**Основные изменения, часть 8: расширенные шаблоны RTK-запросов**

## Введение

В этом разделе мы продолжим перенос нашего примера приложения на использование RTK-запросов для других типов данных

Не все, что щас будет сказано, необходимо делать, тупо демонстрация возможностей.

## Редактирование сообщений

То, то мы тогда делали – добавление. Щас попробуем сделать редактирование поста.

### Обновление формы редактирования записи

Оч похоже на добавление. Это же тоже мутация. И тоже нужен id этому всему делу. Но метод другой PATCH

features/api/apiSlice.js

export const apiSlice = createApi({

reducerPath: 'api',

baseQuery: fetchBaseQuery({ baseUrl: '/fakeApi' }),

tagTypes: ['Post'],

endpoints: builder => ({

getPosts: builder.query({

query: () => '/posts',

providesTags: ['Post']

}),

getPost: builder.query({

query: postId => `/posts/${postId}`

}),

addNewPost: builder.mutation({

query: initialPost => ({ --- типо поч тут initial?

url: '/posts',

method: 'POST',

body: initialPost

}),

invalidatesTags: ['Post']

}),

editPost: builder.mutation({ --- мутация

query: post => ({ -- аргумент

url: `/posts/${post.id}`,

method: 'PATCH', --- новый метод

body: post --- ну и тело также передаешь

})

})

})

})

export const {

useGetPostsQuery,

useGetPostQuery,

useAddNewPostMutation,

useEditPostMutation

} = apiSlice

Как только это будет добавлено, мы сможем обновить <EditPostForm>. Ему необходимо прочитать исходную Post запись из хранилища, использовать ее для инициализации состояния компонента для редактирования полей, а затем отправить обновленные изменения на сервер. В настоящее время мы читаем Post запись с помощью selectPostById и вручную отправляем postUpdated запрос.

Мы можем использовать тот же useGetPostQuery хук, который мы использовали в <SinglePostPage> для чтения Post записи из кэша в хранилище, и мы будем использовать новый useEditPostMutation хук для обработки сохранения изменений.

features/posts/EditPostForm.js

import React, { useState } from 'react'

import { useHistory } from 'react-router-dom'

import { Spinner } from '../../components/Spinner'

import { useGetPostQuery, useEditPostMutation } from '../api/apiSlice'

export const EditPostForm = ({ match }) => {

const { postId } = match.params

const { data: post } = useGetPostQuery(postId) --- чтобы получить пост

const [updatePost, { isLoading }] = useEditPostMutation() --- чтобы мутировать пост

const [title, setTitle] = useState(post.title)

const [content, setContent] = useState(post.content)

const history = useHistory()

const onTitleChanged = e => setTitle(e.target.value)

const onContentChanged = e => setContent(e.target.value)

const onSavePostClicked = async () => {

if (title && content) {

await updatePost({ id: postId, title, content }) --- поменяли значения

history.push(`/posts/${postId}`) ---это че??

}

}

// omit rendering logic

}

### Срок действия подписки на кэширование данных

**RTK-запрос позволяет нескольким компонентам подписываться на одни и те же данные и гарантирует, что каждый уникальный набор данных извлекается только один раз.**

Когда количество активных подписок уменьшается до 0, RTK-запрос запускает внутренний таймер. **Если таймер истечет до того, как будут добавлены какие-либо новые подписки на данные, RTK Query автоматически удалит эти данные из кэша**, поскольку приложению они больше не нужны. Однако, если новая подписка добавляется до истечения срока действия таймера, таймер отменяется, и уже кэшированные данные используются без необходимости их повторной выборки.

В этом случае мы <SinglePostPage> смонтировали и запросили этого пользователя Post по идентификатору. Когда мы нажали на "Редактировать публикацию", <SinglePostPage> компонент был размонтирован маршрутизатором, а активная подписка была удалена из-за размонтирования. RTK-запрос немедленно запустил таймер "удалить эти данные post". Но <EditPostPage> компонент сразу же смонтировался и подписался на те же Post данные с тем же ключом кэша. Итак, RTK-запрос отменил таймер и продолжал использовать те же кэшированные данные вместо того, чтобы извлекать их с сервера.

По умолчанию **неиспользуемые данные удаляются из кэша через 60 секунд**, но это можно настроить.

### Аннулирование определенных элементов

Наш <EditPostForm> компонент теперь может сохранять отредактированную запись на сервер, но у нас проблема. Если мы нажмем "Сохранить запись" во время редактирования, это вернет нас к <SinglePostPage>, но по-прежнему будет отображать старые данные без изменений. В <SinglePostPage> по-прежнему используется кэшированная Post запись, которая была извлечена ранее. Если уж на то пошло, если мы вернемся на главную страницу и посмотрим на <PostsList>, там также будут показаны старые данные. **Нам нужно, чтоб заново подгрузило этот один пост.**

Ранее мы видели, как можно использовать "теги" для аннулирования частей наших кэшированных данных.

