React
Redux ToolKit

Что такое Redux ToolKit
configureStore
createSlice
createAsyncThunk
createReducer
createSelector
createEntityAdapter
RTK Query

Redux ToolKit

Redux Toolkit — это официальный рекомендуемый способ работы с Redux, библиотекой для управления состоянием в JavaScript-приложениях. Он решает основные проблемы, с которыми сталкиваются разработчики при использовании Redux, такие как избыточный код, сложность настроек и работа с иммутабельностью.

Основные моменты:

- RTK упрощает процесс настройки Redux и уменьшает объем кода.
- Встроенные инструменты для работы с асинхронностью и API.
- Поддержка мутабельного синтаксиса с помощью Immer.
- Хорошая интеграция с TypeScript.

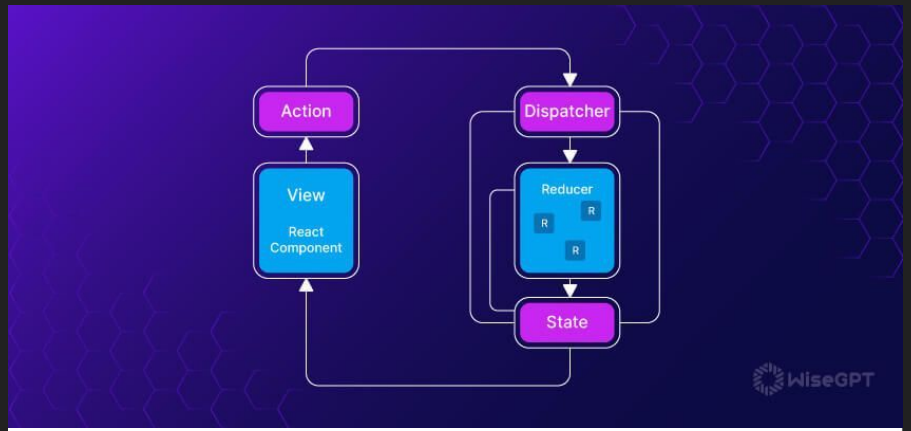
Установка

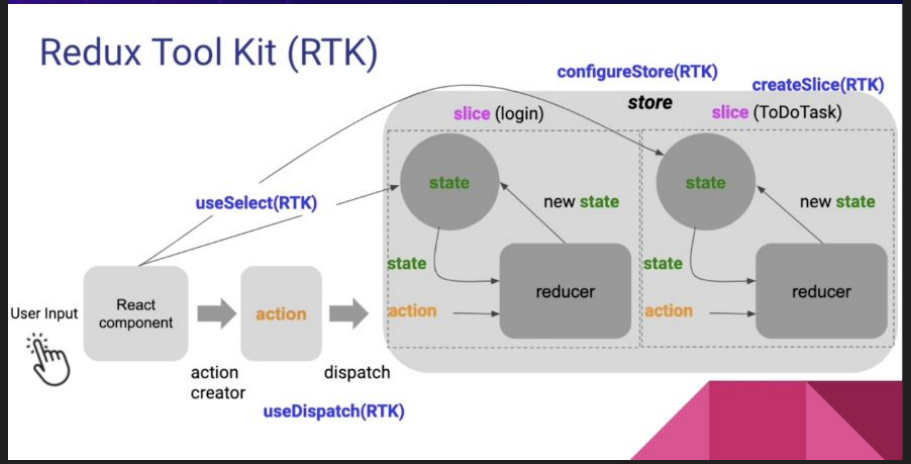
npm i @reduxjs/toolkit

Упрощение настройки и использования Redux

RTK автоматически настраивает многие аспекты Redux, которые раньше нужно было делать вручную. Это включает:

- Настройку хранилища через функцию configureStore, которая автоматизирует подключение редьюсеров, middleware и DevTools.
- Создание экшенов и редьюсеров через функцию createSlice, что уменьшает количество повторяющегося кода.
- Работу с асинхронными операциями с помощью createAsyncThunk, которое упрощает управление побочными эффектами, такими как запросы к API.





configureStore

Функция **configureStore** из библиотеки Redux Toolkit является упрощенным способом создания Redux-хранилища (store). Она автоматизирует множество процессов, которые раньше требовали ручной настройки, таких как подключение middleware, интеграция с DevTools, настройка редьюсеров и работа с асинхронными экшенами.

