Практическая работа №5. Redux, события и бизнес логика.

Цель практической работы заключается в знакомстве с принципами управления состоянием приложения в React и овладении навыками использования Redux, который является одним из наиболее популярных инструментов управления состоянием в React-приложениях, а также в овладении паттернами с разделением слоя представления и бизнес-логики. Это позволит студентам понять, как работает Redux и как его можно использовать для управления состоянием приложения. Это позволит лучше понимать, как разбивать код приложения на модули и как они могут взаимодействовать друг с другом, и в конечном итоге обеспечит создание более легкочитаемого и поддерживаемого кода.

Задачи:

- Создать новый проект.
- Создать форму принятия пользовательского соглашения.
  - Форма должна содержать чекбокс, который следует отметить пользователем, в случае, если он принимает соглашение.
  - Форма должна содержать кнопку подтверждения соглашения, которая становится активной только при активации чекбокса.

Теоретические сведения

Одной из главных проблем в разработке приложений является управление состоянием. Состояние приложения может быть распределено по всему приложению и меняться в ответ на множество событий, таких как ввод пользователя, получение данных из API и т.д. Это может привести к сложной структуре кода, затрудняющей отслеживание и поддержку кода в будущем. Redux решает эту проблему, предоставляя централизованный способ управления состоянием приложения. Использование же паттернов с разделением слоя представления и бизнес-логики играет важную роль в управлении состоянием приложения, предоставляя возможность разделить код, отвечающий за отображение пользовательского интерфейса и код, отвечающий за бизнес-логику.

В предыдущих практических работах рассматривалась реализация бизнес логики приложения прямо в компонентах. И в некоторых ситуациях это удобное решение. Но когда бизнес-логика приложения реализована прямо в компонентах React, это может привести к ряду недостатков:

1. Сложность поддержки: если логика приложения реализована в компонентах, то при каждом изменении логики нужно будет менять соответствующие компоненты. Это может привести к увеличению времени на поддержку приложения и усложнению кода.

2. Ограниченность возможностей: компоненты в React создаются для того, чтобы отображать данные, а не для обработки бизнес-логики. Поэтому, когда логика приложения реализована в компонентах, это может привести к ограниченным возможностям обработки данных и выполнения операций.

3. Нарушение принципа единственной ответственности: если логика приложения реализована прямо в компонентах, то компоненты выполняют сразу несколько функций, таких как обработка данных, отображение пользовательского интерфейса и т.д. Это может привести к усложнению кода, трудностям при тестировании и увеличению вероятности ошибок.

Для решения этих проблем можно использовать паттерн разделения слоя представления и бизнес-логики и инструменты управления состоянием, такие как Redux.

Например, в интернет-магазине можно вынести логику работы с корзиной и каталогом товаров в Redux. Это позволит создавать компоненты, которые будут отображать данные и получать их из хранилища Redux. При изменении данных в хранилище, эти компоненты будут обновляться автоматически. Таким образом, можно избежать проблем, связанных с недостатками реализации бизнес-логики в компонентах.

Redux - это библиотека управления состоянием для JavaScript-приложений, основанная на шаблоне проектирования Flux [23]. Она позволяет управлять состоянием приложения в едином хранилище и обеспечивает предсказуемую модель данных. Основная цель Redux заключается в упрощении управления состоянием приложения, предоставляя стандартный способ изменения состояния через действия (actions) и управления доступом к этому состоянию через хранилище (store).

Redux работает на основе трех основных принципов: только единственный источник истины, состояние только для чтения, все изменения только через чистые функции.

Только единственный источник истины (англ. Single source of truth (SSOT) - это концепция, которая используется в различных областях программирования и информационных технологий. В контексте управления состоянием приложений, SSOT означает, что в приложении должен быть

только один источник правды (или "истина") для всей информации, которую приложение использует [24]. Единственный источник истины: в Redux все состояние приложения хранится в единственном хранилище (store), которое является неизменяемым объектом.

В контексте Redux, SSOT означает, что состояние всего приложения хранится в одном глобальном объекте, который называется "Store" [24]. В этом объекте хранятся все данные, которые нужны приложению для функционирования, включая данные о пользователе, настройки приложения, данные, полученные из внешних источников и т.д. Как только состояние приложения изменяется, оно автоматически обновляется в "Store" и затем отображается на соответствующих компонентах. Это гарантирует, что все компоненты всегда будут иметь актуальную информацию о состоянии приложения и будут работать с одним и тем же набором данных.