Таким образом, каждый раз, когда мы добавляем новую запись, мы заставляем RTK-запрос повторно извлекать весь список записей из getQuery конечной точки.

Мы могли бы добавить 'Post' тег как к getPost запросу, так и к editPost мутации, но это вынудило бы повторно извлекать все остальные отдельные записи. К счастью, **RTK-запрос позволяет нам определять конкретные теги, которые позволяют нам более избирательно аннулировать данные**. Эти конкретные теги выглядят как {type: 'Post', id: 123}.

Чтобы добиться правильного поведения, нам нужно настроить каждую конечную точку с правильными тегами:

* getPosts: предоставляет общий 'Post' тег для всего списка, а также конкретный {type: 'Post', id} тег для каждого полученного объекта post
* getPost: предоставляет конкретный {type: 'Post', id} объект для отдельного объекта post
* addNewPost: делает недействительным тег general 'Post' для повторной выборки всего списка
* editPost: делает недействительным конкретный тег {type: 'Post', id}. Это приведет к повторной выборке как *отдельной* записи из getPost, так и *всего* списка записей из getPosts, потому что они оба предоставляют тег, соответствующий этому {type, id} значению.

features/api/apiSlice.js

export const apiSlice = createApi({

reducerPath: 'api',

baseQuery: fetchBaseQuery({ baseUrl: '/fakeApi' }),

tagTypes: ['Post'],

endpoints: builder => ({

getPosts: builder.query({

query: () => '/posts',

providesTags: (result = [], error, arg) => [

'Post',

...result.map(({ id }) => ({ type: 'Post', id }))

]

}),

getPost: builder.query({

query: postId => `/posts/${postId}`,

providesTags: (result, error, arg) => [{ type: 'Post', id: arg }]

}),

addNewPost: builder.mutation({

query: initialPost => ({

url: '/posts',

method: 'POST',

body: initialPost

}),

invalidatesTags: ['Post']

}),

editPost: builder.mutation({

query: post => ({

url: `posts/${post.id}`,

method: 'PATCH',

body: post

}),

invalidatesTags: (result, error, arg) => [{ type: 'Post', id: arg.id }]

})

})

})

Я ничего про эти теги не понимаю

Поскольку мы определили связи между конечными точками с помощью тегов, **RTK Query знал, что ему нужно повторно выбрать отдельную запись и список записей, когда мы внесли эту правку и конкретный тег с этим идентификатором был признан недействительным** - никаких дальнейших изменений не требовалось! Тем временем, когда мы редактировали сообщение, истек срок действия таймера удаления кеша для getPosts данных, поэтому они были удалены из кеша. Когда мы снова открыли <PostsList> компонент, RTK Query увидел, что у него нет данных в кэше, и повторно извлек их.

## Управление данными пользователей

Посты сделали, очередь юзеров.

RTK Query не зависит от пользовательского интерфейса и может использоваться с любым уровнем пользовательского интерфейса, а не только с React. Обычно вам следует придерживаться использования перехватов, но здесь мы собираемся работать с пользовательскими данными, используя только API RTK Query core, чтобы вы могли увидеть, как его использовать.

Ааааа, зачем

### Выборка пользователей вручную

В настоящее время мы определяем fetchUsers асинхронный блок в usersSlice.js и отправляем этот блок вручную в index.js, чтобы список пользователей был доступен как можно скорее. Мы можем выполнить тот же процесс с помощью RTK-запросов.

Нчали

features/api/apiSlice.js

export const apiSlice = createApi({

reducerPath: 'api',

baseQuery: fetchBaseQuery({ baseUrl: '/fakeApi' }),

tagTypes: ['Post'],

endpoints: builder => ({

// omit other endpoints

getUsers: builder.query({ - пустой пока

query: () => '/users'

})

})

})

export const {

useGetPostsQuery,

useGetPostQuery,

useGetUsersQuery,

useAddNewPostMutation,

useEditPostMutation

} = apiSlice

Что внутри(в девтулсах см)

Каждый конечный объект содержит:

* Тот же самый первичный хук запроса / мутации, который мы экспортировали из объекта фрагмента корневого API, но с именем useQuery или useMutation
* Для конечных точек запросов - дополнительный набор перехватчиков запросов для таких сценариев, как "отложенные запросы" или частичные подписки
* Набор [утилит сопоставления](https://redux-toolkit.js.org/api/matching-utilities) для проверки pending/fulfilled/rejected действий, отправляемых запросами для этой конечной точки
* Функция, которая запускает запрос для этой конечной точки. initiate
* select Функция, которая создает [запоминаемые селекторы](https://redux.js.org/usage/deriving-data-selectors), которые могут извлекать кэшированные данные результата + записи состояния для этой конечной точки

Если мы хотим получить список пользователей за пределами React, мы можем отправить getUsers.initiate() запрос в нашем индексном файле

Index.js

// omit other imports

import { apiSlice } from './features/api/apiSlice'

async function main() {

// Start our mock API server

await worker.start({ onUnhandledRequest: 'bypass' })

store.dispatch(apiSlice.endpoints.getUsers.initiate())

ReactDOM.render(

<React.StrictMode>

<Provider store={store}>

<App />

</Provider>

</React.StrictMode>,

document.getElementById('root')

)

}

main()

Вот зачем все главы говорить про реакт, а в конце вдруг вспомнить, что к API не привязано

ВНИМАНИЕ

При ручной отправке RTKQ-запроса thunk будет создана запись о подписке, но затем вам придется [отказаться от подписки на эти данные позже](https://redux-toolkit.js.org/rtk-query/usage/usage-without-react-hooks#removing-a-subscription) - в противном случае данные останутся в кэше навсегда. В этом случае нам всегда нужны пользовательские данные, поэтому мы можем пропустить отмену подписки.

### Выбор пользовательских данных

В настоящее время у нас есть селекторы, такие как selectAllUsers и selectUserById, которые генерируются нашим createEntityAdapter адаптером users и считываются из state.users. Если мы перезагрузим страницу, все наше отображение, связанное с пользователем, будет нарушено, потому что на state.users фрагменте нет данных. Теперь, когда мы извлекаем данные для кэша RTK-запросов, мы должны заменить эти селекторы эквивалентами, которые вместо этого считывают данные из кэша.

endpoint.select() Функция в конечных точках фрагмента API будет создавать новую запоминающуюся функцию выбора каждый раз, когда мы ее вызываем. select() принимает ключ кэша в качестве аргумента, и это должен быть тот же ключ кэша, который вы передаете в качестве аргумента либо перехватчикам запросов, либо initiate() блоку. Сгенерированный селектор использует этот ключ кэша, чтобы точно знать, какой кэшированный результат он должен вернуть из состояния кэша в хранилище.

В этом случае нашей getUsers конечной точке не нужны никакие параметры - мы всегда извлекаем весь список пользователей. Таким образом, мы можем создать селектор кэша без аргумента, и ключ кэша становится undefined.

Иногда я к опирую целиком, потому что норм сказано. Иногда потому что ни черта не понятно

features/users/usersSlice.js

import {

createSlice,

createEntityAdapter,

createSelector

} from '@reduxjs/toolkit'

import { apiSlice } from '../api/apiSlice'

/\* Temporarily ignore adapter - we'll use this again shortly

const usersAdapter = createEntityAdapter()

const initialState = usersAdapter.getInitialState()

\*/

// Calling `someEndpoint.select(someArg)` generates a new selector that will return

// the query result object for a query with those parameters.

// To generate a selector for a specific query argument, call `select(theQueryArg)`.

// In this case, the users query has no params, so we don't pass anything to select()

export const selectUsersResult = apiSlice.endpoints.getUsers.select()

const emptyUsers = []

export const selectAllUsers = createSelector(

selectUsersResult,

usersResult => usersResult?.data ?? emptyUsers

)

export const selectUserById = createSelector(

selectAllUsers,

(state, userId) => userId,

(users, userId) => users.find(user => user.id === userId)

)

/\* Temporarily ignore selectors - we'll come back to this later

export const {

selectAll: selectAllUsers,

selectById: selectUserById,

} = usersAdapter.getSelectors((state) => state.users)

\*/

Как только у нас будет этот начальный selectUsersResult селектор, мы можем заменить существующий selectAllUsers селектор на тот, который возвращает массив пользователей из результата кэша, а затем заменить selectUserById на тот, который находит нужного пользователя из этого массива.

### Внедрение конечных точек

RTK-запросы обычно имеют один "фрагмент API" для каждого приложения, и до сих пор мы определяли все наши конечные точки непосредственно в apiSlice.js. Что произойдет, если мы захотим разделить некоторые из наших определений конечных точек с помощью кода или переместить их в другой файл, чтобы файл фрагмента API не становился слишком большим?