Особенности configureStore

- **Автоматическое подключение middleware:** По умолчанию configureStore подключает несколько полезных middleware, таких как redux-thunk (для асинхронных операций) и проверки на мутацию состояния с помощью immutableStateInvariant.
- **Интеграция с Redux DevTools:** DevTools включены по умолчанию, что позволяет отслеживать состояние и экшены приложения в режиме разработки.
- **Простота настройки редьюсеров:** Вы указываете редьюсеры в виде объекта, где каждый ключ соответствует части состояния.
- **Лёгкое добавление и настройка middleware:** Вы можете дополнительно добавлять свои middleware или настраивать существующие через функцию getDefaultMiddleware.

Пример configureStore

```
import { configureStore } from '@reduxjs/toolkit';
import counterReducer from './counterSlice';
import logger from 'redux-logger';

const store = configureStore({
  reducer: {
    counter: counterReducer,
  },
  middleware: (getDefaultMiddleware) => getDefaultMiddleware().concat(logger),
  devTools: process.env.NODE_ENV !== 'production',
});

export default store;
```

createSlice

createSlice из Redux Toolkit упрощает создание слайсов состояния, объединяя действия и редюсеры в одном месте. Это помогает избежать многословия и упрощает процесс управления состоянием.

Основные элементы createSlice

- **name:** Имя слайса, которое используется в качестве префикса для сгенерированных действий.
- **initialState:** Начальное состояние слайса.
- **reducers:** Объект, содержащий редюсеры, которые определяют, как состояние будет изменяться в ответ на действия.

Пример createSlice

Как это работает

createSlice генерирует действия и редюсеры на основе переданных настроек.

Мы можем легко добавлять новые действия или редюсеры, просто обновляя объект reducers.

Использование useSelector и useDispatch позволяет нам взаимодействовать с состоянием Redux в компонентах.

Таким образом, createSlice значительно упрощает работу с Redux, делая код более чистым и понятным.

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

// Создаем слайс
const todoSlice = createSlice({
  name: 'todos',
  initialState: [],
  reducers: {
    addTodo: (state, action) => {
      state.push({ id: Date.now(), text: action.payload, completed: false });
    },
    toggleTodo: (state, action) => {
      const todo = state.find(todo => todo.id === action.payload);
      if (todo) {
        todo.completed = !todo.completed;
      }
    },
    removeTodo: (state, action) => {
      return state.filter(todo => todo.id !== action.payload);
    },
  },
});

// Экспортируем действия
export const { addTodo, toggleTodo, removeTodo } = todoSlice.actions;

