RTK Query

Version (rtk) - **`1.9.5`**

https://redux-toolkit.js.org/rtk-query/overview

RTK Query Overview

RTK Query is a powerful data fetching and caching tool. It is designed to simplify common cases for loading data in a web application, **eliminating the need to hand-write data fetching & caching logic yourself**.

RTK Query is **an optional addon included in the Redux Toolkit package**, and its functionality is built on top of the other APIs in Redux Toolkit.

Motivation

Web applications normally need to fetch data from a server in order to display it. They also usually need to make updates to that data, send those updates to the server, and keep the cached data on the client in sync with the data on the server. This is made more complicated by the need to implement other behaviors used in today's applications:

- Tracking loading state in order to show UI spinners
- Avoiding duplicate requests for the same data
- Optimistic updates to make the UI feel faster
- Managing cache lifetimes as the user interacts with the UI

The Redux core has always been very minimal - it's up to developers to write all the actual logic. That means that Redux has never included anything built in to help solve these use cases. The Redux docs have taught some common patterns for dispatching actions around the request lifecycle to track loading state and request results, and Redux Toolkit's `createAsyncThunk` API was designed to abstract that typical pattern. However, users still have to write significant amounts of reducer logic to manage the loading state and the cached data.

Over the last couple years, the React community has come to realize that **"data fetching and caching" is really a different set of concerns than "state management"**. While you can use a state management library like Redux to cache data, the use cases are different enough that it's worth using tools that are purpose-built for the data fetching use case.

RTK Query takes inspiration from other tools that have pioneered solutions for data fetching, like Apollo Client, React Query, Urql, and SWR, but adds a unique approach to its API design:

- The data fetching and caching logic is built on top of Redux Toolkit's `createSlice` and `createAsyncThunk` APIs
- Because Redux Toolkit is UI-agnostic, RTK Query's functionality can be used with any UI layer
- API endpoints are defined ahead of time, including how to generate query parameters from arguments and transform responses for caching
- RTK Query can also generate React hooks that encapsulate the entire data fetching process, provide `data` and `isLoading` fields to components, and manage the lifetime of cached data as components mount and unmount
- RTK Query provides "cache entry lifecycle" options that enable use cases like streaming cache updates via websocket messages after fetching the initial data
- We have early working examples of code generation of API slices from OpenAPI and GraphQL schemas
- Finally, RTK Query is completely written in TypeScript, and is designed to provide an excellent TS usage experience

What's included

APIs

RTK Query is included within the installation of the core Redux Toolkit package. It is available via either of the two entry points below:

```
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 Query includes these APIs:

- `createApi()`: The core of RTK Query's functionality. It allows you to define a set of endpoints describe how to retrieve data from a series of endpoints, including configuration of how to fetch and transform that data. In most cases, you should use this once per app, with "one API slice per base URL" as a rule of thumb.
- `fetchBaseQuery()`: A small wrapper around `fetch` that aims to simplify requests. Intended as the recommended `baseQuery` to be used in `createApi` for the majority of users.
- `<ApiProvider />`: Can be used as a `Provider` if you **do not already have a Redux store**.
- `setupListeners()`: A utility used to enable `refetchOnMount` and `refetchOnReconnect` behaviors.

Bundle Size

RTK Query adds a fixed one-time amount to your app's bundle size. Since RTK Query builds on top of Redux Toolkit and React-Redux, the added size varies depending on whether you are already using those in your app. The estimated min+gzip bundle sizes are:

- If you are using RTK already: ~9kb for RTK Query and ~2kb for the hooks.
- If you are not using RTK already:
  - Without React: 17 kB for RTK+dependencies+RTK Query
  - With React: 19kB + React-Redux, which is a peer dependency

Adding additional endpoint definitions should only increase size based on the actual code inside the `endpoints` definitions, which will typically be just a few bytes.

The functionality included in RTK Query quickly pays for the added bundle size, and the elimination of hand-written data fetching logic should be a net improvement in size for most meaningful applications.

Basic Usage

Create an API Slice

RTK Query is included within the installation of the core Redux Toolkit package. It is available via either of the two entry points below:

```
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'
```

For typical usage with React, start by importing `createApi` and defining an "API slice" that lists the server's base URL and which endpoints we want to interact with:

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
import type { Pokemon } from './types'

// Define a service using a base URL and expected endpoints
export const pokemonApi = createApi({
  reducerPath: 'pokemonApi',
  baseQuery: fetchBaseQuery({ baseUrl: 'https://pokeapi.co/api/v2/' }),
  endpoints: (builder) => ({
    getPokemonByName: builder.query<Pokemon, string>({
      query: (name) => `pokemon/${name}`,
    }),
  }),
})

// Export hooks for usage in functional components, which are
// auto-generated based on the defined endpoints
export const { useGetPokemonByNameQuery } = pokemonApi
```

Configure the Store

The "API slice" also contains an auto-generated Redux slice reducer and a custom middleware that manages subscription lifetimes. Both of those need to be added to the Redux store:

```
import { configureStore } from '@reduxjs/toolkit'
// Or from '@reduxjs/toolkit/query/react'
import { setupListeners } from '@reduxjs/toolkit/query'
import { pokemonApi } from './services/pokemon'

export const store = configureStore({
  reducer: {
    // Add the generated reducer as a specific top-level slice
    [pokemonApi.reducerPath]: pokemonApi.reducer,
  },
  // Adding the api middleware enables caching, invalidation, polling,
  // and other useful features of `rtk-query`.
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware().concat(pokemonApi.middleware),
})

// optional, but required for refetchOnFocus/refetchOnReconnect behaviors
// see `setupListeners` docs - takes an optional callback as the 2nd arg for customization
setupListeners(store.dispatch)
```

Use Hooks in Components

Finally, import the auto-generated React hooks from the API slice into your component file, and call the hooks in your component with any needed parameters. RTK Query will automatically fetch data on mount, re-fetch when parameters change, provide `{data, isFetching}` values in the result, and re-render the component as those values change:

```
import * as React from 'react'
import { useGetPokemonByNameQuery } from './services/pokemon'

export default function App() {
  // Using a query hook automatically fetches data and returns query values
  const { data, error, isLoading } = useGetPokemonByNameQuery('bulbasaur')
  // Individual hooks are also accessible under the generated endpoints:
  // const { data, error, isLoading } = pokemonApi.endpoints.getPokemonByName.useQuery('bulbasaur')

  // render UI based on data and loading state
}
```

Further Information

*Last updated on **Jun 24, 2022***

Comparison with Other Tools

RTK Query takes inspiration from many other data fetching libraries in the ecosystem. Much like the Redux core library was inspired by tools like Flux and Elm, RTK Query builds on API design patterns and feature concepts popularized by libraries like React Query, SWR, Apollo, and Urql. RTK Query has been written from scratch, but tries to use the best concepts from those libraries and other data fetching tools, with an eye towards leveraging the unique strengths and capabilities of Redux.

We think that all of those tools are great! If you're using one of them, you're happy with it, and it solves the problems you are facing in your app, keep using that tool. The information on this page is meant to help show **where there are differences in features, implementation approaches, and API design**. The goal is to help you **make informed decisions and understand tradeoffs**, rather than argue that tool X is better than tool Y.

When Should You Use RTK Query?

In general, the main reasons to use RTK Query are:

- You already have a Redux app and you want to simplify your existing data fetching logic
- You want to be able to use the Redux DevTools to see the history of changes to your state over time
- You want to be able to integrate the RTK Query behavior with the rest of the Redux ecosystem
- Your app logic needs to work outside of React

Unique Capabilities

RTK Query has some unique API design aspects and capabilities that are worth considering.

- With React Query and SWR, you usually define your hooks yourself, and you can do that all over the place and on the fly. With RTK Query, you do so in one central place by defining an "API slice" with multiple endpoints ahead of time. This allows for a more tightly integrated model of mutations automatically invalidating/refetching queries on trigger.
- Because RTK Query dispatches normal Redux actions as requests are processed, all actions are visible in the Redux DevTools. Additionally, every request is automatically visible to your Redux reducers and can easily update the global application state if necessary (see example). You can use the endpoint matcher functionality to do additional processing of cache-related actions in your own reducers.
- Like Redux itself, the main RTK Query functionality is UI-agnostic and can be used with any UI layer
- You can easily invalidate entities or patch existing query data (via `util.updateQueryData`) from middleware.

- RTK Query enables streaming cache updates, such as updating the initial fetched data as messages are received over a websocket, and has built in support for optimistic updates as well.
- RTK Query ships a very tiny and flexible fetch wrapper: `fetchBaseQuery`. It's also very easy to swap our client with your own, such as using `axios`, `redaxios`, or something custom.
- RTK Query has a (currently experimental) code-gen tool that will take an OpenAPI spec or GraphQL schema and give you a typed API client, as well as provide methods for enhancing the generated client after the fact.

Tradeoffs

No Normalized or Deduplicated Cache

RTK Query deliberately **does *not* implement a cache that would deduplicate identical items across multiple requests**. There are several reasons for this:

- A fully normalized shared-across-queries cache is a *hard* problem to solve
- We don't have the time, resources, or interest in trying to solve that right now
- In many cases, simply refetching data when it's invalidated works well and is easier to understand
- At a minimum, RTKQ can help solve the general use case of "fetch some data", which is a big pain point for a lot of people

Bundle Size

RTK Query adds a fixed one-time amount to your app's bundle size. Since RTK Query builds on top of Redux Toolkit and React-Redux, the added size varies depending on whether you are already using those in your app. The estimated min+gzip bundle sizes are:

- If you are using RTK already: ~9kb for RTK Query and ~2kb for the hooks.
- If you are not using RTK already:
  - Without React: 17 kB for RTK+dependencies+RTK Query
  - With React: 19kB + React-Redux, which is a peer dependency

Adding additional endpoint definitions should only increase size based on the actual code inside the `endpoints` definitions, which will typically be just a few bytes.

The functionality included in RTK Query quickly pays for the added bundle size, and the elimination of hand-written data fetching logic should be a net improvement in size for most meaningful applications.

Comparing Feature Sets

It's worth comparing the feature sets of all these tools to get a sense of their similarities and differences.

This comparison table strives to be as accurate and as unbiased as possible. If you use any of these libraries and feel the information could be improved, feel free to suggest changes (with notes or evidence of claims) by opening an issue.

Feature	rtk-query	react-query	apollo	urql
Supported Protocols	any, REST included	any, none included	GraphQL	GraphQL
API Definition	declarative	on use, declarative	GraphQL schema	GraphQL schema
Cache by	endpoint + serialized arguments	user-defined query-key	type/id	type/id?
Invalidation Strategy + Refetching	declarative, by type and/or type/id	manual by cache key	automatic cache updates on per-entity level, manual query invalidation by cache key	declarative, by type OR automatic cache updates on per-entity level, manual query invalidation by cache key
Polling	yes	yes	yes	yes
Parallel queries	yes	yes	yes	yes
Dependent queries	yes	yes	yes	yes
Skip queries	yes	yes	yes	yes

Lagged queries	yes	yes	no	?
Auto garbage collection	yes	yes	no	?
Normalized caching	no	no	yes	yes
Infinite scrolling	TODO	yes	requires manual code	?
Prefetching	yes	yes	yes	yes?
Retrying	yes	yes	requires manual code	?
Optimistic updates	can update cache by hand	can update cache by hand	`optimisticResponse`	?
Manual cache manipulation	yes	yes	yes	yes
Platforms	hooks for React, everywhere Redux works	hooks for React	various	various

Further Information

- The React Query "Comparison" page has an additional detailed feature set comparison table and discussion of capabilities
- Urql maintainer Phil Pluckthun wrote an excellent explanation of what a "normalized cache" is and how Urql's cache works
- The RTK Query "Cache Behavior" page has further details on why RTK Query does not implement a normalized cache

*Last updated on **Apr 28, 2023***

Usage With TypeScript

Introduction

As with the rest of the Redux Toolkit package, RTK Query is written in TypeScript, and its API is designed for seamless use in TypeScript applications.

This page provides details for using APIs included in RTK Query with TypeScript and how to type them correctly.

`createApi`

Using auto-generated React Hooks

The React-specific entry point for RTK Query exports a version of `createApi` which automatically generates React hooks for each of the defined query & mutation `endpoints`.

To use the auto-generated React Hooks as a TypeScript user, **you'll need to use TS4.1+**.

```
// Need to use the React-specific entry point to allow generating React hooks
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
import type { Pokemon } from './types'

// Define a service using a base URL and expected endpoints
export const pokemonApi = createApi({
  reducerPath: 'pokemonApi',
  baseQuery: fetchBaseQuery({ baseUrl: 'https://pokeapi.co/api/v2/' }),
  endpoints: (builder) => ({
    getPokemonByName: builder.query<Pokemon, string>({
      query: (name) => `pokemon/${name}`,
    }),
  }),
})

// Export hooks for usage in function components, which are
// auto-generated based on the defined endpoints
export const { useGetPokemonByNameQuery } = pokemonApi
```

For older versions of TS, you can use `api.endpoints.[endpointName].useQuery/useMutation` to access the same hooks.

```
import { pokemonApi } from './pokemon'

const useGetPokemonByNameQuery = pokemonApi.endpoints.getPokemonByName.useQuery
```

Typing a `baseQuery`

Typing a custom `baseQuery` can be done using the `BaseQueryFn` type exported by RTK Query.

```
export type BaseQueryFn<
  Args = any,
  Result = unknown,
  Error = unknown,
  DefinitionExtraOptions = {},
  Meta = {}
> = (
  args: Args,
  api: BaseQueryApi,
  extraOptions: DefinitionExtraOptions
) => MaybePromise<QueryReturnValue<Result, Error, Meta>>

export interface BaseQueryApi {
  signal: AbortSignal
  dispatch: ThunkDispatch<any, any, any>
  getState: () => unknown
}

export type QueryReturnValue<T = unknown, E = unknown, M = unknown> =
  | {
      error: E
      data?: undefined
      meta?: M
    }
  | {
      error?: undefined
      data: T
      meta?: M
    }
```

The `BaseQueryFn` type accepts the following generics:

- `Args` - The type for the first parameter of the function. The result returned by a `query` property on an endpoint will be passed here.
- `Result` - The type to be returned in the `data` property for the success case. Unless you expect all queries and mutations to return the same type, it is recommended to keep this typed as `unknown`, and specify the types individually as shown below.
- `Error` - The type to be returned for the `error` property in the error case. This type also applies to all `queryFn` functions used in endpoints throughout the API definition.

- `DefinitionExtraOptions` - The type for the third parameter of the function. The value provided to the `extraOptions` property on an endpoint will be passed here.
- `Meta` - the type of the `meta` property that may be returned from calling the `baseQuery`. The `meta` property is accessible as the second argument to `transformResponse` and `transformErrorResponse`.

NOTE

The `meta` property returned from a `baseQuery` will always be considered as potentially undefined, as a `throw` in the error case may result in it not being provided. When accessing values from the `meta` property, this should be accounted for, e.g. using optional chaining

```
import { createApi } from '@reduxjs/toolkit/query'
import type { BaseQueryFn } from '@reduxjs/toolkit/query'

const simpleBaseQuery: BaseQueryFn<
  string, // Args
  unknown, // Result
  { reason: string }, // Error
  { shout?: boolean }, // DefinitionExtraOptions
  { timestamp: number } // Meta
> = (arg, api, extraOptions) => {
  // `arg` has the type `string`
  // `api` has the type `BaseQueryApi` (not configurable)
  // `extraOptions` has the type `{ shout?: boolean }

  const meta = { timestamp: Date.now() }

  if (arg === 'forceFail') {
    return {
      error: {
        reason: 'Intentionally requested to fail!',
        meta,
      },
    }
  }

  if (extraOptions.shout) {
    return { data: 'CONGRATULATIONS', meta }
  }

  return { data: 'congratulations', meta }
}

const api = createApi({
  baseQuery: simpleBaseQuery,
  endpoints: (builder) => ({
    getSupport: builder.query({
```

```
    query: () => 'support me',
    extraOptions: {
      shout: true,
    },
   }),
  }),
})
```

Typing query and mutation `endpoints`

`endpoints` for an api are defined as an object using the builder syntax.
Both `query` and `mutation` endpoints can be typed by providing types to the generics in `<ResultType, QueryArg>` format.

- `ResultType` - The type of the final data returned by the query, factoring an optional `transformResponse`.
  - If `transformResponse` is not provided, then it is treated as though a successful query will return this type instead.
  - If `transformResponse` *is* provided, the input type for `transformResponse` must also be specified, to indicate the type that the initial query returns. The return type for `transformResponse` must match `ResultType`.
  - If `queryFn` is used rather than `query`, then it must return the following shape for the success case:

```
{
 data: ResultType
}
```

- `QueryArg` - The type of the input that will be passed as the only parameter to the `query` property of the endpoint, or the first parameter of a `queryFn` property if used instead.
  - If `query` doesn't have a parameter, then `void` type has to be provided explicitly.
  - If `query` has an optional parameter, then a union type with the type of parameter, and `void` has to be provided, e.g. `number | void`.

NOTE

`queries` and `mutations` can also have their return type defined by a `baseQuery` rather than the method shown above, however, unless you expect all of your queries and mutations to return the same type, it is recommended to leave the return type of the `baseQuery` as `unknown`.

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
interface Post {
  id: number
  name: string
}

const api = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  endpoints: (build) => ({
    //            ResultType  QueryArg
    //                 v      v
    getPost: build.query<Post, number>({
      // inferred as `number` from the `QueryArg` type
      //      v
      query: (id) => `post/${id}`,
      // An explicit type must be provided to the raw result that the query returns
      // when using `transformResponse`
      //                        v
      transformResponse: (rawResult: { result: { post: Post } }, meta) => {
        //                                        ^
        // The optional `meta` property is available based on the type for the `baseQuery` used

        // The return value for `transformResponse` must match `ResultType`
        return rawResult.result.post
      },
    }),
  }),
})
```

Typing a `queryFn`

As mentioned in Typing query and mutation endpoints, a `queryFn` will receive its result & arg types from the generics provided to the corresponding built endpoint.

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
import { getRandomName } from './randomData'

interface Post {
  id: number
  name: string
}

const api = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  endpoints: (build) => ({
    //            ResultType  QueryArg
    //                 v      v
    getPost: build.query<Post, number>({
      // inferred as `number` from the `QueryArg` type
```

```
    //       v
    queryFn: (arg, queryApi, extraOptions, baseQuery) => {
      const post: Post = {
        id: arg,
        name: getRandomName(),
      }
      // For the success case, the return type for the `data` property
      // must match `ResultType`
      //            v
      return { data: post }
    },
  }),
 }),
})
```

The error type that a `queryFn` must return is determined by the `baseQuery` provided to `createApi`.

With `fetchBaseQuery`, the error type is like so:

```
{
 status: number
 data: any
}
```

An error case for the example above using `queryFn` and the error type from `fetchBaseQuery` could look like:

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
import { getRandomName } from './randomData'

interface Post {
 id: number
 name: string
}

const api = createApi({
 baseQuery: fetchBaseQuery({ baseUrl: '/' }),
 endpoints: (build) => ({
  getPost: build.query<Post, number>({
   queryFn: (arg, queryApi, extraOptions, baseQuery) => {
    if (arg <= 0) {
     return {
      error: {
       status: 500,
       statusText: 'Internal Server Error',
       data: 'Invalid ID provided.',
      },
     }
    }
```

```
      const post: Post = {
        id: arg,
        name: getRandomName(),
      }
      return { data: post }
    },
  }),
 }),
})
```

For users who wish to *only* use `queryFn` for each endpoint and not include a `baseQuery` at all, RTK Query provides a `fakeBaseQuery` function that can be used to easily specify the error type each `queryFn` should return.

```
import { createApi, fakeBaseQuery } from '@reduxjs/toolkit/query'

type CustomErrorType = { reason: 'too cold' | 'too hot' }

const api = createApi({
 // This type will be used as the error type for all `queryFn` functions provided
 //                         v
 baseQuery: fakeBaseQuery<CustomErrorType>(),
 endpoints: (build) => ({
  eatPorridge: build.query<'just right', 1 | 2 | 3>({
    queryFn(seat) {
      if (seat === 1) {
        return { error: { reason: 'too cold' } }
      }

      if (seat === 2) {
        return { error: { reason: 'too hot' } }
      }

      return { data: 'just right' }
    },
  }),
  microwaveHotPocket: build.query<'delicious!', number>({
    queryFn(duration) {
      if (duration < 110) {
        return { error: { reason: 'too cold' } }
      }
      if (duration > 140) {
        return { error: { reason: 'too hot' } }
      }

      return { data: 'delicious!' }
    },
  }),
 }),
})
```

Typing `providesTags`/`invalidatesTags`

RTK Query utilizes a cache tag invalidation system in order to provide automated re-fetching of stale data.

When using the function notation, both the `providesTags` and `invalidatesTags` properties on endpoints are called with the following arguments:

- result: `ResultType` | `undefined` - The result returned by a successful query. The type corresponds with `ResultType` as supplied to the built endpoint. In the error case for a query, this will be `undefined`.
- error: `ErrorType` | `undefined` - The error returned by an errored query. The type corresponds with `Error` as supplied to the `baseQuery` for the api. In the success case for a query, this will be `undefined`.
- arg: `QueryArg` - The argument supplied to the `query` property when the query itself is called. The type corresponds with `QueryArg` as supplied to the built endpoint.

A recommended use-case with `providesTags` when a query returns a list of items is to provide a tag for each item in the list using the entity ID, as well as a 'LIST' ID tag (see Advanced Invalidation with abstract tag IDs).

This is often written by spreading the result of mapping the received data into an array, as well as an additional item in the array for the `'LIST'` ID tag. When spreading the mapped array, by default, TypeScript will broaden the `type` property to `string`. As the tag `type` must correspond to one of the string literals provided to the `tagTypes` property of the api, the broad `string` type will not satisfy TypeScript. In order to alleviate this, the tag `type` can be cast `as const` to prevent the type being broadened to `string`.

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
interface Post {
  id: number
  name: string
}
type PostsResponse = Post[]

const api = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  tagTypes: ['Posts'],
  endpoints: (build) => ({
    getPosts: build.query<PostsResponse, void>({
      query: () => 'posts',
      providesTags: (result) =>
        result
```

```
        ? [
            ...result.map(({ id }) => ({ type: 'Posts' as const, id })),
            { type: 'Posts', id: 'LIST' },
          ]
        : [{ type: 'Posts', id: 'LIST' }],
    }),
  }),
})
```

Skipping queries with TypeScript using `skipToken`

RTK Query provides the ability to conditionally skip queries from automatically running using the `skip` parameter as part of query hook options (see Conditional Fetching).

TypeScript users may find that they encounter invalid type scenarios when a query argument is typed to not be `undefined`, and they attempt to `skip` the query when an argument would not be valid.

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
import type { Post } from './types'

export const api = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  endpoints: (build) => ({
    // Query argument is required to be `number`, and can't be `undefined`
    //                       V
    getPost: build.query<Post, number>({
      query: (id) => `post/${id}`,
    }),
  }),
})

export const { useGetPostQuery } = api
```

```
import { useGetPostQuery } from './api'

function MaybePost({ id }: { id?: number }) {
  // This will produce a typescript error:
  // Argument of type 'number | undefined' is not assignable to parameter of type 'number | unique
symbol'.
  // Type 'undefined' is not assignable to type 'number | unique symbol'.

  // @ts-expect-error id passed must be a number, but we don't call it when it isn't a number
  const { data } = useGetPostQuery(id, { skip: !id })

  return <div>...</div>
}
```

While you might be able to convince yourself that the query won't be called unless the `id` arg is a `number` at the time, TypeScript won't be convinced so easily.

RTK Query provides a `skipToken` export which can be used as an alternative to the `skip` option in order to skip queries, while remaining type-safe.
When `skipToken` is passed as the query argument to `useQuery`, `useQueryState` or `useQuerySubscription`, it provides the same effect as setting `skip: true` in the query options, while also being a valid argument in scenarios where the `arg` might be undefined otherwise.

```
import { skipToken } from '@reduxjs/toolkit/query/react'
import { useGetPostQuery } from './api'

function MaybePost({ id }: { id?: number }) {
  // When `id` is nullish, we will still skip the query.
  // TypeScript is also happy that the query will only ever be called with a `number` now
  const { data } = useGetPostQuery(id ?? skipToken)

  return <div>...</div>
}
```

Type safe error handling

When an error is gracefully provided from a `base query`, RTK query will provide the error directly. If an unexpected error is thrown by user code rather than a handled error, that error will be transformed into a `SerializedError` shape. Users should make sure that they are checking which kind of error they are dealing with before attempting to access its properties. This can be done in a type safe manner either by using a type guard, e.g. by checking for discriminated properties, or using a type predicate.

When using `fetchBaseQuery`, as your base query, errors will be of type `FetchBaseQueryError | SerializedError`. The specific shapes of those types can be seen below.

```
export type FetchBaseQueryError =
  | {
      /**
       * * `number`:
       *   HTTP status code
       */
      status: number
      data: unknown
    }
```

```
  | {
      /**
       * * `"FETCH_ERROR"`:
       *   An error that occurred during execution of `fetch` or the `fetchFn` callback option
       **/
      status: 'FETCH_ERROR'
      data?: undefined
      error: string
    }
  | {
      /**
       * * `"PARSING_ERROR"`:
       *   An error happened during parsing.
       *   Most likely a non-JSON-response was returned with the default `responseHandler` "JSON",
       *   or an error occurred while executing a custom `responseHandler`.
       **/
      status: 'PARSING_ERROR'
      originalStatus: number
      data: string
      error: string
    }
  | {
      /**
       * * `"CUSTOM_ERROR"`:
       *   A custom error type that you can return from your `queryFn` where another error might not
make sense.
       **/
      status: 'CUSTOM_ERROR'
      data?: unknown
      error: string
    }
```

```
export interface SerializedError {
  name?: string
  message?: string
  stack?: string
  code?: string
}
```

Error result example

When using `fetchBaseQuery`, the `error` property returned from a hook will have the type `FetchBaseQueryError | SerializedError | undefined`. If an error is present, you can access error properties after narrowing the type to either `FetchBaseQueryError` or `SerializedError`.

```
import { api } from './services/api'

function PostDetail() {
  const { data, error, isLoading } = usePostsQuery()

  if (isLoading) {
    return <div>Loading...</div>
  }

  if (error) {
    if ('status' in error) {
      // you can access all properties of `FetchBaseQueryError` here
      const errMsg = 'error' in error ? error.error : JSON.stringify(error.data)

      return (
        <div>
          <div>An error has occurred:</div>
          <div>{errMsg}</div>
        </div>
      )
    } else {
      // you can access all properties of `SerializedError` here
      return <div>{error.message}</div>
    }
  }

  if (data) {
    return (
      <div>
        {data.map((post) => (
          <div key={post.id}>Name: {post.name}</div>
        ))}
      </div>
    )
  }

  return null
}
```

Inline error handling example

When handling errors inline after `unwrapping` a mutation call, a thrown error will have a type of `any` for typescript versions below 4.4, or `unknown` for versions 4.4+. In order to safely access properties of the error, you must first narrow the type to a known type. This can be done using a type predicate as shown below.

```
import { FetchBaseQueryError } from '@reduxjs/toolkit/query'

/**
 * Type predicate to narrow an unknown error to `FetchBaseQueryError`
 */
export function isFetchBaseQueryError(
  error: unknown
): error is FetchBaseQueryError {
  return typeof error === 'object' && error != null && 'status' in error
}

/**
 * Type predicate to narrow an unknown error to an object with a string 'message' property
 */
export function isErrorWithMessage(
  error: unknown
): error is { message: string } {
  return (
    typeof error === 'object' &&
    error != null &&
    'message' in error &&
    typeof (error as any).message === 'string'
  )
}
```

```
import { useState } from 'react'
import { useSnackbar } from 'notistack'
import { api } from './services/api'
import { isFetchBaseQueryError, isErrorWithMessage } from './services/helpers'

function AddPost() {
  const { enqueueSnackbar, closeSnackbar } = useSnackbar()
  const [name, setName] = useState('')
  const [addPost] = useAddPostMutation()

  async function handleAddPost() {
    try {
      await addPost(name).unwrap()
      setName('')
    } catch (err) {
      if (isFetchBaseQueryError(err)) {
        // you can access all properties of `FetchBaseQueryError` here
        const errMsg = 'error' in err ? err.error : JSON.stringify(err.data)
        enqueueSnackbar(errMsg, { variant: 'error' })
      } else if (isErrorWithMessage(err)) {
        // you can access a string 'message' property here
        enqueueSnackbar(err.message, { variant: 'error' })
      }
    }
  }

  return (
    <div>
      <input value={name} onChange={(e) => setName(e.target.value)} />
      <button>Add post</button>
    </div>
  )}
```

*Last updated on **Apr 2, 2023***

Queries

Overview

This is the most common use case for RTK Query. A query operation can be performed with any data fetching library of your choice, but the general recommendation is that you only use queries for requests that retrieve data. For anything that alters data on the server or will possibly invalidate the cache, you should use a Mutation.

By default, RTK Query ships with `fetchBaseQuery`, which is a lightweight `fetch` wrapper that automatically handles request headers and response parsing in a manner similar to common libraries like `axios`.
See Customizing Queries if `fetchBaseQuery` does not handle your requirements.

> INFO
>
> Depending on your environment, you may need to polyfill `fetch` with `node-fetch` or `cross-fetch` if you choose to use `fetchBaseQuery` or `fetch` on its own.

See `useQuery` for the hook signature and additional details.

Defining Query Endpoints

Query endpoints are defined by returning an object inside the `endpoints` section of `createApi`, and defining the fields using the `builder.query()` method.

Query endpoints should define either a `query` callback that constructs the URL (including any URL query params), or a `queryFn` callback that may do arbitrary async logic and return a result.

If the `query` callback needs additional data to generate the URL, it should be written to take a single argument. If you need to pass in multiple parameters, pass them formatted as a single "options object".

Query endpoints may also modify the response contents before the result is cached, define "tags" to identify cache invalidation, and provide cache entry lifecycle callbacks to run additional logic as cache entries are added and removed.