**RTK-запросы поддерживают разделение определений конечных точек с помощью apiSlice.injectEndpoints()**.

features/users/usersSlice.js

import { apiSlice } from '../api/apiSlice'

export const extendedApiSlice = apiSlice.injectEndpoints({ --- в инджекте весь смысл

endpoints: builder => ({

getUsers: builder.query({

query: () => '/users'

})

})

})

export const { useGetUsersQuery } = extendedApiSlice

export const selectUsersResult = extendedApiSlice.endpoints.getUsers.select()

injectEndpoints() **изменяет исходный объект фрагмента API для добавления дополнительных определений конечных точек, а затем возвращает его**.

index.js

// omit other imports

- import { apiSlice } from './features/api/apiSlice'---убрали

+ import { extendedApiSlice } from './features/users/usersSlice'

async function main() {

// Start our mock API server

await worker.start({ onUnhandledRequest: 'bypass' })

- store.dispatch(apiSlice.endpoints.getUsers.initiate()) - убрали

+ store.dispatch(extendedApiSlice.endpoints.getUsers.initiate())

ReactDOM.render(

<React.StrictMode>

<Provider store={store}>

<App />

</Provider>

</React.StrictMode>,

document.getElementById('root')

)

}

main()

## Манипулирование данными ответа

Мы просто брали ответ с сервера как он есть. А что если у него не подъоядщий вид? что, если /getPost запрос возвращает тело типа {post: {id}} с вложенными данными?

Есть пара способов, которыми мы могли бы справиться с этим концептуально. Одним из вариантов было бы извлечь responseData.post поле и сохранить его в кэше вместо всего тела. Другой вариант - хранить все данные ответа в кэше, но при этом наши компоненты должны указывать только определенную часть этих кэшированных данных, которая им нужна.

### Преобразование ответов

**Конечные точки могут определять transformResponse обработчик, который может извлекать или изменять данные, полученные от сервера, до их кэширования**. Для getPost примера у нас мог бы быть transformResponse: (responseData) => responseData.post, и он кэшировал бы только фактический Post объект вместо всего тела ответа.

В [части 6: Производительность и нормализация](https://redux.js.org/tutorials/essentials/part-6-performance-normalization) мы обсудили причины, по которым полезно хранить данные в нормализованной структуре. В частности, это позволяет нам искать и обновлять элементы на основе идентификатора, вместо того, чтобы перебирать массив в поисках нужного элемента.

Я мало что пониманию

features/users/usersSlice.js

import { apiSlice } from '../api/apiSlice'

const usersAdapter = createEntityAdapter()

const initialState = usersAdapter.getInitialState()

export const extendedApiSlice = apiSlice.injectEndpoints({

endpoints: builder => ({

getUsers: builder.query({

query: () => '/users',

transformResponse: responseData => {

return usersAdapter.setAll(initialState, responseData)

}

})

})

})

export const { useGetUsersQuery } = extendedApiSlice

// Calling `someEndpoint.select(someArg)` generates a new selector that will return

// the query result object for a query with those parameters.

// To generate a selector for a specific query argument, call `select(theQueryArg)`.

// In this case, the users query has no params, so we don't pass anything to select()

export const selectUsersResult = extendedApiSlice.endpoints.getUsers.select()

const selectUsersData = createSelector(

selectUsersResult,

usersResult => usersResult.data

)

export const { selectAll: selectAllUsers, selectById: selectUserById } =

usersAdapter.getSelectors(state => selectUsersData(state) ?? initialState)

Мы добавили transformResponse опцию к getUsers конечной точке. Она получает все тело данных ответа в качестве аргумента и должна возвращать фактические данные для кэширования. При вызове usersAdapter.setAll(initialState, responseData) программа вернет стандартную {ids: [], entities: {}} нормализованную структуру данных, содержащую все полученные элементы.

adapter.getSelectors() Функции необходимо предоставить "селектор входных данных", чтобы она знала, где найти эти нормализованные данные. В этом случае данные вложены в RTK Query cache reducer, поэтому мы выбираем нужное поле из состояния кэша.

RTK Query uses a "document cache" approach, not a "normalized cache".

### Нормализованные кэши против кэшей документов

**RTK-запрос использует подход "кэш документов", а не "нормализованный кэш"**.

Полностью нормализованный кэш пытается дедуплицировать похожие элементы во *всех* запросах на основе типа элемента и идентификатора. В качестве примера предположим, что у нас есть фрагмент API с конечными точками getTodos и getTodo, и наши компоненты выполняют следующие запросы:

* getTodos()
* getTodos({filter: 'odd'})
* getTodo({id: 1})

Каждый из этих результатов запроса будет включать объект Todo , который выглядит как {id: 1}.