// Экспортируем редюсер
export default todoSlice.reducer;
```

createAsyncThunk

createAsyncThunk из Redux Toolkit упрощает работу с асинхронными операциями, такими как запросы к API. Он автоматически генерирует действия для различных состояний асинхронной операции (запуск, успешное завершение и ошибка).

Основные элементы createAsyncThunk

- **typePrefix:** Префикс для сгенерированных действий.
- **payloadCreator:** Функция, которая выполняет асинхронную операцию и возвращает промис.

Преимущества createAsyncThunk

- Упрощает работу с асинхронными операциями.
- Автоматически генерирует действия для управления состоянием загрузки и ошибок.
- Позволяет легко интегрировать асинхронные операции в Redux Store.

Пример createAsyncThunk

Как это работает

Определение thunk: В fetchTodos мы используем fetch для получения данных. Эта функция возвращает промис, который будет обрабатываться Redux Toolkit.

Создание слайса: В extraReducers мы обрабатываем три состояния:

pending: При вызове thunk устанавливаем статус в loading.

fulfilled: При успешном завершении операции обновляем состояние и добавляем полученные данные.

rejected: В случае ошибки обновляем статус и сохраняем сообщение об ошибке.

Использование в компоненте: Мы используем useEffect, чтобы запустить thunk при монтировании компонента. В зависимости от статуса отображаем загрузку, ошибку или список задач.

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

// Определяем асинхронный thunk
export const fetchTodos = createAsyncThunk('todos/fetchTodos', async () => {
  const response = await fetch('https://jsonplaceholder.typicode.com/todos');
  return await response.json();
});

// Создаем слайс
const todoSlice = createSlice({
  name: 'todos',
  initialState: {
    items: [],
    status: 'idle', // 'idle' | 'loading' | 'succeeded' | 'failed'
    error: null,
  },
  reducers: {},
  extraReducers: (builder) => {
    builder
      .addCase(fetchTodos.pending, (state) => {
        state.status = 'loading';
      })
      .addCase(fetchTodos.fulfilled, (state, action) => {
        state.status = 'succeeded';
        // Добавляем полученные задачи в массив items
        state.items = action.payload;
      })
      .addCase(fetchTodos.rejected, (state, action) => {
        state.status = 'failed';
        state.error = action.error.message;
      });
  },
});

// Экспортируем редюсер
export default todoSlice.reducer;
```

createReducer

createReducer из Redux Toolkit упрощает создание редюсеров, позволяя использовать "мутаторы" для обновления состояния. Это позволяет писать более читабельный и краткий код по сравнению с традиционными подходами, где требуется возвращать новое состояние.

Основные элементы createReducer

- **initialState:** Начальное состояние редюсера.
- **reducers:** Объект, где ключи — это названия действий, а значения — функции-редюсеры, которые обрабатывают эти действия.

Преимущества createReducer

- **Простота:** Упрощает создание редюсеров, позволяя писать меньше кода.
- **Читаемость:** Использование "мутаторов" делает код более понятным, так как не нужно явно возвращать новое состояние.
- **Поддержка Immer:** Благодаря интеграции с Immer вы можете безопасно изменять состояние, и Redux Toolkit позаботится о создании нового состояния.

Пример createReducer

Как это работает

Определение начального состояния: Мы определяем начальное состояние как пустой массив.

Создание редюсера: Используя createReducer, мы передаем начальное состояние и объект с редюсерами. Внутри builder мы определяем, как обрабатывать каждое действие.

addCase: Указывает, как обрабатывать конкретные действия. Здесь мы используем "мутацию" состояния (что допустимо благодаря Immer, встроенному в Redux Toolkit).

Использование в Store: Мы интегрируем редюсер в Redux Store, позволяя другим компонентам взаимодействовать с состоянием.

```
import { createReducer } from '@reduxjs/toolkit';
import { addTodo, toggleTodo, removeTodo } from './todoActions';

const initialState = [];

const todoReducer = createReducer(initialState, (builder) => {
  builder
    .addCase(addTodo, (state, action) => {
      state.push({ id: Date.now(), text: action.payload, completed: false });
    })
    .addCase(toggleTodo, (state, action) => {
      const todo = state.find(todo => todo.id === action.payload);
      if (todo) {
        todo.completed = !todo.completed;
      }
    })
    .addCase(removeTodo, (state, action) => {
      return state.filter(todo => todo.id !== action.payload);
    });
});

export default todoReducer;
```

createSelector

createSelector из Redux Toolkit (или reselect в более общем контексте) используется для создания мемоизированных селекторов, которые позволяют эффективно извлекать данные из состояния Redux. Он помогает избежать повторных вычислений при изменении состояния и обеспечивает производительность.

Основные характеристики createSelector

- **Мемоизация:** Селектор будет повторно использовать вычисленное значение, если входные данные (аргументы) не изменились, что повышает производительность.
- **Композиция:** Позволяет комбинировать несколько селекторов в один, что упрощает выборку сложных данных.
- **Упрощение логики:** Логика выборки состояния может быть вынесена в отдельные функции, что делает компоненты более чистыми и сосредоточенными на отображении.