```
// Or from '@reduxjs/toolkit/query/react'
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query'
import type { Post } from './types'

const api = createApi({
  baseQuery: fetchBaseQuery({
    baseUrl: '/',
  }),
  tagTypes: ['Post'],
  endpoints: (build) => ({
    getPost: build.query<Post, number>({
      // note: an optional `queryFn` may be used in place of `query`
      query: (id) => ({ url: `post/${id}` }),
      // Pick out data and prevent nested properties in a hook or selector
      transformResponse: (response: { data: Post }, meta, arg) => response.data,
      // Pick out errors and prevent nested properties in a hook or selector
      transformErrorResponse: (
        response: { status: string | number },
        meta,
        arg
      ) => response.status,
      providesTags: (result, error, id) => [{ type: 'Post', id }],
      // The 2nd parameter is the destructured `QueryLifecycleApi`
      async onQueryStarted(
        arg,
        {
          dispatch,
          getState,
          extra,
          requestId,
          queryFulfilled,
          getCacheEntry,
          updateCachedData,
        }
      ) {},
      // The 2nd parameter is the destructured `QueryCacheLifecycleApi`
      async onCacheEntryAdded(
        arg,
        {
          dispatch,
          getState,
          extra,
          requestId,
          cacheEntryRemoved,
          cacheDataLoaded,
          getCacheEntry,
          updateCachedData,
        }
      ) {},
    }),
  }),
})
```

Performing Queries with React Hooks

If you're using React Hooks, RTK Query does a few additional things for you. The primary benefit is that you get a render-optimized hook that allows you to have 'background fetching' as well as derived booleans for convenience.

Hooks are automatically generated based on the name of the `endpoint` in the service definition. An endpoint field with `getPost: builder.query()` will generate a hook named `useGetPostQuery`.

Hook types

There are 5 query-related hooks:

1. `useQuery`
    - Composes `useQuerySubscription` and `useQueryState` and is the primary hook. Automatically triggers fetches of data from an endpoint, 'subscribes' the component to the cached data, and reads the request status and cached data from the Redux store.
2. `useQuerySubscription`
    - Returns a `refetch` function and accepts all hook options. Automatically triggers fetches of data from an endpoint, and 'subscribes' the component to the cached data.
3. `useQueryState`
    - Returns the query state and accepts `skip` and `selectFromResult`. Reads the request status and cached data from the Redux store.
4. `useLazyQuery`
    - Returns a tuple with a `trigger` function, the query result, and last promise info. Similar to `useQuery`, but with manual control over when the data fetching occurs. **Note: the `trigger` function takes a second argument of `preferCacheValue?: boolean` in the event you want to skip making a request if cached data already exists.**
5. `useLazyQuerySubscription`
    - Returns a tuple with a `trigger` function, and last promise info. Similar to `useQuerySubscription`, but with manual control over when the data fetching occurs. **Note: the `trigger` function takes a second argument of `preferCacheValue?: boolean` in the event you want to skip making a request if cached data already exists.**

In practice, the standard `useQuery`-based hooks such as `useGetPostQuery` will be the primary hooks used in your application, but the other hooks are available for specific use cases.

Query Hook Options

The query hooks expect two parameters: `(queryArg?, queryOptions?)`.

The `queryArg` param will be passed through to the underlying `query` callback to generate the URL.

The `queryOptions` object accepts several additional parameters that can be used to control the behavior of the data fetching:

- skip - Allows a query to 'skip' running for that render. Defaults to `false`
- pollingInterval - Allows a query to automatically refetch on a provided interval, specified in milliseconds. Defaults to `0` *(off)*
- selectFromResult - Allows altering the returned value of the hook to obtain a subset of the result, render-optimized for the returned subset.
- refetchOnMountOrArgChange - Allows forcing the query to always refetch on mount (when `true` is provided). Allows forcing the query to refetch if enough time (in seconds) has passed since the last query for the same cache (when a `number` is provided). Defaults to `false`
- refetchOnFocus - Allows forcing the query to refetch when the browser window regains focus. Defaults to `false`
- refetchOnReconnect - Allows forcing the query to refetch when regaining a network connection. Defaults to `false`

All `refetch`-related options will override the defaults you may have set in createApi

Frequently Used Query Hook Return Values

The query hook returns an object containing properties such as the latest `data` for the query request, as well as status booleans for the current request lifecycle state. Below are some of the most frequently used properties. Refer to `useQuery` for an extensive list of all returned properties.

- `data` - The latest returned result regardless of hook arg, if present.
- `currentData` - The latest returned result for the current hook arg, if present.
- `error` - The error result if present.
- `isUninitialized` - When true, indicates that the query has not started yet.
- `isLoading` - When true, indicates that the query is currently loading for the first time, and has no data yet. This will be `true` for the first request fired off, but *not* for subsequent requests.

- `isFetching` - When true, indicates that the query is currently fetching, but might have data from an earlier request. This will be `true` for both the first request fired off, as well as subsequent requests.
- `isSuccess` - When true, indicates that the query has data from a successful request.
- `isError` - When true, indicates that the query is in an `error` state.
- `refetch` - A function to force refetch the query

In most cases, you will probably read `data` and either `isLoading` or `isFetching` in order to render your UI.

Query Hook Usage Example

Here is an example of a `PostDetail` component:

```
export const PostDetail = ({ id }: { id: string }) => {
 const {
   data: post,
   isFetching,
   isLoading,
 } = useGetPostQuery(id, {
   pollingInterval: 3000,
   refetchOnMountOrArgChange: true,
   skip: false,
 })

 if (isLoading) return <div>Loading...</div>
 if (!post) return <div>Missing post!</div>

 return (
   <div>
     {post.name} {isFetching ? '...refetching' : ''}
   </div>
 )
}
```

The way that this component is setup would have some nice traits:

1. It only shows 'Loading...' on the **initial load**
    - **Initial load** is defined as a query that is pending and does not have data in the cache
2. When the request is re-triggered by the polling interval, it will add '...refetching' to the post name
3. If a user closed this `PostDetail`, but then re-opened it within the allowed time, they would immediately be served a cached result and polling would resume with the previous behavior.

Query Loading State

The auto-generated React hooks created by the React-specific version of `createApi` provide derived booleans that reflect the current state of a given query. Derived booleans are preferred for the generated React hooks as opposed to a `status` flag, as the derived booleans are able to provide a greater amount of detail which would not be possible with a single `status` flag, as multiple statuses may be true at a given time (such as `isFetching` and `isSuccess`).

For query endpoints, RTK Query maintains a semantic distinction between `isLoading` and `isFetching` in order to provide more flexibility with the derived information provided.

- `isLoading` refers to a query being in flight for the *first time* for the given hook. No data will be available at this time.
- `isFetching` refers to a query being in flight for the given endpoint + query param combination, but not necessarily for the first time. Data may be available from an earlier request done by this hook, maybe with the previous query param.

This distinction allows for greater control when handling UI behavior. For example, `isLoading` can be used to display a skeleton while loading for the first time, while `isFetching` can be used to grey out old data when changing from page 1 to page 2 or when data is invalidated and re-fetched.

```
import { Skeleton } from './Skeleton'
import { useGetPostsQuery } from './api'

function App() {
  const { data = [], isLoading, isFetching, isError } = useGetPostsQuery()

  if (isError) return <div>An error has occurred!</div>

  if (isLoading) return <Skeleton />

  return (
    <div className={isFetching ? 'posts--disabled' : ''}>
      {data.map((post) => (
        <Post
          key={post.id}
          id={post.id}
          name={post.name}
          disabled={isFetching}
        />
      ))}
    </div>
  )
}
```

While `data` is expected to be used in the majority of situations, `currentData` is also provided, which allows for a further level of granularity. For example, if you wanted to show data in the UI as translucent to represent a re-fetching state, you can use `data` in combination with `isFetching` to achieve this. However, if you also wish to *only* show data corresponding to the current arg, you can instead use `currentData` to achieve this.

In the example below, if posts are being fetched for the first time, a loading skeleton will be shown. If posts for the current user have previously been fetched, and are re-fetching (e.g. as a result of a mutation), the UI will show the previous data, but will grey out the data. If the user changes, it will instead show the skeleton again as opposed to greying out data for the previous user.

```
import { Skeleton } from './Skeleton'
import { useGetPostsByUserQuery } from './api'

function PostsList({ userName }: { userName: string }) {
  const { currentData, isFetching, isError } = useGetPostsByUserQuery(userName)

  if (isError) return <div>An error has occurred!</div>

  if (isFetching && !currentData) return <Skeleton />

  return (
    <div className={isFetching ? 'posts--disabled' : ''}>
      {currentData
        ? currentData.map((post) => (
            <Post
              key={post.id}
              id={post.id}
              name={post.name}
              disabled={isFetching}
            />
          ))
        : 'No data available'}
    </div>
  )
}
```

Query Cache Keys

When you perform a query, RTK Query automatically serializes the request parameters and creates an internal `queryCacheKey` for the request. Any future request that produces the same `queryCacheKey` will be de-duped against the original, and will share updates if a `refetch` is trigged on the query from any subscribed component.

Selecting data from a query result

Sometimes you may have a parent component that is subscribed to a query, and then in a child component you want to pick an item from that query. In most cases you don't want to perform an additional request for a `getItemById`-type query when you know that you already have the result.

`selectFromResult` allows you to get a specific segment from a query result in a performant manner. When using this feature, the component will not rerender unless the underlying data of the selected item has changed. If the selected item is one element in a larger collection, it will disregard changes to elements in the same collection.

```
function PostsList() {
 const { data: posts } = api.useGetPostsQuery()

 return (
  <ul>
   {posts?.data?.map((post) => (
    <PostById key={post.id} id={post.id} />
   ))}
  </ul>
 )
}

function PostById({ id }: { id: number }) {
 // Will select the post with the given id, and will only rerender if the given posts data changes
 const { post } = api.useGetPostsQuery(undefined, {
  selectFromResult: ({ data }) => ({
   post: data?.find((post) => post.id === id),
  }),
 })

 return <li>{post?.name}</li>
}
```

Note that a shallow equality check is performed on the overall return value of `selectFromResult` to determine whether to force a rerender. i.e. it will trigger a rerender if any of the returned object values change reference. If a new array/object is created and used as a return value within the callback, it will hinder the performance benefits due to being identified as a new item each time the callback is run. When intentionally providing an empty array/object, in order to avoid re-creating it each time the callback runs, you can declare an empty array/object outside of the component in order to maintain a stable reference.

```
// An array declared here will maintain a stable reference rather than be re-created again
const emptyArray: Post[] = []

function PostsList() {
  // This call will result in an initial render returning an empty array for `posts`,
  // and a second render when the data is received.
  // It will trigger additional rerenders only if the `posts` data changes
  const { posts } = api.useGetPostsQuery(undefined, {
    selectFromResult: ({ data }) => ({
      posts: data ?? emptyArray,
    }),
  })

  return (
    <ul>
      {posts.map((post) => (
        <PostById key={post.id} id={post.id} />
      ))}
    </ul>
  )
}
```

To summarize the above behaviour - the returned values must be correctly memoized. See also Deriving Data with Selectors and Redux Essentials - RTK Query Advanced Patterns for additional information.

Avoiding unnecessary requests

By default, if you add a component that makes the same query as an existing one, no request will be performed.

In some cases, you may want to skip this behavior and force a refetch - in that case, you can call `refetch` that is returned by the hook.

If you're not using React Hooks, you can access `refetch` like this:

```
const { status, data, error, refetch } = dispatch(
  pokemonApi.endpoints.getPokemon.initiate('bulbasaur')
)
```

*Last updated on **Apr 2, 2023***

Mutations

Overview

Mutations are used to send data updates to the server and apply the changes to the local cache. Mutations can also invalidate cached data and force re-fetches.

Defining Mutation Endpoints

Mutation endpoints are defined by returning an object inside the `endpoints` section of `createApi`, and defining the fields using the `build.mutation()` method.

Mutation endpoints should define either a `query` callback that constructs the URL (including any URL query params), or a `queryFn` callback that may do arbitrary async logic and return a result. The `query` callback may also return an object containing the URL, the HTTP method to use and a request body.

If the `query` callback needs additional data to generate the URL, it should be written to take a single argument. If you need to pass in multiple parameters, pass them formatted as a single "options object".

Mutation endpoints may also modify the response contents before the result is cached, define "tags" to identify cache invalidation, and provide cache entry lifecycle callbacks to run additional logic as cache entries are added and removed.

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query'
import type { Post } from './types'

const api = createApi({
  baseQuery: fetchBaseQuery({
    baseUrl: '/',
  }),
  tagTypes: ['Post'],
  endpoints: (build) => ({
    updatePost: build.mutation<Post, Partial<Post> & Pick<Post, 'id'>>({
      // note: an optional `queryFn` may be used in place of `query`
      query: ({ id, ...patch }) => ({
        url: `post/${id}`,
        method: 'PATCH',
        body: patch,
      }),
      // Pick out data and prevent nested properties in a hook or selector
      transformResponse: (response: { data: Post }, meta, arg) => response.data,
      // Pick out errors and prevent nested properties in a hook or selector
      transformErrorResponse: (
```

```
      response: { status: string | number },
      meta,
      arg
    ) => response.status,
    invalidatesTags: ['Post'],
    // onQueryStarted is useful for optimistic updates
    // The 2nd parameter is the destructured `MutationLifecycleApi`
    async onQueryStarted(
      arg,
      { dispatch, getState, queryFulfilled, requestId, extra, getCacheEntry }
    ) {},
    // The 2nd parameter is the destructured `MutationCacheLifecycleApi`
    async onCacheEntryAdded(
      arg,
      {
        dispatch,
        getState,
        extra,
        requestId,
        cacheEntryRemoved,
        cacheDataLoaded,
        getCacheEntry,
      }
    ) {},
  }),
 }),
})
```

The `onQueryStarted` method can be used for optimistic updates

Performing Mutations with React Hooks

Mutation Hook Behavior

Unlike `useQuery`, `useMutation` returns a tuple. The first item in the tuple is the "trigger" function and the second element contains an object with `status`, `error`, and `data`.

Unlike the `useQuery` hook, the `useMutation` hook doesn't execute automatically. To run a mutation you have to call the trigger function returned as the first tuple value from the hook.

See `useMutation` for the hook signature and additional details.

Frequently Used Mutation Hook Return Values

The `useMutation` hook returns a tuple containing a "mutation trigger" function, as well as an object containing properties about the "mutation result".

The "mutation trigger" is a function that when called, will fire off the mutation request for that endpoint. Calling the "mutation trigger" returns a promise with an `unwrap` property, which can be called to unwrap the mutation call and provide the raw response/error. This can be useful if you wish to determine whether the mutation succeeds/fails inline at the call-site.

The "mutation result" is an object containing properties such as the latest `data` for the mutation request, as well as status booleans for the current request lifecycle state.

Below are some of the most frequently used properties on the "mutation result" object. Refer to `useMutation` for an extensive list of all returned properties.

- `data` - The data returned from the latest trigger response, if present. If subsequent triggers from the same hook instance are called, this will return undefined until the new data is received. Consider component level caching if the previous response data is required for a smooth transition to new data.
- `error` - The error result if present.
- `isUninitialized` - When true, indicates that the mutation has not been fired yet.
- `isLoading` - When true, indicates that the mutation has been fired and is awaiting a response.
- `isSuccess` - When true, indicates that the last mutation fired has data from a successful request.
- `isError` - When true, indicates that the last mutation fired resulted in an error state.
- `reset` - A method to reset the hook back to it's original state and remove the current result from the cache

NOTE

With RTK Query, a mutation does not contain a semantic distinction between 'loading' and 'fetching' in the way that a query does. For a mutation, subsequent calls are not assumed to be necessarily related, so a mutation is either 'loading' or 'not loading', with no concept of 're-fetching'.

Shared Mutation Results

By default, separate instances of a `useMutation` hook are not inherently related to each other. Triggering one instance will not affect the result for a separate instance. This applies regardless of whether the hooks are called within the same component, or different components.

```
export const ComponentOne = () => {
  // Triggering `updatePostOne` will affect the result in this component,
  // but not the result in `ComponentTwo`, and vice-versa
  const [updatePost, result] = useUpdatePostMutation()

  return <div>...</div>
}

export const ComponentTwo = () => {
  const [updatePost, result] = useUpdatePostMutation()

  return <div>...</div>
}
```

RTK Query provides an option to share results across mutation hook instances using the `fixedCacheKey` option. Any `useMutation` hooks with the same `fixedCacheKey` string will share results between each other when any of the trigger functions are called. This should be a unique string shared between each mutation hook instance you wish to share results.

```
export const ComponentOne = () => {
  // Triggering `updatePostOne` will affect the result in both this component,
  // but as well as the result in `ComponentTwo`, and vice-versa
  const [updatePost, result] = useUpdatePostMutation({
    fixedCacheKey: 'shared-update-post',
  })

  return <div>...</div>
}

export const ComponentTwo = () => {
  const [updatePost, result] = useUpdatePostMutation({
    fixedCacheKey: 'shared-update-post',
  })

  return <div>...</div>
}
```

When using `fixedCacheKey`, the `originalArgs` property is not able to be shared and will always be `undefined`.

Standard Mutation Example

This is a modified version of the complete example you can see at the bottom of the page to highlight the `updatePost` mutation. In this scenario, a post is fetched with `useQuery`, and then an `EditablePostName` component is rendered that allows us to edit the name of the post.

```
export const PostDetail = () => {
  const { id } = useParams<{ id: any }>()

  const { data: post } = useGetPostQuery(id)

  const [
    updatePost, // This is the mutation trigger
    { isLoading: isUpdating }, // This is the destructured mutation result
  ] = useUpdatePostMutation()

  return (
    <Box p={4}>
      <EditablePostName
        name={post.name}
        onUpdate={(name) => {
          // If you want to immediately access the result of a mutation, you need to chain `.unwrap()`
          // if you actually want the payload or to catch the error.
          // Example: `updatePost().unwrap().then(fulfilled => console.log(fulfilled)).catch(rejected =>
console.error(rejected))

          return (
            // Execute the trigger with the `id` and updated `name`
            updatePost({ id, name })
          )
        }}
        isLoading={isUpdating}
      />
    </Box>
  )
}
```

Advanced Mutations with Revalidation

In the real world, it's very common that a developer would want to resync their local data cache with the server after performing a mutation (aka "revalidation"). RTK Query takes a more centralized approach to this and requires you to configure the invalidation behavior in your API service definition. See Advanced Invalidation with abstract tag IDs for details on advanced invalidation handling with RTK Query.

Revalidation Example

This is an example of a CRUD service for Posts. This implements the Selectively invalidating lists strategy and will most likely serve as a good foundation for real applications.

```
// Or from '@reduxjs/toolkit/query' if not using the auto-generated hooks
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'

export interface Post {
  id: number
  name: string
}

type PostsResponse = Post[]

export const postApi = createApi({
  reducerPath: 'postsApi',
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  tagTypes: ['Posts'],
  endpoints: (build) => ({
    getPosts: build.query<PostsResponse, void>({
      query: () => 'posts',
      // Provides a list of `Posts` by `id`.
      // If any mutation is executed that `invalidate`s any of these tags, this query will re-run to be always up-to-date.
      // The `LIST` id is a "virtual id" we just made up to be able to invalidate this query specifically if a new `Posts`
element was added.
      providesTags: (result) =>
        // is result available?
        result
          ? // successful query
            [
              ...result.map(({ id }) => ({ type: 'Posts', id } as const)),
              { type: 'Posts', id: 'LIST' },
            ]
          : // an error occurred, but we still want to refetch this query when `{ type: 'Posts', id: 'LIST' }` is invalidated
            [{ type: 'Posts', id: 'LIST' }],
    }),
    addPost: build.mutation<Post, Partial<Post>>({
      query(body) {
        return {
          url: `post`,
          method: 'POST',
          body,
        }
      },
      // Invalidates all Post-type queries providing the `LIST` id - after all, depending of the sort order,
      // that newly created post could show up in any lists.
      invalidatesTags: [{ type: 'Posts', id: 'LIST' }],
    }),
    getPost: build.query<Post, number>({
      query: (id) => `post/${id}`,
      providesTags: (result, error, id) => [{ type: 'Posts', id }],
    }),
    updatePost: build.mutation<Post, Partial<Post>>({
      query(data) {
        const { id, ...body } = data
        return {
          url: `post/${id}`,
          method: 'PUT',
          body,
        }
      },
      // Invalidates all queries that subscribe to this Post `id` only.
      // In this case, `getPost` will be re-run. `getPosts` *might*  rerun, if this id was under its results.
```

```
    invalidatesTags: (result, error, { id }) => [{ type: 'Posts', id }],
   }),

   deletePost: build.mutation<{ success: boolean; id: number }, number>({
    query(id) {
     return {
      url: `post/${id}`,
      method: 'DELETE',
     }
    },
    // Invalidates all queries that subscribe to this Post `id` only.
    invalidatesTags: (result, error, id) => [{ type: 'Posts', id }],
   }),
  }),
})

export const {
  useGetPostsQuery,
  useAddPostMutation,
  useGetPostQuery,
  useUpdatePostMutation,
  useDeletePostMutation,
} = postApi
```

*Last updated on **Apr 2, 2023***

Cache Behavior

A key feature of RTK Query is its management of cached data. When data is fetched from the server, RTK Query will store the data in the Redux store as a 'cache'. When an additional request is performed for the same data, RTK Query will provide the existing cached data rather than sending an additional request to the server.

RTK Query provides a number of concepts and tools to manipulate the cache behaviour and adjust it to your needs.

Default Cache Behavior

With RTK Query, caching is based on:

- API endpoint definitions
- The serialized query parameters used when components subscribe to data from an endpoint
- Active subscription reference counts

When a subscription is started, the parameters used with the endpoint are serialized and stored internally as a `queryCacheKey` for the request. Any future request that produces the same `queryCacheKey` (i.e. called with the same parameters, factoring serialization) will be de-duped against the original, and will share the same data and updates. i.e. two separate components performing the same request will use the same cached data.

When a request is attempted, if the data already exists in the cache, then that data is served and no new request is sent to the server. Otherwise, if the data does not exist in the cache, then a new request is sent, and the returned response is stored in the cache.

Subscriptions are reference-counted. Additional subscriptions that ask for the same endpoint+params increment the reference count. As long as there is an active 'subscription' to the data (e.g. if a component is mounted that calls a `useQuery` hook for the endpoint), then the data will remain in the cache. Once the subscription is removed (e.g. when last component subscribed to the data unmounts), after an amount of time (default 60 seconds), the data will be removed from the cache. The expiration time can be configured with the `keepUnusedDataFor` property for the API definition as a whole, as well as on a per-endpoint basis.

Cache lifetime & subscription example

Imagine an endpoint that expects an `id` as the query param, and 4 components mounted which are requesting data from this same endpoint:

```
import { useGetUserQuery } from './api.ts'

function ComponentOne() {
  // component subscribes to the data
  const { data } = useGetUserQuery(1)

  return <div>...</div>
}

function ComponentTwo() {
  // component subscribes to the data
  const { data } = useGetUserQuery(2)

  return <div>...</div>
}

function ComponentThree() {
  // component subscribes to the data
  const { data } = useGetUserQuery(3)

  return <div>...</div>
```

```
}

function ComponentFour() {
  // component subscribes to the *same* data as ComponentThree,
  // as it has the same query parameters
  const { data } = useGetUserQuery(3)

  return <div>...</div>
}
```

While four components are subscribed to the endpoint, there are only three distinct combinations of endpoint + query parameters. Query parameters `1` and `2` will each have a single subscriber, while query parameter `3` has two subscribers. RTK Query will make three distinct fetches; one for each unique set of query parameters per endpoint.

Data is kept in the cache as long as at least one active subscriber is interested in that endpoint + parameter combination. When the subscriber reference count reaches zero, a timer is set, and if there are no new subscriptions to that data by the time the timer expires, the cached data will be removed. The default expiration is 60 seconds, which can be configured both for the API definition as a whole, as well as on a per-endpoint basis.

If 'ComponentThree' is unmounted in the example above, regardless of how much time passes, the data will remain in the cache due to 'ComponentFour' still being subscribed to the same data, and the subscribe reference count will be `1`. However, once 'ComponentFour' unmounts, the subscriber reference count will be `0`. The data will remain in the cache for the remainder of the expiration time. If no new subscription has been created before the timer expires, the cached data will finally be removed.

Manipulating Cache Behavior

On top of the default behaviour, RTK Query provides a number of methods to re-fetch data earlier in scenarios where it should be considered invalid, or is otherwise deemed suitable to be 'refreshed'.

Reducing subscription time with `keepUnusedDataFor`

As mentioned above under Default Cache Behavior and Cache lifetime & subscription example, by default, data will remain in the cache for 60 seconds after the subscriber reference count hits zero.

This value can be configured using the `keepUnusedDataFor` option for both the API definition, as well as per-endpoint. Note that the per-endpoint version, if provided, will overrule a setting on the API definition.

Providing a value to `keepUnusedDataFor` as a number in seconds specifies how long the data should be kept in the cache after the subscriber reference count reaches zero.

keepUnusedDataFor configuration

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
import type { Post } from './types'

export const api = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  // global configuration for the api
  keepUnusedDataFor: 30,
  endpoints: (builder) => ({
    getPosts: builder.query<Post[], number>({
      query: () => `posts`,
      // configuration for an individual endpoint, overriding the api setting
      keepUnusedDataFor: 5,
    }),
  }),
})
```

Re-fetching on demand with `refetch`/`initiate`

In order to achieve complete granular control over re-fetching data, you can use the `refetch` function returned as a result property from a `useQuery` or `useQuerySubscription` hook.

Calling the `refetch` function will force refetch the associated query.

Alternatively, you can dispatch the `initiate` thunk action for an endpoint, passing the option `forceRefetch: true` to the thunk action creator for the same effect.

Force refetch example

```
import { useDispatch } from 'react-redux'
import { useGetPostsQuery } from './api'

const Component = () => {
  const dispatch = useDispatch()
  const { data, refetch } = useGetPostsQuery({ count: 5 })

  function handleRefetchOne() {
    // force re-fetches the data
    refetch()
  }

  function handleRefetchTwo() {
    // has the same effect as `refetch` for the associated query
    dispatch(
      api.endpoints.getPosts.initiate(
        { count: 5 },
        { subscribe: false, forceRefetch: true }
      )
    )
  }

  return (
    <div>
      <button onClick={handleRefetchOne}>Force re-fetch 1</button>
      <button onClick={handleRefetchTwo}>Force re-fetch 2</button>
    </div>
  )
}
```

Encouraging re-fetching with `refetchOnMountOrArgChange`

Queries can be encouraged to re-fetch more frequently than usual via the `refetchOnMountOrArgChange` property. This can be passed to the endpoint as a whole, to individual hook calls, or when dispatching the `initiate` action (the name of the action creator's option is `forceRefetch`).

`refetchOnMountOrArgChange` is used to encourage re-fetching in additional situations where the default behavior would instead serve cached data.

`refetchOnMountOrArgChange` accepts either a boolean value, or a number as time in seconds.

Passing `false` (the default value) for this property will use the default behavior described above.

Passing `true` for this property will cause the endpoint to always refetch when a new subscriber to the query is added. If passed to an individual hook call and not the api definition itself, then this applies only to that hook call. I.e., when the component calling the hook mounts, or the argument changes, it will always refetch, regardless of whether cached data for the endpoint + arg combination already exists.

Passing a `number` as a value in seconds will use the following behavior:

- At the time a query subscription is created:
  - if there is an existing query in the cache, it will compare the current time vs the last fulfilled timestamp for that query,
  - It will refetch if the provided amount of time in seconds has elapsed.
- If there is no query, it will fetch the data.
- If there is an existing query, but the amount of time specified since the last query has not elapsed, it will serve the existing cached data.

Configuring re-fetching on subscription if data exceeds a given time

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
import type { Post } from './types'

export const api = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  // global configuration for the api
  refetchOnMountOrArgChange: 30,
  endpoints: (builder) => ({
    getPosts: builder.query<Post[], number>({
      query: () => `posts`,
    }),
  }),
})
```

Forcing refetch on component mount

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

const Component = () => {
  const { data } = useGetPostsQuery(
    { count: 5 },
    // this overrules the api definition setting,
    // forcing the query to always fetch when this component is mounted
    { refetchOnMountOrArgChange: true }
  )

  return <div>...</div>
}
```

Re-fetching on window focus with `refetchOnFocus`

The `refetchOnFocus` option allows you to control whether RTK Query will try to refetch all subscribed queries after the application window regains focus.

If you specify this option alongside `skip: true`, this will not be evaluated until skip is false.

Note that this requires `setupListeners` to have been called.

This option is available on both the api definition with `createApi`, as well as on the `useQuery`, `useQuerySubscription`, `useLazyQuery`, and `useLazyQuerySubscription` hooks.

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
import type { Post } from './types'

export const api = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  // global configuration for the api
  refetchOnFocus: true,
  endpoints: (builder) => ({
    getPosts: builder.query<Post[], number>({
      query: () => `posts`,
    }),
  }),
})
```

```
import { configureStore } from '@reduxjs/toolkit'
import { setupListeners } from '@reduxjs/toolkit/query'
import { api } from './services/api'

export const store = configureStore({
  reducer: {
    [api.reducerPath]: api.reducer,
  },
  middleware: (gDM) => gDM().concat(api.middleware),
})

// enable listener behavior for the store
setupListeners(store.dispatch)

export type RootState = ReturnType<typeof store.getState>
```

Re-fetching on network reconnection with `refetchOnReconnect`

The `refetchOnReconnect` option on `createApi` allows you to control whether RTK Query will try to refetch all subscribed queries after regaining a network connection.

If you specify this option alongside `skip: true`, this **will not be evaluated** until `skip` is false.

Note that this requires `setupListeners` to have been called.

This option is available on both the api definition with `createApi`, as well as on the `useQuery`, `useQuerySubscription`, `useLazyQuery`, and `useLazyQuerySubscription` hooks.

