**Redux Essentials, Part 7: RTK Query Basics**

**RTK Query** — мощный инструмент извлечения и кэширования данных. Он предназначен для упрощения распространенных случаев загрузки данных в веб-приложение, **устраняя необходимость самостоятельно писать логику выборки и кэширования данных** .

RTK Query — это **дополнительная надстройка, включенная в пакет Redux Toolkit** , и ее функциональность построена поверх других API в Redux Toolkit.

За последние пару лет сообщество React пришло к пониманию того, что **«извлечение и кэширование данных» на самом деле представляет собой другой набор проблем, чем «управление состоянием»** . Хотя вы можете использовать библиотеку управления состоянием, такую ​​как Redux, для кэширования данных, варианты использования достаточно различны, поэтому стоит использовать инструменты, специально созданные для варианта использования выборки данных.

RTK Query включен в установку основного пакета Redux Toolkit. Он доступен через любую из двух точек входа ниже:

import { createApi } from '@reduxjs/toolkit/query'  
  
/\* React-specific entry point that automatically generates  
 hooks corresponding to the defined endpoints \*/  
import { createApi } from '@reduxjs/toolkit/query/react'

RTK-запрос в основном состоит из двух API:

* [createApi()](https://redux-toolkit.js.org/rtk-query/api/createApi): ядро ​​функциональности RTK Query. Он позволяет вам определить набор конечных точек, описывающих, как получать данные из ряда конечных точек, включая настройку способа получения и преобразования этих данных. В большинстве случаев вам следует использовать это один раз для каждого приложения, используя, как правило, «один фрагмент API на каждый базовый URL».
* [fetchBaseQuery()](https://redux-toolkit.js.org/rtk-query/api/fetchBaseQuery): Небольшая оболочка [fetch](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API" \t "_blank), призванная упростить запросы. Предназначен как рекомендуемый baseQueryдля использования createApiбольшинством пользователей.

Весит немного

* [Думая о кэшировании запросов RTK](https://redux.js.org/tutorials/essentials/part-7-rtk-query-basics#thinking-in-rtk-query-caching)

**мы теперь думаем об «управлении кэшированными данными »** . Вместо того, чтобы пытаться писать редукторы самостоятельно, мы теперь сосредоточимся на определении **«откуда берутся эти данные?», «как следует отправить это обновление?», «когда следует повторно получить эти кэшированные данные?», и «как следует обновлять кэшированные данные?»** .

[Настройка RTK-запроса](https://redux.js.org/tutorials/essentials/part-7-rtk-query-basics#setting-up-rtk-query)

пришло время перенести всю асинхронную логику на использование RTK Query

При использовании RTK Query **логика управления кэшированными данными централизована в одном «срезе API» для каждого приложения** . Почти так же, как у вас есть одно хранилище Redux для каждого приложения, теперь у нас есть один срез для всех наших кэшированных данных.

Начнем с определения нового apiSlice.jsфайла. Поскольку это не относится ни к одной из других «функций», которые мы уже написали, мы добавим новую features/api/папку и поместим apiSlice.js --- крч лежит отдельно

// Import the RTK Query methods from the React-specific entry point

import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'

// Define our single API slice object

export const apiSlice = createApi({

// The cache reducer expects to be added at `state.api` (already default - this is optional)

reducerPath: 'api',

// All of our requests will have URLs starting with '/fakeApi'

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

// The "endpoints" represent operations and requests for this server

endpoints: builder => ({

// The `getPosts` endpoint is a "query" operation that returns data

getPosts: builder.query({

// The URL for the request is '/fakeApi/posts'

query: () => '/posts'

})

})

})

// Export the auto-generated hook for the `getPosts` query endpoint

export const { useGetPostsQuery } = apiSlice

Когда мы вызываем createApi, есть два обязательных поля:

