

Цели занятия

- узнать что такое middleware в Redux;
- написать собственный middleware для логирования;
- научиться работать с асинхронными действиями в Redux;

Полезные материалы

- [Официальная документация про middleware](#)
- [Документация по createAsyncThunk](#)

Что такое Redux Middleware?

Redux Middleware переводится как промежуточное программное обеспечение и позволяет разработчикам писать функции, которые могут перехватывать любые действия, отправляемые в reducer, и выполнять дополнительную логику.

В предыдущем занятии мы с вами познакомились с термином редуктор и их правилами. Одно из правил написания редуктора было таким — редуктор должен быть чистой функцией без сайд-эффектов. Но в разработке мы не можем взять и просто отказаться от сайд-эффектов, именно для этого разработчики Redux придумали middleware.

Эта концепция позволяет нам реагировать на **любое** действие, которое у нас есть, видоизменять или полностью его отменять, а также выполнять дополнительную логику. Причём мы имеем возможность включать/выключать middleware буквально одной строкой кода и это никак не повлияет на уже написанные нами действия и редукторы.

Middleware на практике: проблема

Давайте представим, что у нас с вами возникла потребность выводить в консоль любое действие, которое у нас есть в хранилище. А также нам нужно выводить состояние, которое мы получили после выполнения этого действия.

И первый подход к решению этой задачи может быть таким (на основе кода, который мы писали в предыдущем занятии):

```
// Файл cart-reducer.js
import { createSlice } from "@reduxjs/toolkit";

export const cartSlice = createSlice({
  name: "cart",
  initialState: {
    products: []
  },
  reducers: {
    addProduct: (prevState, action) => {
      console.log("action", action);

      const product = action.payload;
      const newState = {
        ...prevState,
        products: [...prevState.products, product]
      };

      // Вынесли новое состояние в переменную,
      // чтобы иметь возможность вывести его в консоль
      console.log("new state", newState);

      return newState;
    }
  }
});

export const { addProduct } = cartSlice.actions;
export default cartSlice.reducer;
```

И с первого взгляда может показаться, что всё хорошо — мы выполнили задачу. Но! Это только одно действие и когда мы добавим второе, то нам придется писать там то же самое (а в реальных проектах десятки действий). Более того, мы не можем выключить это логирование, если нам нужно. Только если удалять код, а потом заново добавлять, когда потребуется — рутинная работа и бесполезная трата времени

Middleware на практике: решение

Именно здесь нам на помощь и приходит middleware, который мы сейчас и напишем:

```
// Файл store.js
import { configureStore } from "@reduxjs/toolkit";
import cartReducer from "./cart-reducer";

// Это и есть middleware
const logger = (store) => (next) => (action) => {
  console.log("action", action);

  // Функция next применяет действие к хранилищу
  // и возвращает новое состояние хранилища
  let result = next(action);

  // Выводим в консоль новое состояние
  console.log("next state", store.getState());
  return result;
};

export const store = configureStore({
  reducer: {
    cart: cartReducer,
  },

  // Подключаем middleware ко всем хранилищу
  // Он будет работать для ВСЕХ действий сразу!
  middleware: [logger],
});
```

Это всё, что потребуется написать для решения задачи, причём это будет работать одинаково для любого действия, даже если их будут сотни, а включить/выключить можно одной строкой — просто уберите logger из массива middleware. Более того, этот логгер можно использовать для любых хранилищ, которые написаны на Redux и везде он будет работать одинаково!

Redux Thunk

На текущий момент времени мы с вами умеем работать только с синхронными действиями в Redux. Но в реальной разработке есть огромное количество асинхронных действий, например, когда вы добавляете товар в корзину, то делаете запрос на бэкенд, который может выполняться несколько секунд и нам нужно показывать пользователю индикатор загрузки и только после получения ответа мы можем обновить состояние в хранилище.

Для всего этого есть библиотека, которая называется [redux-thunk](#), но мы используем Redux Toolkit и она уже у нас установлена и доступна в виде функции createAsyncThunk.

Включаем redux-thunk

```
import { configureStore, getDefaultMiddleware } from "@reduxjs/toolkit";
import cartReducer from "../features/cart-reducer";

// Не меняется из предыдущего примера кода
const logger = ...

export const store = configureStore({
  reducer: {
    cart: cartReducer
  },

  // Функция getDefaultMiddleware вернет нам стандартные middleware
  // которые уже встроены в Redux Toolkit и к которым мы добавили
  // ещё и нам логгер
  middleware: [...getDefaultMiddleware(), logger]
});
```

Пишем асинхронное действие

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

// Это асинхронное действие, вместо реального запроса на бэкенд
// мы будем использовать таймаут

export const fetchProduct = createAsyncThunk(
  "cart/fetchProduct",
  (productId) => {
    return new Promise((resolve) => {
      console.log(`get info from backend: ${productId}`);

      setTimeout(() => {
        // Представим, что бэкенд через три секунды
        // вернул нам вот эту информацию
        resolve({ id: productId, name: "iPhone 13 Pro" });
      }, 3000);
    });
  }
);

export const cartSlice = createSlice({
  name: "cart",
  initialState: {
    // У нас добавился статус загрузки с несколькими значениями
    // idle — не загружается
    // loading — загружается
    // success — загрузка закончена
    status: "idle",
    products: [],
  },
  // Не меняем с прошлого урока
  reducers: {},

  // Это ключ с асинхронными редукторами
  extraReducers: (builder) => {
    // Что должно произойти с хранилищем, когда запрос начал загружаться?
    builder.addCase(fetchProduct.pending, (prevState, action) => {
      return { ...prevState, status: "loading" };
    });

    // Что должно произойти с хранилищем, когда запрос загрузился?
    builder.addCase(fetchProduct.fulfilled, (prevState, action) => {
      const product = action.payload;
```

```

    return {
      ...prevState,
      status: "success",
      products: [...prevState.products, product],
    };
  });
},
});

export const { addProduct } = cartSlice.actions;

export default cartSlice.reducer;

```

Обновляем компонент

```

import { useSelector, useDispatch } from "react-redux";
import { fetchProduct } from "../features/cart-reducer";

function Cart() {
  // Получаем частичку глобального состояния
  const count = useSelector((state) => state.cart.products.length);
  const status = useSelector((state) => state.cart.status);
  const dispatch = useDispatch();

  function handleClickAdd() {
    // Теперь мы используем другое действие, чтобы добавить товар
    // Передаём в функцию fetchProduct идентификатор товара
    // и информацию о товаре должен вернуть бэкенд
    dispatch(fetchProduct(4884));
  }

  return (
    <div>
      <div>{`В корзине ${count} товаров`}</div>
      <div>{`Статус: ${status}`}</div>
      <button onClick={handleClickAdd}>Добавить товар</button>
    </div>
  );
}

```

Подробнее о том, что здесь происходит мы будем на онлайн-занятии, поэтому заглядывайте или смотрите в записи. Либо можно обратиться к страницам [официальной документации](#), где есть больше примеров и разных случаев