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
import type { Post } from './types'

export const api = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  // global configuration for the api
  refetchOnReconnect: true,
  endpoints: (builder) => ({
    getPosts: builder.query<Post[], number>({
      query: () => `posts`,
    }),
  }),
})
```

```
import { configureStore } from '@reduxjs/toolkit'
import { setupListeners } from '@reduxjs/toolkit/query'
import { api } from './services/api'

export const store = configureStore({
  reducer: {
    [api.reducerPath]: api.reducer,
  },
  middleware: (gDM) => gDM().concat(api.middleware),
})

// enable listener behavior for the store
setupListeners(store.dispatch)

export type RootState = ReturnType<typeof store.getState>
```

Re-fetching after mutations by invalidating cache tags

RTK Query uses an optional cache tag system to automate re-fetching for query endpoints that have data affected by mutation endpoints.

See Automated Re-fetching for full details on this concept.

Tradeoffs

No Normalized or De-duplicated Cache

RTK Query deliberately **does *not* implement a cache that would deduplicate identical items across multiple requests**. There are several reasons for this:

- A fully normalized shared-across-queries cache is a *hard* problem to solve
- We don't have the time, resources, or interest in trying to solve that right now

- In many cases, simply re-fetching data when it's invalidated works well and is easier to understand
- At a minimum, RTKQ can help solve the general use case of "fetch some data", which is a big pain point for a lot of people

As an example, say that we have an API slice with `getTodos` and `getTodo` endpoints, and our components make the following queries:

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

Each of these query results would include a Todo object that looks like `{id: 1}`.

In a fully normalized de-duplicating cache, only a single copy of this Todo object would be stored. However, RTK Query saves each query result independently in the cache. So, this would result in three separate copies of this Todo being cached in the Redux store. However, if all the endpoints are consistently providing the same tags (such as `{type: 'Todo', id: 1}`), then invalidating that tag will force all the matching endpoints to refetch their data for consistency.

The Redux docs have always recommended keeping data in a normalized lookup table to enable easily finding items by ID and updating them in the store, and RTK's `createEntityAdapter` was designed to help manage normalized state. Those concepts are still valuable and don't go away. However, if you're using RTK Query to manage caching data, there's less need to manipulate the data that way yourself.

There are a couple additional points that can help here:

- The generated query hooks have a `selectFromResult` option that allow components to read individual pieces of data from a query result. As an example, a `<TodoList>` component might call `useTodosQuery()`, and each individual `<TodoListItem>` could use the same query hook but select from the result to get the right todo object.
- You can use the `transformResponse` endpoint option to modify the fetched data so that it's stored in a different shape, such as using `createEntityAdapter` to normalize the data *for this one response* before it's inserted into the cache.

Further information

- Reddit: discussion of why RTKQ doesn't have a normalized cache, and tradeoffs

Automated Re-fetching

As seen under Default Cache Behavior, when a subscription is added for a query endpoint, a request will be sent only if the cache data does not already exist. If it exists, the existing data will be served instead.

RTK Query uses a "cache tag" system to automate re-fetching for query endpoints that have data affected by mutation endpoints. This enables designing your API such that firing a specific mutation will cause a certain query endpoint to consider its cached data *invalid*, and re-fetch the data if there is an active subscription.

Each endpoint + parameter combination contributes its own `queryCacheKey`. The cache tag system enables the ability to inform RTK Query that a particular query cache has *provided* specific tags. If a mutation is fired which is said to `invalidate` tags that a query cache has *provided*, the cached data will be considered *invalidated*, and re-fetch if there is an active subscription to the cached data.

For triggering re-fetching through other means, see Manipulating Cache Behavior.

Definitions

Tags

see also: tagTypes API reference

For RTK Query, *tags* are just a name that you can give to a specific collection of data to control caching and invalidation behavior for re-fetching purposes. It can be considered as a 'label' attached to cached data that is read after a mutation, to decide whether the data should be affected by the mutation.

Tags are defined in the `tagTypes` argument when defining an api. For example, in an application that has both `Posts` and `Users`, you might define `tagTypes: ['Post', 'User']` when calling `createApi`.

An individual `tag` has a `type`, represented as a `string` name, and an optional `id`, represented as a `string` or `number`. It can be represented as a plain string (such as `'Post'`), or an object in the shape `{type: string, id?: string|number}` (such as `[{type: 'Post', id: 1}]`).

Providing tags

see also: providesTags API reference

A *query* can have its cached data *provide* tags. Doing so determines which 'tag' is attached to the cached data returned by the query.

The `providesTags` argument can either be an array of `string` (such as `['Post']`), `{type: string, id?: string|number}` (such as `[{type: 'Post', id: 1}]`), or a callback that returns such an array. That function will be passed the result as the first argument, the response error as the second argument, and the argument originally passed into the `query` method as the third argument. Note that either the result or error arguments may be undefined based on whether the query was successful or not.

Invalidating tags

see also: invalidatesTags API reference

A *mutation* can *invalidate* specific cached data based on the tags. Doing so determines which cached data will be either refetched or removed from the cache.

The `invalidatesTags` argument can either be an array of `string` (such as `['Post']`), `{type: string, id?: string|number}` (such as `[{type: 'Post', id: 1}]`), or a callback that returns such an array. That function will be passed the result as the first argument, the response error as the second argument, and the argument originally passed into the `query` method as the third argument. Note that either the result or error arguments may be undefined based on whether the mutation was successful or not.

Cache tags

RTK Query uses the concept of 'tags' to determine whether a mutation for one endpoint intends to *invalidate* some data that was *provided* by a query from another endpoint.

If cache data is being invalidated, it will either refetch the providing query (if components are still using that data) or remove the data from the cache.

When defining an API slice, `createApi` accepts an array of tag type names for the `tagTypes` property, which is a list of possible tag name options that the queries for the API slice could provide.

The example below declares that endpoints can possibly provide 'Posts' and/or 'Users' to the cache:

Example of declaring cache tags

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query'
import type { Post, User } from './types'

const api = createApi({
  baseQuery: fetchBaseQuery({
    baseUrl: '/',
  }),
  tagTypes: ['Post', 'User'],
  endpoints: (build) => ({
    getPosts: build.query<Post[], void>({
      query: () => '/posts',
    }),
    getUsers: build.query<User[], void>({
      query: () => '/users',
    }),
    addPost: build.mutation<Post, Omit<Post, 'id'>>({
      query: (body) => ({
        url: 'post',
        method: 'POST',
        body,
      }),
    }),
    editPost: build.mutation<Post, Partial<Post> & Pick<Post, 'id'>>({
      query: (body) => ({
        url: `post/${body.id}`,
        method: 'POST',
        body,
      }),
    }),
  }),
})
```

By declaring these tags as what can possibly be provided to the cache, it enables control for individual mutation endpoints to claim whether they affect specific portions of the cache or not, in conjunction with `providesTags` and `invalidatesTags` on individual endpoints.

Providing cache data

Each individual `query` endpoint can have its cached data *provide* particular tags. Doing so enables a relationship between cached data from one or more query endpoints and the behaviour of one or more mutation endpoints.

The `providesTags` property on a `query` endpoint is used for this purpose.

INFO

Provided tags have no inherent relationship across separate `query` endpoints. Provided tags are used to determine whether cached data returned by an endpoint should be `invalidated` and either be refetched or removed from the cache. If two separate endpoints provide the same tags, they will still contribute their own distinct cached data, which could later both be invalidated by a single tag declared from a mutation.

The example below declares that the `getPosts query` endpoint `provides` the `'Post'` tag to the cache, using the `providesTags` property for a `query` endpoint.

Example of providing tags to the cache

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query'
import type { Post, User } from './types'

const api = createApi({
  baseQuery: fetchBaseQuery({
    baseUrl: '/',
  }),
  tagTypes: ['Post', 'User'],
  endpoints: (build) => ({
    getPosts: build.query<Post[], void>({
      query: () => '/posts',
      providesTags: ['Post'],
    }),
    getUsers: build.query<User[], void>({
      query: () => '/users',
      providesTags: ['User'],
    }),
    addPost: build.mutation<Post, Omit<Post, 'id'>>({
      query: (body) => ({
        url: 'posts',
        method: 'POST',
        body,
      }),
    }),
    editPost: build.mutation<Post, Partial<Post> & Pick<Post, 'id'>>({
      query: (body) => ({
        url: `post/${body.id}`,
        method: 'POST',
        body,
      }),
    }),
  }),
})
```

For more granular control over the provided data, provided `tags` can have an associated `id`. This enables a distinction between 'any of a particular tag type', and 'a specific instance of a particular tag type'.

The example below declares that the provided posts are associated with particular IDs as determined by the result returned by the endpoint:

Example of providing tags with IDs to the cache

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query'
import type { Post, User } from './types'

const api = createApi({
  baseQuery: fetchBaseQuery({
    baseUrl: '/',
  }),
  tagTypes: ['Post', 'User'],
  endpoints: (build) => ({
    getPosts: build.query<Post[], void>({
      query: () => '/posts',
      providesTags: (result, error, arg) =>
        result
          ? [...result.map(({ id }) => ({ type: 'Post' as const, id })),
'Post']
          : ['Post'],
    }),
    getUsers: build.query<User[], void>({
      query: () => '/users',
      providesTags: ['User'],
    }),
    addPost: build.mutation<Post, Omit<Post, 'id'>>({
      query: (body) => ({
        url: 'post',
        method: 'POST',
        body,
      }),
    }),
    editPost: build.mutation<Post, Partial<Post> & Pick<Post, 'id'>>({
      query: (body) => ({
        url: `post/${body.id}`,
        method: 'POST',
        body,
      }),
    }),
  }),
})
```

Note that for the example above, the `id` is used where possible on a successful result. In the case of an error, no result is supplied, and we still consider that it has provided the general `'Post'` tag type rather than any specific instance of that tag.

ADVANCED LIST INVALIDATION

In order to provide stronger control over invalidating the appropriate data, you can use an arbitrary ID such as `'LIST'` for a given tag. See Advanced Invalidation with abstract tag IDs for additional details.

Invalidating cache data

Each individual mutation endpoint can `invalidate` particular tags for existing cached data. Doing so enables a relationship between cached data from one or more query endpoints and the behaviour of one or more mutation endpoints.

The `invalidatesTags` property on a mutation endpoint is used for this purpose.

The example below declares that the `addPost` and `editPost` mutation endpoints `invalidate` any cached data with the `'Post'` tag, using the `invalidatesTags` property for a mutation endpoint:

Example of invalidating tags in the cache

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query'
import type { Post, User } from './types'

const api = createApi({
  baseQuery: fetchBaseQuery({
    baseUrl: '/',
  }),
  tagTypes: ['Post', 'User'],
  endpoints: (build) => ({
    getPosts: build.query<Post[], void>({
      query: () => '/posts',
      providesTags: (result, error, arg) =>
        result
          ? [...result.map(({ id }) => ({ type: 'Post' as const, id })),
'Post']
          : ['Post'],
    }),
    getUsers: build.query<User[], void>({
      query: () => '/users',
      providesTags: ['User'],
    }),
    addPost: build.mutation<Post, Omit<Post, 'id'>>({
      query: (body) => ({
        url: 'post',
        method: 'POST',
        body,
      }),
      invalidatesTags: ['Post'],
    }),
    editPost: build.mutation<Post, Partial<Post> & Pick<Post, 'id'>>({
      query: (body) => ({
        url: `post/${body.id}`,
        method: 'POST',
        body,
      }),
      invalidatesTags: ['Post'],}}}}}
```

For the example above, this tells RTK Query that after the `addPost` and/or `editPost` mutations are called and completed, any cache data supplied with the `'Post'` tag is no longer valid. If a component is currently subscribed to the cached data for a `'Post'` tag after the above mutations are

called and complete, it will automatically re-fetch in order to retrieve up to date data from the server.

An example scenario would be like so:

1. A component is rendered which is using the `useGetPostsQuery()` hook to subscribe to that endpoint's cached data
2. The `/posts` request is fired off, and server responds with posts with IDs 1, 2 & 3
3. The `getPosts` endpoint stores the received data in the cache, and internally registers that the following tags have been provided:

```
[
  { type: 'Post', id: 1 },
  { type: 'Post', id: 2 },
  { type: 'Post', id: 3 },
]
```

4. The `editPost` mutation is fired off to alter a particular post
5. Upon completion, RTK Query internally registers that the `'Post'` tag is now invalidated, and removes the previously provided `'Post'` tags from the cache
6. Since the `getPosts` endpoint has provided tags of type `'Post'` which now has invalid cache data, and the component is still subscribed to the data, the `/posts` request is automatically fired off again, fetching new data and registering new tags for the updated cached data

For more granular control over the invalidated data, invalidated `tags` can have an associated `id` in the same manner as `providesTags`. This enables a distinction between 'any of a particular tag type' and 'a specific instance of a particular tag type'.

The example below declares that the `editPost` mutation invalidates a specific instance of a `Post` tag, using the ID passed in when calling the mutation function:

- TypeScript
- JavaScript

Example of invalidating tags with IDs to the cache

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query'
import type { Post, User } from './types'

const api = createApi({
  baseQuery: fetchBaseQuery({
    baseUrl: '/',
  }),
  tagTypes: ['Post', 'User'],
  endpoints: (build) => ({
    getPosts: build.query<Post[], void>({
      query: () => '/posts',
      providesTags: (result, error, arg) =>
        result
          ? [...result.map(({ id }) => ({ type: 'Post' as const, id })),
'Post']
          : ['Post'],
    }),
    getUsers: build.query<User[], void>({
      query: () => '/users',
      providesTags: ['User'],
    }),
    addPost: build.mutation<Post, Omit<Post, 'id'>>({
      query: (body) => ({
        url: 'post',
        method: 'POST',
        body,
      }),
      invalidatesTags: ['Post'],
    }),
    editPost: build.mutation<Post, Partial<Post> & Pick<Post, 'id'>>({
      query: (body) => ({
        url: `post/${body.id}`,
        method: 'POST',
        body,
      }),
      invalidatesTags: (result, error, arg) => [{ type: 'Post', id: arg.id
}],
    }),
  }),
})
```

For the example above, rather than invalidating any tag with the type `'Post'`, calling the `editPost` mutation function will now only invalidate a tag for the provided `id`. I.e. if cached data from an endpoint does not provide a `'Post'` for that same `id`, it will remain considered as 'valid', and will not be triggered to automatically re-fetch.

USING ABSTRACT TAG IDS

In order to provide stronger control over invalidating the appropriate data, you can use an arbitrary ID such as `'LIST'` for a given tag. See Advanced Invalidation with abstract tag IDs for additional details.

Tag Invalidation Behavior

The matrix below shows examples of which invalidated tags will affect and invalidate which provided tags:

Provided / Invalidated	**General tag A** ['Post'] / [{ type: 'Post' }]	**General tag B** ['User'] / [{ type: 'User' }]	**Specific tag A1** [{ type: 'Post', id: 1 }]	**Specific tag A2** [{ type: 'Post', id: 'LIST' }]	**Specific tag B1** [{ type: 'User', id: 1 }]	**Specific tag B2** [{ type: 'User', id: 2 }]
General tag A ['Post'] / [{ type: 'Post' }]	✔		✔	✔		
General tag B ['User'] / [{ type: 'User' }]		✔			✔	✔
Specific tag A1 [{ type: 'Post', id: 1 }]			✔			
Specific tag A2 [{ type: 'Post', id: 'LIST' }]				✔		
Specific tag B1 [{ type: 'User', id: 1 }]					✔	
Specific tag B2 [{ type: 'User', id: 2 }]						✔

The invalidation behavior is summarized based on tag specificity in the sections below.

General tag

e.g. `['Post'] / [{ type: 'Post' }]`

Will `invalidate` any `provided` tag with the matching type, including general and specific tags.

Example:
If a general tag of `Post` was invalidated, endpoints whose data `provided` the following tags would all have their data invalidated:

- `['Post']`
- `[{ type: 'Post' }]`
- `[{ type: 'Post' }, { type: 'Post', id: 1 }]`
- `[{ type: 'Post', id: 1 }]`
- `[{ type: 'Post', id: 1 }, { type: 'User' }]`
- `[{ type: 'Post', id: 'LIST' }]`
- `[{ type: 'Post', id: 1 }, { type: 'Post', id: 'LIST' }]`

Endpoints whose data `provided` the following tags would *not* have their data invalidated:

- `['User']`
- `[{ type: 'User' }]`
- `[{ type: 'User', id: 1 }]`
- `[{ type: 'User', id: 'LIST' }]`
- `[{ type: 'User', id: 1 }, { type: 'User', id: 'LIST' }]`

Specific tag

e.g. `[{ type: 'Post', id: 1 }]`

Will `invalidate` any `provided` tag with both the matching type, *and* matching id. Will not cause a `general` tag to be invalidated directly, but *might* invalidate data for an endpoint that provides a `general` tag *if* it also provides a matching `specific` tag.

Example 1: If a specific tag of `{ type: 'Post', id: 1 }` was invalidated, endpoints whose data `provided` the following tags would all have their data invalidated:

- `[{ type: 'Post' }, { type: 'Post', id: 1 }]`
- `[{ type: 'Post', id: 1 }]`
- `[{ type: 'Post', id: 1 }, { type: 'User' }]`
- `[{ type: 'Post', id: 1 }, { type: 'Post', id: 'LIST' }]`

Endpoints whose data `provided` the following tags would *not* have their data invalidated:

- `['Post']`
- `[{ type: 'Post' }]`
- `[{ type: 'Post', id: 'LIST' }]`
- `['User']`
- `[{ type: 'User' }]`
- `[{ type: 'User', id: 1 }]`
- `[{ type: 'User', id: 'LIST' }]`
- `[{ type: 'User', id: 1 }, { type: 'User', id: 'LIST' }]`

Example 2: If a specific tag of `{ type: 'Post', id: 'LIST' }` was invalidated, endpoints whose data `provided` the following tags would all have their data invalidated:

- `[{ type: 'Post', id: 'LIST' }]`
- `[{ type: 'Post', id: 1 }, { type: 'Post', id: 'LIST' }]`

Endpoints whose data `provided` the following tags would *not* have their data invalidated:

- `['Post']`
- `[{ type: 'Post' }]`
- `[{ type: 'Post' }, { type: 'Post', id: 1 }]`
- `[{ type: 'Post', id: 1 }]`
- `[{ type: 'Post', id: 1 }, { type: 'User' }]`
- `['User']`
- `[{ type: 'User' }]`
- `[{ type: 'User', id: 1 }]`
- `[{ type: 'User', id: 'LIST' }]`
- `[{ type: 'User', id: 1 }, { type: 'User', id: 'LIST' }]`

Recipes

Advanced Invalidation with abstract tag IDs

While using an 'entity ID' for a tag `id` is a common use case, the `id` property is not intended to be limited to database IDs alone. The `id` is simply a way to label a subset of a particular collection of data for a particular `tag type`.

A powerful use-case is to use an ID like `'LIST'` as a label for data provided by a bulk query, *as well as* using entity IDs for the individual items. Doing so allows future `mutations` to declare whether they invalidate the data only if it contains a particular item (e.g. `{ type: 'Post', id: 5 }`), or invalidate the data if it is a `'LIST'` (e.g. `{ type: 'Post', id: 'LIST' }`).

'LIST' TAG AND IDS

1. `LIST` is an arbitrary string - technically speaking, you could use anything you want here, such as `ALL` or `*`. The important thing when choosing a custom id is to make sure there is no possibility of it colliding with an id that is returned by a query result. If you have unknown ids in your query results and don't want to risk it, you can go with point 3 below.
2. You can add *many* tag types for even more control
    - `[{ type: 'Posts', id: 'LIST' }, { type: 'Posts', id: 'SVELTE_POSTS' }, { type: 'Posts', id: 'REACT_POSTS' }]`

3. If the concept of using an `id` like 'LIST' seems strange to you, you can always add another `tagType` and invalidate its root, but we recommend using the `id` approach as shown.

We can compare the scenarios below to see how using a `'LIST'` id can be leveraged to optimize behaviour.

Invalidating everything of a type

API Definition

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
import type { Post, User } from './types'

export const api = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  tagTypes: ['Posts'],
  endpoints: (build) => ({
    getPosts: build.query<Post[], void>({
      query: () => 'posts',
      providesTags: (result) =>
        result ? result.map(({ id }) => ({ type: 'Posts', id })) : ['Posts'],
    }),
    addPost: build.mutation<Post, Partial<Post>>({
      query: (body) => ({
        url: `post`,
        method: 'POST',
        body,
      }),
      invalidatesTags: ['Posts'],
    }),
    getPost: build.query<Post, number>({
      query: (id) => `post/${id}`,
      providesTags: (result, error, id) => [{ type: 'Posts', id }],
    }),
  }),
})

export const { useGetPostsQuery, useGetPostQuery, useAddPostMutation } = api
```

App.tsx

```
function App() {
  const { data: posts } = useGetPostsQuery()
  const [addPost] = useAddPostMutation()

  return (
    <div>
      <AddPost onAdd={addPost} />
      <PostsList />
      {/* Assume each PostDetail is subscribed via `const {data} =
useGetPostQuery(id)` */}
      <PostDetail id={1} />
      <PostDetail id={2} />
      <PostDetail id={3} />
    </div>
  )}
```

What to expect

When `addPost` is triggered, it would cause each `PostDetail` component to go back into a `isFetching` state because `addPost` invalidates the root tag, which causes *every query* that provides 'Posts' to be re-run. In most cases, this may not be what you want to do. Imagine if you had 100 posts on the screen that all subscribed to a `getPost` query – in this case, you'd create 100 requests and send a ton of unnecessary traffic to your server, which we're trying to avoid in the first place! Even though the user would still see the last good cached result and potentially not notice anything other than their browser hiccuping, you still want to avoid this.

Selectively invalidating lists

API Definition

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
import type { Post, User } from './types'

export const api = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  tagTypes: ['Posts'],
  endpoints: (build) => ({
    getPosts: build.query<Post[], void>({
      query: () => 'posts',
      providesTags: (result) =>
        result
          ? [
              ...result.map(({ id }) => ({ type: 'Posts' as const, id })),
              { type: 'Posts', id: 'LIST' },
            ]
          : [{ type: 'Posts', id: 'LIST' }],
    }),
    addPost: build.mutation<Post, Partial<Post>>({
      query(body) {
        return {
          url: `post`,
          method: 'POST',
          body,
        }
      },
      invalidatesTags: [{ type: 'Posts', id: 'LIST' }],
    }),
    getPost: build.query<Post, number>({
      query: (id) => `post/${id}`,
      providesTags: (result, error, id) => [{ type: 'Posts', id }],
    }),
  }),
})

export const { useGetPostsQuery, useAddPostMutation, useGetPostQuery } = api
```

App.tsx

```
function App() {
  const { data: posts } = useGetPostsQuery()
  const [addPost] = useAddPostMutation()
```

```
  return (
    <div>
      <AddPost onAdd={addPost} />
      <PostsList />
      {/* Assume each PostDetail is subscribed via `const {data} =
useGetPostQuery(id)` */}
      <PostDetail id={1} />
      <PostDetail id={2} />
      <PostDetail id={3} />
    </div>
  )
}
```

What to expect

When `addPost` is fired, it will only cause the `PostsList` to go into an `isFetching` state because `addPost` only invalidates the `'LIST'` id, which causes `getPosts` to rerun (because it provides that specific id). So in your network tab, you would only see 1 new request fire for `GET /posts`. As the singular `getPost` queries have not been invalidated, they will not re-run as a result of `addPost`.

INFO

If you intend for the `addPost` mutation to refresh all posts including individual `PostDetail` components while still only making 1 new `GET /posts` request, this can be done by selecting a part of the data using `selectFromResult`.

Providing errors to the cache

The information provided to the cache is not limited to successful data fetches. The concept can be used to inform RTK Query that when a particular failure has been encountered, to `provide` a specific `tag` for that failed cache data. A separate endpoint can then `invalidate` the data for that `tag`, telling RTK Query to re-attempt the previously failed endpoints if a component is still subscribed to the failed data.

The example below demonstrates an example with the following behaviour:

- Provides an `UNAUTHORIZED` cache tag if a query fails with an error code of `401 UNAUTHORIZED`
- Provides an `UNKNOWN_ERROR` cache tag if a query fails with a different error
- Enables a 'login' mutation, which when *successful*, will `invalidate` the data with the `UNAUTHORIZED` tag.
  This will trigger the `postById` endpoint to re-fire if:

    i. The last call for `postById` had encountered an unauthorized error, and
    ii. A component is still subscribed to the cached data
- Enables a 'refetchErroredQueries' mutation which when *called*, will `invalidate` the data with the `UNKNOWN_ERROR` tag.
  This will trigger the `postById` endpoint to re-fire if:
    i. The last call for `postById` had encountered an unknown error, and
    ii. A component is still subscribed to the cached data

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query'
import { Post, LoginResponse } from './types'

const api = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: 'https://example.com' }),
  tagTypes: ['Post', 'UNAUTHORIZED', 'UNKNOWN_ERROR'],
  endpoints: (build) => ({
    postById: build.query<Post, number>({
      query: (id) => `post/${id}`,
      providesTags: (result, error, id) =>
        result
          ? [{ type: 'Post', id }]
          : error?.status === 401
          ? ['UNAUTHORIZED']
          : ['UNKNOWN_ERROR'],
    }),
    login: build.mutation<LoginResponse, void>({
      query: () => '/login',
      // on successful login, will refetch all currently
      // 'UNAUTHORIZED' queries
      invalidatesTags: (result) => (result ? ['UNAUTHORIZED'] : []),
    }),
    refetchErroredQueries: build.mutation<null, void>({
      queryFn: () => ({ data: null }),
      invalidatesTags: ['UNKNOWN_ERROR'],
    }),
  }),
})
```

Abstracting common provides/invalidates usage

The code written to `provide` & `invalidate` tags for a given API slice will be dependent on multiple factors, including:

- The shape of the data returned by your backend
- Which tags you expect a given query endpoint to provide
- Which tags you expect a given mutation endpoint to invalidate
- The extent that you wish to use the invalidation feature for

When declaring your API slice, you may feel as though you're duplicating your code. For instance, for two separate endpoints that both provide a list of a particular entity, the `providesTags` declaration may only differ in the `tagType` provided.

e.g.

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query'
import type { Post, User } from './types'

const api = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: 'https://example.com' }),
  tagTypes: ['Post', 'User'],
  endpoints: (build) => ({
    getPosts: build.query<Post[], void>({
      query: () => `posts`,
      providesTags: (result) =>
        result
          ? [
              { type: 'Post', id: 'LIST' },
              ...result.map(({ id }) => ({ type: 'Post' as const, id })),
            ]
          : [{ type: 'Post', id: 'LIST' }],
    }),
    getUsers: build.query<User[], void>({
      query: () => `users`,
      providesTags: (result) =>
        result
          ? [
              { type: 'User', id: 'LIST' },
              ...result.map(({ id }) => ({ type: 'User' as const, id })),
            ]
          : [{ type: 'User', id: 'LIST' }],
    }),
  }),
})
```

You may find it beneficial to define helper functions designed for your particular api to reduce this boilerplate across endpoint definitions, e.g.

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query'
import type { Post, User } from './types'

function providesList<R extends { id: string | number }[], T extends string>(
  resultsWithIds: R | undefined,
  tagType: T
) {
  return resultsWithIds
    ? [
        { type: tagType, id: 'LIST' },
        ...resultsWithIds.map(({ id }) => ({ type: tagType, id })),
      ]
    : [{ type: tagType, id: 'LIST' }]
}

const api = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: 'https://example.com' }),
  tagTypes: ['Post', 'User'],
  endpoints: (build) => ({
    getPosts: build.query({
```

```
      query: () => `posts`,
      providesTags: (result) => providesList(result, 'Post'),
    }),
    getUsers: build.query({
      query: () => `users`,
      providesTags: (result) => providesList(result, 'User'),
    }),
  }),
})
```

An example of various abstractions for tag providing/invalidating designed for common rest data formats can be seen in the following gist, including typescript support, and factoring both 'LIST' style advanced tag invalidation and 'error' style tag invalidation: **RTK Query cache utils**.

*Last updated on **Apr 18, 202***

Manual Cache Updates

Overview

For most cases, in order to receive up to date data after a triggering a change in the backend, you can take advantage of `cache tag invalidation` to perform automated re-fetching, which will cause a query to re-fetch its data when it has been told that a mutation has occurred which would cause its data to become out of date. In most cases, we recommend using `automated re-fetching` as a preference over `manual cache updates`, unless you encounter the need to do so.

However, in some cases, you may want to update the cache manually. When you wish to update cache data that *already exists* for query endpoints, you can do so using the `updateQueryData` thunk action available on the `util` object of your created API.

Anywhere you have access to the `dispatch` method for the store instance, you can dispatch the result of calling `updateQueryData` in order to update the cache data for a query endpoint, if the corresponding cache entry exists.

Use cases for manual cache updates include:

- Providing immediate feedback to the user when a mutation is attempted
- After a mutation, updating a single item in a large list of items that is already cached, rather than re-fetching the whole list
- Debouncing a large number of mutations with immediate feedback as though they are being applied, followed by a single request sent to the server to update the debounced attempts

NOTE

`updateQueryData` is strictly intended to perform *updates* to existing cache entries, not create new entries. If an `updateQueryData` thunk action is dispatched that corresponds to no existing cache entry for the provided `endpointName` + `args` combination, the provided `recipe` will not be called, and no `patches` or `inversePatches` will be returned.

Recipes

Optimistic Updates

When you wish to perform an update to cache data immediately after a `mutation` is triggered, you can apply an `optimistic update`. This can be a useful pattern for when you want to give the user the impression that their changes are immediate, even while the mutation request is still in flight.

The core concepts for an optimistic update are:

- when you start a query or mutation, `onQueryStarted` will be executed
- you manually update the cached data by dispatching `api.util.updateQueryData` within `onQueryStarted`
- then, in the case that `queryFulfilled` rejects:
  - you roll it back via the `.undo` property of the object you got back from the earlier dispatch,
    OR
  - you invalidate the cache data via `api.util.invalidateTags` to trigger a full re-fetch of the data

TIP

Where many mutations are potentially triggered in short succession causing overlapping requests, you may encounter race conditions if attempting to roll back patches using the `.undo` property on failures. For these scenarios, it is often simplest and safest to invalidate the tags on error instead, and re-fetch truly up-to-date data from the server.

Optimistic update mutation example (async await)

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query'
import type { Post } from './types'

const api = createApi({
  baseQuery: fetchBaseQuery({
    baseUrl: '/',
  }),
  tagTypes: ['Post'],
  endpoints: (build) => ({
    getPost: build.query<Post, number>({
      query: (id) => `post/${id}`,
      providesTags: ['Post'],
    }),
    updatePost: build.mutation<void, Pick<Post, 'id'> & Partial<Post>>({
      query: ({ id, ...patch }) => ({
        url: `post/${id}`,
        method: 'PATCH',
        body: patch,
```

```
      }),
      async onQueryStarted({ id, ...patch }, { dispatch, queryFulfilled }) {
        const patchResult = dispatch(
          api.util.updateQueryData('getPost', id, (draft) => {
            Object.assign(draft, patch)
          })
        )
        try {
          await queryFulfilled
        } catch {
          patchResult.undo()

          /**
           * Alternatively, on failure you can invalidate the corresponding
cache tags
           * to trigger a re-fetch:
           * dispatch(api.util.invalidateTags(['Post']))
           */
        }
      },
    }),
  }),
})
```

or, if you prefer the slightly shorter version with `.catch`

```
-      async onQueryStarted({ id, ...patch }, { dispatch, queryFulfilled }) {
+      onQueryStarted({ id, ...patch }, { dispatch, queryFulfilled }) {
        const patchResult = dispatch(
          api.util.updateQueryData('getPost', id, (draft) => {
            Object.assign(draft, patch)
          })
        )
-       try {
-         await queryFulfilled
-       } catch {
-         patchResult.undo()
-       }
+       queryFulfilled.catch(patchResult.undo)
      }
```

Example

React Optimistic Updates

Pessimistic Updates

When you wish to perform an update to cache data based on the response received from the server after a `mutation` is triggered, you can apply a `pessimistic update`. The distinction between a `pessimistic update` and an `optimistic update` is that the `pessimistic update` will instead wait for the response from the server prior to updating the cached data.