* baseQuery: функция, которая умеет получать данные с сервера. RTK Query включает в себя fetchBaseQueryнебольшую оболочку стандартной fetch()функции, которая выполняет типичную обработку запросов и ответов. Когда мы создаем fetchBaseQueryэкземпляр, мы можем передать базовый URL-адрес всех будущих запросов, а также переопределить поведение, например изменить заголовки запросов.
* endpoints: набор операций, которые мы определили для взаимодействия с этим сервером. Конечными точками могут быть ***запросы*** , которые возвращают данные для кэширования, или ***мутации*** , которые отправляют обновление на сервер. Конечные точки определяются с помощью функции обратного вызова, которая принимает builderпараметр и возвращает объект, содержащий определения конечных точек, созданные с помощью builder.query()и builder.mutation().

Есть еще третье – редьюсер путь.  которое определяет ожидаемое поле среза состояния верхнего уровня для сгенерированного редуктора. Хз, что это, но там есть дефолт, если будет что-то не так, выдаст ошибку.

Конечные точки

На первом этапе мы хотим добавить конечную точку, которая будет возвращать весь список сообщений с поддельного сервера API. Мы добавим конечную точку с именем getPostsи определим ее как **конечную точку запроса,** используя builder.query()

По умолчанию конечные точки запроса будут использовать GET HTTP-запрос, но вы можете переопределить это, вернув объект типа {url: '/posts', method: 'POST', body: newPost}

#### Экспорт фрагментов API и хуков

**Интеграция React RTK Query автоматически генерирует перехватчики React для *каждой* определяемой нами конечной точки!** Эти перехватчики инкапсулируют процесс запуска запроса при монтировании компонента и повторной визуализации компонента по мере обработки запроса и доступности данных. Мы можем экспортировать эти перехватчики из этого файла фрагмента API для использования в наших компонентах React.

Хуки автоматически именуются на основе стандартного соглашения:

* use, обычный префикс для любого хука React.
* Имя конечной точки, с заглавной буквы
* Тип конечной точки QueryилиMutation

В данном случае нашей конечной точкой является getPostsконечная точка запроса, поэтому сгенерированный хук — useGetPostsQuery.

### Настройка хранилища

Мидлваре тут управляет временем жизни и истечением срока действия кэша.

app/store.js

import postsReducer from '../features/posts/postsSlice'

import usersReducer from '../features/users/usersSlice'

import notificationsReducer from '../features/notifications/notificationsSlice'

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

export default configureStore({

reducer: {

posts: postsReducer,

users: usersReducer,

notifications: notificationsReducer,

[apiSlice.reducerPath]: apiSlice.reducer ---- типо имя ключа так вычисляется []

},

middleware: getDefaultMiddleware =>

getDefaultMiddleware().concat(apiSlice.middleware) --- если нужны будут еще мидлвары, их тоже можно conact как я пон

})

## Отображение записей с запросами

### Использование перехватчиков запросов (Query Hooks ) в компонентах

Давай заюзаем хук, который сделали. Импортнем его в PostList сначала

features/posts/PostsList.js

import React from 'react'

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

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

import { PostAuthor } from './PostAuthor'

import { TimeAgo } from './TimeAgo'

import { ReactionButtons } from './ReactionButtons'

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

let PostExcerpt = ({ post }) => {

return (

<article className="post-excerpt" key={post.id}>

<h3>{post.title}</h3>

<div>

<PostAuthor userId={post.user} />

<TimeAgo timestamp={post.date} />

</div>

<p className="post-content">{post.content.substring(0, 100)}</p>

<ReactionButtons post={post} />

<Link to={`/posts/${post.id}`} className="button muted-button">

View Post

</Link>

</article>

)

}

export const PostsList = () => {

const {

data: posts,

isLoading,

isSuccess,

isError,

error

} = useGetPostsQuery() - вытаскиваем из объекта, который возвращает наш хук разные штуки. Data: posts – posts это имя для data

let content

if (isLoading) {

content = <Spinner text="Loading..." />

} else if (isSuccess) {

content = posts.map(post => <PostExcerpt key={post.id} post={post} />)

} else if (isError) {

content = <div>{error.toString()}</div>

}

return (

<section className="posts-list">

<h2>Posts</h2>

{content}

</section>

)

}

<PostsList> все еще выполняет всю ту же работу, что и раньше, но **мы смогли заменить несколько useSelector вызовов и useEffect отправку одним вызовом useGetPostsQuery()**.