В полностью нормализованном кэше с удалением дубликатов будет храниться только одна копия этого объекта Todo. Однако **RTK-запрос сохраняет каждый результат запроса в кэше независимо**. Таким образом, это приведет к тому, что три отдельные копии этого Todo будут кэшированы в хранилище Redux.

**Дедупликация**от лат. deduplicatio — «устранение дубликатов») — специализированный метод сжатия массива данных, использующий в качестве алгоритма сжатия исключение дублирующих копий повторяющихся данных.

Однако, если все конечные точки последовательно предоставляют одни и те же теги (например, {type: 'Todo', id: 1}), то аннулирование этого тега вынудит все соответствующие конечные точки повторно извлекать свои данные для обеспечения согласованности.

RTK-запрос намеренно **не реализует кэш, который дедуплицировал бы идентичные элементы в нескольких запросах**.

Для сравнения, мы просто нормализовали данные ответа для getUsers конечной точки, поскольку они хранятся в виде {[id]: value} справочной таблицы. Однако, **это не то же самое, что "нормализованный кэш" - мы только изменили способ хранения этого ответа**, а не дедуплицировали результаты между конечными точками или запросами.

### Выбор значений из результатов

<UserPage>, который фильтрует список записей на основе текущего пользователя. Мы уже видели, что можем получить весь список записей с помощью useGetPostsQuery() , а затем преобразовать его в компоненте, например, отсортировав внутри useMemo. Перехватчики запросов также дают нам возможность выбирать фрагменты кэшированного состояния, предоставляя selectFromResult опцию, и повторно отображать только при изменении выбранных фрагментов.

features/users/UsersPage.js

import { createSelector } from '@reduxjs/toolkit'

import { selectUserById } from '../users/usersSlice'

import { useGetPostsQuery } from '../api/apiSlice'

export const UserPage = ({ match }) => {

const { userId } = match.params

const user = useSelector(state => selectUserById(state, userId))

const selectPostsForUser = useMemo(() => {

const emptyArray = []

// Return a unique selector instance for this page so that

// the filtered results are correctly memoized

return createSelector(

res => res.data,

(res, userId) => userId,

(data, userId) => data?.filter(post => post.user === userId) ?? emptyArray

)

}, [])

// Use the same posts query, but extract only part of its data

const { postsForUser } = useGetPostsQuery(undefined, {

selectFromResult: result => ({

// We can optionally include the other metadata fields from the result here

...result,

// Include a field called `postsForUser` in the hook result object,

// which will be a filtered list of posts

postsForUser: selectPostsForUser(result, userId)

})

})

// omit rendering logic

}

### Сравнение подходов к преобразованию

Теперь мы рассмотрели три различных способа управления преобразованием ответов:

* Сохраняйте исходный ответ в кэше, считывайте полный результат в компоненте и извлекайте значения
* Сохраняйте исходный ответ в кэше, считывайте производный результат с помощью selectFromResult
* Преобразуйте ответ перед сохранением в кэше

Каждый из этих подходов может быть полезен в различных ситуациях. Вот несколько советов о том, когда вам следует рассмотреть возможность их использования:

* transformResponse: всем потребителям конечной точки нужен определенный формат, например, нормализация ответа для обеспечения более быстрого поиска по идентификатору
* selectFromResult: некоторым потребителям конечной точки нужны только частичные данные, такие как отфильтрованный список
* для каждого компонента / useMemo: когда только некоторым конкретным компонентам требуется преобразовать кэшированные данные

## Расширенные обновления кэша

работа с реакциями и уведомлениями

### Постоянные реакции

Изначально мы отслеживали реакции только на стороне клиента и не сохраняли их на сервере. Давайте добавим новую addReaction мутацию

features/api/apiSlice.js

export const apiSlice = createApi({

reducerPath: 'api',

baseQuery: fetchBaseQuery({ baseUrl: '/fakeApi' }),

tagTypes: ['Post'],

endpoints: builder => ({

// omit other endpoints

addReaction: builder.mutation({

query: ({ postId, reaction }) => ({

url: `posts/${postId}/reactions`,

method: 'POST',

// In a real app, we'd probably need to base this on user ID somehow

// so that a user can't do the same reaction more than once

body: { reaction }

}),

invalidatesTags: (result, error, arg) => [

{ type: 'Post', id: arg.postId }

]

})

})

})

export const {

useGetPostsQuery,

useGetPostQuery,

useAddNewPostMutation,

useEditPostMutation,

useAddReactionMutation

} = apiSlice

Заюзай в компоненте

features/posts/ReactionButtons.js

import React from 'react'

import { useAddReactionMutation } from '../api/apiSlice'

const reactionEmoji = {

thumbsUp: '👍',

hooray: '🎉',

heart: '❤️',

rocket: '🚀',

eyes: '👀'