The core concepts for a pessimistic update are:

- when you start a query or mutation, `onQueryStarted` will be executed
- you await `queryFulfilled` to resolve to an object containing the transformed response from the server in the `data` property
- you manually update the cached data by dispatching `api.util.updateQueryData` within `onQueryStarted`, using the data in the response from the server for your draft updates

- TypeScript
- JavaScript

Pessimistic update mutation example (async await)

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query'
import type { Post } from './types'

const api = createApi({
  baseQuery: fetchBaseQuery({
    baseUrl: '/',
  }),
  tagTypes: ['Post'],
  endpoints: (build) => ({
    getPost: build.query<Post, number>({
      query: (id) => `post/${id}`,
      providesTags: ['Post'],
    }),
    updatePost: build.mutation<Post, Pick<Post, 'id'> & Partial<Post>>({
      query: ({ id, ...patch }) => ({
        url: `post/${id}`,
        method: 'PATCH',
        body: patch,
      }),
      async onQueryStarted({ id, ...patch }, { dispatch, queryFulfilled }) {
        try {
          const { data: updatedPost } = await queryFulfilled
          const patchResult = dispatch(
            api.util.updateQueryData('getPost', id, (draft) => {
              Object.assign(draft, updatedPost)
            })
          )
        } catch {}
      },
    }),
  }),
})
```

General Updates

If you find yourself wanting to update cache data elsewhere in your application, you can do so anywhere you have access to the `store.dispatch` method, including within React components via the useDispatch hook (or a typed version such as useAppDispatch for typescript users).

INFO

You should generally avoid manually updating the cache outside of the `onQueryStarted` callback for a mutation without a good reason, as RTK Query is intended to be used by considering your cached data as a reflection of the server-side state.

General manual cache update example

```
import { api } from './api'
import { useAppDispatch } from './store/hooks'

function App() {
  const dispatch = useAppDispatch()

  function handleClick() {
    /**
     * This will update the cache data for the query corresponding to the
`getPosts` endpoint,
     * when that endpoint is used with no argument (undefined).
     */
    const patchCollection = dispatch(
      api.util.updateQueryData('getPosts', undefined, (draftPosts) => {
        draftPosts.push({ id: 1, name: 'Teddy' })
      })
    )
  }

  return <button onClick={handleClick}>Add post to cache</button>
}
```

*Last updated on **Apr 2, 2023***

Conditional Fetching

Overview

Query hooks automatically begin fetching data as soon as the component is mounted. But, there are use cases where you may want to delay fetching data until some condition becomes true. RTK Query supports conditional fetching to enable that behavior.

If you want to prevent a query from automatically running, you can use the `skip` parameter in a hook.

Skip example

```
const Pokemon = ({ name, skip }: { name: string; skip: boolean }) => {
  const { data, error, status } = useGetPokemonByNameQuery(name, {
    skip,
  });

  return (
    <div>
      {name} - {status}
    </div>
  );
};
```

When `skip` is true (or `skipToken` is passed in as `arg`):

- **If the query has cached data:**
  - The cached data **will not be used** on the initial load, and will ignore updates from any identical query until the `skip` condition is removed
  - The query will have a status of `uninitialized`
  - If `skip: false` is set after the initial load, the cached result will be used
- **If the query does not have cached data:**
  - The query will have a status of `uninitialized`
  - The query will not exist in the state when viewed with the dev tools
  - The query will not automatically fetch on mount
  - The query will not automatically run when additional components with the same query are added that do run

TIP

Typescript users may wish to use `skipToken` as an alternative to the `skip` option in order to skip running a query, while still keeping types for the endpoint accurate.

*Last updated on **Jun 19, 2021***

Error Handling

Overview

If your query or mutation happens to throw an error when using fetchBaseQuery, it will be returned in the `error` property of the respective hook. The component will re-render when that occurs, and you can show appropriate UI based on the error data if desired.

Error Display Examples

Query Error

```
function PostsList() {
  const { data, error } = useGetPostsQuery()

  return (
    <div>
      {error.status} {JSON.stringify(error.data)}
    </div>
  )
}
```

Mutation Error

```
function AddPost() {
  const [addPost, { error }] = useAddPostMutation()

  return (
    <div>
      {error.status} {JSON.stringify(error.data)}
    </div>
  )
}
```

If you need to access the error or success payload immediately after a mutation, you can chain `.unwrap()`.

Using .unwrap

```
addPost({ id: 1, name: 'Example' })
  .unwrap()
  .then((payload) => console.log('fulfilled', payload))
  .catch((error) => console.error('rejected', error))
```

Manually selecting an error

```
function PostsList() {
  const { error } = useSelector(api.endpoints.getPosts.select())

  return (
    <div>
      {error.status} {JSON.stringify(error.data)}
    </div> )}
```

Errors with a custom `baseQuery`

Whether a response is returned as `data` or `error` is dictated by the `baseQuery` provided.

Ultimately, you can choose whatever library you prefer to use with your `baseQuery`, but it's important that you return the correct response format. If you haven't tried `fetchBaseQuery` yet, give it a chance! Otherwise, see Customizing Queries for information on how to alter the returned errors.

Handling errors at a macro level

There are quite a few ways that you can manage your errors, and in some cases, you may want to show a generic toast notification for any async error. Being that RTK Query is built on top of Redux and Redux-Toolkit, you can easily add a middleware to your store for this purpose.

TIP

Redux Toolkit has action matching utilities that we can leverage for additional custom behaviors.
Error catching middleware example

```
import { isRejectedWithValue } from '@reduxjs/toolkit'
import type { MiddlewareAPI, Middleware } from '@reduxjs/toolkit'
import { toast } from 'your-cool-library'

/**
 * Log a warning and show a toast!
 */
export const rtkQueryErrorLogger: Middleware =
  (api: MiddlewareAPI) => (next) => (action) => {
    // RTK Query uses `createAsyncThunk` from redux-toolkit under the hood,
so we're able to utilize these matchers!
    if (isRejectedWithValue(action)) {
      console.warn('We got a rejected action!')
      toast.warn({ title: 'Async error!', message: action.error.data.message
})
    }

    return next(action)
  }
```

*Last updated on **Apr 2, 2023***

Pagination

RTK Query does not include any built-in pagination behavior. However, RTK Query does make it straightforward to integrate with a standard index-based pagination API. This is the most common form of pagination that you'll need to implement.

Pagination Recipes

Setup an endpoint to accept a page `arg`

src/app/services/posts.ts

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
interface Post {
  id: number
  name: string
}
interface ListResponse<T> {
  page: number
  per_page: number
  total: number
  total_pages: number
  data: T[]
}

export const api = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  endpoints: (builder) => ({
    listPosts: builder.query<ListResponse<Post>, number | void>({
      query: (page = 1) => `posts?page=${page}`,
    }),
  }),
})

export const { useListPostsQuery } = api
```

Trigger the next page by incrementing the `page` state variable

src/features/posts/PostsManager.tsx

```
const PostList = () => {
  const [page, setPage] = useState(1)
  const { data: posts, isLoading, isFetching } = useListPostsQuery(page)

  if (isLoading) {
    return <div>Loading</div>
  }

  if (!posts?.data) {
    return <div>No posts :(</div>
  }

  return (
    <div>
      {posts.data.map(({ id, title, status }) => (
```

```
        <div key={id}>
          {title} - {status}
        </div>
      ))}
      <button onClick={() => setPage(page - 1)} isLoading={isFetching}>
        Previous
      </button>
      <button onClick={() => setPage(page + 1)} isLoading={isFetching}>
        Next
      </button>
    </div>
  )
}
```

Automated Re-fetching of Paginated Queries

It is a common use-case to utilize tag invalidation to perform automated re-fetching with RTK Query.

A potential pitfall when combining this with pagination is that your paginated query may only provide a *partial* list at any given time, and hence not `provide` tags for entity IDs that fall on pages which aren't currently shown. If a specific entity is deleted that falls on an earlier page, the paginated query will not be providing a tag for that specific ID, and will not be invalidated to trigger re-fetching data. As a result, items on the current page that should shift one item up will not have done so, and the total count of items and/or pages may be incorrect.

A strategy to overcome this is to ensure that the `delete` mutation always `invalidates` the paginated query, even if the deleted item is not *currently* provided on that page. We can leverage the concept of advanced invalidation with abstract tag ids to do this by `providing` a `'Posts'` tag with the `'PARTIAL-LIST'` ID in our paginated query, and `invalidating` that corresponding tag for any mutation that should affect it.

Example of invalidating cache for paginated queries

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
interface Post {
  id: number
  name: string
}
interface ListResponse<T> {
  page: number
  per_page: number
  total: number
  total_pages: number
  data: T[]
}

export const postApi = createApi({
```

```
  reducerPath: 'postsApi',
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  tagTypes: ['Posts'],
  endpoints: (build) => ({
    listPosts: build.query<ListResponse<Post>, number | void>({
      query: (page = 1) => `posts?page=${page}`,
      providesTags: (result, error, page) =>
        result
          ? [
              // Provides a tag for each post in the current page,
              // as well as the 'PARTIAL-LIST' tag.
              ...result.data.map(({ id }) => ({ type: 'Posts' as const, id
})),
              { type: 'Posts', id: 'PARTIAL-LIST' },
            ]
          : [{ type: 'Posts', id: 'PARTIAL-LIST' }],
    }),
    deletePost: build.mutation<{ success: boolean; id: number }, number>({
      query(id) {
        return {
          url: `post/${id}`,
          method: 'DELETE',
        }
      },
      // Invalidates the tag for this Post `id`, as well as the `PARTIAL-
LIST` tag,
      // causing the `listPosts` query to re-fetch if a component is
subscribed to the query.
      invalidatesTags: (result, error, id) => [
        { type: 'Posts', id },
        { type: 'Posts', id: 'PARTIAL-LIST' },
      ],
    }),
  }),
})
```

General Pagination Example

*Last updated on **Apr 2, 2023***

Prefetching

The goal of prefetching is to make data fetch *before* the user navigates to a page or attempts to load some known content.

There are a handful of situations that you may want to do this, but some very common use cases are:

1. User hovers over a navigation element
2. User hovers over a list element that is a link
3. User hovers over a next pagination button
4. User navigates to a page and you know that some components down the tree will require said data. This way, you can prevent fetching waterfalls.

Prefetching with React Hooks

Similar to the `useMutation` hook, the `usePrefetch` hook will not run automatically — it returns a "trigger function" that can be used to initiate the behavior.

It accepts two arguments: the first is the key of a query action that you defined in your API service, and the second is an object of two optional parameters:

usePrefetch Signature

```
export type PrefetchOptions =
  | { force?: boolean }
  | {
      ifOlderThan?: false | number;
    };

usePrefetch<EndpointName extends QueryKeys<Definitions>>(
    endpointName: EndpointName,
    options?: PrefetchOptions
  ): (arg: QueryArgFrom<Definitions[EndpointName]>, options?:
PrefetchOptions) => void;
```

Customizing the Hook Behavior

You can specify these prefetch options when declaring the hook or at the call site. The call site will take priority over the defaults.

1. `ifOlderThan` - (default: `false` | `number`) - *number is value in seconds*
    - If specified, it will only run the query if the difference between `new Date()` and the last `fulfilledTimeStamp` is greater than the given value
2. `force`

- If `force: true`, it will ignore the `ifOlderThan` value if it is set and the query will be run even if it exists in the cache.

Trigger Function Behavior

1. The trigger function *always* returns `void`.
2. If `force: true` is set during the declaration or at the call site, the query will be run no matter what. The one exception to that is if the same query is already in-flight.
3. If no options are specified and the query exists in the cache, the query will not be performed.
4. If no options are specified and the query *does not exist* in the cache, the query will be performed.
    - **Assuming** you have a `useQuery` hook in the tree that is subscribed to the same query that you are prefetching:
        - `useQuery` will return `{isLoading: true, isFetching: true, ...rest}`
5. If `ifOlderThan` is specified but evaluates to false and the query is in the cache, the query will not be performed.
6. If `ifOlderThan` is specified and evaluates to true, the query will be performed even if there is an existing cache entry.
    - **Assuming** you have a `useQuery` hook in the tree that is subscribed to the same query that you are prefetching:
        - `useQuery` will return `{isLoading: false, isFetching: true, ...rest}`

usePrefetch Example

```
function User() {
  const prefetchUser = usePrefetch('getUser')

  // Low priority hover will not fire unless the last request happened more
than 35s ago
  // High priority hover will _always_ fire
  return (
    <div>
      <button onMouseEnter={() => prefetchUser(4, { ifOlderThan: 35 })}>
        Low priority
      </button>
      <button onMouseEnter={() => prefetchUser(4, { force: true })}>
        High priority
      </button>
    </div>
  )
}
```

Recipe: Prefetch Immediately

In some cases, you may want to prefetch a resource immediately. You can implement this in just a few lines of code:

hooks/usePrefetchImmediately.ts

```
type EndpointNames = keyof typeof api.endpoints

export function usePrefetchImmediately<T extends EndpointNames>(
  endpoint: T,
  arg: Parameters<typeof api.endpoints[T]['initiate']>[0],
  options: PrefetchOptions = {}
) {
  const dispatch = useAppDispatch()
  useEffect(() => {
    dispatch(api.util.prefetch(endpoint, arg as any, options))
  }, [])
}

// In a component
usePrefetchImmediately('getUser', 5)
```

Prefetching Without Hooks

If you're not using the `usePrefetch` hook, you can recreate the same behavior on your own in any framework.

When dispatching the `prefetch` thunk as shown below you will see the same exact behavior as described here.

Non-hook prefetching example

```
store.dispatch(
  api.util.prefetch(endpointName, arg, { force: false, ifOlderThan: 10 })
)
```

You can also dispatch the query action, but you would be responsible for implementing any additional logic.

Alternate method of manual prefetching

```
 dispatch(api.endpoints[endpointName].initiate(arg, { forceRefetch: true }))
```

*Last updated on **Apr 2, 2023***

Polling

Polling Overview

Polling gives you the ability to have a 'real-time' effect by causing a query to run at a specified interval. To enable polling for a query, pass a `pollingInterval` to the `useQuery` hook or action creator with an interval in milliseconds:

src/Pokemon.tsx

```
import * as React from 'react'
import { useGetPokemonByNameQuery } from './services/pokemon'

export const Pokemon = ({ name }: { name: string }) => {
  // Automatically refetch every 3s
  const { data, status, error, refetch } = useGetPokemonByNameQuery(name, {
    pollingInterval: 3000,
  })

  return <div>{data}</div>
}
```

In an action creator without React Hooks:

```
const { data, status, error, refetch } = store.dispatch(
  endpoints.getCountById.initiate(id, {
    subscriptionOptions: { pollingInterval: 3000 },
  })
)
```

Polling Without React Hooks

If you use polling without the convenience of React Hooks, you will need to manually call `updateSubscriptionOptions` on the promise ref to update the interval. This approach varies by framework but is possible everywhere. See the Svelte Example for one possibility, and the Usage Without React Hooks page for more details on working with subscriptions manually.

```
queryRef.updateSubscriptionOptions({ pollingInterval: 0 })
```

*Last updated on **Apr 2, 2023***

Streaming Updates

Overview

RTK Query gives you the ability to receive **streaming updates** for persistent queries. This enables a query to establish an ongoing connection to the server (typically using WebSockets), and apply updates to the cached data as additional information is received from the server.

Streaming updates can be used to enable the API to receive real-time updates to the back-end data, such as new entries being created, or important properties being updated.

To enable streaming updates for a query, pass the asynchronous `onCacheEntryAdded` function to the query, including the logic for how to update the query when streamed data is received.
See `onCacheEntryAdded` API reference for more details.

When to use streaming updates

Primarily updates to query data should be done via `polling` intermittently on an interval, using `cache invalidation` to invalidate data based on tags associated with queries & mutations, or with `refetchOnMountOrArgChange` to fetch fresh data when a component using the data mounts.

However, streaming updates is particularly useful for scenarios involving:

- *Small, frequent changes to large objects*. Rather than repeatedly polling for a large object, the object can be fetched with an initial query, and streaming updates can update individual properties as updates are received.
- *External event-driven updates*. Where data may be changed by the server or otherwise external users and where real-time updates are expected to be shown to an active user, polling alone would result in periods of stale data in between queries, causing state to easily get out of sync. Streaming updates can update all active clients as the updates occur rather than waiting for the next interval to elapse.

Example use cases that benefit from streaming updates are:

- GraphQL subscriptions
- Real-time chat applications

- Real-time multiplayer games
- Collaborative document editing with multiple concurrent users

Using the `onCacheEntryAdded` Lifecycle

The `onCacheEntryAdded` lifecycle callback lets you write arbitrary async logic that will be executed after a new cache entry is added to the RTK Query cache (ie, after a component has created a new subscription to a given endpoint+params combination).

`onCacheEntryAdded` will be called with two arguments: the `arg` that was passed to the subscription, and an options object containing "lifecycle promises" and utility functions. You can use these to write sequenced logic that waits for data to be added, initiates server connections, applies partial updates, and cleans up the connection when the query subscription is removed.

Typically, you will `await cacheDataLoaded` to determine when the first data has been fetched, then use the `updateCacheData` utility to apply streaming updates as messages are received. `updateCacheData` is an Immer-powered callback that receives a `draft` of the current cache value. You may "mutate" the draft value to update it as needed based on the received values. RTK Query will then dispatch an action that applies a diffed patch based on those changes.

Finally, you can `await cacheEntryRemoved` to know when to clean up any server connections.

Streaming Update Examples

Websocket Chat API

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
import { isMessage } from './schemaValidators'

export type Channel = 'redux' | 'general'

export interface Message {
  id: number
  channel: Channel
  userName: string
  text: string
}

export const api = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  endpoints: (build) => ({
    getMessages: build.query<Message[], Channel>({
      query: (channel) => `messages/${channel}`,
      async onCacheEntryAdded(
        arg,
        { updateCachedData, cacheDataLoaded, cacheEntryRemoved }
```

```
      ) {
        // create a websocket connection when the cache subscription starts
        const ws = new WebSocket('ws://localhost:8080')
        try {
          // wait for the initial query to resolve before proceeding
          await cacheDataLoaded

          // when data is received from the socket connection to the server,
          // if it is a message and for the appropriate channel,
          // update our query result with the received message
          const listener = (event: MessageEvent) => {
            const data = JSON.parse(event.data)
            if (!isMessage(data) || data.channel !== arg) return

            updateCachedData((draft) => {
              draft.push(data)
            })
          }

          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 { useGetMessagesQuery } = api
```

What to expect

When the `getMessages` query is triggered (e.g. via a component mounting with the `useGetMessagesQuery()` hook), a `cache entry` will be added based on the serialized arguments for the endpoint. The associated query will be fired off based on the `query` property to fetch the initial data for the cache. Meanwhile, the asynchronous `onCacheEntryAdded` callback will begin, and create a new WebSocket connection. Once the response for the initial query is received, the cache will be populated with the response data, and the `cacheDataLoaded` promise will resolve. After awaiting the `cacheDataLoaded` promise, the `message` event listener will be added to the WebSocket connection, which updates the cache data when an associated message is received.

When there are no more active subscriptions to the data (e.g. when the subscribed components remain unmounted for a sufficient amount of time), the `cacheEntryRemoved` promise will resolve, allowing the remaining code to run

and close the websocket connection. RTK Query will also remove the associated data from the cache.

If a query for the corresponding cache entry runs later, it will overwrite the whole cache entry, and the streaming update listeners will continue to work on the updated data.

Websocket Chat API with a transformed response shape

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
import { createEntityAdapter } from '@reduxjs/toolkit'
import type { EntityState } from '@reduxjs/toolkit'
import { isMessage } from './schemaValidators'

export type Channel = 'redux' | 'general'

export interface Message {
  id: number
  channel: Channel
  userName: string
  text: string
}

const messagesAdapter = createEntityAdapter<Message>()
export const api = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  endpoints: (build) => ({
    getMessages: build.query<EntityState<Message>, Channel>({
      query: (channel) => `messages/${channel}`,
      transformResponse(response: Message[]) {
        return messagesAdapter.addMany(
          messagesAdapter.getInitialState(),
          response
        )
      },
      async onCacheEntryAdded(
        arg,
        { updateCachedData, cacheDataLoaded, cacheEntryRemoved }
      ) {
        const ws = new WebSocket('ws://localhost:8080')
        try {
          await cacheDataLoaded

          const listener = (event: MessageEvent) => {
            const data = JSON.parse(event.data)
            if (!isMessage(data) || data.channel !== arg) return

            updateCachedData((draft) => {
              messagesAdapter.upsertOne(draft, data)
            })
          }

          ws.addEventListener('message', listener)
        } catch {}
        await cacheEntryRemoved
        ws.close()
      },
    }),
  }),
```

```
})

export const { useGetMessagesQuery } = api
```

This example demonstrates how the previous example can be altered to allow for transforming the response shape when adding data to the cache.

For example, the data is transformed from this shape:

```
[
  {
    id: 0
    channel: 'redux'
    userName: 'Mark'
    text: 'Welcome to #redux!'
  },
  {
    id: 1
    channel: 'redux'
    userName: 'Lenz'
    text: 'Glad to be here!'
  },
]
```

To this:

```
{
  // The unique IDs of each item. Must be strings or numbers
  ids: [0, 1],
  // A lookup table mapping entity IDs to the corresponding entity objects
  entities: {
    0: {
      id: 0,
      channel: "redux",
      userName: "Mark",
      text: "Welcome to #redux!",
    },
    1: {
      id: 1,
      channel: "redux",
      userName: "Lenz",
      text: "Glad to be here!",
    },
  },
};
```

A key point to keep in mind is that updates to the cached data within the `onCacheEntryAdded` callback must respect the transformed data shape which will be present for the cached data. The example shows how `createEntityAdapter` can be used for the initial `transformResponse`, and again

when streamed updates are received to upsert received items into the cached data, while maintaining the normalized state structure.

*Last updated on **Jun 24, 2022***

Code Splitting

RTK Query makes it possible to trim down your initial bundle size by allowing you to inject additional endpoints after you've set up your initial service definition. This can be very beneficial for larger applications that may have *many* endpoints.

`injectEndpoints` accepts a collection of endpoints, as well as an optional `overrideExisting` parameter.

Calling `injectEndpoints` will inject the endpoints into the original API, but also give you that same API with correct types for these endpoints back. (Unfortunately, it cannot modify the types for the original definition.)

A typical approach would be to have one empty central API slice definition:

Basic setup

```
// Or from '@reduxjs/toolkit/query' if not using the auto-generated hooks
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'

// initialize an empty api service that we'll inject endpoints into later as
needed
export const emptySplitApi = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  endpoints: () => ({}),
})
```

and then inject the api endpoints in other files and export them from there - that way you will be sure to always import the endpoints in a way that they are definitely injected.

Injecting & exporting additional endpoints

```
import { emptySplitApi } from './emptySplitApi'

const extendedApi = emptySplitApi.injectEndpoints({
  endpoints: (build) => ({
    example: build.query({
      query: () => 'test',
    }),
  }),
  overrideExisting: false,
})

export const { useExampleQuery } = extendedApi
```

If you inject an endpoint that already exists and don't explicitly specify `overrideExisting: true`, the endpoint will not be overridden. In development mode, you will get a warning about this.

*Last updated on **Mar 28, 2023***

Code Generation

RTK Query's API and architecture is oriented around declaring API endpoints up front. This lends itself well to automatically generating API slice definitions from external API schema definitions, such as OpenAPI and GraphQL.

We have early previews of code generation capabilities available as separate tools.

GraphQL

We provide a Plugin for GraphQL Codegen. You can find the documentation to that on the graphql-codegen homepage.

For a full example on how to use it, you can see this example project.

OpenAPI

We provide a package for RTK Query code generation from OpenAPI schemas. It is published as `@rtk-query/codegen-openapi` and you can find the source code at `packages/rtk-query-codegen-openapi`.

Usage

Create an empty api using `createApi` like

src/store/emptyApi.ts

```
// Or from '@reduxjs/toolkit/query' if not using the auto-generated hooks
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'

// initialize an empty api service that we'll inject endpoints into later as
needed
export const emptySplitApi = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  endpoints: () => ({}),
})
```

Generate a config file (json, js or ts) with contents like

openapi-config.ts

```
import type { ConfigFile } from '@rtk-query/codegen-openapi'

const config: ConfigFile = {
  schemaFile: 'https://petstore3.swagger.io/api/v3/openapi.json',
  apiFile: './src/store/emptyApi.ts',
  apiImport: 'emptySplitApi',
  outputFile: './src/store/petApi.ts',
  exportName: 'petApi',
  hooks: true,
}

export default config
```

and then call the code generator:

```
npx @rtk-query/codegen-openapi openapi-config.ts
```

Generating tags

If your OpenAPI specification uses tags, you can specify the `tag` option to the codegen.
That will result in all generated endpoints
having `providesTags`/`invalidatesTags` declarations for the `tags` of their respective operation definition.

Note that this will only result in string tags with no ids, so it might lead to scenarios where too much is invalidated and unneccessary requests are made on mutation.

In that case it is still recommended to manually specify tags by using `enhanceEndpoints` on top of the generated api and manually declare `providesTags`/`invalidatesTags`.

Programmatic usage

src/store/petApi.ts

```
import { generateEndpoints } from '@rtk-query/codegen-openapi'

const api = await generateEndpoints({
  apiFile: './fixtures/emptyApi.ts',
  schemaFile: resolve(__dirname, 'fixtures/petstore.json'),
  filterEndpoints: ['getPetById', 'addPet'],
  hooks: true,
})
```

Config file options

Simple usage

```
interface SimpleUsage {
  apiFile: string
  schemaFile: string
  apiImport?: string
  exportName?: string
  argSuffix?: string
  responseSuffix?: string
  hooks?:
    | boolean
    | { queries: boolean; lazyQueries: boolean; mutations: boolean }
  tag?: boolean
  outputFile: string
  filterEndpoints?:
    | string
    | RegExp
    | EndpointMatcherFunction
    | Array<string | RegExp | EndpointMatcherFunction>
  endpointOverrides?: EndpointOverrides[]
  flattenArg?: boolean
}

export type EndpointMatcherFunction = (
  operationName: string,
  operationDefinition: OperationDefinition
) => boolean
```

Filtering endpoints

If you only want to include a few endpoints, you can use the `filterEndpoints` config option to filter your endpoints.

openapi-config.ts

```
const filteredConfig: ConfigFile = {
  // ...
  // should only have endpoints loginUser, placeOrder, getOrderById,
deleteOrder
  filterEndpoints: ['loginUser', /Order/],
}
```

Endpoint overrides

If an endpoint is generated as a mutation instead of a query or the other way round, you can override that:

openapi-config.ts

```
const withOverride: ConfigFile = {
  // ...
  endpointOverrides: [
    {
      pattern: 'loginUser',
      type: 'mutation',
    },
  ],
}
```

Generating hooks

Setting `hooks: true` will generate `useQuery` and `useMutation` hook exports. If you also want `useLazyQuery` hooks generated or more granular control, you can also pass an object in the shape of: `{ queries: boolean; lazyQueries: boolean; mutations: boolean }`.

Multiple output files

openapi-config.ts

```
const config: ConfigFile = {
  schemaFile: 'https://petstore3.swagger.io/api/v3/openapi.json',
  apiFile: './src/store/emptyApi.ts',
  outputFiles: {
    './src/store/user.ts': {
      filterEndpoints: [/user/i],
    },
    './src/store/order.ts': {
      filterEndpoints: [/order/i],
    },
    './src/store/pet.ts': {
      filterEndpoints: [/pet/i],
    },
  },
}
```

Persistence and Rehydration

RTK Query supports rehydration via the `extractRehydrationInfo` option on `createApi`. This function is passed every dispatched action, and where it returns a value other than `undefined`, that value is used to rehydrate the API state for fulfilled & errored queries.

See also Server Side Rendering.

> INFO
>
> Generally, persisting API slices is not recommended and instead, mechanisms like `Cache-Control` Headers should be used in browsers to define cache behaviour. Persisting and rehydrating an api slice might always leave the user with very stale data if the user has not visited the page for some time. Nonetheless, in environments like Native Apps, where there is no browser cache to take care of this, persistance might still be a viable option.

Redux Persist

API state rehydration can be used in conjunction with Redux Persist by leveraging the `REHYDRATE` action type imported from `redux-persist`. This can be used out of the box with the `autoMergeLevel1` or `autoMergeLevel2` state reconcilers when persisting the root reducer, or with the `autoMergeLevel1` reconciler when persisting just the api reducer.

redux-persist rehydration example

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
import { REHYDRATE } from 'redux-persist'

export const api = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  extractRehydrationInfo(action, { reducerPath }) {
    if (action.type === REHYDRATE) {
      return action.payload[reducerPath]
    }
  },
  endpoints: (build) => ({
    // omitted
  }),
})
```

*Last updated on **Oct 29, 2021***

Customizing `createApi`

Currently, RTK Query includes two variants of `createApi`:

- `createBaseApi`, which contains only the UI-agnostic Redux logic (the core module)
- `createApi`, which contains both the core and React hooks modules

You can create your own versions of `createApi` by either specifying non-default options for the modules or by adding your own modules.

Customizing the React-Redux Hooks

If you want the hooks to use different versions of `useSelector` or `useDispatch`, such as if you are using a custom context, you can pass these in at module creation:

```
import * as React from 'react'
import { createDispatchHook, ReactReduxContextValue } from 'react-redux'
import {
  buildCreateApi,
  coreModule,
  reactHooksModule,
} from '@reduxjs/toolkit/query/react'

const MyContext = React.createContext<ReactReduxContextValue>(null as any)
const customCreateApi = buildCreateApi(
  coreModule(),
  reactHooksModule({ useDispatch: createDispatchHook(MyContext) })
)
```

Creating your own module

If you want to create your own module, you should review the react-hooks module to see what an implementation would look like.

