**ОТЧЕТ**

**Step-1**

Изображение выглядит как текст, снимок экрана, Шрифт, меню

Автоматически созданное описание

Для начала создали стили и нужные нам компоненты

import React from "react";

import ReactDOM from "react-dom";

import App from "./components/App/App";

import "./index.css";

ReactDOM.render(<App />, document.getElementById("root"));

Мы импортируем библиотеку React, которая используется для создания компонентов в React-приложениях.

Далее импортируем модуль ReactDOM из библиотеки React, который используется для взаимодействия с DOM (Document Object Model), а именно для отрисовки React-компонентов в реальном DOM.

Дальше мы импортируем компонент App из файла "./components/App/App". Важно отметить, что путь к файлу начинается с текущего каталога (./), что означает, что файл App находится в подкаталоге "components", который в свою очередь расположен в корневом каталоге проекта. <App />: Это синтаксис JSX, который предоставляет удобный способ создания React-элемента на основе компонента App.

document.getElementById("root"): Это DOM-элемент, в который будет производиться отрисовка компонента App. Обычно в HTML-документе должен существовать элемент с идентификатором "root".

ReactDOM.render(<App />, document.getElementById("root"));: Эта строка кода отвечает за отображение React-компонента App в элементе с идентификатором "root" внутри HTML-документа.

**Step-2**

import React from "react";

import "./Header.css";

const Header = () => {

    return (

        <div className="rmdb-header">

            <div className="rmdb-header-content">

                <img src="./images/reactMovie\_logo.png"

                    className="rmdb-logo"

                    alt="rmdb-logo"/>

                <img src="../images/tmdb\_logo.png"

                    className="rmdb-tmdb-logo"

                    alt="rmdb-tmdb-logo"/>

            </div>

        </div>

    );

};

export default Header;

Этот код представляет собой React-компонент Header, который, предназначен для отображения верхней части (заголовка) интерфейса приложения. Для начала мы определяем функциональный компонент Header. Функциональные компоненты представляют собой простые функции, которые принимают props (свойства) и возвращают React-элементы.

**Step-3**

import React, { Component } from "react";

import "./Home.css";

class Home extends Component {

  state = {};

  render() {

    return <div>Home</div>;

  }

}

export default Home;

Также импортировали нужные библиотеки и создали React-компонент Home, который является классовым компонентом и предназначен для отображения главной (страницы) интерфейса приложения. Для начала Мы создаем классовый компонент Home, который расширяет базовый класс Component из библиотеки React. Классовые компоненты обычно используются, когда требуется состояние (state) или методы жизненного цикла компонента. Далее здесь определено состояние компонента. В данном случае, у компонента Home нет конкретного состояния, поэтому используется пустой объект. Если бы у компонента было бы состояние, мы могли бы инициализировать его здесь.

**Step-4**

import React from "react";

import Header from "../elements/Header/Header";

import Home from "../Home/Home";

class App extends React.Component{

    render() {

        return(

            <div>

                <Home />

                <Header />

            </div>

        );

    }

}

export default App;

Этот код представляет собой основной компонент приложения App. Импортируем Реакт и нужные компоненты. Далее мы создаем классовый компонент App, который расширяет базовый класс Component из библиотеки React. Класс App представляет собой основной компонент всего приложения.

render() { ... }: Этот метод отвечает за отображение компонента App. Внутри метода render мы возвращаем JSX-элемент, который представляет собой корневой элемент приложения.

Изображение выглядит как текст, снимок экрана, Шрифт, программное обеспечение

Автоматически созданное описание

**Step-5**

import React, { Component } from "react";

import "./Home.css";

import HeroImage from "../elements/HeroImage/HeroImage";

import SearchBar from "../elements/SearchBar/SearchBar";

import FourColGrid from "../elements/FourColGrid/FourColGrid";

import MovieThumb from "../elements/MovieThumb/MovieThumb";

import LoadMoreBtn from "../elements/LoadMoreBtn/LoadMoreBtn";

import Spinner from "../elements/Spinner/Spinner";

class Home extends Component {

  state = {};

  render() {

    return (

      <div className="rmdb-home">

        <HeroImage />

        <SearchBar />

        <FourColGrid />

        <Spinner />

        <LoadMoreBtn />

      </div>

    );

  }

}

export default Home;

В этом коде компонент Home теперь возвращает блок JSX с различными дочерними компонентами, такими как HeroImage, SearchBar, FourColGrid, Spinner и LoadMoreBtn. Kаждый из этих компонентов отвечает за какую-то часть интерфейса домашнего экрана.

