсах должно быть передано ровно столько параметров, сколько заявлено. Если какой-нибудь параметр не обязателен, использовать знак вопроса:
```
type CategoriesProps = {
 value: number;
 onChangeCategory?: (idx: number) => void; // описываем параметры и возвращаемое значение
};
```

Типизируем функцию с Event-параметром:
```
 const onChangeInput = (event: React.ChangeEvent<HTMLInputElement>) => {
   setValue(event.target.value);
   updateSearchValue(event.target.value);
 };
```
Здесь посмотрели в коде в подсказке, что требуется **React.ChangeEvent**. **<HTMLInputElement>** требуется указать, так как для него определено **value**. Тогда ошибки не будет.

Следующие записи эквивалентны в пропсах:
```
onClick:React.MouseEventHandler<HTMLButtonElement>;
onClick:(event: React.MouseEvent<HTMLButtonElement>) => void;
```

Для **сортировки** используем обработчик клика, навешанный на body. Для его типизации используем "костыль":
В **Sort.tsx** создаем свой тип:

```
type PopupClick = MouseEvent & {   // добавляем к типу MouseEvent свойство path
  path: Node[];
};
...
const Sort: React.FC = () => {
...
  React.useEffect(() => {
    const handleClickOutSide = (event: MouseEvent) => {
      const _event = event as PopupClick;
      if (sortRef.current && !_event.path.includes(sortRef.current)) {
        setOpen(false);
      }
    };
    document.body.addEventListener('click', handleClickOutSide);
...
```

Костыль нужен, так как у события **MountEvent** может не быть **path**

Другой вариант - вместо **path** использовать composedPath (есть у event):

```
  React.useEffect(() => {
    const handleClickOutSide = (event: MouseEvent) => {
      if (sortRef.current && !event.composedPath().includes(sortRef.current)) {
        setOpen(false);
      }
    };
```

JavaScript метод **composedPath()** объекта Event возвращает путь события, представляющий собой массив объектов, на которых будут вызваны обработчики событий.

10. **enum - Перечисления -** это удобный способ создания наборов значений, количество и названия которых известны заранее и не должны изменяться. Каждое перечисление создает свой собственный тип данных.

```
enum Status {
  LOADING = 'loading',
  SUCCESS = 'success',
  ERROR = 'error',
}
...
const initialState: PizzaSliceState = {
  items: [],
  status: Status.LOADING, // будет 'loading'
};
...
```

Типизация Redux Toolkit

https://redux-toolkit.js.org/usage/usage-with-typescript

Всем файлам из папки **redux** дать расширение .ts

Тип never указывается, если typescript не знает, что это за тип, н-р, в массиве не знает какого типа элементы в нем

Типизируем состояние

Для **initialState** сделаем interface. Интерфейс от **type** отличается тем, что принимает только объект. Для **state** обычно используют интерфейс.

В **store.ts:**

```
// Вытаскиваем глобальный state, ReturnType - превратить содержимое функции в тип
export type RootState = ReturnType<typeof store.getState>;
```

В **cartSlice.ts:**

```
import { RootState } from '../store';

interface CartSliceState {   // созданный интерфейс
  totalPrice: number;
```

```
  items: CartItem[];
}

type CartItem = {  // созданный тип для товара из корзины
  id: string;
  title: string;
  price: number;
  imageUrl: string;
  type: number;
  size: number;
  count: number;
};

const initialState: CartSliceState = {
  totalPrice: 0,
  items: [],
};

const cartSlice = createSlice({
  name: 'cart',
  initialState,
  reducers: {
    addItem(state, action) {
      const findItem = state.items.find((item) => item.id === action.payload.id);

      if (findItem) {
        findItem.count++;
      } else {
        state.items.push({
          ...action.payload,
          count: 1,
        });
      }
...

export const selectCart = (state: RootState) => state.cart;
// с параметром
export const selectCartItemById = (id: string) => (state: RootState) =>
  state.cart.items.find((item) => item.id === id);
...
```

Типизируем actions. Асинхронность

https://redux-toolkit.js.org/usage/usage-with-typescript#createslice

См. из документации **action: PayloadAction<тип для payload>**

Следующую запись можно сократить:

```
type FetchPizzasArgs = {
  order: string;
  sortBy: string;
  category: string;
  search: string;
  currentPage: string;
};
```

