Бібліотека Redux, Redux Toolkit

1) createSlice()
2) configureStore()
3) useDispatch()
4) useSelector()
5) Рекомендована структура папок.
6) Робота з асинхронними операціями
7) Робота з асинхронними операціями createAsyncThank

Redux, Redux toolkit

createSlice

Redux Toolkit використовує підхід до розділення стану за допомогою "slices". Створення slice - це спосіб групувати пов'язані дії та редюсери разом. Ось приклад створення slice для каталогу товарів:

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

const productsSlice = createSlice({
  name: 'products',
  initialState: [],
  reducers: {
    addProduct: (state, action) => {
      state.push(action.payload);
    },
    removeProduct: (state, action) => {
      return state.filter(product => product.id !== action.payload);
    },
  },
});

export const { addProduct, removeProduct } = productsSlice.actions;
export default productsSlice.reducer;
```

Redux, Redux toolkit

createSlice, configureStore

Redux Toolkit також надає функцію `configureStore`, яка автоматично виконує багато стандартних операцій настроювання. Ось приклад створення store з використанням Redux Toolkit:

```
import { configureStore } from '@reduxjs/toolkit';
import productsReducer from './productsSlice';

const store = configureStore({
  reducer: {
    products: productsReducer,
    // Додайте інші редюсери, якщо необхідно
  },
});

export default store;
```

Redux, Redux toolkit

Provider

Після створення store ви можете підключити його до свого додатку як ми робили у минулій лекції. Ось приклад використання Redux Toolkit з React:

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

const Root = () => (
  <Provider store={store}>
    <App />
  </Provider>
);

export default Root;
```

Redux, Redux toolkit

Використання у додатку

Тепер ви можете використовувати створені дії у вашому додатку. Ось приклад використання дії `addProduct` для додавання нового товару до списку:

```
import React from 'react';
import { useDispatch } from 'react-redux';
import { addProduct } from './productsSlice';

const ProductForm = () => {
  const dispatch = useDispatch();

  const handleSubmit = (event) => {
    event.preventDefault();
    const product = { id: 1, name: 'Product 1' };
    dispatch(addProduct(product));
  };

  return (
    <form onSubmit={handleSubmit}>
      <button type="submit">Add Product</button>
    </form>
  );
};

export default ProductForm;
```

Redux, Redux toolkit

useDispatch()

Метод `useDispatch()` в React Redux Toolkit - це спеціальна функція-хук, яка надає можливість виконувати диспетчеризацію (dispatch) дій (actions) у вашому компоненті React, коли ви використовуєте Redux Toolkit для керування станом вашого додатка.

В Redux Toolkit диспетчеризація дій відбувається за допомогою Redux store, який зберігає стан додатка. Для виконання диспетчеризації дій потрібно мати доступ до об'єкта store або мати можливість отримати доступ до нього.

Завдяки `useDispatch()`, вам не потрібно прямо отримувати доступ до store в компоненті. Замість цього ви можете використовувати `useDispatch()` для отримання функції диспетчеризації дій, яку ви можете використовувати для виклику дій Redux.

Redux, Redux toolkit

useDispatch()

У цьому прикладі `useDispatch()` використовується для отримання функції диспетчеризації дій, яка прив'язана до об'єкта store. У нашому компоненті ми визначаємо функцію `handleClick()`, яка викликає диспетчеризацію дії `myAction()` за допомогою `dispatch(myAction())`. При кліку на кнопку "Dispatch Action" буде виконано диспетчеризацію дії `myAction()`.

Отримання доступу до функції диспетчеризації дій за допомогою `useDispatch()` дозволяє зручно і просто керувати станом вашого додатка в компонентах React, які використовують Redux Toolkit.

```
import { useDispatch } from 'react-redux';

function MyComponent() {
  const dispatch = useDispatch();

  const handleClick = () => {
    dispatch(myAction());
  };

  return (
    <button onClick={handleClick}>Dispatch Action</button>
  );
}
```

Redux, Redux toolkit

useSelector()

Для того, щоб отримати дані зі стору Redux та використовувати їх у компонентах, ви можете використовувати хук `useSelector()` з бібліотекою `react-redux`. Ось як це працює:

Імпортуйте необхідні функції:

```
import React from 'react';
import { useSelector } from 'react-redux';
```

Використовуйте хук `useSelector` у вашому компоненті:

```
const MyComponent = () => {
  const data = useSelector((state) => state.data);

  // Використання даних зі стору
  // ...

  return (
    // JSX компонента
  );
};
```

У функції `useSelector` ви передаєте колбек-функцію, яка отримує поточний стан стору як аргумент і повертає потрібні дані. В прикладі вище ми отримуємо дані з ключем `data` зі стору.