Также для всех компонентов определили функциональный компоненты исоздали классовые компоненты

Изображение выглядит как текст, снимок экрана, Шрифт, программное обеспечение

Автоматически созданное описание

**Step-6**

import React, { Component } from "react";

import "./Home.css";

import {

  API\_URL,

  API\_KEY,

  IMAGE\_BASE\_URL,

  POSTER\_SIZE,

  BACKDROP\_SIZE

} from "../../config";

class Home extends Component {

    state = {

    movies: [],

    heroImage: null,

    loading: false,

    currentPage: 0,

    totalPages: 0,

    searchTerm: ""

  };

  componentDidMount() {

    this.setState({ loading: true });

  }

  render() {

Здесь мы импортировали различные константы, такие как URL API, ключ API, базовый URL для изображений, размеры постеров и фонов, из файла конфигурации ("../../config"). Эти константы, будут использоваться для взаимодействия с внешним API и настройки отображения изображений в вашем приложении. Далее мы определяем начальное состояние компонента Home. Здесь есть различные свойства состояния, такие как movies (список фильмов), heroImage (главное изображение), loading (индикатор загрузки), currentPage (текущая страница), totalPages (всего страниц) и searchTerm (строка поиска). Метод жизненного цикла componentDidMount: В методе componentDidMount компонента Home мы устанавливаем состояние loading в true. Обычно этот метод используется для выполнения каких-то инициализаций, например, для загрузки данных с сервера. В нашем случае, это первичная загрузка данных о фильмах.

**Step-7**

  componentDidMount() {

    this.setState({ loading: true });

    const endpoint = `${API\_URL}movie/popular?api\_key=${API\_KEY}&language=en-US&page=1`;

  }

В нашем коде componentDidMount начинается работа с загрузки данных о популярных фильмах. Мы устанавливаем свойство loading в состоянии в true, что является сигналом начала процесса загрузки данных. Мы формируем конечную точку (endpoint) для запроса к API The Movie Database (TMDb). Эта конечная точка указывает на популярные фильмы, используя наш API ключ и указывая язык и страницу запроса.

**Step-8**

componentDidMount() {

    this.setState({ loading: true });

    const endpoint = `${API\_URL}movie/popular?api\_key=${API\_KEY}&language=en-US&page=1`;

    this.fetchItems(endpoint);

    }

    loadMoreItems = () => {

    let endpoint = "";

    this.setState({

      loading: true

    });

    if (this.state.searchTerm === "") {

      endpoint = `${API\_URL}movie/popular?api\_key=${API\_KEY}&language=en-US&page=${this

        .state.currentPage + 1}`;

    } else {

      endpoint = `${API\_URL}search/movie?api\_key=${API\_KEY}&language=en-US&query${

        this.state.searchTerm

      }$page=${this.state.currentPage + 1}`;

    }

    this.fetchItems(endpoint);

  };

  fetchItems = endpoint => {

    fetch(endpoint)

        .then(result => result.json())

        .then(result => {

            this.setState({

                movies: [...this.state.movies, ...result.results],

                heroImage: this.state.heroImage || result.results[0],

                loading: false,

                currentPage: result.page,

                totalPages: result.totalPages

            });

        });

  };

Здесь мы добавили логику для выполнения запросов к АПИ ТМДБ, обработки данных и обновления состояния компонента ХОУМ.

В методе componentDidMount после установки состояния loading в true, мы вызываем функцию fetchItems, передавая ей конечную точку для запроса популярных фильмов.

Метод loadMoreItems вызывается при запросе дополнительных фильмов (например, при нажатии на кнопку "Load More"). В зависимости от того, есть ли поисковый запрос (searchTerm), он формирует соответствующую конечную точку для запроса.