на

```
type FetchPizzasArgs = Record<string, string> // Record<ключ, значение>
```

Применяется для массива объектов одного типа

В асинхронном экшене типизировать нужно параметры и результат асинхронной функции. **Есть два варианта синтаксиса:**

```
export const fetchPizzas = createAsyncThunk(
  'pizza/fetchPizzasStatus',
  async (params: Record<string, string>) => {
    const { order, sortBy, category, search, currentPage } = params;

    const { data } = await axios.get(
```

```
`https://62d162dcdccad0cf176680f0.mockapi.io/items?page=${currentPage}&limit=4&search=${search}&${category}&sortBy=${sortBy}&order=${order}`,
    );

    return data as Pizza[];
  },
);
```

Другой вариант:

```
export const fetchPizzas = createAsyncThunk<Pizza[] //что возвращать, Record<string, string> //параметры >(
  'pizza/fetchPizzasStatus',
  async (params) => {
    const { order, sortBy, category, search, currentPage } = params;
    //return data;  -  глюк axios.get - data:any Поэтому тип строго дадим сами
    const { data } = await axios.get<Pizza[]>(

`https://62d162dcdccad0cf176680f0.mockapi.io/items?page=${currentPage}&limit=4&search=${search}&${category}&sortBy=${sortBy}&order=${order}`,
    );:
    return data;
  },
);
```

extraReducers

Для них код придется переписывать на синтаксис из документации:

```
  extraReducers: (builder) => {
    builder.addCase(fetchPizzas.pending, (state) => {
      state.status = 'loading';
      state.items = [];
    });
    builder.addCase(fetchPizzas.fulfilled, (state, action) => {
      state.status = 'success';
      state.items = action.payload;
    });
    builder.addCase(fetchPizzas.rejected, (state) => {
      state.status = 'error';
      state.items = [];
    });
  },
```

Переделаем dispatch для асинхронного экшена.

Обычный **dispatch** принимает **объект**, а асинхронный экшн возвращает не объект, а функцию.
https://redux-toolkit.js.org/usage/usage-with-typescript#getting-the-dispatch-type

В **Store.js** сделаем тип для **dispatch (хук) (**из документации**)**:

```
export type AppDispatch = typeof store.dispatch;
export const useAppDispatch: () => AppDispatch = useDispatch;
```

?

В компонентах меняем:

```
const dispatch = useAppDispatch();
```

Оптимизация

Перерисовки и др. clsx. ahooks. React.useCallback, React.memo

Разбираемся с уменьшением к-ва товаров в корзине. clsx

Нужно, чтобы к-во товаров было >= 1. Проверку сделаем в компоненте **CartItem.tsx,** а не в **Redux.** Клик нужно сделать по **button.** Кнопку при этом нужно сделать **disable**

```
<button
  disabled={count === 1} ...
</button>
```

Также нужно добавлять свой класс для кнопки при этом. Для можно использовать библиотеки classnames, **clsx** и др.

```
npm install clsx
```

ClickItem.tsx:

```
import clsx from 'clsx';
...
      <button
        disabled={count === 1}
        className={clsx('button button--outline button--circle cart__item-count-minus', {
          'cart__item-count-minus--disabled': count === 1,   })}
        onClick={onClickMinus}> ...
      </button>
```

Добавляем класс **cart__item-count-minus--disabled** и дописываем стили для него

В данном случае библиотеку не нужно было использовать. Можно в **css** см. &:disabled

Перерисовки

Категории и сортировка не должны меняться, н-р, при поиске. В общем случае избавляться от лишних перерисовок нужно тогда, когда это мешает производительности. Мы сделаем это в качестве тренировки.