Пример createSelector

Как это работает

Селектор состояния: selectTodos извлекает массив задач из состояния.

Создание мемоизированного селектора: selectCompletedTodos использует createSelector, чтобы фильтровать завершенные задачи. Если массив todos не изменился, то селектор вернет мемоизированное значение.

Использование в компоненте: useSelector извлекает завершенные задачи, и они отображаются в компоненте. Если задачи не изменились, повторный рендеринг компонента не произойдет.

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

const selectTodos = (state) => state.todos;

export const selectCompletedTodos = createSelector(
  [selectTodos],
  (todos) => todos.filter(todo => todo.completed)
);
```

```
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { addTodo, toggleTodo } from './todoSlice';
import { selectCompletedTodos } from './selectors';

const TodoApp = () => {
  const dispatch = useDispatch();
  const completedTodos = useSelector(selectCompletedTodos);

  const handleAddTodo = (text) => {
    dispatch(addTodo(text));
  };

  return (
    <div>
      <h1>Completed Todos</h1>
      <ul>
        {completedTodos.map(todo => (
          <li key={todo.id}>{todo.text}</li>
        ))}
      </ul>
      <button onClick={() => handleAddTodo('New Todo')}>Add Todo</button>
    </div>
  );
};

export default TodoApp;
```

createEntityAdapter

createEntityAdapter из Redux Toolkit упрощает работу с коллекциями объектов, такими как списки задач или пользователей. Он предоставляет методы для управления состоянием, включая добавление, обновление, удаление и выборку объектов, а также автоматически нормализует данные для более удобной работы.

Основные характеристики createEntityAdapter

- **Нормализация данных:** Состояние хранится в виде объекта, где ключи — это идентификаторы, а значения — соответствующие объекты. Это упрощает доступ к объектам и их обновление.
- **Методы для управления коллекцией:** Предоставляет встроенные методы для работы с коллекцией, такие как addOne, addMany, updateOne, removeOne, и т.д.
- **Гибкость:** Можно использовать с любыми типами данных, предоставляя интерфейсы для работы с объектами.

Пример createEntityAdapter

Как это работает

Создание адаптера: Мы создаем адаптер todosAdapter, который предоставляет методы для управления коллекцией задач.

Инициализация состояния: Используем getInitialState, чтобы задать начальное состояние, включая дополнительные поля (например, loading и error).

Определение редюсеров: Мы используем методы адаптера (например, addOne, updateOne, removeOne) в редюсерах для добавления, обновления и удаления задач.

Селекторы: Используем методы адаптера для создания селекторов, которые позволяют удобно извлекать все задачи или задачу по идентификатору.

Использование в компоненте: Мы используем селекторы для получения списка задач и адаптер для добавления новых задач.

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

const todosAdapter = createEntityAdapter();

const initialState = todosAdapter.getInitialState({
  loading: false,
  error: null,
});

// Создаем слайс
const todoSlice = createSlice({
  name: 'todos',
  initialState,
  reducers: {
    addTodo: todosAdapter.addOne,
    updateTodo: todosAdapter.updateOne,
    removeTodo: todosAdapter.removeOne,
    setLoading: (state, action) => {
      state.loading = action.payload;
    },
    setError: (state, action) => {
      state.error = action.payload;
    },
  },
});

// Экспортируем действия и редюсер
export const { addTodo, updateTodo, removeTodo, setLoading, setError } = todoSlice.actions;
export default todoSlice.reducer;

// Селекторы
export const {
  selectAll: selectAllTodos,
  selectById: selectTodoById,
} = todosAdapter.getSelectors(state => state.todos);
```

RTK Query

Что такое Redux Toolkit Query? Redux Toolkit Query (RTK Query) — это мощный инструмент для управления запросами к API в приложениях, построенных на Redux. Он упрощает взаимодействие с удалёнными данными, предоставляет средства для кэширования, синхронизации и управления состоянием загрузки.