Метод fetchItems выполняет фактический запрос к API с использованием fetch, обрабатывает полученный результат и обновляет состояние компонента. Фильмы добавляются к текущему списку фильмов, а главное изображение устанавливается, если оно еще не установлено. Затем обновляются страница, общее количество страниц и состояние загрузки.

**Step-9**

return (

      <div className="rmdb-home">

        {this.state.heroImage ? (

            <div>

                <HeroImage

                    image={`${IMAGE\_BASE\_URL}${BACKDROP\_SIZE}${

                        this.state.heroImage.backdrop\_path

                    }`}

                    title={this.state.heroImage.original\_title}

                    text={this.state.heroImage.overview}

                />

                <SearchBar />

            </div>

        ) : null}

        <FourColGrid />

        <Spinner />

        <LoadMoreBtn />

      </div>

    );

Hаш метод render теперь использует полученные данные для отображения HeroImage, SearchBar и других компонентов.

Мы используем условный оператор {this.state.heroImage ? ... : null}, чтобы решить, когда рендерить HeroImage и SearchBar. Если this.state.heroImage существует (не является null или undefined), то рендерятся HeroImage и SearchBar, в противном случае они не отображаются.

В компоненте HeroImage передаются свойства image, title, и text. image формируется с использованием констант IMAGE\_BASE\_URL и BACKDROP\_SIZE в сочетании с путем к изображению героического фона из данных о главном изображении.

SearchBar: Если главное изображение существует, отображается строка поиска.

Затем идут компоненты FourColGrid, Spinner, и LoadMoreBtn, которые в настоящий момент рендерятся всегда.

FourColGrid: Пока не переданы данные для отображения фильмов в FourColGrid, этот компонент будет отображать пустую сетку.

Spinner: Отображает индикатор загрузки, если this.state.loading установлено в true.

LoadMoreBtn: Кнопка "Load More", которую можно нажать для загрузки дополнительных фильмов.

const HeroImage = props => {

    return (

        <div

          className="rmdb-heroimage"

          style={{

            background: `linear-gradient(to bottom, rgba(0,0,0,0)

          39%, rgba(0,0,0,0)

          41%, rgba(0,0,0,0.65)