Авторы вщ говорят, что они не нужны, если юзаешь rtk query

Каждый сгенерированный перехватчик запроса возвращает объект "результат", содержащий несколько полей, в том числе:

* data: фактическое содержимое ответа от сервера. **Это поле будет действовать undefined до тех пор, пока не будет получен ответ**.
* isLoading: логическое значение, указывающее, выполняет ли этот перехват в данный момент *первый* запрос к серверу. (Обратите внимание, что если параметры изменятся для запроса других данных, isLoading останется false.)
* isFetching: логическое значение, указывающее, выполняет ли перехватчик в данный момент *какой-либо* запрос к серверу
* isSuccess: логическое значение, указывающее, выполнило ли перехватчик успешный запрос и доступны ли кэшированные данные (т.е., data должно быть определено сейчас)
* isError: логическое значение, указывающее, была ли ошибка в последнем запросе
* error: сериализованный объект ошибки

### Сортировка сообщений

К сожалению, записи теперь отображаются не по порядку. Ранее мы сортировали их по дате на уровне reducer с createEntityAdapter опцией сортировки. Поскольку фрагмент API просто кэширует точный массив, возвращаемый с сервера, никакой конкретной сортировки не происходит - мы получаем то, в каком порядке сервер отправляет обратно.

Как варинт – отсортировать в самом PostsList.

Мы не можем просто вызвать posts.sort() напрямую, потому что Array.sort() изменяет существующий массив, поэтому сначала нам нужно сделать его копию. Чтобы избежать повторной сортировки при каждом повторном отправлении, мы можем выполнить сортировку с помощью useMemo() хука. Мы также захотим предоставить posts пустой массив по умолчанию на случай, если это так undefined, чтобы у нас всегда был массив для сортировки.

// omit setup

export const PostsList = () => {

const {

data: posts = [],

isLoading,

isSuccess,

isError,

error

} = useGetPostsQuery()

const sortedPosts = useMemo(() => { -- закэшируй рез-т

const sortedPosts = posts.slice() --- скопируй все посты в другой массив

// Sort posts in descending chronological order

sortedPosts.sort((a, b) => b.date.localeCompare(a.date)) --- отсортируй по дате

return sortedPosts --- верни рез-т

}, [posts])

let content

if (isLoading) {

content = <Spinner text="Loading..." />

} else if (isSuccess) {

content = sortedPosts.map(post => <PostExcerpt key={post.id} post={post} />) - если ответ с сервера ок то отрисуй для каждого поста из массива отсортированных постов свой эл на экране

} else if (isError) {

content = <div>{error.toString()}</div>

}

return (

<section className="posts-list">

<h2>Posts</h2>

{content}

</section>

)

}

## Отображение отдельных записей

Мы обновили, <PostsList> чтобы получать список всех записей, и мы показываем фрагменты каждой записи Post внутри списка. Но, если мы нажмем "Просмотреть публикацию" для любого из них, наш <SinglePostPage> компонент не сможет найти публикацию в старом state.posts фрагменте и покажет нам ошибку "Публикация не найдена!" Надо обновить этот компонент.

Есть пара способов, которыми мы могли бы это сделать. Можно было бы <SinglePostPage> вызвать один и тот же useGetPostsQuery() хук, получить весь массив posts и найти только один Post объект, который ему нужно отобразить. В перехватчиках запросов также есть selectFromResult опция, которая позволила бы нам выполнить тот же поиск ранее, внутри самого перехватчика - позже мы увидим это в действии.