Redux, Redux toolkit

useSelector()

Використовуйте отримані дані у вашому компоненті:

```
const MyComponent = () => {
  const data = useSelector((state) => state.data);

  // Використання даних зі стору
  return (
    <div>
      <h1>{data.title}</h1>
      <p>{data.description}</p>
    </div>
  );
};
```

В даному прикладі ми використовуємо отримані дані зі стору для відображення заголовку (`data.title`) та опису (`data.description`) у компоненті.

Важливо відмітити, що ви можете використовувати хук `useSelector` в будь-якому компоненті, який обгортається компонентою `<Provider>` з `react-redux`. Цей хук автоматично підписує компонент на зміни стану стору, тому компонент буде оновлюватись, коли дані змінюються у сторі Redux.

Redux, Redux toolkit

createSelector()

`createSelector` - це функція з бібліотеки redux-toolkit, яка використовується для створення мемоізованих селекторів в Redux. Вона дозволяє ефективно обчислювати похідні дані зі стану вашого Redux-стору, забезпечуючи кешування результатів, що покращує продуктивність додатку.

У Redux ви можете мати ситуації, коли вам потрібно обчислювати похідні дані зі стану, наприклад, фільтрувати, сортувати або групувати дані перед їхнім використанням в компоненті. Без мемоізації ці обчислення можуть повторюватися надто часто, що призводить до зайвої роботи і зниження продуктивності.

Визначте селектори та використайте `createSelector`:

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

// Вхідні селектори
const getUsers = state => state.users;
const getFilter = state => state.filter;

// Створення мемоізованого селектора
const filteredUsersSelector = createSelector(
  [getUsers, getFilter],
  (users, filter) => {
    return users.filter(user => user.name.includes(filter));
  }
);
```

У цьому прикладі ми використовуємо два вхідних селектори - `getUsers` і `getFilter` - для отримання відповідних частин стану. Потім визначаємо функцію, яка обчислює відфільтрований список користувачів на основі цих даних.

Redux, Redux toolkit

createSelector()

Використовуйте filteredUsersSelector з Redux Toolkit:

```
import { useSelector } from 'react-redux';

const MyComponent = () => {
  const filteredUsers = useSelector(filteredUsersSelector);

  // Використання filteredUsers у вашому компоненті
  // ...
}
```

Замість прямого використання `getUsers` і `getFilter`, ви використовуєте `filteredUsersSelector` як селектор для отримання відфільтрованого списку користувачів. Селектор буде повертати результат з кешу, якщо вхідні дані не змінилися, зменшуючи непотрібні повторні обчислення.

Redux, Redux toolkit

Folder structure

При використанні Redux Toolkit рекомендовані структура папок та конвенції назв файлів аналогічні попередньо наведеним. Однак Redux Toolkit надає деякі абстракції, які спрощують налаштування Redux та організацію коду.

Структура папок: Організуйте файли, пов'язані з Redux Toolkit, у відповідних папках всередині кореневої директорії вашого проекту. Наприклад:

```
src/
  |- features/
  |- store/
```

- **features**: Ця папка містить підпапки, які представляють різні функціональні можливості або домени вашого додатку. Кожна підпапка містить логіку Redux, пов'язану з конкретною функціональністю.
- **store**: Розмістіть файли налаштування Redux Toolkit у цій папці. Зазвичай вона включає створення Redux-сховища за допомогою `configureStore` з Redux Toolkit.

Redux, Redux toolkit

Folder structure

Підхід на основі функціональності: Використовуйте підхід на основі функціональності всередині папки `features`. Кожна підпапка, що відповідає функціональності, може містити наступні файли:

- **Slice**: Redux Toolkit використовує поняття "slices" (фрагменти), які поєднують визначення дій та редюсерів в одному файлі. У кожній підпапці функціональності повинен бути відповідний файл `slice.js`, наприклад `authSlice.js` або `cartSlice.js`. Цей файл визначає дії та редюсери для конкретної функціональності.
- **Actions**: За потреби ви можете створити окремий файл `actions.js` всередині підпапки функціональності, щоб визначити додаткові дії, специфічні для цієї функціональності.
- **Selectors**: Аналогічно, ви можете створити файл `selectors.js` всередині підпапки функціональності, щоб визначити селектори, які витягують та обчислюють похідний стан для цієї функціональності.

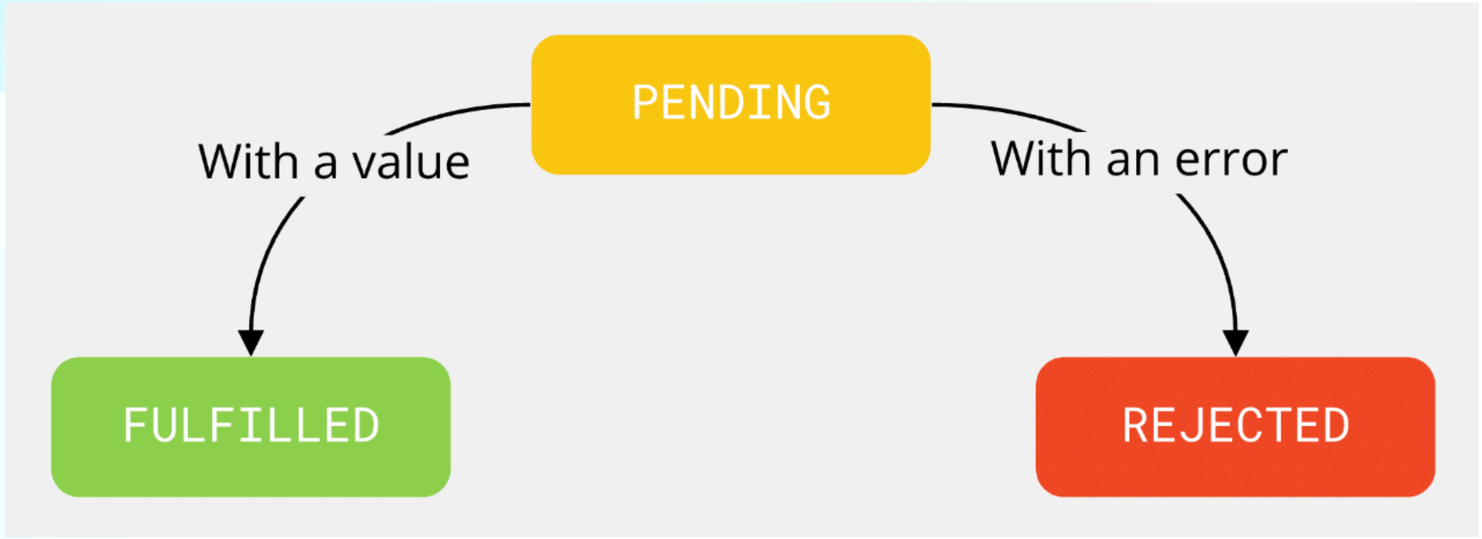
```
src/
  |- features/
     |- auth/
        |- authSlice.js
        |- authActions.js
        |- authSelectors.js
     |- cart/
        |- cartSlice.js
        |- cartActions.js
        |- cartSelectors.js
     |- ...
  |- ...
```

За допомогою Redux Toolkit ідея полягає в тому, щоб кожна функціональність мала свою власну папку, використовуючи "slices" для поєднання дій та редюсерів. Цей підхід спрощує структуру коду та зменшує кількість шаблонного коду, які зазвичай пов'язані з Redux.

Redux, Redux toolkit

Робота з асинхронними операціями

HTTP-запити це асинхронні операції, які представлені промісами, тому їх можна розбити на три складові: процес запиту (pending), успішне завершення запиту (fulfilled) та завершення запиту з помилкою (rejected). Цей шаблон застосуємо до будь-яких запитів читання, створення, видалення та оновлення.



Redux, Redux toolkit

Робота з асинхронними операціями

Основний підхід для роботи з асинхронними операціями в Redux Toolkit - це використання вбудованного у Redux Toolkit пакету Redux Thunk, який дозволяє вам писати асинхронні дії (actions) та обробники (reducers) в Redux.

Створіть файл для опису дій та обробників. Наприклад, `userSlice.js`:

```
const userSlice = createSlice({
  name: 'counter',
  initialState: {
    user: null,
    loading: false,
    error: null,
  },
  reducers: {
    getUserStart(state) {
      state.loading = true;
    },
    getUserSuccess(state, action) {
      state.loading = false;
      state.user = action.payload;
      state.error = null;
    },
    getUserFailure(state, action) {
      state.loading = false;
      state.user = null;
      state.error = action.payload;
    },
  },
});

export const { getUserStart, getUserSuccess, getUserFailure } =
userSlice.actions;
export default userSlice.reducer;
```

В редьюсері описуємо логіку завантаження та обробки помилок

Redux, Redux toolkit

Робота з асинхронними операціями

Додамо Операцію. Ця функція використовує `dispatch` для відправки різних дій Redux, які відповідають різним етапам асинхронного запиту.