Here is a very stripped down version:

```ts
import { CoreModule } from '@internal/core/module'
import {
  BaseQueryFn,
  EndpointDefinitions,
  Api,
  Module,
  buildCreateApi,
  coreModule,
} from '@reduxjs/toolkit/query'

export const customModuleName = Symbol()
export type CustomModule = typeof customModuleName

declare module '../apiTypes' {
  export interface ApiModules<
    BaseQuery extends BaseQueryFn,
    Definitions extends EndpointDefinitions,
    ReducerPath extends string,
    TagTypes extends string
  > {
    [customModuleName]: {
      endpoints: {
        [K in keyof Definitions]: {
          myEndpointProperty: string
        }
      }
    }
  }
}

export const myModule = (): Module<CustomModule> => ({
  name: customModuleName,
  init(api, options, context) {
    // initialize stuff here if you need to

    return {
      injectEndpoint(endpoint, definition) {
        const anyApi = (api as any) as Api<
          any,
          Record<string, any>,
          string,
          string,
          CustomModule | CoreModule
        >
        anyApi.endpoints[endpoint].myEndpointProperty = 'test'
      },
    }
  },
})

export const myCreateApi = buildCreateApi(coreModule(), myModule())
```

*Last updated on **Jun 19, 2021***

Customizing queries

RTK Query is agnostic as to how your requests resolve. You can use any library you like to handle requests, or no library at all. RTK Query provides reasonable defaults expected to cover the majority of use cases, while also allowing room for customization to alter query handling to fit specific needs.

Customizing queries with `baseQuery`

The default method to handle queries is via the `baseQuery` option on `createApi`, in combination with the `query` option on an endpoint definition.

To process queries, endpoints are defined with a `query` option, which passes its return value to a common `baseQuery` function used for the API.

By default, RTK Query ships with `fetchBaseQuery`, which is a lightweight `fetch` wrapper that automatically handles request headers and response parsing in a manner similar to common libraries like `axios`.
If `fetchBaseQuery` alone does not meet your needs, you can customize its behaviour with a wrapper function, or create your own `baseQuery` function from scratch for `createApi` to use.

See also `baseQuery API Reference`.

Implementing a custom `baseQuery`

RTK Query expects a `baseQuery` function to be called with three arguments: `args`, `api`, and `extraOptions`. It is expected to return an object with either a `data` or `error` property, or a promise that resolves to return such an object.

baseQuery function arguments

baseQuery example arguments

```
const customBaseQuery = (
  args,
  { signal, dispatch, getState },
  extraOptions
) => {
  // omitted
}
```

baseQuery function return value

1. Expected success result format

   ```
   return { data: YourData }
   ```

2. Expected error result format

   ```
   return { error: YourError }
   ```

baseQuery example return value

```
const customBaseQuery = (
  args,
  { signal, dispatch, getState },
  extraOptions
) => {
  if (Math.random() > 0.5) return { error: 'Too high!' }
  return { data: 'All good!' }
}
```

NOTE

This format is required so that RTK Query can infer the return types for your responses.

At its core, a `baseQuery` function only needs to have the minimum return value to be valid; an object with a `data` or `error` property. It is up to the user to determine how they wish to use the provided arguments, and how requests are handled within the function itself.

fetchBaseQuery defaults

For `fetchBaseQuery` specifically, the return type is as follows:

Return types of fetchBaseQuery

```
Promise<{
    data: any;
    error?: undefined;
    meta?: { request: Request; response: Response };
} | {
    error: {
        status: number;
        data: any;
    };
    data?: undefined;
    meta?: { request: Request; response: Response };
}>
```

1. Expected success result format with fetchBaseQuery

```
return { data: YourData }
```

2. Expected error result format with fetchBaseQuery

```
return { error: { status: number, data: YourErrorData } }
```

Customizing query responses with `transformResponse`

Individual endpoints on `createApi` accept a `transformResponse` property which allows manipulation of the data returned by a query or mutation before it hits the cache.

`transformResponse` is called with the data that a successful `baseQuery` returns for the corresponding endpoint, and the return value of `transformResponse` is used as the cached data associated with that endpoint call.

By default, the payload from the server is returned directly.

```
function defaultTransformResponse(
  baseQueryReturnValue: unknown,
  meta: unknown,
  arg: unknown
) {
  return baseQueryReturnValue
}
```

To change it, provide a function that looks like:

Unpack a deeply nested collection

```
transformResponse: (response, meta, arg) =>
  response.some.deeply.nested.collection
```

`transformResponse` is called with the `meta` property returned from the `baseQuery` as its second argument, which can be used while determining the transformed response. The value for `meta` is dependent on the `baseQuery` used.

transformResponse meta example

```
transformResponse: (response: { sideA: Tracks; sideB: Tracks }, meta, arg) =>
{
  if (meta?.coinFlip === 'heads') {
    return response.sideA
  }
  return response.sideB
}
```

`transformResponse` is called with the `arg` property provided to the endpoint as its third argument, which can be used while determining the transformed response. The value for `arg` is dependent on the `endpoint` used, as well as the argument used when calling the query/mutation.

transformResponse arg example

```
transformResponse: (response: Posts, meta, arg) => {
  return {
    originalArg: arg,
    data: response,
  }
}
```

While there is less need to store the response in a normalized lookup table with RTK Query managing caching data, `transformResponse` can be leveraged to do so if desired.

Normalize the response data

```
transformResponse: (response) =>
  response.reduce((acc, curr) => {
    acc[curr.id] = curr
    return acc
  }, {})

/*
  will convert:
  [
    {id: 1, name: 'Harry'},
    {id: 2, name: 'Ron'},
    {id: 3, name: 'Hermione'},
  ]

  to:
  {
    1: { id: 1, name: "Harry" },
    2: { id: 2, name: "Ron" },
    3: { id: 3, name: "Hermione" },
  }
*/
```

createEntityAdapter can also be used with transformResponse to normalize data, while also taking advantage of other features provided by createEntityAdapter, including providing an ids array, using sortComparer to maintain a consistently sorted list, as well as maintaining strong TypeScript support.

See also Websocket Chat API with a transformed response shape for an example of transformResponse normalizing response data in combination with createEntityAdapter, while also updating further data using streaming updates.

Customizing query responses with `transformErrorResponse`

Individual endpoints on createApi accept a transformErrorResponse property which allows manipulation of the error returned by a query or mutation before it hits the cache.

transformErrorResponse is called with the error that a failed baseQuery returns for the corresponding endpoint, and the return value of transformErrorResponse is used as the cached error associated with that endpoint call.

By default, the payload from the server is returned directly.

```
function defaultTransformResponse(
  baseQueryReturnValue: unknown,
  meta: unknown,
  arg: unknown
) {
  return baseQueryReturnValue
}
```

To change it, provide a function that looks like:

Unpack a deeply nested error object

```
transformErrorResponse: (response, meta, arg) =>
  response.data.some.deeply.nested.errorObject
```

transformErrorResponse is called with the meta property returned from the baseQuery as its second argument, which can be used while determining the transformed response. The value for meta is dependent on the baseQuery used.

transformErrorResponse meta example

```
transformErrorResponse: (response: { data: { sideA: Tracks; sideB: Tracks }
}, meta, arg) => {
  if (meta?.coinFlip === 'heads') {
    return response.data.sideA
  }
  return response.data.sideB
}
```

`transformErrorResponse` is called with the `arg` property provided to the endpoint as its third argument, which can be used while determining the transformed response. The value for `arg` is dependent on the `endpoint` used, as well as the argument used when calling the query/mutation.

transformErrorResponse arg example

```
transformErrorResponse: (response: Posts, meta, arg) => {
  return {
    originalArg: arg,
    error: response,
  }
}
```

Customizing queries with `queryFn`

Individual endpoints on `createApi` accept a `queryFn` property which allows a given endpoint to ignore `baseQuery` for that endpoint by providing an inline function determining how that query resolves.

This can be useful for scenarios where you want to have particularly different behaviour for a single endpoint, or where the query itself is not relevant. Such situations may include:

- One-off queries that use a different base URL
- One-off queries that use different request handling, such as automatic re-tries
- One-off queries that use different error handling behaviour
- Performing multiple requests with a single query (example)
- Leveraging invalidation behaviour with no relevant query (example)
- Using Streaming Updates with no relevant initial request (example)

See also `queryFn API Reference` for the type signature and available options.

Implementing a `queryFn`

In order to use `queryFn`, it can be treated as an inline `baseQuery`. It will be called with the same arguments as `baseQuery`, as well as the provided `baseQuery` function itself (`arg`, `api`, `extraOptions`, and `baseQuery`). Similarly to `baseQuery`, it is expected to return an object with either a `data` or `error` property, or a promise that resolves to return such an object.

queryFn function arguments

queryFn example arguments

```
const queryFn = (
  args,
  { signal, dispatch, getState },
  extraOptions,
  baseQuery
) => {
  // omitted
}
```

queryFn function return value

1. Expected success result format

   ```
   return { data: YourData }
   ```

2. Expected error result format

   ```
   return { error: YourError }
   ```

queryFn example return value

```
const queryFn = (
  args,
  { signal, dispatch, getState },
  extraOptions,
  baseQuery
) => {
  if (Math.random() > 0.5) return { error: 'Too high!' }
  return { data: 'All good!' }
}
```

Examples - `baseQuery`

Axios baseQuery

This example implements a very basic axios-based `baseQuery` utility.

Basic axios baseQuery

```
import { createApi } from '@reduxjs/toolkit/query'
import type { BaseQueryFn } from '@reduxjs/toolkit/query'
import axios from 'axios'
import type { AxiosRequestConfig, AxiosError } from 'axios'

const axiosBaseQuery =
  (
    { baseUrl }: { baseUrl: string } = { baseUrl: '' }
  ): BaseQueryFn<
    {
      url: string
      method: AxiosRequestConfig['method']
      data?: AxiosRequestConfig['data']
      params?: AxiosRequestConfig['params']
    },
    unknown,
    unknown
  > =>
  async ({ url, method, data, params }) => {
    try {
      const result = await axios({ url: baseUrl + url, method, data, params
})
      return { data: result.data }
    } catch (axiosError) {
      let err = axiosError as AxiosError
      return {
        error: {
          status: err.response?.status,
          data: err.response?.data || err.message,
        },
      }
    }
  }

const api = createApi({
  baseQuery: axiosBaseQuery({
    baseUrl: 'https://example.com',
  }),
  endpoints(build) {
    return {
      query: build.query({ query: () => ({ url: '/query', method: 'get' })
}),
      mutation: build.mutation({
        query: () => ({ url: '/mutation', method: 'post' }),
      }),
    }
  },
})
```

GraphQL baseQuery

This example implements a very basic GraphQL-based `baseQuery`.

Basic GraphQL baseQuery

```
import { createApi } from '@reduxjs/toolkit/query'
import { request, gql, ClientError } from 'graphql-request'

const graphqlBaseQuery =
  ({ baseUrl }: { baseUrl: string }) =>
  async ({ body }: { body: string }) => {
    try {
      const result = await request(baseUrl, body)
      return { data: result }
    } catch (error) {
      if (error instanceof ClientError) {
        return { error: { status: error.response.status, data: error } }
      }
      return { error: { status: 500, data: error } }
    }
  }

export const api = createApi({
  baseQuery: graphqlBaseQuery({
    baseUrl: 'https://graphqlzero.almansi.me/api',
  }),
  endpoints: (builder) => ({
    getPosts: builder.query({
      query: () => ({
        body: gql`
          query {
            posts {
              data {
                id
                title
              }
            }
          }
        `,
      }),
      transformResponse: (response) => response.posts.data,
    }),
    getPost: builder.query({
      query: (id) => ({
        body: gql`
        query {
          post(id: ${id}) {
            id
            title
            body
          }
        }
        `,
      }),
      transformResponse: (response) => response.post,
    }),
  }),
})
```

Automatic re-authorization by extending fetchBaseQuery

This example wraps `fetchBaseQuery` such that when encountering a `401 Unauthorized` error, an additional request is sent to attempt to refresh an authorization token, and re-try to initial query after re-authorizing.

Simulating axios-like interceptors with a custom base query

```
import { fetchBaseQuery } from '@reduxjs/toolkit/query'
import type {
  BaseQueryFn,
  FetchArgs,
  FetchBaseQueryError,
} from '@reduxjs/toolkit/query'
import { tokenReceived, loggedOut } from './authSlice'

const baseQuery = fetchBaseQuery({ baseUrl: '/' })
const baseQueryWithReauth: BaseQueryFn<
  string | FetchArgs,
  unknown,
  FetchBaseQueryError
> = async (args, api, extraOptions) => {
  let result = await baseQuery(args, api, extraOptions)
  if (result.error && result.error.status === 401) {
    // try to get a new token
    const refreshResult = await baseQuery('/refreshToken', api, extraOptions)
    if (refreshResult.data) {
      // store the new token
      api.dispatch(tokenReceived(refreshResult.data))
      // retry the initial query
      result = await baseQuery(args, api, extraOptions)
    } else {
      api.dispatch(loggedOut())
    }
  }
  return result
}
```

Preventing multiple unauthorized errors

Using `async-mutex` to prevent multiple calls to '/refreshToken' when multiple calls fail with `401 Unauthorized` errors.

Preventing multiple calls to '/refreshToken'

```
import { fetchBaseQuery } from '@reduxjs/toolkit/query'
import type {
  BaseQueryFn,
  FetchArgs,
  FetchBaseQueryError,
} from '@reduxjs/toolkit/query'
import { tokenReceived, loggedOut } from './authSlice'
import { Mutex } from 'async-mutex'

// create a new mutex
const mutex = new Mutex()
const baseQuery = fetchBaseQuery({ baseUrl: '/' })
const baseQueryWithReauth: BaseQueryFn<
  string | FetchArgs,
  unknown,
  FetchBaseQueryError
> = async (args, api, extraOptions) => {
  // wait until the mutex is available without locking it
  await mutex.waitForUnlock()
  let result = await baseQuery(args, api, extraOptions)
  if (result.error && result.error.status === 401) {
    // checking whether the mutex is locked
    if (!mutex.isLocked()) {
      const release = await mutex.acquire()
      try {
        const refreshResult = await baseQuery(
          '/refreshToken',
          api,
          extraOptions
        )
        if (refreshResult.data) {
          api.dispatch(tokenReceived(refreshResult.data))
          // retry the initial query
          result = await baseQuery(args, api, extraOptions)
        } else {
          api.dispatch(loggedOut())
        }
      } finally {
        // release must be called once the mutex should be released again.
        release()
      }
    } else {
      // wait until the mutex is available without locking it
      await mutex.waitForUnlock()
      result = await baseQuery(args, api, extraOptions)
    }
  }
  return result
}
```

Automatic retries

RTK Query exports a utility called `retry` that you can wrap the `baseQuery` in your API definition with. It defaults to 5 attempts with a basic exponential backoff.

The default behavior would retry at these intervals:

1. 600ms * random(0.4, 1.4)
2. 1200ms * random(0.4, 1.4)

3. 2400ms * random(0.4, 1.4)
4. 4800ms * random(0.4, 1.4)
5. 9600ms * random(0.4, 1.4)

- TypeScript
- JavaScript

Retry every request 5 times by default

```
import { createApi, fetchBaseQuery, retry } from
'@reduxjs/toolkit/query/react'
interface Post {
  id: number
  name: string
}
type PostsResponse = Post[]

// maxRetries: 5 is the default, and can be omitted. Shown for documentation
purposes.
const staggeredBaseQuery = retry(fetchBaseQuery({ baseUrl: '/' }), {
  maxRetries: 5,
})
export const api = createApi({
  baseQuery: staggeredBaseQuery,
  endpoints: (build) => ({
    getPosts: build.query<PostsResponse, void>({
      query: () => ({ url: 'posts' }),
    }),
    getPost: build.query<PostsResponse, string>({
      query: (id) => ({ url: `post/${id}` }),
      extraOptions: { maxRetries: 8 }, // You can override the retry behavior
on each endpoint
    }),
  }),
})

export const { useGetPostsQuery, useGetPostQuery } = api
```

In the event that you didn't want to retry on a specific endpoint, you can just set `maxRetries: 0`.

INFO

It is possible for a hook to return `data` and `error` at the same time. By default, RTK Query will keep whatever the last 'good' result was in `data` until it can be updated or garbage collected.

Bailing out of error re-tries

The `retry` utility has a `fail` method property attached which can be used to bail out of retries immediately. This can be used for situations where it is known that additional re-tries would be guaranteed to all fail and would be redundant.

Bailing out of error re-tries

```
import { createApi, fetchBaseQuery, retry } from
'@reduxjs/toolkit/query/react'
import type { FetchArgs } from '@reduxjs/toolkit/dist/query/fetchBaseQuery'
interface Post {
  id: number
  name: string
}
type PostsResponse = Post[]

const staggeredBaseQueryWithBailOut = retry(
  async (args: string | FetchArgs, api, extraOptions) => {
    const result = await fetchBaseQuery({ baseUrl: '/api/' })(
      args,
      api,
      extraOptions
    )

    // bail out of re-tries immediately if unauthorized,
    // because we know successive re-retries would be redundant
    if (result.error?.status === 401) {
      retry.fail(result.error)
    }

    return result
  },
  {
    maxRetries: 5,
  }
)

export const api = createApi({
  baseQuery: staggeredBaseQueryWithBailOut,
  endpoints: (build) => ({
    getPosts: build.query<PostsResponse, void>({
      query: () => ({ url: 'posts' }),
    }),
    getPost: build.query<Post, string>({
      query: (id) => ({ url: `post/${id}` }),
      extraOptions: { maxRetries: 8 }, // You can override the retry behavior
on each endpoint
    }),
  }),
})
export const { useGetPostsQuery, useGetPostQuery } = api
```

Adding Meta information to queries

A `baseQuery` can also include a `meta` property in its return value. This can be beneficial in cases where you may wish to include additional information associated with the request such as a request ID or timestamp.

In such a scenario, the return value would look like so:

1. Expected success result format with meta

```
return { data: YourData, meta: YourMeta }
```

2. Expected error result format with meta

```
return { error: YourError, meta: YourMeta }
```

baseQuery example with meta information

```
import { fetchBaseQuery, createApi } from '@reduxjs/toolkit/query'
import type {
  BaseQueryFn,
  FetchArgs,
  FetchBaseQueryError,
} from '@reduxjs/toolkit/query'
import type { FetchBaseQueryMeta } from
'@reduxjs/toolkit/dist/query/fetchBaseQuery'
import { uuid } from './idGenerator'

type Meta = {
  requestId: string
  timestamp: number
}

const metaBaseQuery: BaseQueryFn<
  string | FetchArgs,
  unknown,
  FetchBaseQueryError,
  {},
  Meta & FetchBaseQueryMeta
> = async (args, api, extraOptions) => {
  const requestId = uuid()
  const timestamp = Date.now()

  const baseResult = await fetchBaseQuery({ baseUrl: '/' })(
    args,
    api,
    extraOptions
  )

  return {
    ...baseResult,
    meta: baseResult.meta && { ...baseResult.meta, requestId, timestamp },
  }
}

const DAY_MS = 24 * 60 * 60 * 1000

interface Post {
  id: number
  name: string
  timestamp: number
}
type PostsResponse = Post[]

const api = createApi({
  baseQuery: metaBaseQuery,
  endpoints: (build) => ({
    // a theoretical endpoint where we only want to return data
    // if request was performed past a certain date
    getRecentPosts: build.query<PostsResponse, void>({
      query: () => 'posts',
      transformResponse: (returnValue: PostsResponse, meta) => {
        // `meta` here contains our added `requestId` & `timestamp`, as well
as
```

```
        // `request` & `response` from fetchBaseQuery's meta object.
        // These properties can be used to transform the response as desired.
        if (!meta) return []
        return returnValue.filter(
          (post) => post.timestamp >= meta.timestamp - DAY_MS
        )
      },
    }),
  }),
})
```

Constructing a Dynamic Base URL using Redux state

In some cases, you may wish to have a dynamically altered base url determined from a property in your Redux state. A `baseQuery` has access to a `getState` method that provides the current store state at the time it is called. This can be used to construct the desired url using a partial url string, and the appropriate data from your store state.

Dynamically generated Base URL example

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
import type {
  BaseQueryFn,
  FetchArgs,
  FetchBaseQueryError,
} from '@reduxjs/toolkit/query/react'
import type { Post } from './types'
import { selectProjectId } from './projectSlice'
import type { RootState } from '../store'

const rawBaseQuery = fetchBaseQuery({
  baseUrl: 'www.my-cool-site.com/',
})

const dynamicBaseQuery: BaseQueryFn<
  string | FetchArgs,
  unknown,
  FetchBaseQueryError
> = async (args, api, extraOptions) => {
  const projectId = selectProjectId(api.getState() as RootState)
  // gracefully handle scenarios where data to generate the URL is missing
  if (!projectId) {
    return {
      error: {
        status: 400,
        statusText: 'Bad Request',
        data: 'No project ID received',
      },
    }
  }

  const urlEnd = typeof args === 'string' ? args : args.url
  // construct a dynamically generated portion of the url
  const adjustedUrl = `project/${projectId}/${urlEnd}`
```

```
  const adjustedArgs =
    typeof args === 'string' ? adjustedUrl : { ...args, url: adjustedUrl }
  // provide the amended url and other params to the raw base query
  return rawBaseQuery(adjustedArgs, api, extraOptions)
}

export const api = createApi({
  baseQuery: dynamicBaseQuery,
  endpoints: (builder) => ({
    getPosts: builder.query<Post[], void>({
      query: () => 'posts',
    }),
  }),
})

export const { useGetPostsQuery } = api

/*
  Using `useGetPostsQuery()` where a `projectId` of 500 is in the redux state
will result in
  a request being sent to www.my-cool-site.com/project/500/posts
*/
```

Examples - `transformResponse`

Unpacking deeply nested GraphQL data

GraphQL transformation example

```
import { createApi } from '@reduxjs/toolkit/query'
import { graphqlBaseQuery, gql } from './graphqlBaseQuery'

interface Post {
  id: number
  title: string
}

export const api = createApi({
  baseQuery: graphqlBaseQuery({
    baseUrl: '/graphql',
  }),
  endpoints: (builder) => ({
    getPosts: builder.query<Post[], void>({
      query: () => ({
        body: gql`
          query {
            posts {
              data {
                id
                title
              }
            }
          }
        `,
      }),
      transformResponse: (response: { posts: { data: Post[] } }) =>
        response.posts.data,
    }),
  }),
})
```

Normalizing data with `createEntityAdapter`

In the example below, `transformResponse` is used in conjunction with `createEntityAdapter` to normalize the data before storing it in the cache.

For a response such as:

```
[
  { id: 1, name: 'Harry' },
  { id: 2, name: 'Ron' },
  { id: 3, name: 'Hermione' },
]
```

The normalized cache data will be stored as:

```
{
  ids: [1, 3, 2],
  entities: {
    1: { id: 1, name: "Harry" },
    2: { id: 2, name: "Ron" },
    3: { id: 3, name: "Hermione" },
  }
}
```

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
import { createEntityAdapter } from '@reduxjs/toolkit'
import type { EntityState } from '@reduxjs/toolkit'

export interface Post {
  id: number
  name: string
}

const postsAdapter = createEntityAdapter<Post>({
  sortComparer: (a, b) => a.name.localeCompare(b.name),
})

export const api = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  endpoints: (build) => ({
    getPosts: build.query<EntityState<Post>, void>({
      query: () => `posts`,
      transformResponse(response: Post[]) {
        return postsAdapter.addMany(postsAdapter.getInitialState(), response)
      },
    }),
  }),
})

export const { useGetPostsQuery } = api
```

Examples - `queryFn`

Using a no-op queryFn

In certain scenarios, you may wish to have a `query` or `mutation` where sending a request or returning data is not relevant for the situation. Such a scenario would be to leverage the `invalidatesTags` property to force re-fetch specific `tags` that have been provided to the cache.

See also `providing errors to the cache` to see additional detail and an example for such a scenario to 'refetch errored queries'.

Using a no-op queryFn

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query'
import type { Post, User } from './types'

const api = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  tagTypes: ['Post', 'User'],
  endpoints: (build) => ({
    getPosts: build.query<Post[], void>({
      query: () => 'posts',
      providesTags: ['Post'],
    }),

    getUsers: build.query<User[], void>({
      query: () => 'users',
      providesTags: ['User'],
    }),

    refetchPostsAndUsers: build.mutation<null, void>({
      // The query is not relevant here, so a `null` returning `queryFn` is
used
      queryFn: () => ({ data: null }),
      // This mutation takes advantage of tag invalidation behaviour to
trigger
      // any queries that provide the 'Post' or 'User' tags to re-fetch if
the queries
      // are currently subscribed to the cached data
      invalidatesTags: ['Post', 'User'],
    }),
  }),
})
```

Streaming data with no initial request

RTK Query provides the ability for an endpoint to send an initial request for data, followed up with recurring streaming updates that perform further updates to the

cached data as the updates occur. However, the initial request is optional, and you may wish to use streaming updates without any initial request fired off.

In the example below, a `queryFn` is used to populate the cache data with an empty array, with no initial request sent. The array is later populated using streaming updates via the `onCacheEntryAdded` endpoint option, updating the cached data as it is received.

Streaming data with no initial request

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query'
import type { Message } from './types'

const api = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  tagTypes: ['Message'],
  endpoints: (build) => ({
    streamMessages: build.query<Message[], void>({
      // The query is not relevant here as the data will be provided via
streaming updates.
      // A queryFn returning an empty array is used, with contents being
populated via
      // streaming updates below as they are received.
      queryFn: () => ({ data: [] }),
      async onCacheEntryAdded(arg, { updateCachedData, cacheEntryRemoved }) {
        const ws = new WebSocket('ws://localhost:8080')
        // populate the array with messages as they are received from the
websocket
        ws.addEventListener('message', (event) => {
          updateCachedData((draft) => {
            draft.push(JSON.parse(event.data))
          })
        })
        await cacheEntryRemoved
        ws.close()
      },
    }),
  }),
})
```

Performing multiple requests with a single query

In the example below, a query is written to fetch all posts for a random user. This is done using a first request for a random user, followed by getting all posts for that user. Using `queryFn` allows the two requests to be included within a single query, avoiding having to chain that logic within component code.

Performing multiple requests with a single query

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query'
import type { FetchBaseQueryError } from '@reduxjs/toolkit/query'
import type { Post, User } from './types'

const api = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: '/ ' }),
  endpoints: (build) => ({
    getRandomUserPosts: build.query<Post, void>({
      async queryFn(_arg, _queryApi, _extraOptions, fetchWithBQ) {
        // get a random user
        const randomResult = await fetchWithBQ('users/random')
        if (randomResult.error)
          return { error: randomResult.error as FetchBaseQueryError }
        const user = randomResult.data as User
        const result = await fetchWithBQ(`user/${user.id}/posts`)
        return result.data
          ? { data: result.data as Post }
          : { error: result.error as FetchBaseQueryError }
      },
    }),
  }),
})
```

*Last updated on **Nov 27, 2022***

Usage Without React Hooks

Like the Redux core and Redux Toolkit, RTK Query's primary functionality is UI-agnostic and can be used with any UI layer. RTK Query also includes a version of `createApi` designed specifically for use with React, which automatically generates React hooks.

While React hooks are the primary way that the majority of users are expected to be using RTK Query, the library itself uses plain JS logic and can be used both with React Class components, and independent of React itself.

This page documents how to interact with RTK Query when used without React Hooks, in order to make proper use of RTK Query `cache behavior`.

Adding a subscription

Cache subscriptions are used to tell RTK Query that it needs to fetch data for an endpoint. A subscription for an endpoint can be added by dispatching the result of the `initiate` thunk action creator attached to a query endpoint.

With React hooks, this behavior is instead handled within `useQuery`, `useQuerySubscription`, `useLazyQuery`, and `useLazyQuerySubscription`.

Subscribing to cached data

```
dispatch(api.endpoints.getPosts.initiate())
```

Removing a subscription

Removing a cache subscription is necessary for RTK Query to identify that cached data is no longer required. This allows RTK Query to clean up and remove old cache data.

The result of dispatching the `initiate` thunk action creator of a query endpoint is an object with an `unsubscribe` property. This property is a function that when called, will remove the corresponding cache subscription.

With React hooks, this behavior is instead handled within `useQuery`, `useQuerySubscription`, `useLazyQuery`, and `useLazyQuerySubscription`.

Unsubscribing from cached data

```
// Adding a cache subscription
const result = dispatch(api.endpoints.getPosts.initiate())

// Removing the corresponding cache subscription
result.unsubscribe()
```

Accessing cached data & request status

Accessing cache data and request status information can be performed using the `select` function property of a query endpoint to create a selector and call that with the Redux state. This provides a snapshot of the cache data and request status information at the time it is called.

CAUTION

The `endpoint.select()` function creates a *new* selector instance - it isn't the actual selector function itself!

With React hooks, this behaviour is instead handled within `useQuery`, `useQueryState`, and `useLazyQuery`.

Accessing cached data & request status

```
const result = api.endpoints.getPosts.select()(state)
const { data, status, error } = result
```

Note that unlike the auto-generated query hooks, derived booleans such as `isLoading`, `isFetching`, `isSuccess` are not available here. The raw `status` enum is provided instead.

Performing mutations

Mutations are used in order to update data on the server. Mutations can be performed by dispatching the result of the `initiate` thunk action creator attached to a mutation endpoint.

With React hooks, this behavior is instead handled within `useMutation`.

Triggering a mutation endpoint

```
dispatch(api.endpoints.addPost.initiate({ name: 'foo' }))
```

*Last updated on **Jun 19, 2021***

Migrating to RTK Query

Overview

The most common use case for side effects in Redux apps is fetching data. Redux apps typically use a tool like thunks, sagas, or observables to make an AJAX request, and dispatch actions based on the results of the request. Reducers then listen for those actions to manage loading state and cache the fetched data.

RTK Query is purpose-built to solve the use case of data fetching. While it can't replace all of the situations where you'd use thunks or other side effects approaches, **using RTK Query should eliminate the need for most of that hand-written side effects logic**.

RTK Query is expected to cover a lot of overlapping behaviour that users may have previously used `createAsyncThunk` for, including caching purposes, and request lifecycle management (e.g. `isUninitialized`, `isLoading`, `isError` states).

In order to migrate data-fetching features from existing Redux tools to RTK Query, the appropriate endpoints should be added to an RTK Query API slice, and the previous feature code deleted. This generally will not include much common code kept between the two, as the tools work differently and one will replace the other.

If you're looking to get started with RTK Query from scratch, you may also wish to see `RTK Query Quick Start`.

Example - Migrating data-fetching logic from Redux Toolkit to RTK Query

A common method used to implement simple, cached, data-fetching logic with Redux is to set up a slice using `createSlice`, with state containing the associated `data` and `status` for a query, using `createAsyncThunk` to handle the asynchronous request lifecycles. Below we will explore an example of such an implementation, and how we can later go about migrating that code to use RTK Query instead.

NOTE

> RTK Query also provides many more features than what is created with the thunk example shown below. The example is only intended to demonstrate how the particular implementation could be replaced with RTK Query.

Design specifications

For our example, the design specifications required for the tool are as follows:

- Provide a hook to fetch data for a `pokemon` using the api: https://pokeapi.co/api/v2/pokemon/bulbasaur, where bulbasaur can be any pokemon name
- A request for any given name should only be sent if it hasn't already done so for the session
- The hook should provide us with the current status of the request for the supplied pokemon name; whether it is in an 'uninitialized', 'pending', 'fulfilled', or 'rejected' state
- The hook should provide us with the current data for the supplied pokemon name

With the above specifications in mind, lets first look at an overview of how this could be implemented traditionally using `createAsyncThunk` combined with `createSlice`.

Implementation using `createSlice` & `createAsyncThunk`

Slice file

The three snippets below make up our slice file. This file is concerned with managing our asynchronous request lifecycles, as well as storing our data & request statuses for a given pokemon name.

Thunk action creator

Below we create a thunk action creator using `createAsyncThunk` in order to manage asynchronous request lifecycles. This will be accessible within components & hooks to be dispatched, in order to fire off a request for some pokemon data. `createAsyncThunk` itself will handle dispatching lifecycle methods for our request: `pending`, `fulfilled`, and `rejected`, which we will handle within our slice.

src/services/pokemonSlice.ts - Thunk Action Creator