Такой подход имеет множество преимуществ. Во-первых, он делает код более понятным и легко поддерживаемым, так как все данные хранятся в одном месте и не разбросаны по всему приложению. Во-вторых, это улучшает производительность приложения, так как компоненты не должны постоянно обращаться к базе данных или другим источникам данных, чтобы получать актуальную информацию. В-третьих, это упрощает процесс отладки приложения, так как все ошибки, связанные с данными, будут происходить только в одном месте - в "Store".

Состояние только для чтения (англ. read-only state, иногда, особоенно в англоязычной литературе, может использоваться термин immutable state) в Redux - это концепция, в которой любые изменения состояния происходят только путем создания новых объектов состояния вместо изменения существующего состояния. Это гарантирует, что состояние Redux остается неизменным и предсказуемым в любом месте приложения.

Redux использует принцип непрямого изменения состояния, который означает, что компоненты не могут изменять состояние Redux напрямую [24]. Вместо этого они вызывают действия (actions), которые передают данные и определяют, что должно измениться в состоянии. Далее, эти действия передаются в редьюсеры (reducers), которые обрабатывают изменения и возвращают новый объект состояния.

Состояние только для чтения является ключевой концепцией в Redux, потому что оно позволяет гарантировать предсказуемость состояния во всем приложении. Это означает, что любые компоненты могут получить доступ к

состоянию Redux, не беспокоясь о том, что другой компонент изменит его непредсказуемым образом.

Концепция «состояние только для чтения» является широко используемым подходом для управления состоянием в современных JavaScript-фреймворках и библиотеках, таких как React, Vue и Redux.

Все изменения только через чистые функции является важной концепцией, которая констатирует, что все изменения состояния приложения должны происходить только через, так называемые, чистые функции, которые не изменяют данные напрямую, а создают новую копию состояния на основе старого. Чистые функции в Redux называются "редьюсерами". Они принимают на вход текущее состояние приложения и объект "действия", описывающий, какое изменение нужно произвести в состоянии. Редьюсер должен вернуть новое состояние приложения на основе текущего состояния и действия.

Для иллюстрации этой концепции рассмотрим простой пример редьюсера для счетчика, исходный код которого показан в листинге 5.1.

Листинг 5.1 - Пример реализации редьюсера для счетчика

```
const initialState = { count: 0 };

function counterReducer(state = initialState, action) {
  switch (action.type) {
    case 'INCREMENT':
      return { count: state.count + 1 };
    case 'DECREMENT':
      return { count: state.count - 1 };
    case 'RESET':
      return { count: 0 };
    default:
      return state;
  }
}
```

В этом примере есть начальное состояние с одним свойством *count*, которое равно нулю. Затем мы определяем функцию-редьюсер *counterReducer*, которая принимает текущее состояние (initialState) и действие (action), и

возвращает новое состояние. Внутри функции мы используем оператор switch для обработки различных типов действий (action.type).

Например, если действие имеет тип INCREMENT, то возвращаем новый объект состояния с увеличенным значением count. Аналогично, если действие имеет тип DECREMENT, то возвращаем новый объект состояния с уменьшенным значением count. Если действие имеет тип RESET, возвращаем объект состояния со значением count, равным нулю. И, наконец, если действие неизвестно, мы просто возвращаем текущее состояние.

Теперь мы можем использовать этот редьюсер в связке с Redux store, о котором мы говорили выше, для управления состоянием счетчика. Когда мы отправляем действие на увеличение счетчика (например, dispatch({ type: 'INCREMENT' })), Redux вызывает наш редьюсер, который изменяет состояние, и все подписчики Redux store (например, компоненты React) получают обновленное состояние и обновляют свой интерфейс. Это даёт гарантию того, что изменение состояния приложения полностью контролируемое и предсказуемое и все подписчики получать и будут работать с обновлёнными данными.

Как видно, в Redux, как и во многих похожих решениях используется событийная модель, основанная на классическом паттерне издатель-подписчик , где Redux store (единственный источник истины) является издателем, а компоненты-подписчики, которые подписываются на изменения состояния, чтобы реагировать на них. Когда состояние Redux store изменяется, Redux генерирует событие (event), и все компоненты, которые подписались на это событие, получают обновленное состояние. Этот процесс позволяет обновлять пользовательский интерфейс (UI) приложения в реальном времени, отображая актуальное состояние приложения.