Основные принципы

- **Автоматизация работы с API:** Упрощает создание, отправку и обработку запросов к API, устраняя необходимость ручного написания большого количества кода для управления состоянием.
- **Кэширование:** Позволяет кэшировать данные, полученные от API, что снижает количество запросов и улучшает производительность.
- **Интерфейс API:** Позволяет описывать API в одном месте, что упрощает управление.
- **Параллельная работа с запросами:** Обрабатывает запросы асинхронно и параллельно, что улучшает пользовательский опыт.

Встроенный функционал Redux ToolKit

Плюсы и минусы RTK Query

Плюсы:

- Скорость разработки: Быстрая настройка и уменьшение количества шаблонного кода.
- Эффективное кэширование: Автоматическое обновление кэша при изменении данных.
- Интеграция с Redux: Прямо использует Redux, что облегчает понимание для разработчиков, знакомых с этой библиотекой.
- Поддержка асинхронных запросов: Удобное API для работы с асинхронными функциями.

Минусы:

- Сложность для новичков: Может быть сложнее для понимания, если у вас нет опыта работы с Redux.
- Зависимость от Redux: Необходимость в Redux может быть избыточной для простых приложений.
- Объёмный пакет: Использование RTK может увеличивать размер итогового бандла.

Как работает под капотом?

RTK Query использует концепцию slice (среза) в Redux для управления состоянием запросов. Он предоставляет slice reducer, который отслеживает состояния загрузки, ошибки и данные для каждого API вызова.

Когда вы делаете запрос, RTK Query:

- Проверяет наличие данных в кэше.
- Если данные не найдены или они устарели, отправляет запрос к API.
- При получении ответа обновляет кэш и состояние приложения.

Пример RTK Query

```jsx
// components/UserList.js
import React from 'react';
import { useGetUsersQuery } from '../features/apiSlice';

const UserList = () => {
  const { data: users, error, isLoading } = useGetUsersQuery();

  if (isLoading) return <div>Loading...</div>;
  if (error) return <div>Error loading users!</div>;

  return (
    <ul>
      {users.map(user => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
};

export default UserList;
```

```jsx
// features/apiSlice.js
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react';

export const apiSlice = createApi({
  reducerPath: 'api',
  baseQuery: fetchBaseQuery({ baseUrl: 'https://jsonplaceholder.typicode.com' }),
  endpoints: (builder) => ({
    getUsers: builder.query({
      query: () => '/users',
    }),
  }),
});

export const { useGetUsersQuery } = apiSlice;
```

```jsx
// app/store.js
import { configureStore } from '@reduxjs/toolkit';
import { apiSlice } from '../features/apiSlice';

export const store = configureStore({
  reducer: {
    [apiSlice.reducerPath]: apiSlice.reducer,
  },
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware().concat(apiSlice.middleware),
});
```

Объяснение примера:

API Slice: В apiSlice.js мы создали срез API, описывающий базовый URL и конечные точки. Метод getUsers определяет, как получить список пользователей.

Store: В store.js мы настроили Redux Store, добавив middleware для RTK Query, что позволяет ему отслеживать состояния.

Компонент: В UserList.js мы используем хук useGetUsersQuery, который автоматически инициирует запрос. Хук возвращает состояние загрузки, данные и возможные ошибки.

RTK Query значительно упрощает работу с API, предлагая мощные средства для управления состоянием и кэширования данных. С помощью простого API разработчики могут быстро настраивать запросы и управлять состоянием приложения, минимизируя шаблонный код.

Ресурсы

Код урока (git репозиторий) - ТЫК

Оф. документация Redux ToolKit (en) - ТЫК

Учебник Redux ToolKit (ru) - ТЫК

Оф. документация Redux ToolKit Query - ТЫК

Статья про Redux ToolKit - ТЫК

Мини документация по RTK Query (ru) - ТЫК