Но мы сделаем иначе. По сути, это же отдельная «просьба» - дай мне такой-то пост. Сделаем для этого отдельный endpoint. В RTK Query можно делать запросы на основе полученных аргументов. В нашем случае на основе id поста.

Хотя авторы говорят мол это избыточно, но фиг с ним

### Добавление единой конечной точки Post-запроса

features/api/apiSlice.js

export const apiSlice = createApi({

reducerPath: 'api',

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

endpoints: builder => ({

getPosts: builder.query({

query: () => '/posts'

}),

getPost: builder.query({ - то же query

query: postId => `/posts/${postId}` --- тут уже в отличе от getPosts есть аргумент, т.е. не (). Ну и путь также пропиывактся, ток с вычислением в конце. Аргумент мб ток 1! Если тебе надо много – пихай в объект

})

})

})

export const { useGetPostsQuery, useGetPostQuery } = apiSlice - не забывай экспортить хуки. Имя по той же схеме строится

### Аргументы запроса и ключи кэша

Меняем компонент SinglePost, теперь будем применять ток что сделанный хук

features/posts/SinglePostPage.js

import React from 'react'

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

import { Spinner } from '../../components/Spinner' --- видать лоадер еще заготовили

import { useGetPostQuery } from '../api/apiSlice' ---- импортим хук

import { PostAuthor } from './PostAuthor'

import { TimeAgo } from './TimeAgo'

import { ReactionButtons } from './ReactionButtons'

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

const { postId } = match.params

const { data: post, isFetching, isSuccess } = useGetPostQuery(postId) -- вызываем наш хук с аргументом id. Срзу деструктурируем результат, даем название data – назвали post. Хз, поч тут именно isFetching, а не isLoading

let content

if (isFetching) {

content = <Spinner text="Loading..." />

} else if (isSuccess) {

content = (

<article className="post">

<h2>{post.title}</h2>

<div>

<PostAuthor userId={post.user} />

<TimeAgo timestamp={post.date} />

</div>

<p className="post-content">{post.content}</p>

<ReactionButtons post={post} />

<Link to={`/editPost/${post.id}`} className="button">

Edit Post

</Link>

</article>

)

}

return <section>{content}</section>

}

RTK-запрос создает "ключ кэша" для каждой уникальной комбинации конечная точка + аргумент и сохраняет результаты для каждого ключа кэша отдельно. Это означает, что **вы можете использовать один и тот же перехват запроса несколько раз, передавать ему разные параметры запроса, и каждый результат будет кэшироваться отдельно в хранилище Redux**

Нужна одна и та же инфа в разных компонентах? Ниче если вызовешь хук в нескольких? Да вообще не проблема. Если аргумент один и тот же RTK Query позаботится, чтобы запрос был сделан единожды. Т.е. он кэширует.

Аргумент дб один. Если тебе надо много – пихай в 1 объект.

Все это можно проверить в инструментах разраба.

## Создание записей с изменениями

Ок, получать научились данные. А если их надо обносить? Пост отредачить н-р.

RTK-запрос позволяет нам определять **конечные точки мутации**, которые обновляют данные на сервере

### Добавление новой конечной точки после мутации

Это то же самое, что и query. Но теперь это называется mutation – типо не запрос, а мутация уже имеющегося. И теперь метод POST. А еще дб тело (body) запроса, куда свежеизмененные данные запихнем.

features/api/apiSlice.js

export const apiSlice = createApi({

reducerPath: 'api',

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

endpoints: builder => ({

getPosts: builder.query({

query: () => '/posts'

}),

getPost: builder.query({

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

}),

addNewPost: builder.mutation({

query: initialPost => ({

url: '/posts', --- я так и не пон, зачем

method: 'POST', --- метод

// Include the entire post object as the body of the request

body: initialPost --- это типо свежак??