Событийная модель в Redux реализуется с помощью функции *subscribe()*, которая принимает функцию-обработчик (handler) и добавляет ее в список подписчиков. При каждом изменении состояния, Redux вызывает все зарегистрированные функции-обработчики, которые получают обновленное состояние и могут производить необходимые действия.

Пример использования событийной модели в Redux показан в листинге 5.2.

Листинг 5.2. - Пример использования событийной модели

```
import { createStore } from 'redux';


// Редьюсер для счетчика
```

```
const counterReducer = (state = 0, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1;
    case 'DECREMENT':
      return state - 1;
    default:
      return state;
  }
};

// Создание хранилища
const store = createStore(counterReducer);

// Подписка компонента на изменения состояния
const unsubscribe = store.subscribe(() => {
  console.log(store.getState());
});

// Диспетчеризация действий для изменения состояния
store.dispatch({ type: 'INCREMENT' });
store.dispatch({ type: 'INCREMENT' });
store.dispatch({ type: 'DECREMENT' });

// Отписка компонента от изменений состояния
unsubscribe();
```

В этом примере мы создаем объект Redux store с помощью функции *createStore()*, передавая ей редьюсер (*counterReducer*), который обрабатывает действия (actions) для изменения состояния. Затем мы подписываемся на изменения состояния с помощью функции *subscribe()* и диспетчеризируем (dispatch) действия для изменения состояния. При каждом изменении состояния, функция-обработчик, переданная в *subscribe()*, будет вызываться и получать обновленное состояние. После выполнения всех действий мы отписываемся от изменений состояния с помощью функции *unsubscribe()*.

Слой бизнес логики и использование Redux в React приложениях

До появления функций (хуков), с помощью которых можно «подцепиться» к состоянию и методам жизненного цикла React из функциональных компонентов, общую логику компонентов выносили либо в функции, которые принимали компонент и возвращали новый компонент, либо описывали прямо в корневом для этой логики компоненте. Хуки отлично подходят для слоя бизнеса логики и переиспользования общей логики в разных компонентах. Хуки не работают внутри классов — они дают вам возможность использовать React без классов.

Первое, чего не хватает при работе с хуками и то что даёт Redux — общее состояние всего приложения. Общее состояние — это отдельный слой, который позволяет передвигать по истории и сохранять состояние приложения. Redux предоставляет инструменты для отладки общего состояния приложения. Таким образом можно пролистать все виды интерфейса пользователя просто меняя состояние.

Это все хорошо, однако что такое Redux? **Redux** — это паттерн и библиотека для управления и обновления состояния приложения с использованием специальных событий, называемых *action*. Он предоставляет централизованное хранилище состояние, которое используется во всём приложении с правилами, гарантирующими предсказуемое изменение этого состояния. Поток данных в Redux продемонстрирован на рисунке 5.1.

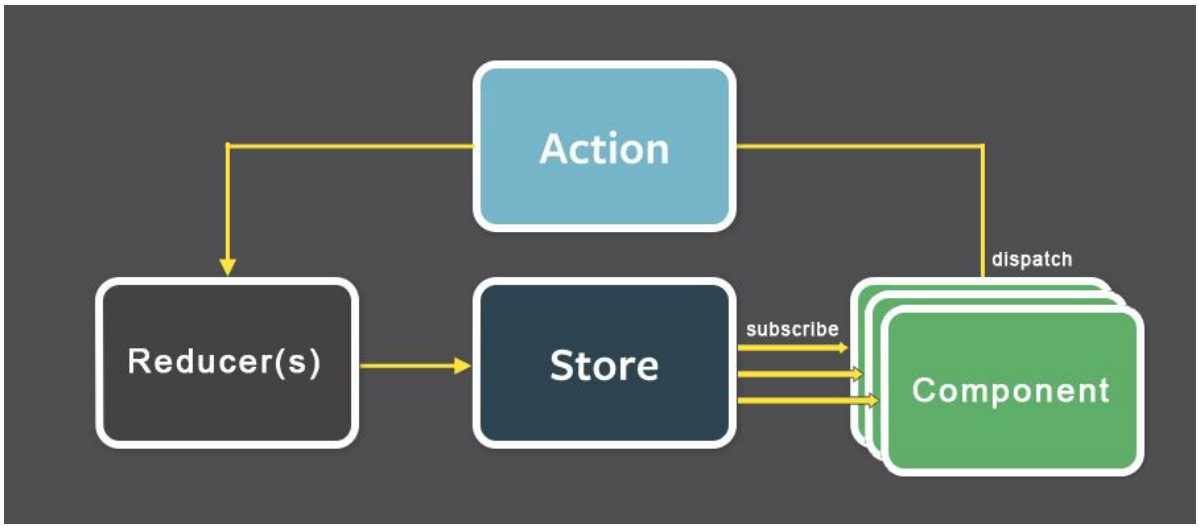

Рисунок 5.1. Схема потока данных в Redux.