```
export const fetchUserAsync = () => async (dispatch) => {
  dispatch(getUserStart());
  try {
    const user = await fetchUser(); // Виконуємо асинхронний запит
    dispatch(getUserSuccess(user));
  } catch (error) {
    dispatch(getUserFailure(error.message));
  }
};
```

- Спочатку, викликається `dispatch(getUserStart())`, що ініціює стан завантаження (`loading`) перед початком асинхронного запиту.
- Потім, виконується сам асинхронний запит `fetchUser()`, який очікує дані користувача.
- Якщо запит успішний, то викликається `dispatch(getUserSuccess(user))`, де `user` - отримані дані про користувача. Це оновлює стан з отриманими даними та очищує будь-які помилки.
- У разі помилки під час виконання запиту, викликається `dispatch(getUserFailure(error.message))`, де `error.message` - повідомлення про помилку. Це оновлює стан з повідомленням про помилку та очищує отримані дані.

Redux, Redux toolkit

Робота з асинхронними операціями

Створіть ваш store з використанням `configureStore` з Redux Toolkit

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

const store = configureStore({
  reducer: {
    counter: counterReducer,
  },

});

export default store;
```

Redux, Redux toolkit

Робота з асинхронними операціями

Використовуйте ваші дії та обробники в компонентах React:

```
import { useDispatch, useSelector } from 'react-redux';
import { fetchUserAsync } from './counterSlice';

function UserComponent() {
  const dispatch = useDispatch();
  const user = useSelector((state) => state.counter.user);
  const loading = useSelector((state) => state.counter.loading);
  const error = useSelector((state) => state.counter.error);

  const handleClick = () => {
    dispatch(fetchUserAsync());
  };

  return (
    <div>
      {loading ? (
        <p>Loading...</p>
      ) : error ? (
        <p>Error: {error}</p>
      ) : (
        <div>
          {user && <p>User: {user.name}</p>}
          <button onClick={handleClick}>Fetch User</button>
        </div>
      )}
    </div>
  );
}
```

В прикладі вище, коли натискається кнопка "Fetch User", викликається асинхронна дія `fetchUserAsync`, яка виконує запит до сервера за даними користувача. Під час виконання запиту встановлюється прапорець `loading`, а після успішного отримання даних користувача - змінюється стан `user`. У разі помилки - стан `error`.

Redux, Redux toolkit

createAsyncThank

`createAsyncThunk` - це функція з Redux Toolkit, яка дозволяє створювати асинхронні дії (async actions) швидко та зручно. Вона автоматично генерує три дії для обробки асинхронних операцій: `pending`, `fulfilled` та `rejected`. Це спрощує обробку асинхронних запитів та керування станом додатку.

У цьому прикладі `createAsyncThunk` генерує асинхронну дію `fetchUserAsync`. Вона автоматично створює три додаткові дії: `fetchUserAsync.pending`, `fetchUserAsync.fulfilled` та `fetchUserAsync.rejected`.

- `pending` - дія, яка спрацьовує перед початком асинхронного запиту.
- `fulfilled` - дія, яка спрацьовує при успішному завершенні асинхронного запиту.
- `rejected` - дія, яка спрацьовує в разі помилки при виконанні асинхронного запиту.