}

export const ReactionButtons = ({ post }) => {

const [addReaction] = useAddReactionMutation()

const reactionButtons = Object.entries(reactionEmoji).map(

([reactionName, emoji]) => {

return (

<button

key={reactionName}

type="button"

className="muted-button reaction-button"

onClick={() => {

addReaction({ postId: post.id, reaction: reactionName })

}}

>

{emoji} {post.reactions[reactionName]}

</button>

)

}

)

return <div>{reactionButtons}</div>

}

В итоге Ого-го. Весь <PostsList> компонент был выделен серым цветом, потому что мы только что повторно извлекли весь список записей в ответ на обновление этой записи. Это намеренно сделано более заметным, потому что наш макет сервера API настроен на 2-секундную задержку перед ответом, но даже если ответ будет быстрее, это все равно не очень удобно для пользователя.

### Внедрение оптимистичных обновлений

Для небольшого обновления, такого как добавление реакции, нам, вероятно, не нужно повторно извлекать весь список сообщений. Вместо этого мы могли бы попробовать просто обновить уже кэшированные данные на клиенте, чтобы они соответствовали тому, что мы ожидаем получить на сервере. Кроме того, если мы немедленно обновляем кэш, пользователь получает мгновенную обратную связь при нажатии кнопки, вместо того, чтобы ждать ответа. **Такой подход к немедленному обновлению состояния клиента называется "оптимистичным обновлением"**, и это распространенный шаблон в веб-приложениях.

**RTK-запросы позволяют реализовывать оптимистичные обновления путем изменения кэша на стороне клиента на основе обработчиков "жизненного цикла запроса"**

features/api/apiSlice.js

export const apiSlice = createApi({

reducerPath: 'api',

baseQuery: fetchBaseQuery({ baseUrl: '/fakeApi' }),

tagTypes: ['Post'],

endpoints: builder => ({

// omit other endpoints

addReaction: builder.mutation({

query: ({ postId, reaction }) => ({

url: `posts/${postId}/reactions`,

method: 'POST',

// In a real app, we'd probably need to base this on user ID somehow

// so that a user can't do the same reaction more than once

body: { reaction }

}),

async onQueryStarted({ postId, reaction }, { dispatch, queryFulfilled }) {

// `updateQueryData` requires the endpoint name and cache key arguments,

// so it knows which piece of cache state to update

const patchResult = dispatch(

apiSlice.util.updateQueryData('getPosts', undefined, draft => {

// The `draft` is Immer-wrapped and can be "mutated" like in createSlice

const post = draft.find(post => post.id === postId)

if (post) {

post.reactions[reaction]++

}

})

)

try {

await queryFulfilled

} catch {

patchResult.undo()

}

}

})

})

})

onQueryStarted Обработчик получает два параметра. Первый - это ключ кэша, arg который был передан при запуске запроса. Второй - это объект, который содержит некоторые из тех же полей, что и thunkApi в createAsyncThunk ( {dispatch, getState, extra, requestId}), но также Promise вызываемый queryFulfilled. Это Promise разрешится, когда запрос вернется, и либо будет выполнено, либо отклонено в зависимости от запроса.

**updateQueryData использует Immer, поэтому вы можете "мутировать" подготовленные данные кэша так же, как и в createSlice**.

Я мало что пон

### Обновления потокового кэша

Теперь уведомления.

Обычно приложения делают первоначальный запрос на получение данных с сервера, а затем открывают подключение к Websocket для получения дополнительных обновлений с течением времени. **RTK-запросы предоставляют onCacheEntryAdded обработчик жизненного цикла конечной точки, который позволяет нам реализовывать "потоковые обновления" для кэшированных данных**.

В нашем src/api/server.js файле уже настроен макет сервера Websocket, аналогичный макету HTTP-сервера. Мы напишем новую getNotifications конечную точку, которая извлекает начальный список уведомлений, а затем устанавливает подключение к Websocket для прослушивания будущих обновлений. Нам все еще нужно вручную указывать макетному серверу, когда отправлять новые уведомления, поэтому мы продолжим имитировать это, нажимая кнопку для принудительного обновления.