При необходимости изменения состояния, например, при клике на элемент в DOM, вызывается **Action creator**, который создаёт определенный **Action**. Этот **Action** с помощью метода **Dispatch** отправляется в **Store**, где он

передаётся на обработку в **Reducers**. Редьюсеры, в свою очередь, на основании текущего состояния и информации, которая находится в экшене, возвращают новое состояние приложения, которое принимает **React** с помощью **Selectors** для нового рендера DOM. Более подробно о каждом компоненте **Redux** будет рассказано ниже по ходу разработки приложения.

Такой однонаправленный поток данных даёт множество преимуществ, таких как — ослабление связанности компонентов, один источник информации о том, как должно выглядеть и действовать приложение, плюс разделение бизнес-логики и отображения, что приводит к значительному упрощению тестирования кода.

Action

Что такое **Action**? Это обычный Javascript объект, у которого есть обязательное свойство **type**, в котором содержится, как правило, осознанное имя экшена. Создатели Redux рекомендуют формировать строку для свойства type по шаблону **домен/событие**. Также в нём может присутствовать дополнительная информация, которая, обычно, складывается в свойство **payload**. Экшены создаются с помощью **Action Creators** - функций, которые возвращают экшены.

3. Reducer

Редьюсеры - это специальные чистые функции, которые принимают на вход текущий **state** и **action**, решают как нужно изменить состояние, если это требуется, и возвращают новый **state**. Редьюсеры должны содержать всю бизнес-логику приложения, насколько это возможно.

Хранить состояние чекбоксов (отмечены они или нет) мы будем простым объектом, где ключом будет выступать название чекбокса, а в булевом значении непосредственно его состояние.

По поводу Reducer в официальной документации по React приводится следующее (ссылка: https://redux.js.org/style-guide/style-guide#put-as-much-logic-as-possible-in-reducers):

«Поместите как можно больше логики в редьюсеры.

По возможности старайтесь поместить как можно больше логики для вычисления нового состояния в соответствующий редьюсер, а не в код, который подготавливает и отправляет действие (например, обработчик кликов). Это помогает обеспечить легкость тестирования большей части фактической логики приложения, позволяет более эффективно использовать отладку с перемещением во времени и помогает избежать распространенных ошибок, которые могут привести к мутациям и ошибкам.

Существуют допустимые случаи, когда некоторые или все новые состояния должны быть сначала рассчитаны (например, создание уникального идентификатора), но это должно быть сведено к минимуму.»

4. Selectors

Селекторы - это функции, которые умеют извлекать требуемую информацию из общего состояния приложения. Если в нескольких частях приложения требуется одинаковая информация, используется один и тот же селектор.

5. Акторы

Часть логики приложения при использовании Redux попадает в Reducer. Однако стоит заметить, что редьюсеры синхронные. Нельзя в нём сделать запрос на сервер и вернуть результат Redux. Для добавления синхронности был разработан специальный механизм - middleware. Этот механизм позволяет выполнить асинхронные действия *вместо*, *до* или *после* момента отправки действия. Поток данных в таком случае выглядит однонаправленным.

Но есть акторы. которые подписываются на изменение состояния и делают асинхронные действия в ответ на изменения какого-то значения в состоянии. В этом случае поток данных выглядит следующим образом: Вызов события в интерфейсе пользователя —> вызов экшен Redux и изменение состояние Store —> актор, подписанный на изменение этого состояния, сделал асинхронные действия и вызвал один или несколько экшенов Redux.

Рассмотрим пример использования Redux в React приложении

Рассмотрим подробно пример простого React приложения с разделением слоя представления и бизнес-логики, использующего Redux. Для этого представим, что у нас есть простое приложение, представляющее собой планировщик ежедневных задач, в котором пользователь может создавать и удалять задачи. Ниже на рисунке 5.2 представлена структура данного проекта.