Разбираемся с причинами перерисовки. В компонент **Category** в пропсах передается функция (обработка клика), а она пересоздается при каждой перерисовке в родительском компоненте **Home.tsx,** а перерисовка происходит при каждом вызове, н-р, useSelector

См. библиотеку **ahooks**
https://ahooks.js.org/hooks/use-request/index
npm install ahooks

Будем использовать для дебажинга хук **useWhyDidYouUpdate.** С его помощью можно узнать - почему компонент перерисовался. Хук импортировать, вызвать внутри компонента. Передать два параметра - название компонента и какие пропсы будут меняться

Categories.tsx:

```
import useWhyDidYouUpdate from 'ahooks/lib/useWhyDidYouUpdate';

const Categories: React.FC<CategoriesProps> = ({ value, onChangeCategory }) => {
  useWhyDidYouUpdate('Categories', { value, onChangeCategory });
...
```

Далее см. консоль при выполнении. Видим, что функция из пропсов в начальную загрузку пересоздается 3 раза. Решение проблемы перерисовки - обернуть функцию из пропсов в React.useCallback:

Home.tsx:

```
  const onChangeCategory = React.useCallback((id: number) => {
    dispatch(setCategoryId(id));
  }, []);  // 2-ой аргумент, зависимости - как у useEffect
```

Дебаггинг - useWhyDidYouUpdate показывает, что перерисовки нет, но в инструментах браузера мы ее видим. Что такое перерисовка - определения ее могут быть немного разными. **Перерисовка компонента** делается в двух случаях - когда меняется **state** или когда меняются **пропсы**.

Чтобы полностью исключить перерисовку, можно весь компонент обернуть в **React.memo**
Если ваш компонент всегда рендерит одно и то же **при неменяющихся пропсах**, вы можете обернуть его в вызов React.memo для повышения производительности в некоторых случаях, мемоизируя тем самым результат. Это значит, что React будет использовать результат последнего рендера, избегая повторного рендеринга.

React.memo затрагивает только **изменения пропсов**. Если функциональный компонент обёрнут в React.memo и использует **useState, useReducer или useContext**, он будет повторно рендериться при изменении состояния или контекста.

По умолчанию он поверхностно сравнивает вложенные объекты в объекте **props**. Если вы хотите контролировать сравнение, вы можете передать свою функцию сравнения в качестве второго аргумента.

Categories.tsx:

```
const Categories: React.FC<CategoriesProps> = React.memo(({ value, onChangeCategory }) => {
   return (...}
);  // memo - без второго аргумента
```

LocalStorage, структурирование папок

LocalStorage

Сохраним пиццы из корзины в LocalStorage.

В **Header.tsx**:

```
 const isMounted = React.useRef(false); // создаем флаг для первого рендера
...
 React.useEffect(() => {
   if (isMounted.current) {
     const json = JSON.stringify(items);
     localStorage.setItem('cart', json); // Сохраняем в LocalStorage только не при первом рендере
   }
   isMounted.current = true;
 }, [items]);
```

Теперь нам нужно из LocalStorage сохранить корзину в redux при первом рендере. Для этого можно было бы в **cartSlice.ts**:

```
const initialState: CartSliceState = {
 totalPrice: 0,
 items: JSON.parse(localStorage.getItem('cart')) || [],
};
```

Однако, typescript не позволяет это сделать. Сделаем 2 вспомогательных функции в спец. папке для таких функций **src/utils/getCartFromLS.ts** и **src/utils/calcTotalPrice.ts:**

```
import { CartItem } from '../redux/slices/cartSlice';
export const calcTotalPrice = (items: CartItem[]) => {
 return items.reduce((sum, obj) => obj.price * obj.count + sum, 0);
};
```

src/utils/getCartFromLS.ts :

```
import { calcTotalPrice } from './calcTotalPrice';
import { CartItem } from '../redux/slices/cartSlice';

export const getCartFromLS = () => {
 const data = localStorage.getItem('cart');
 const items = data ? JSON.parse(data) : [];
 const totalPrice = calcTotalPrice(items);
 return {   items: items as CartItem[],   totalPrice,  };
};
```

cartSlice.ts:

```
...
const initialState: CartSliceState = getCartFromLS();

const cartSlice = createSlice({
 name: 'cart',
 initialState,
 reducers: {
   addItem(state, action: PayloadAction<CartItem>) {
...
     state.totalPrice = calcTotalPrice(state.items);
   },
```