features/notifications/notificationsSlice.js

import { forceGenerateNotifications } from '../../api/server'

import { apiSlice } from '../api/apiSlice'

export const extendedApi = apiSlice.injectEndpoints({

endpoints: builder => ({

getNotifications: builder.query({

query: () => '/notifications',

async onCacheEntryAdded(

arg,

{ updateCachedData, cacheDataLoaded, cacheEntryRemoved }

) {

// create a websocket connection when the cache subscription starts

const ws = new WebSocket('ws://localhost')

try {

// wait for the initial query to resolve before proceeding

await cacheDataLoaded

// when data is received from the socket connection to the server,

// update our query result with the received message

const listener = event => {

const message = JSON.parse(event.data)

switch (message.type) {

case 'notifications': {

updateCachedData(draft => {

// Insert all received notifications from the websocket

// into the existing RTKQ cache array

draft.push(...message.payload)

draft.sort((a, b) => b.date.localeCompare(a.date))

})

break

}

default:

break

}

}

ws.addEventListener('message', listener)

} catch {

// no-op in case `cacheEntryRemoved` resolves before `cacheDataLoaded`,

// in which case `cacheDataLoaded` will throw

}

// cacheEntryRemoved will resolve when the cache subscription is no longer active

await cacheEntryRemoved

// perform cleanup steps once the `cacheEntryRemoved` promise resolves

ws.close()

}

})

})

})

export const { useGetNotificationsQuery } = extendedApi

const emptyNotifications = []

export const selectNotificationsResult =

extendedApi.endpoints.getNotifications.select()

const selectNotificationsData = createSelector(

selectNotificationsResult,

notificationsResult => notificationsResult.data ?? emptyNotifications

)

export const fetchNotificationsWebsocket = () => (dispatch, getState) => {

const allNotifications = selectNotificationsData(getState())

const [latestNotification] = allNotifications

const latestTimestamp = latestNotification?.date ?? ''

// Hardcode a call to the mock server to simulate a server push scenario over websockets

forceGenerateNotifications(latestTimestamp)

}

// omit existing slice code

Внутри onCacheEntryAdded мы создаем реальное Websocket подключение к localhost. В реальном приложении это может быть любая внешняя подписка или подключение для опроса, необходимые для получения текущих обновлений. Всякий раз, когда макет-сервер отправляет нам обновление, мы помещаем все полученные уведомления в кэш и повторно сортируем их.

Когда запись в кэше удаляется, мы очищаем подписку Websocket. В этом приложении запись кэша уведомлений никогда не будет удалена, потому что мы никогда не отказываемся от подписки на данные, но важно посмотреть, как будет работать очистка в реальном приложении.

### Отслеживание состояния на стороне клиента

Нам нужно внести последний набор обновлений. Наш <Navbar> компонент должен инициировать выборку уведомлений и <NotificationsList> должен отображать записи уведомлений с правильным статусом прочитано / непрочитано. Однако ранее мы добавляли прочитанные / непрочитанные поля на стороне клиента в наш notificationsSlice редуктор при получении записей, и теперь записи уведомлений хранятся в кэше RTK-запросов.

Получение новых записей уведомлений происходит в двух случаях: когда мы извлекаем исходный список по HTTP и когда мы получаем обновление, отправленное по соединению Websocket. В идеале мы хотим использовать одну и ту же логику в ответ на оба случая.

features/notifications/notificationsSlice.js

import {

createAction,

createSlice,

createEntityAdapter,

createSelector,

isAnyOf

} from '@reduxjs/toolkit'

import { forceGenerateNotifications } from '../../api/server'

import { apiSlice } from '../api/apiSlice'

const notificationsReceived = createAction(

'notifications/notificationsReceived'

)

export const extendedApi = apiSlice.injectEndpoints({

endpoints: builder => ({

getNotifications: builder.query({

query: () => '/notifications',

async onCacheEntryAdded(

arg,

{ updateCachedData, cacheDataLoaded, cacheEntryRemoved, dispatch }

) {

// create a websocket connection when the cache subscription starts

const ws = new WebSocket('ws://localhost')

try {

// wait for the initial query to resolve before proceeding

await cacheDataLoaded

// when data is received from the socket connection to the server,

// update our query result with the received message

const listener = event => {

const message = JSON.parse(event.data)

switch (message.type) {

case 'notifications': {

updateCachedData(draft => {

// Insert all received notifications from the websocket

// into the existing RTKQ cache array

draft.push(...message.payload)

draft.sort((a, b) => b.date.localeCompare(a.date))

})

// Dispatch an additional action so we can track "read" state

dispatch(notificationsReceived(message.payload))

break

}

default:

break

}

}

ws.addEventListener('message', listener)

} catch {

// no-op in case `cacheEntryRemoved` resolves before `cacheDataLoaded`,

// in which case `cacheDataLoaded` will throw

}

// cacheEntryRemoved will resolve when the cache subscription is no longer active

await cacheEntryRemoved

// perform cleanup steps once the `cacheEntryRemoved` promise resolves

ws.close()