В директории *actions* у нас будет файл *taskActions.js*, где мы определим все необходимые действия (actions) для управления задачами. В нашем случае, это может быть создание задачи (ADD_TASK) и удаление задачи (REMOVE_TASK). Каждое действие будет представлять собой функцию, которая возвращает объект с типом действия и дополнительными данными при необходимости. Исходный код показан в листинге 5.3.

Листинг 5.3 - Содержимое файла taskActions.js

```
export const ADD_TASK = 'ADD_TASK';
export const REMOVE_TASK = 'REMOVE_TASK';

export const addTask = (task) => {
  return {
    type: ADD_TASK,
    payload: task,
  };
};

export const removeTask = (taskId) => {
  return {
    type: REMOVE_TASK,
    payload: taskId,
  };
};
```

В директории *reducers* у нас будет файл *taskReducer.js*, где мы определим редьюсер (reducer), который будет обрабатывать действия, связанные с задачами. Редьюсер будет иметь начальное состояние, которое будет представлять собой массив задач. Когда происходит действие, редьюсер проверяет его тип и возвращает новое состояние на основе текущего состояния и данных, переданных в действии. Исходный код показан в листинге 5.4.

Листинг 5.4 - Содержимое файла taskReducer.js

```
import { ADD_TASK, REMOVE_TASK } from '../actions/taskActions';

const initialState = {
  tasks: [],
};

const taskReducer = (state = initialState, action) => {
  switch (action.type) {
    case ADD_TASK:
      return {
```

```
            ...state,
            tasks: [...state.tasks, action.payload],
          };
        case REMOVE_TASK:
          return {
            ...state,
            tasks: state.tasks.filter((task) => task.id !== action.payload),
          };
        default:
          return state;
      }
    };

    export default taskReducer;
```

В директории *components будет* реализован файл TaskForm.js, который будет отвечать за форму создания новых задач, и файл TaskList.js, который будет отображать список задач. Компоненты будут использовать хук *useSelector* из библиотеки react-redux для доступа к состоянию задач и хук *useDispatch* для отправки действий. Исходный код показан в листингах 5.5 и 5.6.

Листинг 5.5 - Содержимое файла TaskForm.js

```
    //подключаем необходимые модули
    import React, { useState } from 'react';
    import { useDispatch } from 'react-redux';
    import { addTask } from '../actions/taskActions';

    const TaskForm = () => {
      // Использование useState для создания состояния taskName и setTaskName для управления значением поля ввода
      const [taskName, setTaskName] = useState('');
      // Получение экземпляра метода dispatch из хука useDispatch
      const dispatch = useDispatch();
      // Обработчик отправки формы
      const handleSubmit = (e) => {
        e.preventDefault();
```

```
    if (taskName.trim() === '') {
      return;
    }
    // Создание новой задачи
    const newTask = {
      id: Date.now(),
      name: taskName,
    };
    // Отправка действия addTask с новой задачей в хранилище Redux
    dispatch(addTask(newTask));
    // Очистка поля ввода
    setTaskName('');
  };

  // Возвращение разметки формы
  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={taskName}
        onChange={(e) => setTaskName(e.target.value)}
        placeholder="Enter task name"
      />
      <button type="submit">Add Task</button>
    </form>
  );
};

export default TaskForm;
```

Разберём данный исходный код более подробно, для лучшего понимания функционирования приложения. В данном исходном коде используются:

- *React:* модуль React, необходим для создания React-компонентов.
- *useState*: хук, позволяющий использовать состояние в функциональных компонентах.
- *useDispatch*: хук, позволяющий использовать метод dispatch для отправки действий Redux.

• *addTask*: импортируемое действие Redux для добавления задачи.

Хук *useState* используется для создания состояния *taskName* и функции *setTaskName* для управления значением поля ввода. Далее осуществляется получение экземпляра метода dispatch с помощью хука *useDispatch*. Ниже определяется обработчик *handleSubmit* , который необходим для обработки отправки формы. При этом внутри обработчика идёт проверка на заполненость *taskName,* значение которого будет использоваться для объекта задачи. Далее создается новый объект *newTask* с уникальным идентификатором (полученным с помощью *Date.now()*) и значением из *taskName.* Далее происходит отправка действия *addTask* с новой задачей в хранилище Redux с помощью dispatch. После отправки действия, состояние *taskName* очищается путем вызова *setTaskName('')*. После этого возвращается разметка формы, которая содержит поле ввода и кнопку добавления задачи.