структурирование папок

Переделаем **структуру для redux**, делаем отдельно **типы для redux**. Создадим отдельные папки для каждого slice. В них д.б., н-р, redux/**cart/selectors.ts**, redux/**cart/types.ts**, redux/**cart/slice.ts**

Разделим слайсы по этим файлам, правим импорты. Асинхронный экшн перенесем в отдельный файл redux/**pizza/asyncActions.ts**

Оптимизация сборки

Code Splitting

Бандл - единый файл js после сборки. После сборки в консоли:

```
 92.4 kB (+28 B)  build\static\js\main.bb9a7134.js - бандл
 2.75 kB          build\static\css\main.86ff44b1.css
```

Если его размер будет большим (больше 500 Kb), то нужно заняться разделением кода - **код-сплиттингом** - разделением на куски - чанки (chunk). Есть разные способы для этого:

1. Динамический импорт

Создадим utils/**math.ts.** См. документацию по React. Сделаем динамический импорт из math.ts в любом компоненте:

```
import('../utils/math').then((math) => {
  console.log(math.add(77777, 88888));
});
```

В этом случае создается chunk:

```
93.2 kB (+824 B)  build\static\js\main.df06d461.js
2.75 kB           build\static\css\main.86ff44b1.css
202 B             build\static\js\286.394d7948.chunk.js
```

В инструментах браузера можно в network увидеть, что файл chunk загружается отдельно от основного бандла.

2. Разделение с помощью React.lazy для роутера

Будем страницы грузить отдельно. Для **home** ленивую подгрузку делать не надо. Ленивая загрузка **React.lazy** - не загружать, если компонент не нужно сейчас рендерить

В **App.tsx:**

```
const Cart = React.lazy(() => import('./pages/Cart'));
const FullPizza = React.lazy(() => import('./pages/FullPizza'));
const NotFound = React.lazy(() => import('./pages/NotFound'));

function App() {
  return (
    <Routes>
      <Route path="/" element={<MainLayout />}>
        <Route path="" element={<Home />} />
        <Route
          path="cart"
          element={
            <Suspense fallback={<div>Загрузка корзины...</div>}> // пока страница загружается
              <Cart />
            </Suspense>
          }
        />
        <Route
          path="pizza/:id"
          element={
            <Suspense fallback={<div>Загрузка...</div>}>
              <FullPizza />
            </Suspense>
          }
        />
        <Route
          path="*"
          element={
            <Suspense fallback={<div>Загрузка...</div>}>
              <NotFound />
            </Suspense>
          }
        />
      </Route>
    </Routes>
  );
}
```

Suspense можно также навешать сразу на весь роутер, см. документацию по React

Хотим также названия чанкам дать свои. используем возможности Webpack, подправляем:

```
const Cart = React.lazy(() => import(/*webpackChunkName: "Cart"*/ './pages/Cart'));
const FullPizza = React.lazy(() => import(/*webpackChunkName: "FullPizza"*/ './pages/FullPizza'));
const NotFound = React.lazy(() => import(/*webpackChunkName: "NotFound"*/ './pages/NotFound'));
```

3. Бибилиотека react-loadable

Эта библиотека поддерживает рендеринг на стороне сервера. Для этих же целей можно использовать **Loadable Components.**

React.lazy - решение для браузера, не для сервера. **?**

Установка:
npm i react-loadable
npm i @types/react-loadable - для typescript
В **App.tsx:**

```
...
import Loadable from 'react-loadable';

const Cart = Loadable({
  loader: () => import(/*webpackChunkName: "Cart"*/ './pages/Cart'),
  loading: () => <div>Загрузка корзины...</div>,
});
...
function App() {
  return (
...
      <Route path="cart" element={<Cart />} />
```

Tree Shaking
Если импорт нигде не используется, он не будет в бандле - это делает **tree shaking** модуля **Webpack**

```
import { MyComponent } from '...';
```

React.lazy в настоящее время поддерживает только экспорт по умолчанию. Если модуль, который требуется импортировать, использует именованный экспорт, можно создать промежуточный модуль, который повторно экспортирует его как модуль по умолчанию. Это гарантирует работоспособность tree shaking — механизма устранения неиспользуемого кода. - смю документацию по React

