фаза 3 · неделя 1 · день 3

Redux Toolkit, Slice

ELBRUS/BOOTCAMP

План

1. Redux Toolkit
2. Slice

Redux Toolkit

Набор инструментов для упрощения и унификации работы с Redux.

Также известен как RTK.

Redux Toolkit: зачем это всё?

- `Асинхронный код` (fetch и др.) можно вынести из React-компонентов в action creators. Получатся так называемые `Thunk`.
- `Action types, action creators` генерируются автоматически — не нужно писать вручную.
- Проще описывать как меняется состояние после `fetch-запросов` (отправка, успех, ошибка).
- Можно “мутировать” `state` в reducer-функции — менять поля объектов напрямую, пушить в массивы и т. д. RTK позаботится об `иммутабельности` за тебя.

Redux Toolkit: установка

Установить RTK: `npm i @reduxjs/toolkit`

Redux Toolkit: создание store

```
import { configureStore } from '@reduxjs/toolkit';
import { useDispatch, useSelector } from 'react-redux';
// Слайсы - отдельные модули нашего приложения. У каждого слайса - свой редьюсер.
import tasksSliceReducer from './features/tasks/tasksSlice';
import authSliceReducer from './features/auth/authSlice';

const store = configureStore({
    // теперь функция combineReducers не нужна
    reducer: {
        auth: authSliceReducer,
        tasks: tasksSliceReducer,
    },
});
// не забываем про типизацию хуков из redux
type RootState = ReturnType<typeof store.getState>;
type AppDispatch = typeof store.dispatch;
export const useAppSelector = useSelector.withTypes<RootState>();
export const useAppDispatch = useDispatch.withTypes<AppDispatch>();

export default store;
```

Redux Toolkit: применение store

```
import React from 'react';
import { createRoot } from 'react-dom/client';
import { store } from './app/store';
import { Provider } from 'react-redux';
import App from './App';

createRoot(document.getElementById('root')).render(
    <Provider store={store}>
        <App />
    </Provider>,
    document.getElementById('root')
);
```

Slice

Slice в Redux Toolkit (RTK) — это часть глобального состояния приложения, управляемая определенным редьюсером и содержащая свои действия и логику изменения состояния.

Slice: зачем нужен?

Теперь вместо того, чтобы отдельно описывать `reducer`, `actions types` и `actions creators` для каждого раздела приложения мы описываем только `slice`.

В слайсе тебе нужно реализовать все кейсы редьюсера (каждый в отдельной функции), а `actions types` и `actions creators` генерируются автоматически.

Кроме того в редьюсере можно (и нужно) мутировать `state`!

Slice: создание

```
// features/counter/counterSlice.ts
import { createSlice } from '@reduxjs/toolkit';
import CounterState from './types/CounterState';

// начальный state
const initialState: CounterState = { count: 0 };

// объявляем slice с именем “counter”
const counterSlice = createSlice({
    name: 'counter',
    initialState,
    reducers: {
        // отдельные кейсы редьюсера записываются в отдельные функции
        // здесь можно мутировать state, RTK создаст копию state автоматически
        plus: (state) => { state.count += 1; },
        minus: (state) => { state.count -= 1; },
    }
});

// RTK создаёт action creators для каждого кейса редьюсера, экспортируем их
export const { plus, minus } = counterSlice.actions;
export default counterSlice.reducer;
```

Slice: применение синхронных reducers

```tsx
// features/counter/Counter.tsx
import React from 'react';
import { plus, minus } from './counterSlice';
import { useAppDispatch, useAppSelector } from '../../store';

export default function Counter(): JSX.Element {
    const count = useAppSelector((store) => store.countReducer.count);
    const dispatch = useAppDispatch();

    return (
        <div>
            <button onClick={() => dispatch(minus())}>-</button>
            <span>{count}</span>
            <button onClick={() => dispatch(plus())}>+</button>
        </div>
 );
}
```

Slice: createAsyncThunk декомпозиция API

createAsyncThunk позволяет вынести асинхронную логику (например, `fetch`/`axios`) из React-компонента в отдельные функции и запускать их как обычные экшены через `dispatch`. Разберём `createAsyncThunk` на примере загрузки с сервера списка задач.

```
// features/tasks/api.ts
import Task from './types/Task';

export async function getTasks(): Promise<Task[]> {
    const result = await fetch('/api/tasks');
    return result.json();
}
```

Slice: createAsyncThunk описание параметров

Функция `createAsyncThunk` принимает два параметра:

- Первый параметр: тип действия, который будет использоваться в Redux для идентификации этого конкретного асинхронного действия.
- Второй параметр: функция, которая будет выполняться, когда действие будет вызвано. В данном случае, когда вызывается `loadTasks`, эта функция выполнит `api.getTasks()`.

```
export const loadTasks = createAsyncThunk(
 'tasks/loadTasks',
 () => api.getTasks()
);
```

Slice: createAsyncThunk работа с ошибками

Важно: результат `api.getTasks()` будет передан в `action.payload`. Если же `api.getTasks()` завершится с ошибкой, то ошибка будет передана в `action.error`.

```
extraReducers: (builder) => {
  builder
    .addCase(loadTasks.fulfilled, (state, action) => {
      state.tasks = action.payload;
    })
    .addCase(loadTasks.rejected, (state, action) => {
       state.error = action.error.message;
    });
}
```

Slice: createAsyncThunk в extraReducers

```
import { createAsyncThunk, createSlice } from '@reduxjs/toolkit';
import * as api from '../features/tasks/api.ts';
export const loadTasks = createAsyncThunk('tasks/loadTasks',  () =>
api.getTasks());

const tasksSlice = createSlice({
    name: 'tasks',
    initialState,
    reducers: {},
    extraReducers: (builder) => {
        builder
        .addCase(loadTasks.fulfilled, (state, action) => {
            state.tasks = action.payload;
        })
        .addCase(loadTasks.rejected, (state, action) => {
                state.error = action.error.message;
        });
 },
});
export default tasksSlice.reducer;
```

Slice: применение loadTasks в useEffect

```tsx
// features/tasks/TasksList.tsx
export default function TasksList(): JSX.Element {
 const tasks = useAppSelector(selectTasks);
 const error = useAppSelector(selectError);
 const dispatch = useAppDispatch();

 useEffect(() => {
   dispatch(loadTasks());
 }, [dispatch]);

 if (error) return <div className="error">{error}</div>;

 return (
   <div className="tasks list-group">
     {tasks.map((task) => (
       <TaskView key={task.id} task={task} />
     ))}
   </div>
 );
}
```

Slice: применение loadTasks в dispatch

```tsx
// features/tasks/TasksList.tsx
import React, { useEffect } from 'react';
import { selectTasks, selectError } from
'./selectors';
import TaskView from './TaskView';
import { loadTasks } from './tasksSlice';
import { useAppDispatch, useSelector }
from '../../store';
```

```tsx
export default function TasksList():
JSX.Element {
 const tasks = useAppSelector(selectTasks);
 const error = useAppSelector(selectError);
 const dispatch = useAppDispatch();

 useEffect(() => {
   dispatch(loadTasks());
 }, [dispatch]);

 if (error) return <div
className="error">{error}</div>;

 return (
   <div className="tasks list-group">
     {tasks.map((task) => (
       <TaskView key={task.id} task={task} />
     ))}
   </div>
 );
}
```

loadTasks - это thunk.
Диспатчим его как обычный action creator.