})

})

})

})

export const {

useGetPostsQuery,

useGetPostQuery,

useAddNewPostMutation - на конце не query, а mutation

} = apiSlice

### Использование перехватов мутаций (mutation hook) в компонентах

Изменим AddPostForm. Теперь нам не нужны useDispatch и addNewPost thunk.

features/posts/AddPostForm

import React, { useState } from 'react'

import { useSelector } from 'react-redux'

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

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

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

export const AddPostForm = () => {

const [title, setTitle] = useState('')

const [content, setContent] = useState('')

const [userId, setUserId] = useState('')

const [addNewPost, { isLoading }] = useAddNewPostMutation() --- немного инач деструктуризация, видишь?

const users = useSelector(selectAllUsers)

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

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

const onAuthorChanged = e => setUserId(e.target.value)

const canSave = [title, content, userId].every(Boolean) && !isLoading -- проверка и на isLoading

const onSavePostClicked = async () => {

if (canSave) {

try {

await addNewPost({ title, content, user: userId }).unwrap() --- uwrap, чтобы можно было обработать любые потенциальные ошибки стандартным try/catch блоком.

setTitle('')

setContent('')

setUserId('')

} catch (err) {

console.error('Failed to save the post: ', err)

}

}

}

// omit rendering logic

}

Перехваты мутаций возвращают массив с двумя значениями:

* Первое значение - это "функция запуска". При вызове она отправляет запрос на сервер с любым аргументом, который вы укажете. Фактически это похоже на пакет, который уже был завернут для немедленной отправки.
* Второе значение - это объект с метаданными о текущем выполняемом запросе, если таковые имеются. Сюда входит isLoading флаг, указывающий, выполняется ли запрос.

## Обновление кэшированных данных

Когда мы нажимаем "Сохранить публикацию", мы можем просмотреть вкладку Сети в DevTools браузера и подтвердить, что HTTP POST запрос выполнен успешно. Но новое сообщение не отображается в нашем <PostsList> если мы вернемся туда. У нас все те же кэшированные данные в памяти.

Надо пнуть RTK Query, чтобы он обновил закэшированные данные. (а поч он сам это не сделал??)

### Повторная выборка сообщений вручную

Первый вариант - вручную принудительно выполнить RTK-запрос для повторной выборки данных. включают refetch функцию, которую мы можем вызвать для принудительной повторной выборки. Мы можем временно добавить кнопку "Повторная выборка записей" в <PostsList> и нажать на нее после добавления новой записи.

isLoading – первый запрос

isFetching – любой запрос! Вот в чем разница

ы могли бы посмотреть на флаг isFetching и снова заменить весь список записей на загрузочный счетчик, пока выполняется повторная выборка. Но это может немного раздражать, и, кроме того, у нас уже есть все эти записи, почему мы должны их полностью скрывать?

Вместо этого мы могли бы сделать существующий список записей частично прозрачным, чтобы указать, что данные устарели, но сохранить их видимыми, пока происходит повторная выборка. Как только запрос завершится, мы сможем вернуться к отображению списка записей в обычном режиме.

features/posts/PostsList.js

import React, { useMemo } from 'react'

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

import classnames from 'classnames'

// omit other imports and PostExcerpt

export const PostsList = () => {

const {

data: posts = [],

isLoading,

isFetching,

isSuccess,

isError,

error,

refetch

} = useGetPostsQuery()

const sortedPosts = useMemo(() => {

const sortedPosts = posts.slice()

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

return sortedPosts

}, [posts])

let content

if (isLoading) {

content = <Spinner text="Loading..." />

} else if (isSuccess) {

const renderedPosts = sortedPosts.map(post => (

<PostExcerpt key={post.id} post={post} />

))

const containerClassname = classnames('posts-container', {

disabled: isFetching

})

content = <div className={containerClassname}>{renderedPosts}</div>

} else if (isError) {

content = <div>{error.toString()}</div>

}

return (

<section className="posts-list">

<h2>Posts</h2>

<button onClick={refetch}>Refetch Posts</button>

{content}

</section>

)

}

Если мы добавим новую запись, а затем нажмем "Повторно выбрать записи", мы увидим, что список записей станет полупрозрачным на пару секунд, а затем отобразится повторно с добавлением новой записи вверху.