Reexport
В **Home.tsx** есть довольно много импортов компонентов из папки **components**. Можно это подправлять. есть два способа:

1. Создадим components/index.ts:

```
export { default as Skeleton } from './PizzaBlock/Skeleton';
export { default as PizzaBlock } from './PizzaBlock';
export { default as Header } from './Header';
export { default as Categories } from './Categories';
export { default as CartItem } from './CartItem';
export { default as CartEmpty } from './CartEmpty';
export { default as Search } from './Search';
export { default as Pagination } from './Pagination';
export { default as NotFoundBlock } from './NotFoundBlock';
export { default as Sort } from './Sort';

// это равнозначно
import Skeleton from './PizzaBlock/Skeleton';
import PizzaBlock from './PizzaBlock';
...
export {Skeleton, PizzaBlock...};
```

Тогда в **Home.tsx:**
Вместо

```
// import Categories from '../components/Categories';
// import Sort, { sortList } from '../components/Sort';
// import PizzaBlock from '../components/PizzaBlock';
// import Skeleton from '../components/PizzaBlock/Skeleton';
// import Pagination from '../components/Pagination';
//Можем написать
import { Categories, Sort, PizzaBlock, Skeleton, Pagination } from '../components';
```

Проблема такого способа: из **components/index.ts** будут грузиться, пересматриваться сразу все компоненты, что там указаны, даже если при использовании указаны (н-р, в **Home.tsx**) только некоторые из них.

2. Другой способ barrel-export:
Уберем во всех компонентах экспорт по умолчанию default и сделаем просто экспорт, подправляем импорты, н-р:
pages/NotFound.tsx:

```
import { NotFoundBlock } from '../components';
```

Для **pages/** оставим по умолчанию - нужно для ленивой подгрузки

components/index.ts:

```
export * from './PizzaBlock/Skeleton';
export * from './PizzaBlock';
export * from './FullPizzaBlock';
export * from './Header';
export * from './Categories';
export * from './CartItem';
export * from './CartEmpty';
export * from './Search';
export * from './Pagination';
export * from './NotFoundBlock';
export * from './Sort';
```

Мои правки

useSearchParams

Хук **useSearchParams** используется для чтения и изменения строки запроса в URL-адресе для текущего местоположения. Как и собственный хук **React useState**, useSearchParams возвращает массив из двух значений: параметры поиска текущего местоположения и функцию, которая может использоваться для их обновления. Например:

```
const [searchParams, setSearchParams] = useSearchParams();
```

Здесь объект **searchParams** - по сути объект URLSearchParams, который доступен в стандартном коде JavaScript и который позволяет получить из строки запроса параметры и управлять ими.

Применение у меня в **Home.tsx**:

```
const [searchParams, setSearchParams] = useSearchParams();
...
 // не добавлять URL параметры при первом рендере, при всех остальных добавлять
 React.useEffect(() => {
   if (isMounted.current) {
     setSearchParams({   // добавление объекта в адресную строку
       sortProperty: sort.sortProperty,
       categoryId: String(categoryId),
       currentPage: String(currentPage),
     });
   }
   isMounted.current = true;
 }, [categoryId, sort.sortProperty, currentPage]);

 // если рендер первый, то проверяем есть ли URL параметры и сохраняем в redux
 React.useEffect(() => {
   if (window.location.search) {
     const params = Object.fromEntries(new URLSearchParams(searchParams)); // Получим объект из параметров адресной строки
...
 }, []);
```

Использование pickBy и identiti из библиотеки lodash

Используется для того, чтобы исключить из объекта свойства с 0
В **asyncActions.ts:**

```
import pickBy from 'lodash/pickBy';
import identity from 'lodash/identity';

export const fetchPizzas = createAsyncThunk<Pizza[], SearchPizzaParams>(
 'pizza/fetchPizzasStatus',
 async (params) => {
   const { order, sortBy, category, title, page } = params;

   const { data } = await axios.get<Pizza[]>(`https://62d162dcdccad0cf176680f0.mockapi.io/items`, {
     params: pickBy(
       { page,  limit: 8, category, sortBy, order, title  },   identity,  ),
   });

   return data;
 },
);
```