          100%), url('${props.image}'), #1c1c1c`

          }}

        >

          <div className="rmdb-heroimage-content">

            <div className="rmdb-heroimage-text">

              <h1>{props.title}</h1>

              <p>{props.text}</p>

            </div>

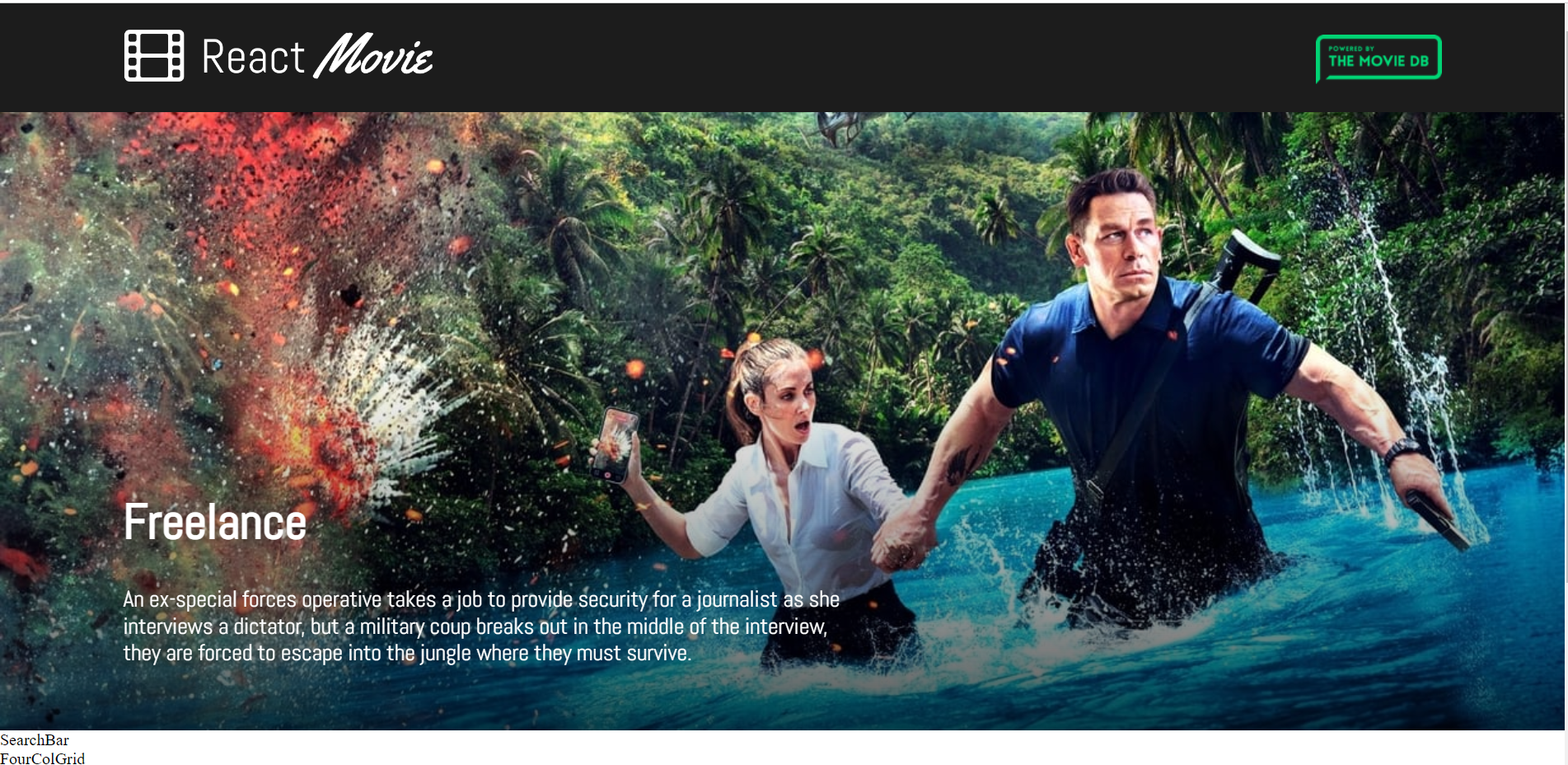
          </div>

        </div>

      );

};

Наш функциональный компонент HeroImage отвечает за отображение главного изображения на домашней странице.



**Step-10**

import React, { Component } from "react";

import "./Home.css";

import {

  API\_URL,

  API\_KEY,

  IMAGE\_BASE\_URL,

  POSTER\_SIZE,

  BACKDROP\_SIZE

} from "../../config";

import HeroImage from "../elements/HeroImage/HeroImage";

import SearchBar from "../elements/SearchBar/SearchBar";

import FourColGrid from "../elements/FourColGrid/FourColGrid";

import MovieThumb from "../elements/MovieThumb/MovieThumb";

import LoadMoreBtn from "../elements/LoadMoreBtn/LoadMoreBtn";

import Spinner from "../elements/Spinner/Spinner";

class Home extends Component {

    state = {

    movies: [],

    heroImage: null,

    loading: false,

    currentPage: 0,

    totalPages: 0,

    searchTerm: ""

  };

  componentDidMount() {

    this.setState({ loading: true });

    const endpoint = `${API\_URL}movie/popular?api\_key=${API\_KEY}&language=en-US&page=1`;

    this.fetchItems(endpoint);

    }

    searchTerm = searchTerm => {

      console.log(searchTerm);

      let endpoint = "";

      this.setState({

        movies: [],

        loading: true,

        searchTerm

      });

      if (searchTerm === "" ) {

        endpoint = `${API\_URL}movie/popular?api\_key=${API\_KEY}&language=en-US&page=1`;

      } else {

        endpoint = `${API\_URL}search/movie?api\_key=${API\_KEY}&language=en-US&query=${searchTerm}`;

      }

      this.fetchItems(endpoint);

    };

    loadMoreItems = () => {

    let endpoint = "";

    this.setState({

      loading: true

    });

    if (this.state.searchTerm === "") {

      endpoint = `${API\_URL}movie/popular?api\_key=${API\_KEY}&language=en-US&page=${this

        .state.currentPage + 1}`;

    } else {

      endpoint = `${API\_URL}search/movie?api\_key=${API\_KEY}&language=en-US&query${

        this.state.searchTerm

      }$page=${this.state.currentPage + 1}`;