### Автоматическое обновление с аннулированием кэша

Типо это ок кликать на кнопку, чтобы посты обновились? Нет, конечно. Надо, чтоб оно само.

Мы знаем, что на нашем "сервере" есть полный список всех записей, включая ту, которую мы только что добавили. В идеале мы хотим, чтобы наше приложение автоматически повторно извлекало обновленный список записей сразу после завершения запроса на изменение. Таким образом, мы знаем, что наши кэшированные данные на стороне клиента синхронизированы с данными, имеющимися на сервере.

**RTK-запрос позволяет нам определять взаимосвязи между запросами и мутациями, чтобы включить автоматическую повторную выборку данных с использованием "тегов"**

* Корневое tagTypes поле в объекте API slice, объявляющее массив имен строковых тегов для таких типов данных, как 'Post'
* providesTags Массив в конечных точках запроса, содержащий список наборов тегов, описывающих данные в этом запросе
* invalidatesTags Массив в конечных точках мутации, содержащий список наборов тегов, которые становятся недействительными при каждом выполнении мутации

Мы можем добавить единственный тег с именем 'Post' в наш фрагмент API, который позволит нам автоматически выполнять повторную выборку нашей getPosts конечной точки каждый раз, когда мы добавляем новую запись:

features/api/apiSlice.js

export const apiSlice = createApi({

reducerPath: 'api',

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

tagTypes: ['Post'], --- объявили тег

endpoints: builder => ({

getPosts: builder.query({

query: () => '/posts',

providesTags: ['Post'] ---- для query

}),

getPost: builder.query({

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

}),

addNewPost: builder.mutation({

query: initialPost => ({

url: '/posts',

method: 'POST',

body: initialPost

}),

invalidatesTags: ['Post'] -- для mutation

})

})

})

Обратите внимание, что здесь нет ничего особенного в строковом литерале 'Post'. Мы могли бы назвать это 'Fred', 'qwerty' или как-нибудь еще. Просто в каждом поле должна быть одна и та же строка, чтобы RTK-запрос знал: "когда произойдет эта мутация, сделайте недействительными все конечные точки, в которых указана та же строка тега".

КРАТКИЕ СВЕДЕНИЯ

* **RTK-запрос - это решение для извлечения и кэширования данных, входящее в состав Redux Toolkit**
  + RTK-запросы упрощают процесс управления кэшированными данными сервера и избавляют от необходимости писать логику для состояния загрузки, хранения результатов и выполнения запросов
  + RTK-запросы создаются поверх тех же шаблонов, что используются в Redux, например, асинхронных блоков
* **RTK-запрос использует один "фрагмент API" для каждого приложения, определенный с помощью createApi**
  + RTK Query предоставляет версии, не зависящие от пользовательского интерфейса и зависящие от React createApi
  + Фрагменты API определяют несколько "конечных точек" для различных серверных операций
  + Фрагмент API включает автоматически генерируемые перехватчики React, если используется интеграция с React
* **Конечные точки запросов позволяют извлекать и кэшировать данные с сервера**
  + Перехватчики запросов возвращают data значение плюс флаги состояния загрузки
  + Запрос может быть повторно извлечен вручную или автоматически с использованием "тегов" для аннулирования кэша
* **Конечные точки мутации позволяют обновлять данные на сервере**
  + Перехваты мутаций возвращают функцию "триггера", которая отправляет запрос на обновление, а также состояние загрузки
  + Функция триггера возвращает обещание, которое можно "развернуть" и ожидать