MobX

Ссылка видео - <https://www.youtube.com/watch?v=Rq-sIE2ceEU&t=482s&ab_channel=WebStack-Frontend>

1. Инсталл

npm i mobx mobx-react-lite mobx-utils

2. Создать состояние

Добавить в конструктор makeAutoObservable(this) и вернуть новую сущность - обьект через new

import { makeAutoObservable } from "mobx"

class CountClass {

count = 42;

constructor() {

makeAutoObservable(this)

}

increment() {

this.count += 1

}

}

export const counterState = new CountClass()

3. Использование

Обернуть всю компоненту в observer. Импортировать counterState и брать из него свойства и методы. (у методов возможна потеря контекста - подключать как на рисунке)

export const Counter = observer(() => {

return (

<div>

{counterState.count}

<button onClick={() => {counterState.increment()}}>Increment</button>

</div>

)

})

4. Асинхронные запросы

Все так же как и выше, только в классе создаем асинхронный action и закидываем в него функцию для получения запроса

const getPost = async() => {

const res = await fetch('https://jsonplaceholder.typicode.com/posts')

return res.json()

}

// далее в классе

getPostsAction = async() => {

try {

this.posts = await getPost()

} catch (error) {

console.log(error)

}

}

Затем в самой компоненте при монтировании вызывается getPostsAction.

!!! Так же нужно учитывать что mobX работает с прокси обьектами - поэтому в компоннету может прийти странный массив. Для этого есть утилита toJS

import { toJS } from "mobx";

const posts = toJS(postsState.posts);

которая преобразует массив к человеческому виду.

Утилита runInAction из mobx предназначена для дото что б действия внутри ее вызывали только один рендер реакта.

в том же запросе в классе

const post = await getPosts();

runInAction(() => { - вместо двух рендеров компоненты будет только один

this.posts = post;

this.isLoading = false;

})

Так же для удодства и чтобы избежать toJS можно использовать fromPromise из mobx-utils. С ней в экшне сохраняем данные не как данные а как результат промиса с полями state и value - и далее в компоненте на основании этого отрисовываем данные и разные состояния (загрузка ошибка фуллфилд)

getPostsAction = () => {

this.posts = fromPromise(getPost());

}

5. Дебагинг

Через эвенты - их очень много, поэтому лучше сортировать по type='action'

import { spy } from "mobx";

spy((e) => { if(e.type === 'action') console.log(e) });

6. Множество сторов (хорошая практика)

В папке store создаем фаил root-store.tsx

class Rootstore {

postsStore = postsState

counterStore = counterState

и т д

}

export RootStore;

Далее создаем контекст, оборачиваем все проиложение в рут-стор-провайдер, пишем хук useStore(можно и не писать а использовать обычный useContext) и из него забираем все эти сторы. (пример в видео по ссылке вверху)