```
import { createAsyncThunk, createSlice } from '@reduxjs/toolkit'
import type { Pokemon } from './types'
import type { RootState } from '../store'

export const fetchPokemonByName = createAsyncThunk<Pokemon, string>(
  'pokemon/fetchByName',
  async (name, { rejectWithValue }) => {
    const response = await fetch(`https://pokeapi.co/api/v2/pokemon/${name}`)
    const data = await response.json()
    if (response.status < 200 || response.status >= 300) {
      return rejectWithValue(data)
    }
    return data
  }
)

// slice & selectors omitted
```

Slice

Below we have our `slice` created with `createSlice`. We have our reducers containing our request handling logic defined here, storing the appropriate 'status' and 'data' in our state based on the name we search with.

src/services/pokemonSlice.ts - slice logic

```
// imports & thunk action creator omitted

type RequestState = 'pending' | 'fulfilled' | 'rejected'

export const pokemonSlice = createSlice({
  name: 'pokemon',
  initialState: {
    dataByName: {} as Record<string, Pokemon | undefined>,
    statusByName: {} as Record<string, RequestState | undefined>,
  },
  reducers: {},
  extraReducers: (builder) => {
    // When our request is pending:
    // - store the 'pending' state as the status for the corresponding
pokemon name
    builder.addCase(fetchPokemonByName.pending, (state, action) => {
      state.statusByName[action.meta.arg] = 'pending'
    })
    // When our request is fulfilled:
    // - store the 'fulfilled' state as the status for the corresponding
pokemon name
    // - and store the received payload as the data for the corresponding
pokemon name
    builder.addCase(fetchPokemonByName.fulfilled, (state, action) => {
      state.statusByName[action.meta.arg] = 'fulfilled'
      state.dataByName[action.meta.arg] = action.payload
    })
    // When our request is rejected:
    // - store the 'rejected' state as the status for the corresponding
pokemon name
    builder.addCase(fetchPokemonByName.rejected, (state, action) => {
```

```
      state.statusByName[action.meta.arg] = 'rejected'
    })
  },
})

// selectors omitted
```

Selectors

Below we have our selectors defined, allowing us to later access the appropriate status & data for any given pokemon name.

src/services/pokemonSlice.ts - selectors

```
// imports, thunk action creator & slice omitted

export const selectStatusByName = (state: RootState, name: string) =>
  state.pokemon.statusByName[name]
export const selectDataByName = (state: RootState, name: string) =>
  state.pokemon.dataByName[name]
```

Store

In our `store` for our app, we include the corresponding reducer from our slice under the `pokemon` branch in our state tree. This lets our store handle the appropriate actions for our requests we will dispatch when running the app, using the logic defined previously.

src/services/store.ts

```
import { configureStore } from '@reduxjs/toolkit'
import { pokemonSlice } from './services/pokemonSlice'

export const store = configureStore({
  reducer: {
    pokemon: pokemonSlice.reducer,
  },
})

export type RootState = ReturnType<typeof store.getState>
```

In order to have the store accessible within our app, we will wrap our `App` component with a `Provider` component from `react-redux`.

src/index.ts

```
import { render } from 'react-dom'
import { Provider } from 'react-redux'

import App from './App'
import { store } from './store'

const rootElement = document.getElementById('root')
render(
```

```
  <Provider store={store}>
    <App />
  </Provider>,
  rootElement
)
```

Custom hook

Below we create a hook to manage sending our request at the appropriate time, as well as obtaining the appropriate data & status from the store. `useDispatch` and `useSelector` are used from `react-redux` in order to communicate with the Redux store. At the end of our hook, we return the information in a neat, packaged object to be accessed in components.

src/hooks.ts

```
import { useEffect } from 'react'
import { useSelector } from 'react-redux'
import { useAppDispatch } from './store'
import type { RootState } from './store'
import {
  fetchPokemonByName,
  selectStatusByName,
  selectDataByName,
} from './services/pokemonSlice'

export function useGetPokemonByNameQuery(name: string) {
  const dispatch = useAppDispatch()
  // select the current status from the store state for the provided name
  const status = useSelector((state: RootState) =>
    selectStatusByName(state, name)
  )
  // select the current data from the store state for the provided name
  const data = useSelector((state: RootState) => selectDataByName(state,
name))
  useEffect(() => {
    // upon mount or name change, if status is uninitialized, send a request
    // for the pokemon name
    if (status === undefined) {
      dispatch(fetchPokemonByName(name))
    }
  }, [status, name, dispatch])

  // derive status booleans for ease of use
  const isUninitialized = status === undefined
  const isLoading = status === 'pending' || status === undefined
  const isError = status === 'rejected'
  const isSuccess = status === 'fulfilled'

  // return the import data for the caller of the hook to use
  return { data, isUninitialized, isLoading, isError, isSuccess }
}
```

Using the custom hook

Our code above meets all of the design specifications, so let's use it! Below we can see how the hook can be called in a component, and return the relevant data & status booleans.

Our implementation below provides the following behaviour in the component:

- When our component is mounted, if a request for the provided pokemon name has not already been sent for the session, send the request off
- The hook always provides the latest received `data` when available, as well as the request status
  booleans `isUninitialized`, `isPending`, `isFulfilled` & `isRejected` in order to determine the current UI at any given moment as a function of our state.

src/App.tsx

```
import * as React from 'react'
import { useGetPokemonByNameQuery } from './hooks'

export default function App() {
  const { data, isError, isLoading } = useGetPokemonByNameQuery('bulbasaur')

  return (
    <div className="App">
      {isError ? (
        <>Oh no, there was an error</>
      ) : isLoading ? (
        <>Loading...</>
      ) : data ? (
        <>
          <h3>{data.species.name}</h3>
          <img src={data.sprites.front_shiny} alt={data.species.name} />
        </>
      ) : null}
    </div>
  )
}
```

A runnable example of the above code can be seen below:

Converting to RTK Query

Our implementation above *does* work perfectly fine for the requirements specified, however, extending the code to include further endpoints could involve a lot of repetition. It also has some certain limitations that may not be immediately obvious. For example, multiple components rendering simultaneously calling our hook would each send off a request for bulbasaur at the same time!

Below we will walk through how a lot of the boilerplate can be avoided by migrating the above code to use RTK Query instead. RTK Query will also handle many other situations for us, including de-duping requests on a more granular level to prevent sending unnecessary duplicate requests like that brought up above.

API Slice File

Our code below is for our API slice definition. This acts as our network API interface layer, and is created using `createApi`. This file will contain our endpoint definition, and `createApi` will provide us with an auto-generated hook which manages firing our request only when necessary, as well as providing us with request status lifecycle booleans.

This will completely cover our logic implemented above for the entire slice file, including the thunk, slice definition, selectors, *and* our custom hook!

src/services/api.ts

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
import type { Pokemon } from './types'

export const api = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: 'https://pokeapi.co/api/v2/' }),
  reducerPath: 'pokemonApi',
  endpoints: (build) => ({
    getPokemonByName: build.query<Pokemon, string>({
      query: (name) => `pokemon/${name}`,
    }),
  }),
})

export const { useGetPokemonByNameQuery } = api
```

Connecting the API slice to the store

Now that we have our API definition created, we need to hook it up to our store. In order to do that, we will need to use the `reducerPath` and `middleware` properties from our created `api`. This will allow the store to process the internal actions that the generated hook uses, allows the generated API logic to find the state correctly, and adds the logic for managing caching, invalidation, subscriptions, polling, and more.

src/store.ts

```
import { configureStore } from '@reduxjs/toolkit'
import { pokemonSlice } from './services/pokemonSlice'
import { api } from './services/api'

export const store = configureStore({
  reducer: {
    pokemon: pokemonSlice.reducer,
    [api.reducerPath]: api.reducer,
  },
  middleware: (gDM) => gDM().concat(api.middleware),
})

export type RootState = ReturnType<typeof store.getState>
```

Using our auto-generated hook

At this basic level, the usage of the auto-generated hook is identical to our custom hook! All we need to do is change our import path and we're good to go!

src/App.tsx

```
  import * as React from 'react'
- import { useGetPokemonByNameQuery } from './hooks'
+ import { useGetPokemonByNameQuery } from './services/api'

  export default function App() {
    const { data, isError, isLoading } =
useGetPokemonByNameQuery('bulbasaur')


    return (
      <div className="App">
        {isError ? (
          <>Oh no, there was an error</>
        ) : isLoading ? (
          <>Loading...</>
        ) : data ? (
          <>
            <h3>{data.species.name}</h3>
            <img src={data.sprites.front_shiny} alt={data.species.name} />
          </>
        ) : null}
      </div>
    )
  }
```

Cleaning up unused code

As mentioned previously, our `api` definition has replaced all of the logic that we implemented previously using `createAsyncThunk`, `createSlice`, and our custom hook definition.

Given that we're no longer using that slice any longer, we can remove the import and reducer from our store:

src/store.ts

```
  import { configureStore } from '@reduxjs/toolkit'
- import { pokemonSlice } from './services/pokemonSlice'
  import { api } from './services/api'


  export const store = configureStore({
    reducer: {
-     pokemon: pokemonSlice.reducer,
      [api.reducerPath]: api.reducer,
    },
    middleware: (gDM) => gDM().concat(api.middleware),
  })

  export type RootState = ReturnType<typeof store.getState>
```

We can also remove the *entire slice and hook files* completely!

```
- src/services/pokemonSlice.ts (-51 lines)
- src/hooks.ts (-34 lines)
```

createApi

`createApi` is the core of RTK Query's functionality. It allows you to define a set of "endpoints" that describe how to retrieve data from backend APIs and other async sources, including the configuration of how to fetch and transform that data. It generates an "API slice" structure that contains Redux logic (and optionally React hooks) that encapsulate the data fetching and caching process for you.

TIP

Typically, you should only have one API slice per base URL that your application needs to communicate with. For example, if your site fetches data from both `/api/posts` and `/api/users`, you would have a single API slice with `/api/` as the base URL, and separate endpoint definitions for `posts` and `users`. This allows you to effectively take advantage of automated re-fetching by defining tag relationships across endpoints.

For maintainability purposes, you may wish to split up endpoint definitions across multiple files, while still maintaining a single API slice which includes all of these endpoints. See code splitting for how you can use the `injectEndpoints` property to inject API endpoints from other files into a single API slice definition.

- TypeScript
- JavaScript

Example: src/services/pokemon.ts

```
// Need to use the React-specific entry point to allow generating React hooks
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
import type { Pokemon } from './types'

// Define a service using a base URL and expected endpoints
export const pokemonApi = createApi({
  reducerPath: 'pokemonApi',
  baseQuery: fetchBaseQuery({ baseUrl: 'https://pokeapi.co/api/v2/' }),
  endpoints: (builder) => ({
    getPokemonByName: builder.query<Pokemon, string>({
      query: (name) => `pokemon/${name}`,
    }),
  }),
})

// Export hooks for usage in function components, which are
// auto-generated based on the defined endpoints
export const { useGetPokemonByNameQuery } = pokemonApi
```

Parameters

`createApi` accepts a single configuration object parameter with the following options:

```
  baseQuery(args: InternalQueryArgs, api: BaseQueryApi, extraOptions?:
DefinitionExtraOptions): any;
  endpoints(build: EndpointBuilder<InternalQueryArgs, TagTypes>):
Definitions;
  extractRehydrationInfo?: (
    action: AnyAction,
    {
      reducerPath,
    }: {
      reducerPath: ReducerPath
    }
  ) =>
    | undefined
    | CombinedState<Definitions, TagTypes, ReducerPath>
  tagTypes?: readonly TagTypes[];
  reducerPath?: ReducerPath;
  serializeQueryArgs?: SerializeQueryArgs<InternalQueryArgs>;
  keepUnusedDataFor?: number; // value is in seconds
  refetchOnMountOrArgChange?: boolean | number; // value is in seconds
  refetchOnFocus?: boolean;
  refetchOnReconnect?: boolean;
```

`baseQuery`

The base query used by each endpoint if no `queryFn` option is specified. RTK Query exports a utility called fetchBaseQuery as a lightweight wrapper around `fetch` for common use-cases. See Customizing Queries if `fetchBaseQuery` does not handle your requirements.

baseQuery function arguments

- `args` - The return value of the `query` function for a given endpoint
- `api` - The `BaseQueryApi` object contains:
    - `signal` - An `AbortSignal` object that may be used to abort DOM requests and/or read whether the request is aborted.
    - `abort` - The `abort()` method of the AbortController attached to `signal`.
    - `dispatch` - The `store.dispatch` method for the corresponding Redux store
    - `getState` - A function that may be called to access the current store state
    - `extra` - Provided as thunk.extraArgument to the configureStore getDefaultMiddleware option.
    - `endpoint` - The name of the endpoint.
    - `type` - Type of request (`query` or `mutation`).
    - `forced` - Indicates if a query has been forced.

- `extraOptions` - The value of the optional `extraOptions` property provided for a given endpoint

baseQuery function signature

Base Query signature

```
export type BaseQueryFn<
  Args = any,
  Result = unknown,
  Error = unknown,
  DefinitionExtraOptions = {},
  Meta = {}
> = (
  args: Args,
  api: BaseQueryApi,
  extraOptions: DefinitionExtraOptions
) => MaybePromise<QueryReturnValue<Result, Error, Meta>>

export interface BaseQueryApi {
  signal: AbortSignal
  abort: (reason?: string) => void
  dispatch: ThunkDispatch<any, any, any>
  getState: () => unknown
  extra: unknown
  endpoint: string
  type: 'query' | 'mutation'
  forced?: boolean
}

export type QueryReturnValue<T = unknown, E = unknown, M = unknown> =
  | {
      error: E
      data?: undefined
      meta?: M
    }
  | {
      error?: undefined
      data: T
      meta?: M
    }
```

- TypeScript
- JavaScript

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

const api = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  endpoints: (build) => ({
    // ...endpoints
  }),
})
```

`endpoints`

Endpoints are just a set of operations that you want to perform against your server. You define them as an object using the builder syntax. There are two basic endpoint types: `query` and `mutation`.

See Anatomy of an endpoint for details on individual properties.

Query endpoint definition

Query endpoint definition

```
export type QueryDefinition<
  QueryArg,
  BaseQuery extends BaseQueryFn,
  TagTypes extends string,
  ResultType,
  ReducerPath extends string = string
> = {
  query(arg: QueryArg): BaseQueryArg<BaseQuery>

  /* either `query` or `queryFn` can be present, but not both simultaneously
*/
  queryFn(
    arg: QueryArg,
    api: BaseQueryApi,
    extraOptions: BaseQueryExtraOptions<BaseQuery>,
    baseQuery: (arg: Parameters<BaseQuery>[0]) => ReturnType<BaseQuery>
  ): MaybePromise<QueryReturnValue<ResultType, BaseQueryError<BaseQuery>>>

  /* transformResponse only available with `query`, not `queryFn` */
  transformResponse?(
    baseQueryReturnValue: BaseQueryResult<BaseQuery>,
    meta: BaseQueryMeta<BaseQuery>,
    arg: QueryArg
  ): ResultType | Promise<ResultType>

  /* transformErrorResponse only available with `query`, not `queryFn` */
  transformErrorResponse?(
    baseQueryReturnValue: BaseQueryError<BaseQuery>,
    meta: BaseQueryMeta<BaseQuery>,
    arg: QueryArg
  ): unknown

  extraOptions?: BaseQueryExtraOptions<BaseQuery>

  providesTags?: ResultDescription<
    TagTypes,
    ResultType,
    QueryArg,
    BaseQueryError<BaseQuery>
  >

  keepUnusedDataFor?: number

  onQueryStarted?(
    arg: QueryArg,
    {
      dispatch,
      getState,
```

```
      extra,
      requestId,
      queryFulfilled,
      getCacheEntry,
      updateCachedData, // available for query endpoints only
    }: QueryLifecycleApi
  ): Promise<void>

  onCacheEntryAdded?(
    arg: QueryArg,
    {
      dispatch,
      getState,
      extra,
      requestId,
      cacheEntryRemoved,
      cacheDataLoaded,
      getCacheEntry,
      updateCachedData, // available for query endpoints only
    }: QueryCacheLifecycleApi
  ): Promise<void>
}
```

Mutation endpoint definition

Mutation endpoint definition

```
export type MutationDefinition<
  QueryArg,
  BaseQuery extends BaseQueryFn,
  TagTypes extends string,
  ResultType,
  ReducerPath extends string = string,
  Context = Record<string, any>
> = {
  query(arg: QueryArg): BaseQueryArg<BaseQuery>

  /* either `query` or `queryFn` can be present, but not both simultaneously
*/
  queryFn(
    arg: QueryArg,
    api: BaseQueryApi,
    extraOptions: BaseQueryExtraOptions<BaseQuery>,
    baseQuery: (arg: Parameters<BaseQuery>[0]) => ReturnType<BaseQuery>
  ): MaybePromise<QueryReturnValue<ResultType, BaseQueryError<BaseQuery>>>

  /* transformResponse only available with `query`, not `queryFn` */
  transformResponse?(
    baseQueryReturnValue: BaseQueryResult<BaseQuery>,
    meta: BaseQueryMeta<BaseQuery>,
    arg: QueryArg
  ): ResultType | Promise<ResultType>

  /* transformErrorResponse only available with `query`, not `queryFn` */
  transformErrorResponse?(
    baseQueryReturnValue: BaseQueryError<BaseQuery>,
    meta: BaseQueryMeta<BaseQuery>,
    arg: QueryArg
  ): unknown

  extraOptions?: BaseQueryExtraOptions<BaseQuery>
```

```
  invalidatesTags?: ResultDescription<TagTypes, ResultType, QueryArg>

  onQueryStarted?(
    arg: QueryArg,
    {
      dispatch,
      getState,
      extra,
      requestId,
      queryFulfilled,
      getCacheEntry,
    }: MutationLifecycleApi
  ): Promise<void>

  onCacheEntryAdded?(
    arg: QueryArg,
    {
      dispatch,
      getState,
      extra,
      requestId,
      cacheEntryRemoved,
      cacheDataLoaded,
      getCacheEntry,
    }: MutationCacheLifecycleApi
  ): Promise<void>
}
```

How endpoints get used

When defining a key like `getPosts` as shown below, it's important to know that this name will become exportable from `api` and be able to referenced under `api.endpoints.getPosts.useQuery()`, `api.endpoints.getPosts.initiate()` and `api.endpoints.getPosts.select()`. The same thing applies to `mutation`s but they reference `useMutation` instead of `useQuery`.

- TypeScript
- JavaScript

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
interface Post {
  id: number
  name: string
}
type PostsResponse = Post[]

const api = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  tagTypes: ['Posts'],
  endpoints: (build) => ({
    getPosts: build.query<PostsResponse, void>({
      query: () => 'posts',
      providesTags: (result) =>
        result ? result.map(({ id }) => ({ type: 'Posts', id })) : [],
    }),
    addPost: build.mutation<Post, Partial<Post>>({
```

```
      query: (body) => ({
        url: `posts`,
        method: 'POST',
        body,
      }),
      invalidatesTags: ['Posts'],
    }),
  }),
})

// Auto-generated hooks
export const { useGetPostsQuery, useAddPostMutation } = api

// Possible exports
export const { endpoints, reducerPath, reducer, middleware } = api
// reducerPath, reducer, middleware are only used in store configuration
// endpoints will have:
// endpoints.getPosts.initiate(), endpoints.getPosts.select(),
endpoints.getPosts.useQuery()
// endpoints.addPost.initiate(), endpoints.addPost.select(),
endpoints.addPost.useMutation()
// see `createApi` overview for _all exports_
```

`extractRehydrationInfo`

A function that is passed every dispatched action. If this returns something other than `undefined`, that return value will be used to rehydrate fulfilled & errored queries.

- TypeScript
- JavaScript

next-redux-wrapper rehydration example

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
import { HYDRATE } from 'next-redux-wrapper'

export const api = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  extractRehydrationInfo(action, { reducerPath }) {
    if (action.type === HYDRATE) {
      return action.payload[reducerPath]
    }
  },
  endpoints: (build) => ({
    // omitted
  }),
})
```

See also Server Side Rendering and Persistence and Rehydration.

`tagTypes`

An array of string tag type names. Specifying tag types is optional, but you should define them so that they can be used for caching and invalidation. When

defining a tag type, you will be able to provide them with `providesTags` and invalidate them with `invalidatesTags` when configuring endpoints.

- TypeScript
- JavaScript

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

const api = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  tagTypes: ['Post', 'User'],
  endpoints: (build) => ({
    // ...endpoints
  }),
})
```

`reducerPath`

The `reducerPath` is a *unique* key that your service will be mounted to in your store. If you call `createApi` more than once in your application, you will need to provide a unique value each time. Defaults to `'api'`.

- TypeScript
- JavaScript

apis.js

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

const apiOne = createApi({
  reducerPath: 'apiOne',
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  endpoints: (builder) => ({
    // ...endpoints
  }),
})

const apiTwo = createApi({
  reducerPath: 'apiTwo',
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  endpoints: (builder) => ({
    // ...endpoints
  }),
})
```

`serializeQueryArgs`

Accepts a custom function if you have a need to change the creation of cache keys for any reason.

By default, this function will take the query arguments, sort object keys where applicable, stringify the result, and concatenate it with the endpoint name. This

creates a cache key based on the combination of arguments + endpoint name (ignoring object key order), such that calling any given endpoint with the same arguments will result in the same cache key.

`keepUnusedDataFor`

Defaults to `60` *(this value is in seconds)*. This is how long RTK Query will keep your data cached for **after** the last component unsubscribes. For example, if you query an endpoint, then unmount the component, then mount another component that makes the same request within the given time frame, the most recent value will be served from the cache.

- TypeScript
- JavaScript

keepUnusedDataFor example

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
interface Post {
  id: number
  name: string
}
type PostsResponse = Post[]

const api = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  endpoints: (build) => ({
    getPosts: build.query<PostsResponse, void>({
      query: () => 'posts',
      keepUnusedDataFor: 5,
    }),
  }),
})
```

`refetchOnMountOrArgChange`

Defaults to `false`. This setting allows you to control whether if a cached result is already available RTK Query will only serve a cached result, or if it should `refetch` when set to `true` or if an adequate amount of time has passed since the last successful query result.

- `false` - Will not cause a query to be performed *unless* it does not exist yet.
- `true` - Will always refetch when a new subscriber to a query is added. Behaves the same as calling the `refetch` callback or passing `forceRefetch: true` in the action creator.
- `number` - **Value is in seconds**. If a number is provided and there is an existing query in the cache, it will compare the current time vs the last fulfilled timestamp, and only refetch if enough time has elapsed.

If you specify this option alongside `skip: true`, this **will not be evaluated** until `skip` is false.

NOTE

You can set this globally in `createApi`, but you can also override the default value and have more granular control by passing `refetchOnMountOrArgChange` to each individual hook call or similarly by passing `forceRefetch: true` when dispatching the `initiate` action.

`refetchOnFocus`

Defaults to `false`. This setting allows you to control whether RTK Query will try to refetch all subscribed queries after the application window regains focus.

If you specify this option alongside `skip: true`, this **will not be evaluated** until `skip` is false.

Note: requires `setupListeners` to have been called.

NOTE

You can set this globally in `createApi`, but you can also override the default value and have more granular control by passing `refetchOnFocus` to each individual hook call or when dispatching the `initiate` action.

If you specify `track: false` when manually dispatching queries, RTK Query will not be able to automatically refetch for you.

`refetchOnReconnect`

Defaults to `false`. This setting allows you to control whether RTK Query will try to refetch all subscribed queries after regaining a network connection.

If you specify this option alongside `skip: true`, this **will not be evaluated** until `skip` is false.

Note: requires `setupListeners` to have been called.

NOTE

You can set this globally in `createApi`, but you can also override the default value and have more granular control by passing `refetchOnReconnect` to each individual hook call or when dispatching the `initiate` action.

If you specify `track: false` when manually dispatching queries, RTK Query will not be able to automatically refetch for you.

Anatomy of an endpoint

`query`

(required if no `queryFn` provided)

query signature

```
export type query = <QueryArg>(
  arg: QueryArg
) => string | Record<string, unknown>

// with `fetchBaseQuery`
export type query = <QueryArg>(arg: QueryArg) => string | FetchArgs
```

`query` can be a function that returns either a `string` or an `object` which is passed to your `baseQuery`. If you are using fetchBaseQuery, this can return either a `string` or an `object` of properties in `FetchArgs`. If you use your own custom `baseQuery`, you can customize this behavior to your liking.

- TypeScript
- JavaScript

query example

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
interface Post {
  id: number
  name: string
}
type PostsResponse = Post[]

const api = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  tagTypes: ['Post'],
  endpoints: (build) => ({
    getPosts: build.query<PostsResponse, void>({
      query: () => 'posts',
    }),
    addPost: build.mutation<Post, Partial<Post>>({
      query: (body) => ({
        url: `posts`,
        method: 'POST',
        body,
      }),
      invalidatesTags: [{ type: 'Post', id: 'LIST' }],
    }),
  }),
})
```

`queryFn`

(required if no `query` provided)

Can be used in place of `query` as an inline function that bypasses `baseQuery` completely for the endpoint.

Called with the same arguments as `baseQuery`, as well as the provided `baseQuery` function itself. It is expected to return an object with either a `data` or `error` property, or a promise that resolves to return such an object.

See also Customizing queries with queryFn.

queryFn signature

```
queryFn(
  arg: QueryArg,
  api: BaseQueryApi,
  extraOptions: BaseQueryExtraOptions<BaseQuery>,
  baseQuery: (arg: Parameters<BaseQuery>[0]) => ReturnType<BaseQuery>
): MaybePromise<
| {
    error: BaseQueryError<BaseQuery>
    data?: undefined
  }
| {
    error?: undefined
    data: ResultType
  }
>

export interface BaseQueryApi {
  signal: AbortSignal
  dispatch: ThunkDispatch<any, any, any>
  getState: () => unknown
}
```

queryFn function arguments

- `args` - The argument provided when the query itself is called
- `api` - The `BaseQueryApi` object, containing `signal`, `dispatch` and `getState` properties
  - `signal` - An `AbortSignal` object that may be used to abort DOM requests and/or read whether the request is aborted.
  - `dispatch` - The `store.dispatch` method for the corresponding Redux store
  - `getState` - A function that may be called to access the current store state
- `extraOptions` - The value of the optional `extraOptions` property provided for the endpoint

- `baseQuery` - The `baseQuery` function provided to the api itself

- TypeScript
- JavaScript

Basic queryFn example

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
interface Post {
  id: number
  name: string
}
type PostsResponse = Post[]

const api = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  endpoints: (build) => ({
    getPosts: build.query<PostsResponse, void>({
      query: () => 'posts',
    }),
    flipCoin: build.query<'heads' | 'tails', void>({
      queryFn(arg, queryApi, extraOptions, baseQuery) {
        const randomVal = Math.random()
        if (randomVal < 0.45) {
          return { data: 'heads' }
        }
        if (randomVal < 0.9) {
          return { data: 'tails' }
        }
        return {
          error: {
            status: 500,
            statusText: 'Internal Server Error',
            data: "Coin landed on it's edge!",
          },
        }
      },
    }),
  }),
})
```

`transformResponse`

(optional, not applicable with `queryFn`)

A function to manipulate the data returned by a query or mutation.

In some cases, you may want to manipulate the data returned from a query before you put it in the cache. In this instance, you can take advantage of `transformResponse`.

See also Customizing query responses with `transformResponse`

Unpack a deeply nested collection

```
transformResponse: (response, meta, arg) =>
  response.some.deeply.nested.collection
```

`transformErrorResponse`

(optional, not applicable with `queryFn`)

A function to manipulate the data returned by a failed query or mutation.

In some cases, you may want to manipulate the error returned from a query before you put it in the cache. In this instance, you can take advantage of `transformErrorResponse`.

See also Customizing query responses with `transformErrorResponse`

Unpack a deeply nested error object

```
transformErrorResponse: (response, meta, arg) =>
  response.data.some.deeply.nested.errorObject
```

`extraOptions`

(optional)

Passed as the third argument to the supplied `baseQuery` function

`providesTags`

(optional, only for query endpoints)

Used by `query` endpoints. Determines which 'tag' is attached to the cached data returned by the query. Expects an array of tag type strings, an array of objects of tag types with ids, or a function that returns such an array.

1. `['Post']` - equivalent to `2`
2. `[{ type: 'Post' }]` - equivalent to `1`
3. `[{ type: 'Post', id: 1 }]`
4. `(result, error, arg) => ['Post']` - equivalent to `5`
5. `(result, error, arg) => [{ type: 'Post' }]` - equivalent to `4`
6. `(result, error, arg) => [{ type: 'Post', id: 1 }]`

See also Providing cache data.