    }

    this.fetchItems(endpoint);

  };

  fetchItems = endpoint => {

    fetch(endpoint)

        .then(result => result.json())

        .then(result => {

            this.setState({

                movies: [...this.state.movies, ...result.results],

                heroImage: this.state.heroImage || result.results[0],

                loading: false,

                currentPage: result.page,

                totalPages: result.totalPages

            });

        });

  };

  render() {

    return (

      <div className="rmdb-home">

        {this.state.heroImage ? (

            <div>

                <HeroImage

                    image={`${IMAGE\_BASE\_URL}${BACKDROP\_SIZE}${

                        this.state.heroImage.backdrop\_path

                    }`}

                    title={this.state.heroImage.original\_title}

                    text={this.state.heroImage.overview}

                />

                <SearchBar callback={this.searchItems}/>

            </div>

        ) : null}

        <FourColGrid />

        <Spinner />

        <LoadMoreBtn />

      </div>

    );

  }

}

export default Home;

Метод searchTerm: Этот метод вызывается при изменении строки поиска в компоненте SearchBar. Он устанавливает новое значение searchTerm в состоянии и, в зависимости от того, есть ли поисковый запрос или нет, формирует соответствующую конечную точку запроса. Затем вызывает fetchItems для выполнения запроса и обновления состояния.

Метод render: в компонент SearchBar добавлено свойство callback, которое передает функцию searchTerm в компонент SearchBar. Теперь, при изменении строки поиска в компоненте SearchBar, вызывается функция searchTerm, которая обновляет состояние и выполняет запрос.

class SearchBar extends Component {

  state = {

    value: ""

  };

  timeout = null;

  doSearch = event => {

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

    clearTimeout(this.timeout)

    this.timeout = setTimeout(() => {

      this.props.callback(this.state.value);

    }, 500);

  };

  render() {

    return (

      <div className="rmdb-searchbar">

        <div className="rmdb-searchbar-content">

          <FontAwesome className="rmdb-fa-search" name="search" size="2x"/>

          <input

            type="text"

            className="rmdb-searchbar-input"

            placeholder="Search"

            onChange={this.doSearch}

            value={this.state.value}

          />

        </div>

      </div>

    );

  }

}

timeout: В состоянии компонента есть переменная timeout, которая используется для отслеживания таймера задержки перед выполнением поиска.

Метод doSearch вызывается при изменении значения в поле ввода. Он обновляет состояние value с текущим значением поля ввода. Затем используется clearTimeout для очистки предыдущего таймера (если он был установлен). Устанавливается новый таймер, который вызовет функцию обратного вызова (callback) через 500 миллисекунд после окончания ввода текста. Это позволяет создать задержку перед выполнением фактического поиска, что полезно для уменьшения числа запросов в случае быстрого ввода.

render: JSX содержит иконку поиска (FontAwesome), поле ввода и соответствующие стили. Обработчик onChange связан с методом doSearch, который отслеживает изменения в поле ввода.

const FourColGrid = props => {

    const renderElements = () => {

        const gridElements = props.children.map((element, i ) => {

            return (

                <div key={i} className="rmdb-grid-element">

                    {element}

                </div>

            );

        });

    };

    return (

        <div className="rmdb-grid">

            {props.header && !props.loading ? <h1>{props.header}</h1>

            : null}

            <div className="rmdb-grid-content">

                {renderElements()}

            </div>

        </div>

    )

};

Компонент FourColGrid предназначен для создания сетки с четырьмя колонками. Это может быть полезно, например, для отображения постеров фильмов в четырех колонках.

Добавлена функция renderElements, которая создает массив gridElements на основе дочерних элементов, переданных в компонент через props.children. Возвращается массив gridElements.

Возвращение из render: Возвращение массива gridElements вместо undefined. Теперь функция renderElements вызывается внутри JSX возвращаемого из компонента.

Возвращаемый JSX: Компонент возвращает блок с классом "rmdb-grid". Если переданы props.header и при этом props.loading равно false, то отображается заголовок (h1). В блоке "rmdb-grid-content" отображаются результаты вызова функции renderElements.