}

})

})

})

export const { useGetNotificationsQuery } = extendedApi

// omit selectors and websocket thunk

const notificationsAdapter = createEntityAdapter()

const matchNotificationsReceived = isAnyOf(

notificationsReceived,

extendedApi.endpoints.getNotifications.matchFulfilled

)

const notificationsSlice = createSlice({

name: 'notifications',

initialState: notificationsAdapter.getInitialState(),

reducers: {

allNotificationsRead(state, action) {

Object.values(state.entities).forEach(notification => {

notification.read = true

})

}

},

extraReducers(builder) {

builder.addMatcher(matchNotificationsReceived, (state, action) => {

// Add client-side metadata for tracking new notifications

const notificationsMetadata = action.payload.map(notification => ({

id: notification.id,

read: false,

isNew: true

}))

Object.values(state.entities).forEach(notification => {

// Any notifications we've read are no longer new

notification.isNew = !notification.read

})

notificationsAdapter.upsertMany(state, notificationsMetadata)

})

}

})

export const { allNotificationsRead } = notificationsSlice.actions

export default notificationsSlice.reducer

export const {

selectAll: selectNotificationsMetadata,

selectEntities: selectMetadataEntities

} = notificationsAdapter.getSelectors(state => state.notifications)

app/Navbar.js

import React from 'react'

import { useDispatch, useSelector } from 'react-redux'

import { Link } from 'react-router-dom'

import {

fetchNotificationsWebsocket,

selectNotificationsMetadata,

useGetNotificationsQuery

} from '../features/notifications/notificationsSlice'

export const Navbar = () => {

const dispatch = useDispatch()

// Trigger initial fetch of notifications and keep the websocket open to receive updates

useGetNotificationsQuery()

const notificationsMetadata = useSelector(selectNotificationsMetadata)

const numUnreadNotifications = notificationsMetadata.filter(

n => !n.read

).length

const fetchNewNotifications = () => {

dispatch(fetchNotificationsWebsocket())

}

let unreadNotificationsBadge

if (numUnreadNotifications > 0) {

unreadNotificationsBadge = (

<span className="badge">{numUnreadNotifications}</span>

)

}

// omit rendering logic

}

features/notifications/NotificationsList.js

import {

useGetNotificationsQuery,

allNotificationsRead,

selectMetadataEntities,

} from './notificationsSlice'

export const NotificationsList = () => {

const dispatch = useDispatch()

const { data: notifications = [] } = useGetNotificationsQuery()

const notificationsMetadata = useSelector(selectMetadataEntities)

const users = useSelector(selectAllUsers)

useLayoutEffect(() => {

dispatch(allNotificationsRead())

})

const renderedNotifications = notifications.map((notification) => {

const date = parseISO(notification.date)

const timeAgo = formatDistanceToNow(date)

const user = users.find((user) => user.id === notification.user) || {

name: 'Unknown User',

}

const metadata = notificationsMetadata[notification.id]

const notificationClassname = classnames('notification', {

new: metadata.isNew,

})

// omit rendering logic

}

В качестве последнего шага мы можем выполнить дополнительную очистку здесь - postsSlice больше не используется, так что его можно полностью удалить.

КРАТКИЕ СВЕДЕНИЯ

* **Определенные теги кэша могут использоваться для более мелкозернистого аннулирования кэша**
  + Теги кэша могут быть либо 'Post' , либо {type: 'Post', id}
  + Конечные точки могут предоставлять или аннулировать теги кэша на основе результатов и ключей arg кэша
* **API RTK Query не зависят от пользовательского интерфейса и могут использоваться вне React**
  + Объекты конечной точки включают функции для инициирования запросов, генерации селекторов результатов и сопоставления объектов действия запроса
* **Ответы могут быть преобразованы различными способами по мере необходимости**
  + Конечные точки могут определять transformResponse обратный вызов для изменения данных перед кэшированием
  + Хукам можно предоставить selectFromResult возможность извлекать / преобразовывать данные
  + Компоненты могут считывать целое значение и преобразовывать с помощью useMemo
* **RTK Query содержит расширенные опции для управления кэшированными данными для улучшения взаимодействия с пользователем**
  + Жизненный цикл onQueryStarted можно использовать для оптимистичных обновлений, обновляя кэш непосредственно перед возвратом запроса
  + Жизненный цикл onCacheEntryAdded может использоваться для потоковой передачи обновлений путем обновления кэша с течением времени на основе принудительных подключений к серверу

(вот про жизненный цикл понятно стало)