- TypeScript
- JavaScript

providesTags example

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
interface Post {
  id: number
  name: string
}
type PostsResponse = Post[]

const api = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  tagTypes: ['Posts'],
  endpoints: (build) => ({
    getPosts: build.query<PostsResponse, void>({
      query: () => 'posts',
      providesTags: (result) =>
        result
          ? [
              ...result.map(({ id }) => ({ type: 'Posts' as const, id })),
              { type: 'Posts', id: 'LIST' },
            ]
          : [{ type: 'Posts', id: 'LIST' }],
    }),
  }),
})
```

`invalidatesTags`

(optional, only for mutation endpoints)

Used by `mutation` endpoints. Determines which cached data should be either re-fetched or removed from the cache. Expects the same shapes as `providesTags`.

See also Invalidating cache data.

- TypeScript
- JavaScript

invalidatesTags example

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
interface Post {
  id: number
  name: string
}
type PostsResponse = Post[]

const api = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  tagTypes: ['Posts'],
  endpoints: (build) => ({
    getPosts: build.query<PostsResponse, void>({
      query: () => 'posts',
      providesTags: (result) =>
        result
          ? [
              ...result.map(({ id }) => ({ type: 'Posts' as const, id })),
              { type: 'Posts', id: 'LIST' },
            ]
          : [{ type: 'Posts', id: 'LIST' }],
```

```
    }),
    addPost: build.mutation<Post, Partial<Post>>({
      query(body) {
        return {
          url: `posts`,
          method: 'POST',
          body,
        }
      },
      invalidatesTags: [{ type: 'Posts', id: 'LIST' }],
    }),
  }),
})
```

`keepUnusedDataFor`

(optional, only for query endpoints)

Overrides the api-wide definition of `keepUnusedDataFor` for this endpoint only.a

Defaults to `60` *(this value is in seconds)*. This is how long RTK Query will keep your data cached for **after** the last component unsubscribes. For example, if you query an endpoint, then unmount the component, then mount another component that makes the same request within the given time frame, the most recent value will be served from the cache.

- TypeScript
- JavaScript

keepUnusedDataFor example

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react'
interface Post {
  id: number
  name: string
}
type PostsResponse = Post[]

const api = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  endpoints: (build) => ({
    getPosts: build.query<PostsResponse, void>({
      query: () => 'posts',
      keepUnusedDataFor: 5,
    }),
  }),
})
```

`serializeQueryArgs`

(optional, only for query endpoints)

Can be provided to return a custom cache key value based on the query arguments.

This is primarily intended for cases where a non-serializable value is passed as part of the query arg object and should be excluded from the cache key. It may also be used for cases where an endpoint should only have a single cache entry, such as an infinite loading / pagination implementation.

Unlike the `createApi` version which can *only* return a string, this per-endpoint option can also return an an object, number, or boolean. If it returns a string, that value will be used as the cache key directly. If it returns an object / number / boolean, that value will be passed to the built-in `defaultSerializeQueryArgs`. This simplifies the use case of stripping out args you don't want included in the cache key.

- TypeScript
- JavaScript

serializeQueryArgs : exclude value

```
import {
  createApi,
  fetchBaseQuery,
  defaultSerializeQueryArgs,
} from '@reduxjs/toolkit/query/react'
interface Post {
  id: number
  name: string
}

interface MyApiClient {
  fetchPost: (id: string) => Promise<Post>
}

createApi({
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  endpoints: (build) => ({
    // Example: an endpoint with an API client passed in as an argument,
    // but only the item ID should be used as the cache key
    getPost: build.query<Post, { id: string; client: MyApiClient }>({
      queryFn: async ({ id, client }) => {
        const post = await client.fetchPost(id)
        return { data: post }
      },
      serializeQueryArgs: ({ queryArgs, endpointDefinition, endpointName })
=> {
        const { id } = queryArgs
        // This can return a string, an object, a number, or a boolean.
        // If it returns an object, number or boolean, that value
        // will be serialized automatically via `defaultSerializeQueryArgs`
        return { id } // omit `client` from the cache key

        // Alternately, you can use `defaultSerializeQueryArgs` yourself:
        // return defaultSerializeQueryArgs({
        //   endpointName,
        //   queryArgs: { id },
        //   endpointDefinition
        // })
        // Or  create and return a string yourself:
```

```
        // return `getPost(${id})`
      },
    }),
  }),
})
```

`merge`

(optional, only for query endpoints)

Can be provided to merge an incoming response value into the current cache data. If supplied, no automatic structural sharing will be applied - it's up to you to update the cache appropriately.

Since RTKQ normally replaces cache entries with the new response, you will usually need to use this with the `serializeQueryArgs` or `forceRefetch` options to keep an existing cache entry so that it can be updated.

Since this is wrapped with Immer, you , you may either mutate the `currentCacheValue` directly, or return a new value, but *not* both at once.

Will only be called if the existing `currentCacheData` is *not* `undefined` - on first response, the cache entry will just save the response data directly.

Useful if you don't want a new request to completely override the current cache value, maybe because you have manually updated it from another source and don't want those updates to get lost.

- TypeScript
- JavaScript

merge: pagination

```
import {
  createApi,
  fetchBaseQuery,
  defaultSerializeQueryArgs,
} from '@reduxjs/toolkit/query/react'
interface Post {
  id: number
  name: string
}

createApi({
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  endpoints: (build) => ({
    listItems: build.query<string[], number>({
      query: (pageNumber) => `/listItems?page=${pageNumber}`,
      // Only have one cache entry because the arg always maps to one string
      serializeQueryArgs: ({ endpointName }) => {
        return endpointName
      },
```

```
      // Always merge incoming data to the cache entry
      merge: (currentCache, newItems) => {
        currentCache.push(...newItems)
      },
      // Refetch when the page arg changes
      forceRefetch({ currentArg, previousArg }) {
        return currentArg !== previousArg
      },
    }),
  }),
})
```

`forceRefetch`

(optional, only for query endpoints)

forceRefetch signature

```
type forceRefetch = (params: {
  currentArg: QueryArg | undefined
  previousArg: QueryArg | undefined
  state: RootState<any, any, string>
  endpointState?: QuerySubState<any>
}) => boolean
```

Check to see if the endpoint should force a refetch in cases where it normally wouldn't. This is primarily useful for "infinite scroll" / pagination use cases where RTKQ is keeping a single cache entry that is added to over time, in combination with `serializeQueryArgs` returning a fixed cache key and a `merge` callback set to add incoming data to the cache entry each time.

- TypeScript
- JavaScript

forceRefresh: pagination

```
import {
  createApi,
  fetchBaseQuery,
  defaultSerializeQueryArgs,
} from '@reduxjs/toolkit/query/react'
interface Post {
  id: number
  name: string
}

createApi({
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  endpoints: (build) => ({
    listItems: build.query<string[], number>({
      query: (pageNumber) => `/listItems?page=${pageNumber}`,
      // Only have one cache entry because the arg always maps to one string
      serializeQueryArgs: ({ endpointName }) => {
        return endpointName
      },
      // Always merge incoming data to the cache entry
```

```
      merge: (currentCache, newItems) => {
        currentCache.push(...newItems)
      },
      // Refetch when the page arg changes
      forceRefetch({ currentArg, previousArg }) {
        return currentArg !== previousArg
      },
    }),
  }),
})
```

`onQueryStarted`

(optional)

Available to both queries and mutations.

A function that is called when you start each individual query or mutation. The function is called with a lifecycle api object containing properties such as `queryFulfilled`, allowing code to be run when a query is started, when it succeeds, and when it fails (i.e. throughout the lifecycle of an individual query/mutation call).

Can be used in `mutations` for optimistic updates.

Lifecycle API properties

- `dispatch` - The dispatch method for the store.
- `getState` - A method to get the current state for the store.
- `extra` - `extra` as provided as `thunk.extraArgument` to the `configureStore` `getDefaultMiddleware` option.
- `requestId` - A unique ID generated for the query/mutation.
- `queryFulfilled` - A Promise that will resolve with a `data` property (the transformed query result), and a `meta` property (`meta` returned by the `baseQuery`). If the query fails, this Promise will reject with the error. This allows you to `await` for the query to finish.
- `getCacheEntry` - A function that gets the current value of the cache entry.
- `updateCachedData` *(query endpoints only)* - A function that accepts a 'recipe' callback specifying how to update the data for the corresponding cache at the time it is called. This uses `immer` internally, and updates can be written 'mutably' while safely producing the next immutable state.

Mutation onQueryStarted signature

```
async function onQueryStarted(
  arg: QueryArg,
  {
    dispatch,
    getState,
    extra,
```

```
    requestId,
    queryFulfilled,
    getCacheEntry,
  }: MutationLifecycleApi
): Promise<void>
```

Query onQueryStarted signature

```
async function onQueryStarted(
  arg: QueryArg,
  {
    dispatch,
    getState,
    extra,
    requestId,
    queryFulfilled,
    getCacheEntry,
    updateCachedData, // available for query endpoints only
  }: QueryLifecycleApi
): Promise<void>
```

- TypeScript
- JavaScript

onQueryStarted query lifecycle example

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query'
import { messageCreated } from './notificationsSlice'

export interface Post {
  id: number
  name: string
}

const api = createApi({
  baseQuery: fetchBaseQuery({
    baseUrl: '/',
  }),
  endpoints: (build) => ({
    getPost: build.query<Post, number>({
      query: (id) => `post/${id}`,
      async onQueryStarted(id, { dispatch, queryFulfilled }) {
        // `onStart` side-effect
        dispatch(messageCreated('Fetching post...'))
        try {
          const { data } = await queryFulfilled
          // `onSuccess` side-effect
          dispatch(messageCreated('Post received!'))
        } catch (err) {
          // `onError` side-effect
          dispatch(messageCreated('Error fetching post!'))
        }
      },
    }),
  }),
})
```

`onCacheEntryAdded`

(optional)

Available to both queries and mutations.

A function that is called when a new cache entry is added, i.e. when a new subscription for the endpoint + query parameters combination is created. The function is called with a lifecycle api object containing properties such as `cacheDataLoaded` & `cacheDataRemoved`, allowing code to be run when a cache entry is added, when cache data is loaded, and when the cache entry is removed (i.e. throughout the lifecycle of a cache entry).

Can be used for streaming updates.

Cache Lifecycle API properties

- `dispatch` - The dispatch method for the store.
- `getState` - A method to get the current state for the store.
- `extra` - `extra` as provided as `thunk.extraArgument` to the `configureStore` `getDefaultMiddleware` option.
- `requestId` - A unique ID generated for the cache entry.
- `cacheEntryRemoved` - A Promise that allows you to wait for the point in time when the cache entry has been removed from the cache, by not being used/subscribed to any more in the application for too long or by dispatching `api.util.resetApiState`.
- `cacheDataLoaded` - A Promise that will resolve with the first value for this cache key. This allows you to `await` until an actual value is in the cache. Note: If the cache entry is removed from the cache before any value has ever been resolved, this Promise will reject with `new Error('Promise never resolved before cacheEntryRemoved.')` to prevent memory leaks. You can just re-throw that error (or not handle it at all) - it will be caught outside of `cacheEntryAdded`.
- `getCacheEntry` - A function that gets the current value of the cache entry.
- `updateCachedData` *(query endpoints only)* - A function that accepts a 'recipe' callback specifying how to update the data at the time it is called. This uses `immer` internally, and updates can be written 'mutably' while safely producing the next immutable state.

Mutation onCacheEntryAdded signature

```
async function onCacheEntryAdded(
  arg: QueryArg,
  {
    dispatch,
    getState,
```

```
    extra,
    requestId,
    cacheEntryRemoved,
    cacheDataLoaded,
    getCacheEntry,
  }: MutationCacheLifecycleApi
): Promise<void>
```

Query onCacheEntryAdded signature

```
async function onCacheEntryAdded(
  arg: QueryArg,
  {
    dispatch,
    getState,
    extra,
    requestId,
    cacheEntryRemoved,
    cacheDataLoaded,
    getCacheEntry,
    updateCachedData, // available for query endpoints only
  }: QueryCacheLifecycleApi
): Promise<void>
```

Return value

See the "created Api" API reference

*Last updated on **Feb 7, 2023***

fetchBaseQuery

This is a very small wrapper around `fetch` that aims to simplify HTTP requests. It is not a full-blown replacement for `axios`, `superagent`, or any other more heavyweight library, but it will cover the vast majority of your HTTP request needs.

`fetchBaseQuery` is a factory function that generates a data fetching method compatible with RTK Query's `baseQuery` configuration option. It takes all standard options from fetch's `RequestInit` interface, as well as `baseUrl`, a `prepareHeaders` function, an optional `fetch` function, a `paramsSerializer` function, and a `timeout`.

Basic Usage

To use it, import it when you are creating an API service definition, call it as `fetchBaseQuery(options)`, and pass the result as the `baseQuery` field in `createApi`:

- TypeScript
- JavaScript

src/services/pokemon.ts

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

export const pokemonApi = createApi({
  // Set the baseUrl for every endpoint below
  baseQuery: fetchBaseQuery({ baseUrl: 'https://pokeapi.co/api/v2/' }),
  endpoints: (builder) => ({
    getPokemonByName: builder.query({
      // Will make a request like https://pokeapi.co/api/v2/pokemon/bulbasaur
      query: (name: string) => `pokemon/${name}`,
    }),
    updatePokemon: builder.mutation({
      query: ({ name, patch }) => ({
        url: `pokemon/${name}`,
        // When performing a mutation, you typically use a method of
        // PATCH/PUT/POST/DELETE for REST endpoints
        method: 'PATCH',
        // fetchBaseQuery automatically adds `content-type: application/json`
to
        // the Headers and calls `JSON.stringify(patch)`
        body: patch,
      }),
    }),
  }),
})
```

Signature

fetchBaseQuery signature

```
type FetchBaseQuery = (
  args: FetchBaseQueryArgs
) => (
  args: string | FetchArgs,
  api: BaseQueryApi,
  extraOptions: ExtraOptions
) => FetchBaseQueryResult

type FetchBaseQueryArgs = {
  baseUrl?: string
  prepareHeaders?: (
    headers: Headers,
    api: Pick<
      BaseQueryApi,
      'getState' | 'extra' | 'endpoint' | 'type' | 'forced'
    >
  ) => MaybePromise<Headers | void>
  fetchFn?: (
    input: RequestInfo,
    init?: RequestInit | undefined
  ) => Promise<Response>
  paramsSerializer?: (params: Record<string, any>) => string
  isJsonContentType?: (headers: Headers) => boolean
  jsonContentType?: string
  timeout?: number
} & RequestInit

type FetchBaseQueryResult = Promise<
  | {
      data: any
      error?: undefined
      meta?: { request: Request; response: Response }
    }
  | {
      error: {
        status: number
        data: any
      }
      data?: undefined
      meta?: { request: Request; response: Response }
    }
>
```

Parameters

`baseUrl`

(required)

Typically a string like `https://api.your-really-great-app.com/v1/`. If you don't provide a `baseUrl`, it defaults to a relative path from where the request is being made. **You should most likely *always* specify this**.

`prepareHeaders`

(optional)

Allows you to inject headers on every request. You can specify headers at the endpoint level, but you'll typically want to set common headers like `authorization` here. As a convenience mechanism, the second argument allows you to use `getState` to access your redux store in the event you store information you'll need there such as an auth token. Additionally, it provides access to `extra`, `endpoint`, `type`, and `forced` to unlock more granular conditional behaviors.

You can mutate the `headers` argument directly, and returning it is optional.

prepareHeaders signature

```
type prepareHeaders = (
  headers: Headers,
  api: {
    getState: () => unknown
    extra: unknown
    endpoint: string
    type: 'query' | 'mutation'
    forced: boolean | undefined
  }
) => Headers | void
```

`paramsSerializer`

(optional)

A function that can be used to apply custom transformations to the data passed into `params`. If you don't provide this, `params` will be given directly to `new URLSearchParams()`. With some API integrations, you may need to leverage this to use something like the `query-string` library to support different array types.

`fetchFn`

(optional)

A fetch function that overrides the default on the window. Can be useful in SSR environments where you may need to leverage `isomorphic-fetch` or `cross-fetch`.

`timeout`

(optional)

A number in milliseconds that represents the maximum time a request can take before timing out.

`isJsonContentType`

(optional)

A callback that receives a `Headers` object and determines the `body` field of the `FetchArgs` argument should be stringified via `JSON.stringify()`.

The default implementation inspects the `content-type` header, and will match values like `"application/json"` and `"application/vnd.api+json"`.

`jsonContentType`

(optional)

Used when automatically setting the `content-type` header for a request with a jsonifiable body that does not have an explicit `content-type` header. Defaults to `"application/json"`.

Common Usage Patterns

Setting default headers on requests

The most common use case for `prepareHeaders` would be to automatically include `authorization` headers for your API requests.

- TypeScript
- JavaScript

"Setting

```
import { fetchBaseQuery } from '@reduxjs/toolkit/query'
import type { RootState } from './store'

const baseQuery = fetchBaseQuery({
  baseUrl: '/',
  prepareHeaders: (headers, { getState }) => {
    const token = (getState() as RootState).auth.token

    // If we have a token set in state, let's assume that we should be
passing it.
    if (token) {
      headers.set('authorization', `Bearer ${token}`)
    }

    return headers
  },
})
```

Individual query options

There is more behavior that you can define on a per-request basis. The `query` field may return an object containing any of the default `fetch` options available to the `RequestInit` interface, as well as these additional options:

- TypeScript
- JavaScript

endpoint request options

```
interface FetchArgs extends RequestInit {
  url: string
  params?: Record<string, any>
  body?: any
  responseHandler?:
    | 'json'
    | 'text'
    | `content-type`
    | ((response: Response) => Promise<any>)
  validateStatus?: (response: Response, body: any) => boolean
  timeout?: number
}

const defaultValidateStatus = (response: Response) =>
  response.status >= 200 && response.status <= 299
```

Setting the body

By default, `fetchBaseQuery` assumes that every request you make will be `json`, so in those cases all you have to do is set the `url` and pass a `body` object when appropriate. For other implementations, you can manually set the `Headers` to specify the content type.

json

```
 // omitted
  endpoints: (builder) => ({
    updateUser: builder.query({
      query: (user: Record<string, string>) => ({
        url: `users`,
        method: 'PUT',
        body: user // Body is automatically converted to json with the
correct headers
      }),
    }),
```

text

```
 // omitted
  endpoints: (builder) => ({
    updateUser: builder.query({
      query: (user: Record<string, string>) => ({
        url: `users`,
        method: 'PUT',
        headers: {
```

```
            'content-type': 'text/plain',
          },
          body: user
        }),
      }),
```

Setting the query string

`fetchBaseQuery` provides a simple mechanism that converts an `object` to a serialized query string by passing the object to `new URLSearchParms()`. If this doesn't suit your needs, you have two options:

1. Pass the `paramsSerializer` option to `fetchBaseQuery` to apply custom transformations
2. Build your own querystring and set it in the `url`

```
 // omitted
  endpoints: (builder) => ({
    updateUser: builder.query({
      query: (user: Record<string, string>) => ({
        url: `users`,
        // Assuming no `paramsSerializer` is specified, the user object is
automatically converted
        // and produces a url like
/api/users?first_name=test&last_name=example
        params: user
      }),
    }),
```

Parsing a Response

By default, `fetchBaseQuery` assumes that every `Response` you get will be parsed as `json`. In the event that you don't want that to happen, you can customize the behavior by specifying an alternative response handler like `text`, or take complete control and use a custom function that accepts the raw `Response` object — allowing you to use any `Response` method.

The `responseHandler` field can be either:

- TypeScript
- JavaScript

```
type ResponseHandler =
  | 'content-type'
  | 'json'
  | 'text'
  | ((response: Response) => Promise<any>)
```

The `"json"` and `"text"` values instruct `fetchBaseQuery` to the corresponding fetch response methods for reading the body. `content-type` will check the header field

to first determine if this appears to be JSON, and then use one of those two methods. The callback allows you to process the body yourself.

- TypeScript
- JavaScript

Parse a Response as text

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

export const customApi = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: '/api/' }),
  endpoints: (builder) => ({
    getUsers: builder.query({
      query: () => ({
        url: `users`,
        // This is the same as passing 'text'
        responseHandler: (response) => response.text(),
      }),
    }),
  }),
})
```

NOTE ABOUT RESPONSES THAT RETURN AN UNDEFINED BODY

If you make a `json` request to an API that only returns a `200` with an undefined body, `fetchBaseQuery` will pass that through as `undefined` and will not try to parse it as `json`. This can be common with some APIs, especially on `delete` requests.

Handling non-standard Response status codes

By default, `fetchBaseQuery` will `reject` any `Response` that does not have a status code of `2xx` and set it to `error`. This is the same behavior you've most likely experienced with `axios` and other popular libraries. In the event that you have a non-standard API you're dealing with, you can use the `validateStatus` option to customize this behavior.

- TypeScript
- JavaScript

Using a custom validateStatus

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

export const customApi = createApi({
  // Set the baseUrl for every endpoint below
  baseQuery: fetchBaseQuery({ baseUrl: '/api/' }),
  endpoints: (builder) => ({
    getUsers: builder.query({
      query: () => ({
        url: `users`,
        // Example: we have a backend API always returns a 200,
        // but sets an `isError` property when there is an error.
```

```
        validateStatus: (response, result) =>
          response.status === 200 && !result.isError,
      }),
    }),
  }),
})
```

Adding a custom timeout to requests

By default, `fetchBaseQuery` has no default timeout value set, meaning your requests will stay pending until your api resolves the request(s) or it reaches the browser's default timeout (normally 5 minutes). Most of the time, this isn't what you'll want. When using `fetchBaseQuery`, you have the ability to set a `timeout` on the `baseQuery` or on individual endpoints. When specifying both options, the endpoint value will take priority.

- TypeScript
- JavaScript

Setting a timeout value

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

export const api = createApi({
  // Set a default timeout of 10 seconds
  baseQuery: fetchBaseQuery({ baseUrl: '/api/', timeout: 10000 }),
  endpoints: (builder) => ({
    getUsers: builder.query({
      query: () => ({
        url: `users`,
        // Example: we know the users endpoint is _really fast_ because it's
always cached.
        // We can assume if its over > 1000ms, something is wrong and we
should abort the request.
        timeout: 1000,
      }),
    }),
  }),
})
```

*Last updated on **Dec 12, 2022***

ApiProvider

Can be used as a `Provider` if you **do not already have a Redux store**.

Basic usage - wrap your App with ApiProvider

```
import * as React from 'react';
import { ApiProvider } from '@reduxjs/toolkit/query/react';
import { Pokemon } from './features/Pokemon';

function App() {
  return (
    <ApiProvider api={api}>
      <Pokemon />
    </ApiProvider>
  );
}
```

DANGER

Using this together with an existing Redux store will cause them to conflict with each other. If you are already using Redux, please follow the instructions as shown in the Getting Started guide.

*Last updated on **Jun 19, 2021***

setupListeners

A utility used to enable `refetchOnFocus` and `refetchOnReconnect` behaviors. It requires the `dispatch` method from your store.
Calling `setupListeners(store.dispatch)` will configure listeners with the recommended defaults, but you have the option of providing a callback for more granular control.

setupListeners default configuration

```
let initialized = false
export function setupListeners(
  dispatch: ThunkDispatch<any, any, any>,
  customHandler?: (
    dispatch: ThunkDispatch<any, any, any>,
    actions: {
      onFocus: typeof onFocus
      onFocusLost: typeof onFocusLost
      onOnline: typeof onOnline
      onOffline: typeof onOffline
    }
  ) => () => void
) {
  function defaultHandler() {
    const handleFocus = () => dispatch(onFocus())
    const handleFocusLost = () => dispatch(onFocusLost())
    const handleOnline = () => dispatch(onOnline())
    const handleOffline = () => dispatch(onOffline())
    const handleVisibilityChange = () => {
      if (window.document.visibilityState === 'visible') {
        handleFocus()
      } else {
        handleFocusLost()
      }
    }

    if (!initialized) {
      if (typeof window !== 'undefined' && window.addEventListener) {
        // Handle focus events
        window.addEventListener(
          'visibilitychange',
          handleVisibilityChange,
          false
        )
        window.addEventListener('focus', handleFocus, false)

        // Handle connection events
        window.addEventListener('online', handleOnline, false)
        window.addEventListener('offline', handleOffline, false)
        initialized = true
      }
    }
    const unsubscribe = () => {
      window.removeEventListener('focus', handleFocus)
      window.removeEventListener('visibilitychange', handleVisibilityChange)
      window.removeEventListener('online', handleOnline)
      window.removeEventListener('offline', handleOffline)
      initialized = false
    }
    return unsubscribe
  }

  return customHandler
    ? customHandler(dispatch, { onFocus, onFocusLost, onOffline, onOnline })
    : defaultHandler()
}
```

If you notice, `onFocus`, `onFocusLost`, `onOffline`, `onOnline` are all actions that are provided to the callback. Additionally, these actions are made available to `api.internalActions` and are able to be used by dispatching them like this:

Manual onFocus event

```
dispatch(api.internalActions.onFocus())
```

*Last updated on **Jun 19, 2021***

Generated API Slices

API Slice Overview

When you call `createApi`, it automatically generates and returns an API service "slice" object structure containing Redux logic you can use to interact with the endpoints you defined. This slice object includes a reducer to manage cached data, a middleware to manage cache lifetimes and subscriptions, and selectors and thunks for each endpoint. If you imported `createApi` from the React-specific entry point, it also includes auto-generated React hooks for use in your components.

This section documents the contents of that API structure, with the different fields grouped by category. The API types and descriptions are listed on separate pages for each category.

TIP

Typically, you should only have one API slice per base URL that your application needs to communicate with. For example, if your site fetches data from both `/api/posts` and `/api/users`, you would have a single API slice with `/api/` as the base URL, and separate endpoint definitions for `posts` and `users`. This allows you to effectively take advantage of automated re-fetching by defining tag relationships across endpoints.

For maintainability purposes, you may wish to split up endpoint definitions across multiple files, while still maintaining a single API slice which includes all of these endpoints. See code splitting for how you can use the `injectEndpoints` property to inject API endpoints from other files into a single API slice definition.
API Slice Contents

```
const api = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: '/' }),
  endpoints: (build) => ({
    // ...
  }),
})

type Api = {
  // Redux integration
  reducerPath: string
  reducer: Reducer
  middleware: Middleware

  // Endpoint interactions
  endpoints: Record<string, EndpointDefinition>

  // Code splitting and generation
  injectEndpoints: (options: InjectEndpointsOptions) => UpdatedApi
  enhanceEndpoints: (options: EnhanceEndpointsOptions) => UpdatedApi

  // Utilities
  utils: {
    updateQueryData: UpdateQueryDataThunk
    patchQueryData: PatchQueryDataThunk
    prefetch: PrefetchThunk
    invalidateTags: ActionCreatorWithPayload<
      Array<TagTypes | FullTagDescription<TagTypes>>,
      string
    >
    selectInvalidatedBy: (
      state: FullState,
      tags: Array<TagTypes | FullTagDescription<TagTypes>>
    ) => Array<{
      endpointName: string
      originalArgs: any
      queryCacheKey: string
    }>
    resetApiState: ActionCreator<ResetAction>
    getRunningQueryThunk(
      endpointName: EndpointName,
      args: QueryArg
    ): ThunkWithReturnValue<QueryActionCreatorResult | undefined>
    getRunningMutationThunk(
      endpointName: EndpointName,
      fixedCacheKeyOrRequestId: string
    ): ThunkWithReturnValue<MutationActionCreatorResult | undefined>
    getRunningQueriesThunk(): ThunkWithReturnValue<
      Array<QueryActionCreatorResult<any>>
    >
    getRunningMutationsThunk(): ThunkWithReturnValue<
      Array<MutationActionCreatorResult<any>>
    >
  }

  // Internal actions
  internalActions: InternalActions

  // React hooks (if applicable)
  [key in GeneratedReactHooks]: GeneratedReactHooks[key]
}
```

Redux Integration

Internally, `createApi` will call the Redux Toolkit `createSlice` API to generate a slice reducer and corresponding action creators with the appropriate logic for caching fetched data. It also automatically generates a custom Redux middleware that manages subscription counts and cache lifetimes.

The generated slice reducer and the middleware both need to be adding to your Redux store setup in `configureStore` in order to work correctly.

API REFERENCE

- API Slices: Redux Integration

Endpoints

The API slice object will have an `endpoints` field inside. This section maps the endpoint names you provided to `createApi` to the core Redux logic (thunks and selectors) used to trigger data fetches and read cached data for that endpoint. If you're using the React-specific version of `createApi`, each endpoint definition will also contain the auto-generated React hooks for that endpoint.

API REFERENCE

- API Slices: Endpoints

Code Splitting and Generation

Each API slice allows additional endpoint definitions to be injected at runtime after the initial API slice has been defined. This can be beneficial for apps that may have *many* endpoints.

The individual API slice endpoint definitions can also be split across multiple files. This is primarily useful for working with API slices that were code-generated from an API schema file, allowing you to add additional custom behavior and configuration to a set of automatically-generated endpoint definitions.

Each API slice object has `injectEndpoints` and `enhanceEndpoints` functions to support these use cases.

API REFERENCE

- API Slices: Code Splitting and Generation

API Slice Utilities

The `util` field includes various utility functions that can be used to manage the cache, including manually updating query cache data, triggering pre-fetching of data, manually invalidating tags, and manually resetting the api state, as well as other utility functions that can be used in various scenarios, including SSR.

API REFERENCE

- API Slices: Utilities

Internal Actions

The `internalActions` field contains a set of additional thunks that are used for internal behavior, such as managing updates based on focus.

React Hooks

The core RTK Query `createApi` method is UI-agnostic, in the same way that the Redux core library and Redux Toolkit are UI-agnostic. They are all plain JS logic that can be used anywhere.

However, RTK Query also provides the ability to auto-generate React hooks for each of your endpoints. Since this specifically depends on React itself, RTK Query provides an alternate entry point that exposes a customized version of `createApi` that includes that functionality:

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

If you have used the React-specific version of `createApi`, the generated `Api` slice structure will also contain a set of React hooks. These endpoint hooks are available as `api.endpoints[endpointName].useQuery` or `api.endpoints[endpointName].useMutation`, matching how you defined that endpoint.

The same hooks are also added to the `Api` object itself, and given auto-generated names based on the endpoint name and query/mutation type.

For example, if you had endpoints for `getPosts` and `updatePost`, these options would be available:

Generated React Hook names

```
// Hooks attached to the endpoint definition
const { data } = api.endpoints.getPosts.useQuery()
const { data } = api.endpoints.updatePost.useMutation()

// Same hooks, but given unique names and attached to the API slice object
const { data } = api.useGetPostsQuery()
const [updatePost] = api.useUpdatePostMutation()
```

The React-specific version of `createApi` also generates a `usePrefetch` hook, attached to the `Api` object, which can be used to initiate fetching data ahead of time.

*Last updated on **Oct 18, 2022***

API Slices: Redux Integration

Internally, `createApi` will call the Redux Toolkit `createSlice` API to generate a slice reducer and corresponding action creators with the appropriate logic for caching fetched data. It also automatically generates a custom Redux middleware that manages subscription counts and cache lifetimes.