Листинг 5.5 - Содержимое файла TaskList.js

```
import React from 'react';
import { useDispatch, useSelector } from 'react-redux';
import { removeTask } from '../actions/taskActions';

const TaskList = () => {
  const tasks = useSelector((state) => state.tasks);
  const dispatch = useDispatch();

  const handleRemoveTask = (taskId) => {
    dispatch(removeTask(taskId));
  };

  return (
    <ul>
      {tasks.map((task) => (
        <li key={task.id}>
          {task.name}
          <button onClick={() => handleRemoveTask(task.id)}>Remove</button>
        </li>
      ))}
```

```
    </ul>
  );
};

export default TaskList;
```

В данном исходном коде используются практически все те же самые компоненты, что и в предыдущем листинге, кроме добавленного хука *useSelector* который позволяет получать данные из хранилища Redux. При этом хук *useState* не используется. Импортируемое действие Redux *removeTask,* предназначено для удаления задачи.

После импорта требуемых модулей, происходит определение компонента *TaskList.* Здесь хук *useSelector* используется для получения списка задач из хранилища Redux. Функция-селектор принимает текущее состояние Redux и возвращает нужные данные. С помощью хука *useDispatch* осуществляется получение экземпляра метода *dispatch.* Далее определяется обработчик *handleRemoveTask,* который, как видно из названия, используется для удаления задачи. Внутри обработчика вызывается dispatch с действием removeTask и передается идентификатор задачи для удаления. Далее осуществляется возврат разметки списка задач с использованием *tasks.map*, где каждая задача представлена в виде элемента списка *<li>*. Для каждой задачи выводится ее название и кнопка "Remove", при нажатии на которую вызывается обработчик *handleRemoveTask* с передачей идентификатора задачи. Стоит отметить, что каждый элемент списка имеет уникальный ключ *key*, который устанавливается на *task.id*.

В файле App.js мы будем объединять все компоненты вместе. Здесь мы также будем использовать компонент-обёртку *Provider* из react-redux, чтобы обернуть всё приложение и предоставить доступ к хранилищу Redux. *Store* здесь является импортируемым объектом, представляющий хранилище Redux. Исходный код показан в листингах 5.7.

Листинг 5.7 - Содержимое файла App.js

```
import React from 'react';
import { Provider } from 'react-redux';
import store from './store';
import TaskForm from './components/TaskForm';
import TaskList from './components/TaskList';
```

```
const App = () => {
  return (
    <Provider store={store}>
      <div>
        <h1>Task Planner</h1>
        <TaskForm />
        <TaskList />
      </div>
    </Provider>
  );
};

export default App;
```

Здесь, внутри компонента *App* используется компонент *Provider,* который оборачивает все компоненты приложения и предоставляет им доступ к хранилищу Redux. Далее передаётся объект *store* в качестве пропса *store* компоненту *Provider*, чтобы он мог установить связь между хранилищем и компонентами приложения. Далее осуществлена разметка компонента *App*, которая состоит из элемента * *, заголовка *<h1>* и двух компонентов *TaskForm* и *TaskList,* которые были определены ранее. Напомним, что компонент *TaskForm* представляет форму для добавления задач, а компонент *TaskList* представляет список задач.

В файле index.js осуществляется рендеринг компонента *App* в корневой элемент HTML-документа. Исходный код показан в листингах 5.8.

Листинг 5. - Содержимое файла index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

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

Это относительно простой пример React приложения с использованием Redux для управления состоянием и разделением слоя представления и бизнес-логики. В данном примере был создан планировщик задач, где пользователь

может создавать и удалять задачи. Состояние задач хранится в Redux хранилище, а компоненты *TaskForm* и *TaskList* отображают и обрабатывают данные с помощью Redux действий и редьюсеров.

Задание на самостоятельное выполнение

Для выполнения работы необходимо установить Redux. Для этого можно воспользоваться следующей командой: *npm i redux react-redux redux-mock-store @types/redux @types/react-redux @types/redux-mock-store*

Листинг и примеры кода можно найти по следующей ссылке: https://ivahaev.ru/how-to-get-profit-with-redux-and-react/

Дополнительный материал

Краткий обзор хуков - https://ru.reactjs.org/docs/hooks-overview.html (дата обращения: 01.02.2022).

Философия React - https://ru.reactjs.org/docs/thinking-in-react.html (дата обращения: 01.02.2022).

React Hooks и советы по избежанию бесполезного рендеринга компонентов, применяемого к спискам — https://blog.theodo.com/2022/01/react-list-hooks-avoid-render/ (дата обращения: 01.02.2022).