У розділі `extraReducers` ми визначаємо, як обробляти ці дії та змінювати стан додатку. Наприклад, коли спрацьовує `pending`, ми встановлюємо значення `loading` в `true`. Коли спрацьовує `fulfilled`, ми оновлюємо `loading` на `false`, зберігаємо отримані дані в `data` і встановлюємо `error` в `null`. Коли спрацьовує `rejected`, ми встановлюємо значення `loading` в `false`, очищуємо `data` і зберігаємо повідомлення про помилку у `error`.

```
import { createSlice, createAsyncThunk } from '@reduxjs/toolkit';
import { fetchUser } from './api'; // Функція, яка виконує асинхронний запит

// Створюємо асинхронну дію за допомогою createAsyncThunk
export const fetchUserAsync = createAsyncThunk('user/fetchUser', async () =>
  const response = await fetchUser();
  return response.data;
});

const userSlice = createSlice({
  name: 'user',
  initialState: {
    data: null,
    loading: false,
    error: null,
  },
  reducers: {},
  extraReducers: (builder) => {
    builder
      .addCase(fetchUserAsync.pending, (state) => {
        state.loading = true;
      })
      .addCase(fetchUserAsync.fulfilled, (state, action) => {
        state.loading = false;
        state.data = action.payload;
        state.error = null;
      })
      .addCase(fetchUserAsync.rejected, (state, action) => {
        state.loading = false;
        state.data = null;
        state.error = action.error.message;
      });
  },
});

export default userSlice.reducer;
```

Redux, Redux toolkit

createAsyncThank

Потім ви можете використовувати `fetchUserAsync` у своєму компоненті React:

```
import { useDispatch, useSelector } from 'react-redux';
import { fetchUserAsync } from './userSlice';

function UserComponent() {
  const dispatch = useDispatch();
  const user = useSelector((state) => state.user.data);
  const loading = useSelector((state) => state.user.loading);
  const error = useSelector((state) => state.user.error);

  const handleClick = () => {
    dispatch(fetchUserAsync());
  };

  return (
    <div>
      {loading ? (
        <p>Loading...</p>
      ) : error ? (
        <p>Error: {error}</p>
      ) : (
        <div>
          {user && <p>User: {user.name}</p>}
          <button onClick={handleClick}>Fetch User</button>
        </div>
      )}
    </div>
  );
}
```

У компоненті ми використовуємо значення `loading`, `error` та `user` зі стану Redux за допомогою `useSelector`. При кліці на кнопку "Fetch User" ми викликаємо `fetchUserAsync` через `dispatch`.

`createAsyncThunk` спрощує обробку асинхронних операцій в Redux, використовуючи стандартні дії та автоматично генеруючи додаткові дії для вас. Ви можете легко розширювати його функціонал та використовувати його для будь-яких асинхронних операцій у своєму додатку.

Redux, Redux toolkit

RTK Query (стисло)

RTK Query (RTK Query) - це бібліотека від Redux Toolkit, яка надає швидкий та простий спосіб для виконання запитів до сервера і кешування даних у стані Redux. Вона дозволяє легко інтегрувати асинхронні запити до API у ваш додаток з мінімальними зусиллями.

Основні принципи та можливості RTK Query:

- **Декларативний підхід**: RTK Query використовує декларативний підхід для опису запитів і мутацій до сервера, використовуючи спеціальні декларативні "слайси" (slices). Ви можете описувати вхідні параметри, типи даних, URL-шаблони, методи запитів та багато іншого, що робить процес взаємодії з API більш зрозумілим і зручним.

- **Кешування даних**: RTK Query автоматично кешує дані, отримані в результаті запитів, і забезпечує їх оновлення за необхідності. Ви можете визначити, як часто оновлювати дані, коли запускати запити повторно та інші параметри кешування. Це дозволяє покращити продуктивність додатку та знизити кількість запитів до сервера.

- **Автоматичне створення Redux-схеми**: RTK Query автоматично створює Redux-схему для управління станом запитів та кешування даних. Вам не потрібно писати власні редуктори або дії Redux - RTK Query робить це за вас.

- **Реактивне оновлення стану**: Коли дані оновлюються, RTK Query автоматично оновлює стан додатку і сповіщає всі підписки та компоненти, які використовують ці дані. Це дозволяє легко відображати оновлені дані у вашому інтерфейсі користувача.

- **Вбудована обробка помилок**: RTK Query автоматично обробляє помилки, які виникають під час запитів, та забезпечує зручний спосіб управління помилками і відображення їх у вашому додатку.

- **Оптимістичні мутації**: RTK Query підтримує оптимістичні мутації, що дозволяють оновлювати стан додатку негайно після відправки мутації до сервера, навіть перед отриманням підтвердження від сервера. Це забезпечує відчуття миттєвої відповіді та покращує взаємодію з користувачем.

Корисні посилання

Redux Toolkit

Домшнє завдання

Переписати весь ваш додаток з тудушками за допомогою Redux, разом з асинхронними запитами.

Розібратись з RTK Query (схожа на react-query бібліотеку) і при бажанні (не обов’язково) використати її у своєму додатку