The generated slice reducer and the middleware both need to be added to your Redux store setup in `configureStore` in order to work correctly:

- TypeScript
- JavaScript

src/store.ts

```
import { configureStore } from '@reduxjs/toolkit'
import { setupListeners } from '@reduxjs/toolkit/query'
import { pokemonApi } from './services/pokemon'

export const store = configureStore({
  reducer: {
    // Add the generated reducer as a specific top-level slice
    [pokemonApi.reducerPath]: pokemonApi.reducer,
  },
  // Adding the api middleware enables caching, invalidation, polling,
  // and other useful features of `rtk-query`.
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware().concat(pokemonApi.middleware),
})

// configure listeners using the provided defaults
setupListeners(store.dispatch)
```

reducerPath

```
reducerPath: string
```

Contains the `reducerPath` option provided to `createApi`. Use this as the root state key when adding the `reducer` function to the store so that the rest of the generated API logic can find the state correctly.

`reducer`

```
reducer: Reducer
```

A standard Redux slice reducer function containing the logic for updating the cached data. Add this to the Redux store using the `reducerPath` you provided as the root state key.

`middleware`

```
middleware: Middleware
```

A custom Redux middleware that contains logic for managing caching, invalidation, subscriptions, polling, and more. Add this to the store setup after other middleware.

*Last updated on **Aug 2, 2021***

API Slices: Endpoints

The API slice object will have an `endpoints` field inside. This section maps the endpoint names you provided to `createApi` to the core Redux logic (thunks and selectors) used to trigger data fetches and read cached data for that endpoint. If you're using the React-specific version of `createApi`, each endpoint definition will also contain the auto-generated React hooks for that endpoint.

Each endpoint structure contains the following fields:

```
type EndpointLogic = {
  initiate: InitiateRequestThunk
  select: CreateCacheSelectorFactory
  matchPending: Matcher<PendingAction>
  matchFulfilled: Matcher<FulfilledAction>
  matchRejected: Matcher<RejectedAction>
}
```

`initiate`

Signature

```
type InitiateRequestThunk = StartQueryActionCreator |
StartMutationActionCreator;

type StartQueryActionCreator = (
  arg:any,
  options?: StartQueryActionCreatorOptions
) => ThunkAction<QueryActionCreatorResult, any, any, AnyAction>;

type StartMutationActionCreator<D extends MutationDefinition<any, any, any,
any>> = (
  arg: any
  options?: StartMutationActionCreatorOptions
) => ThunkAction<MutationActionCreatorResult<D>, any, any, AnyAction>;

type SubscriptionOptions = {
  /**
   * How frequently to automatically re-fetch data (in milliseconds).
Defaults to `0` (off).
   */
  pollingInterval?: number;
  /**
   * Defaults to `false`. This setting allows you to control whether RTK
Query will try to refetch all subscribed queries after regaining a network
connection.
   *
   * If you specify this option alongside `skip: true`, this **will not be
evaluated** until `skip` is false.
   *
   * Note: requires `setupListeners` to have been called.
   */
  refetchOnReconnect?: boolean;
  /**
   * Defaults to `false`. This setting allows you to control whether RTK
Query will try to refetch all subscribed queries after the application window
regains focus.
   *
   * If you specify this option alongside `skip: true`, this **will not be
evaluated** until `skip` is false.
   *
   * Note: requires `setupListeners` to have been called.
   */
  refetchOnFocus?: boolean;
};

interface StartQueryActionCreatorOptions {
  subscribe?: boolean;
  forceRefetch?: boolean | number;
  subscriptionOptions?: SubscriptionOptions;
}

interface StartMutationActionCreatorOptions {
  /**
   * If this mutation should be tracked in the store.
   * If you just want to manually trigger this mutation using `dispatch` and
don't care about the
   * result, state & potential errors being held in store, you can set this
to false.
   * (defaults to `true`)
   */
  track?: boolean;
}
```

Description

A Redux thunk action creator that you can dispatch to trigger data fetch queries or mutations.

React Hooks users will most likely never need to use these directly, as the hooks automatically dispatch these actions as needed.

USAGE OF ACTIONS OUTSIDE OF REACT HOOKS

When dispatching an action creator, you're responsible for storing a reference to the promise it returns in the event that you want to update that specific subscription. Also, you have to manually unsubscribe once your component unmounts. To get an idea of what that entails, see the Svelte Example or the React Class Components Example

Example

initiate query example

```
import { useState } from 'react'
import { useAppDispatch } from './store/hooks'
import { api } from './services/api'

function App() {
  const dispatch = useAppDispatch()
  const [postId, setPostId] = useState<number>(1)

  useEffect(() => {
    // Add a subscription
    const result = dispatch(api.endpoints.getPost.initiate(postId))

    // Return the `unsubscribe` callback to be called in the `useEffect`
cleanup step
    return result.unsubscribe
  }, [dispatch, postId])

  return (
    <div>
      <div>Initiate query example</div>
    </div>
  )
}
```

initiate mutation example

```
import { useState } from 'react'
import { useAppDispatch } from './store/hooks'
import { api, Post } from './services/api'

function App() {
  const dispatch = useAppDispatch()
  const [newPost, setNewPost] = useState<Omit<Post, 'id'>>({ name: 'Ash' })

  function handleClick() {
```

```
    // Trigger a mutation
    // The `track` property can be set `false` in situations where we aren't
    // interested in the result of the mutation
    dispatch(api.endpoints.addPost.initiate(newPost), { track: false })
  }

  return (
    <div>
      <div>Initiate mutation example</div>
      <button onClick={handleClick}>Add post</button>
    </div>
  )
}
```

`select`

Signature

```
type CreateCacheSelectorFactory =
  | QueryResultSelectorFactory
  | MutationResultSelectorFactory

type QueryResultSelectorFactory = (
  queryArg: QueryArg | SkipToken
) => (state: RootState) => QueryResultSelectorResult<Definition>

type MutationResultSelectorFactory<
  Definition extends MutationDefinition<any, any, any, any>,
  RootState
> = (
  requestId: string | SkipToken
) => (state: RootState) => MutationSubState<Definition> & RequestStatusFlags

type SkipToken = typeof Symbol
```

Description

A function that accepts a cache key argument, and generates a new memoized selector for reading cached data for this endpoint using the given cache key. The generated selector is memoized using Reselect's `createSelector`.

When selecting mutation results rather than queries, the function accepts a request ID instead.

RTKQ defines a `Symbol` named `skipToken` internally. If `skipToken` is passed as the query argument to these selectors, the selector will return a default uninitialized state. This can be used to avoid returning a value if a given query is supposed to be disabled.

React Hooks users will most likely never need to use these directly, as the hooks automatically use these selectors as needed.

CAUTION

Each call to `.select(someCacheKey)` returns a *new* selector function instance. In order for memoization to work correctly, you should create a given selector function once per cache key and reuse that selector function instance, rather than creating a new selector instance each time.

Example

select query example

```
import { useState, useMemo } from 'react'
import { useAppDispatch, useAppSelector } from './store/hooks'
import { api } from './services/api'

function App() {
  const dispatch = useAppDispatch()
  const [postId, setPostId] = useState(1)
  // useMemo is used to only call `.select()` when required.
  // Each call will create a new selector function instance
  const selectPost = useMemo(
    () => api.endpoints.getPost.select(postId),
    [postId]
  )
  const { data, isLoading } = useAppSelector(selectPost)

  useEffect(() => {
    // Add a subscription
    const result = dispatch(api.endpoints.getPost.initiate(postId))

    // Return the `unsubscribe` callback to be called in the cleanup step
    return result.unsubscribe
  }, [dispatch, postId])

  if (isLoading) return <div>Loading post...</div>

  return (
    <div>
      <div>Initiate query example</div>
      <div>Post name: {data.name}</div>
    </div>
  )
}
```

select mutation example

```
import { useState, useMemo } from 'react'
import { skipToken } from '@reduxjs/toolkit/query'
import { useAppDispatch, useAppSelector } from './store/hooks'
import { api } from './services/api'

function App() {
  const dispatch = useAppDispatch()
  const [newPost, setNewPost] = useState({ name: 'Ash' })
  const [requestId, setRequestId] = useState<typeof skipToken | string>(
    skipToken
  )
  // useMemo is used to only call `.select(..)` when required.
  // Each call will create a new selector function instance
```

```
  const selectMutationResult = useMemo(
    () => api.endpoints.addPost.select(requestId),
    [requestId]
  )
  const { isLoading } = useAppSelector(selectMutationResult)

  function handleClick() {
    // Trigger a mutation
    const result = dispatch(api.endpoints.addPost.initiate(newPost))
    // store the requestId to select the mutation result elsewhere
    setRequestId(result.requestId)
  }

  if (isLoading) return <div>Adding post...</div>

  return (
    <div>
      <div>Select mutation example</div>
      <button onClick={handleClick}>Add post</button>
    </div>
  )
}
```

Matchers

A set of Redux Toolkit action matching utilities that match the `pending`, `fulfilled`, and `rejected` actions that will be dispatched by this thunk. These allow you to match on Redux actions for that endpoint, such as in `createSlice.extraReducers` or a custom middleware. Those are implemented as follows:

```
 matchPending: isAllOf(isPending(thunk), matchesEndpoint(endpoint)),
 matchFulfilled: isAllOf(isFulfilled(thunk), matchesEndpoint(endpoint)),
 matchRejected: isAllOf(isRejected(thunk), matchesEndpoint(endpoint)),
```

*Last updated on **Apr 2, 2023***

API Slices: Code Splitting and Generation

Each API slice allows additional endpoint definitions to be injected at runtime after the initial API slice has been defined. This can be beneficial for apps that may have *many* endpoints.

The individual API slice endpoint definitions can also be split across multiple files. This is primarily useful for working with API slices that were code-generated from an API schema file, allowing you to add additional custom behavior and configuration to a set of automatically-generated endpoint definitions.

Each API slice object has `injectEndpoints` and `enhanceEndpoints` functions to support these use cases.

`injectEndpoints`

Signature

```
const injectEndpoints = (endpointOptions: InjectedEndpointOptions) =>
  EnhancedApiSlice

interface InjectedEndpointOptions {
  endpoints: (build: EndpointBuilder) => NewEndpointDefinitions
  overrideExisting?: boolean
}
```

Description

Accepts an options object containing the same `endpoints` builder callback you would pass to `createApi.endpoints`. Any endpoint definitions defined using that builder will be merged into the existing endpoint definitions for this API slice using a shallow merge, so any new endpoint definitions will override existing endpoints with the same name.

Returns an updated and enhanced version of the API slice object, containing the combined endpoint definitions.

In development, endpoints will not be overridden unless `overrideExisting` is set to `true`. If not, a warning will be shown to notify you if there is a name clash between endpoint definitions.

This method is primarily useful for code splitting and hot reloading.

`enhanceEndpoints`

Signature

```
const enhanceEndpoints = (endpointOptions: EnhanceEndpointsOptions) =>
  EnhancedApiSlice

interface EnhanceEndpointsOptions {
  addTagTypes?: readonly string[]
  endpoints?: Record<string, Partial<EndpointDefinition>>
}
```

Description

Any provided tag types or endpoint definitions will be merged into the existing endpoint definitions for this API slice. Unlike `injectEndpoints`, the partial endpoint

definitions will not *replace* existing definitions, but are rather merged together on a per-definition basis (ie, `Object.assign(existingEndpoint, newPartialEndpoint)`).

Returns an updated and enhanced version of the API slice object, containing the combined endpoint definitions.

This is primarily useful for taking an API slice object that was code-generated from an API schema file like OpenAPI, and adding additional specific hand-written configuration for cache invalidation management on top of the generated endpoint definitions.

For example, `enhanceEndpoints` can be used to modify caching behavior by changing the values of `providesTags`, `invalidatesTags`, and `keepUnusedDataFor`:

- TypeScript
- JavaScript

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

const enhancedApi = api.enhanceEndpoints({
  addTagTypes: ['User'],
  endpoints: {
    getUserByUserId: {
      providesTags: ['User'],
    },
    patchUserByUserId: {
      invalidatesTags: ['User'],
    },
    // alternatively, define a function which is called with the endpoint
definition as an argument
    getUsers(endpoint) {
      endpoint.providesTags = ['User']
      endpoint.keepUnusedDataFor = 120
    },
  },
})
```

*Last updated on **Sep 14, 2022***

API Slices: Utilities

The API slice object includes various utilities that can be used for cache management, such as implementing optimistic updates, as well implementing server side rendering.

These are included as `api.util` inside the API object.

INFO

Some of the TS types on this page are pseudocode to illustrate intent, as the actual internal types are fairly complex.

`updateQueryData`

Signature

```
const updateQueryData = (
  endpointName: string,
  args: any,
  updateRecipe: (draft: Draft<CachedState>) => void
) => ThunkAction<PatchCollection, PartialState, any, AnyAction>;

interface PatchCollection {
  patches: Patch[];
  inversePatches: Patch[];
  undo: () => void;
}
```

- **Parameters**
  - `endpointName`: a string matching an existing endpoint name
  - `args`: an argument matching that used for a previous query call, used to determine which cached dataset needs to be updated
  - `updateRecipe`: an Immer `produce` callback that can apply changes to the cached state

Description

A Redux thunk action creator that, when dispatched, creates and applies a set of JSON diff/patch objects to the current state. This immediately updates the Redux state with those changes.

The thunk action creator accepts three arguments: the name of the endpoint we are updating (such as `'getPost'`), any relevant query arguments, and a callback function. The callback receives an Immer-wrapped `draft` of the current state, and may modify the draft to match the expected results after the mutation completes successfully.

The thunk returns an object containing `{patches: Patch[], inversePatches: Patch[], undo: () => void}`. The `patches` and `inversePatches` are generated using Immer's `produceWithPatches` method.

This is typically used as the first step in implementing optimistic updates. The generated `inversePatches` can be used to revert the updates by calling `dispatch(patchQueryData(endpointName, args, inversePatches))`. Alternatively, the `undo` method can be called directly to achieve the same effect.

Note that the first two arguments (`endpointName` and `args`) are used to determine which existing cache entry to update. If no existing cache entry is found, the `updateRecipe` callback will not run.

Example 1

```
const patchCollection = dispatch(
  api.util.updateQueryData('getPosts', undefined, (draftPosts) => {
    draftPosts.push({ id: 1, name: 'Teddy' })
  })
)
```

In the example above, `'getPosts'` is provided for the `endpointName`, and `undefined` is provided for `args`. This will match a query cache key of `'getPosts(undefined)'`.

i.e. it will match a cache entry that may have been created via any of the following calls:

```
api.endpoints.getPosts.useQuery()

useGetPostsQuery()

useGetPostsQuery(undefined, { ...options })

dispatch(api.endpoints.getPosts.initiate())

dispatch(api.endpoints.getPosts.initiate(undefined, { ...options }))
```

Example 2

```
const patchCollection = dispatch(
  api.util.updateQueryData('getPostById', 1, (draftPost) => {
    draftPost.name = 'Lilly'
  })
)
```

In the example above, `'getPostById'` is provided for the `endpointName`, and `1` is provided for `args`. This will match a query cache key of `'getPostById(1)'`.

i.e. it will match a cache entry that may have been created via any of the following calls:

```
api.endpoints.getPostById.useQuery(1)

useGetPostByIdQuery(1)

useGetPostByIdQuery(1, { ...options })

dispatch(api.endpoints.getPostById.initiate(1))

dispatch(api.endpoints.getPostById.initiate(1, { ...options }))
```

`upsertQueryData`

Signature

```
const upsertQueryData = <T>(
  endpointName: string,
  args: any,
  newEntryData: T
) => ThunkAction<Promise<CacheEntry<T>>, PartialState, any, AnyAction>;
```

- **Parameters**
  - `endpointName`: a string matching an existing endpoint name
  - `args`: an argument matching that used for a previous query call, used to determine which cached dataset needs to be updated
  - `newEntryValue`: the value to be written into the corresponding cache entry's `data` field

Description

A Redux thunk action creator that, when dispatched, acts as an artificial API request to upsert a value into the cache.

The thunk action creator accepts three arguments: the name of the endpoint we are updating (such as `'getPost'`), the appropriate query arg values to construct the desired cache key, and the data to upsert.

If no cache entry for that cache key exists, a cache entry will be created and the data added. If a cache entry already exists, this will *overwrite* the existing cache entry data.

The thunk executes *asynchronously*, and returns a promise that resolves when the store has been updated.

If dispatched while an actual request is in progress, both the upsert and request will be handled as soon as they resolve, resulting in a "last result wins" update behavior.

Example

```
await dispatch(
  api.util.upsertQueryData('getPost', { id: 1 }, { id: 1, text: 'Hello!' })
)
```

`patchQueryData`

Signature

```
const patchQueryData = (
  endpointName: string,
  args: any
  patches: Patch[]
) => ThunkAction<void, PartialState, any, AnyAction>;
```

- **Parameters**
  - `endpointName`: a string matching an existing endpoint name
  - `args`: a cache key, used to determine which cached dataset needs to be updated
  - `patches`: an array of patches (or inverse patches) to apply to cached state. These would typically be obtained from the result of dispatching `updateQueryData`

Description

A Redux thunk action creator that, when dispatched, applies a JSON diff/patch array to the cached data for a given query result. This immediately updates the Redux state with those changes.

The thunk action creator accepts three arguments: the name of the endpoint we are updating (such as `'getPost'`), the appropriate query arg values to construct the desired cache key, and a JSON diff/patch array as produced by Immer's `produceWithPatches`.

This is typically used as the second step in implementing optimistic updates. If a request fails, the optimistically-applied changes can be reverted by dispatching `patchQueryData` with the `inversePatches` that were generated by `updateQueryData` earlier.

In cases where it is desired to simply revert the previous changes, it may be preferable to call the `undo` method returned from dispatching `updateQueryData` instead.

Example

```
const patchCollection = dispatch(
  api.util.updateQueryData('getPosts', undefined, (draftPosts) => {
    draftPosts.push({ id: 1, name: 'Teddy' })
  })
)

// later
dispatch(
  api.util.patchQueryData('getPosts', undefined,
patchCollection.inversePatches)
)

// or
patchCollection.undo()
```

prefetch

Signature

```
type PrefetchOptions = { ifOlderThan?: false | number } | { force?: boolean
};

const prefetch = (
  endpointName: string,
  arg: any,
  options: PrefetchOptions
) => ThunkAction<void, any, any, AnyAction>;
```

- **Parameters**
  - `endpointName`: a string matching an existing endpoint name
  - `args`: a cache key, used to determine which cached dataset needs to be updated
  - `options`: options to determine whether the request should be sent for a given situation:
    - `ifOlderThan`: if specified, only runs the query if the difference between `new Date()` and the last`fulfilledTimeStamp` is greater than the given value (in seconds)
    - `force`: if `true`, it will ignore the `ifOlderThan` value if it is set and the query will be run even if it exists in the cache.

Description

A Redux thunk action creator that can be used to manually trigger pre-fetching of data.

The thunk action creator accepts three arguments: the name of the endpoint we are updating (such as `'getPost'`), any relevant query arguments, and a set of options used to determine if the data actually should be re-fetched based on cache staleness.

React Hooks users will most likely never need to use this directly, as the `usePrefetch` hook will dispatch the thunk action creator result internally as needed when you call the prefetching function supplied by the hook.

Example

```
dispatch(api.util.prefetch('getPosts', undefined, { force: true }))
```

`selectInvalidatedBy`

Signature

```
function selectInvalidatedBy(
  state: RootState,
  tags: ReadonlyArray<TagDescription<string>>
): Array<{
  endpointName: string
  originalArgs: any
  queryCacheKey: QueryCacheKey
}>
```

- **Parameters**
  - `state`: the root state
  - `tags`: a readonly array of invalidated tags, where the provided `TagDescription` is one of the strings provided to the `tagTypes` property of the api. e.g.
    - `[TagType]`
    - `[{ type: TagType }]`
    - `[{ type: TagType, id: number | string }]`

Description

A function that can select query parameters to be invalidated.

The function accepts two arguments

- the root state and
- the cache tags to be invalidated.

It returns an array that contains

- the endpoint name,
- the original args and
- the queryCacheKey.

Example

```
dispatch(api.util.selectInvalidatedBy(state, ['Post']))
dispatch(api.util.selectInvalidatedBy(state, [{ type: 'Post', id: 1 }]))
```

```
dispatch(
  api.util.selectInvalidatedBy(state, [
    { type: 'Post', id: 1 },
    { type: 'Post', id: 4 },
  ])
)
```

`invalidateTags`

Signature

```
const invalidateTags = (
  tags: Array<TagTypes | FullTagDescription<TagTypes>>
) => ({
  type: string,
  payload: tags,
})
```

- **Parameters**
  - `tags`: an array of tags to be invalidated, where the provided `TagType` is one of the strings provided to the `tagTypes` property of the api. e.g.
    - `[TagType]`
    - `[{ type: TagType }]`
    - `[{ type: TagType, id: number | string }]`

Description

A Redux action creator that can be used to manually invalidate cache tags for automated re-fetching.

The action creator accepts one argument: the cache tags to be invalidated. It returns an action with those tags as a payload, and the corresponding `invalidateTags` action type for the api.

Dispatching the result of this action creator will invalidate the given tags, causing queries to automatically re-fetch if they are subscribed to cache data that provides the corresponding tags.

Example

```
dispatch(api.util.invalidateTags(['Post']))
dispatch(api.util.invalidateTags([{ type: 'Post', id: 1 }]))
dispatch(
  api.util.invalidateTags([
    { type: 'Post', id: 1 },
    { type: 'Post', id: 'LIST' },
  ])
)
```

`resetApiState`

Signature

```
const resetApiState = () => ({
  type: string,
  payload: undefined,
})
```

Description

A Redux action creator that can be dispatched to manually reset the api state completely. This will immediately remove all existing cache entries, and all queries will be considered 'uninitialized'.

Note that hooks also track state in local component state and might not fully be reset by `resetApiState`.

Example

```
dispatch(api.util.resetApiState())
```

`getRunningQueriesThunk` and `getRunningMutationsThunk`

Signature

```
getRunningQueriesThunk():
ThunkWithReturnValue<Array<QueryActionCreatorResult<any>>>
getRunningMutationsThunk():
ThunkWithReturnValue<Array<MutationActionCreatorResult<any>>>
```

Description

Thunks that (if dispatched) return either all running queries or mutations. These returned values can be awaited like promises.

This is useful for SSR scenarios to await all queries (or mutations) triggered in any way, including via hook calls or manually dispatching `initiate` actions.

Awaiting all currently running queries example

```
await Promise.all(dispatch(api.util.getRunningQueriesThunk()))
```

`getRunningQueryThunk` and `getRunningMutationThunk`

Signature

```
getRunningQueryThunk<EndpointName extends QueryKeys<Definitions>>(
  endpointName: EndpointName,
  args: QueryArgFrom<Definitions[EndpointName]>
): ThunkWithReturnValue<
  | QueryActionCreatorResult<
      Definitions[EndpointName] & { type: 'query' }
    >
  | undefined
>

getRunningMutationThunk<EndpointName extends MutationKeys<Definitions>>(
  endpointName: EndpointName,
  fixedCacheKeyOrRequestId: string
): ThunkWithReturnValue<
  | MutationActionCreatorResult<
      Definitions[EndpointName] & { type: 'mutation' }
    >
  | undefined
>
```

Description

Thunks that (if dispatched) return a single running query (or mutation) for a given endpoint name + argument (or requestId/fixedCacheKey) combination, if it is currently running. If it is not currently running, the function returns `undefined`.

These thunks are primarily added to add experimental support for suspense in the future. They enable writing custom hooks that look up if RTK Query has already got a running query/mutation for a certain endpoint/argument combination, and retrieving that to `throw` it as a promise.

API Slices: React Hooks

Hooks Overview

The core RTK Query `createApi` method is UI-agnostic, in the same way that the Redux core library and Redux Toolkit are UI-agnostic. They are all plain JS logic that can be used anywhere.

However, RTK Query also provides the ability to auto-generate React hooks for each of your endpoints. Since this specifically depends on React itself, RTK Query provides an alternate entry point that exposes a customized version of `createApi` that includes that functionality:

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

If you have used the React-specific version of `createApi`, the generated `Api` slice structure will also contain a set of React hooks. The primary endpoint hooks are available as `api.endpoints[endpointName].useQuery` or `api.endpoints[endpointName].useMutation`, matching how you defined that endpoint.

The same hooks are also added to the `Api` object itself, and given auto-generated names based on the endpoint name and query/mutation type.

For example, if you had endpoints for `getPosts` and `updatePost`, these options would be available:

Generated React Hook names

```
// Hooks attached to the endpoint definition
const { data } = api.endpoints.getPosts.useQuery()
const [updatePost, { data }] = api.endpoints.updatePost.useMutation()

// Same hooks, but given unique names and attached to the API slice object
const { data } = api.useGetPostsQuery()
const [updatePost, { data }] = api.useUpdatePostMutation()
```

The general format is `use(Endpointname)(Query|Mutation)` - `use` is prefixed, the first letter of your endpoint name is capitalized, then `Query` or `Mutation` is appended depending on the type.

RTK Query provides additional hooks for more advanced use-cases, although not all are generated directly on the `Api` object as well. The full list of hooks generated in the React-specific version of `createApi` is as follows:

- `useQuery` (endpoint-specific, also generated on the `Api` object)
- `useMutation` (endpoint-specific, also generated on the `Api` object)
- `useQueryState` (endpoint-specific)
- `useQuerySubscription` (endpoint-specific)
- `useLazyQuery` (endpoint-specific, also generated on the `Api` object)
- `useLazyQuerySubscription` (endpoint-specific)
- `usePrefetch` (endpoint-agnostic)

For the example above, the full set of generated hooks for the api would be like so:

Generated React Hooks

```
/* Hooks attached to the `getPosts` query endpoint definition */
api.endpoints.getPosts.useQuery(arg, options)
api.endpoints.getPosts.useQueryState(arg, options)
api.endpoints.getPosts.useQuerySubscription(arg, options)
api.endpoints.getPosts.useLazyQuery(options)
api.endpoints.getPosts.useLazyQuerySubscription(options)

/* Hooks attached to the `updatePost` mutation endpoint definition */
api.endpoints.updatePost.useMutation(options)

/* Hooks attached to the `Api` object */
api.useGetPostsQuery(arg, options) // same as api.endpoints.getPosts.useQuery
api.useLazyGetPostsQuery(options) // same as
api.endpoints.getPosts.useLazyQuery
api.useUpdatePostMutation(options) // same as
api.endpoints.updatePost.useMutation
api.usePrefetch(endpointName, options)
```

Feature Comparison

The provided hooks have a degree of feature overlap in order to provide options optimized for a given situation. The table below provides a comparison of the core features for each hook.

Feature	useQuery	useMutation	useQueryState	useQuerySubscription	useLazyQuery	useLazyQuerySubscription	usePrefetch
Automatically triggers query requests	✔			✔			
Allows manually triggering query requests	✔			✔	✔	✔	✔
Allows manually triggering mutation requests		✔					
Subscribes a component to keep cached data in the store	✔	✔		✔	✔	✔	
Returns request status and cached data from the store	✔	✔	✔		✔		
Re-renders as request status and data	✔	✔	✔		✔		

Feature	useQuery	useMutation	useQueryState	useQuerySubscription	useLazyQuery	useLazyQuerySubscription	usePrefetch
become available							
Accepts polling/re-fetching options to trigger automatic re-fetches	✔			✔	✔	✔	

useQuery

Accessing a useQuery hook

```
const useQueryResult = api.endpoints.getPosts.useQuery(arg, options)
// or
const useQueryResult = api.useGetPostsQuery(arg, options)
```

Signature

```
type UseQuery = (
  arg: any | SkipToken,
  options?: UseQueryOptions
) => UseQueryResult

type UseQueryOptions = {
  pollingInterval?: number
  refetchOnReconnect?: boolean
  refetchOnFocus?: boolean
  skip?: boolean
  refetchOnMountOrArgChange?: boolean | number
  selectFromResult?: (result: UseQueryStateDefaultResult) => any
}

type UseQueryResult<T> = {
  // Base query state
  originalArgs?: unknown // Arguments passed to the query
  data?: T // The latest returned result regardless of hook arg, if present
  currentData?: T // The latest returned result for the current hook arg, if
present
```

```
  error?: unknown // Error result if present
  requestId?: string // A string generated by RTK Query
  endpointName?: string // The name of the given endpoint for the query
  startedTimeStamp?: number // Timestamp for when the query was initiated
  fulfilledTimeStamp?: number // Timestamp for when the query was completed

  // Derived request status booleans
  isUninitialized: boolean // Query has not started yet.
  isLoading: boolean // Query is currently loading for the first time. No
data yet.
  isFetching: boolean // Query is currently fetching, but might have data
from an earlier request.
  isSuccess: boolean // Query has data from a successful load.
  isError: boolean // Query is currently in an "error" state.

  refetch: () => void // A function to force refetch the query
}
```

- **Parameters**
  - `arg`: The query argument to be used in constructing the query itself, and as a cache key for the query. You can also pass in `skipToken` here as an alternative way of skipping the query, see skipToken
  - `options`: A set of options that control the fetching behavior of the hook
- **Returns**
  - A query result object containing the current loading state, the actual data or error returned from the API call, metadata about the request, and a function to `refetch` the data. Can be customized with `selectFromResult`

Description

A React hook that automatically triggers fetches of data from an endpoint, 'subscribes' the component to the cached data, and reads the request status and cached data from the Redux store. The component will re-render as the loading status changes and the data becomes available.

The query arg is used as a cache key. Changing the query arg will tell the hook to re-fetch the data if it does not exist in the cache already, and the hook will return the data for that query arg once it's available.

This hook combines the functionality of both `useQueryState` and `useQuerySubscription` together, and is intended to be used in the majority of situations.

Features

- Automatically triggers requests to retrieve data based on the hook argument and whether cached data exists by default
- 'Subscribes' the component to keep cached data in the store, and 'unsubscribes' when the component unmounts
- Accepts polling/re-fetching options to trigger automatic re-fetches when the corresponding criteria is met
- Returns the latest request status and cached data from the Redux store
- Re-renders as the request status changes and data becomes available

skipToken

Can be passed into `useQuery`, `useQueryState` or `useQuerySubscription` instead of the query argument to get the same effect as if setting `skip: true` in the query options.

Useful for scenarios where a query should be skipped when `arg` is `undefined` and TypeScript complains about it because `arg` is not allowed to be passed in as `undefined`, such as

will error if the query argument is not allowed to be undefined

```
useSomeQuery(arg, { skip: !!arg })
```

using skipToken instead

```
useSomeQuery(arg ?? skipToken)
```

If passed directly into a query or mutation selector, that selector will always return an uninitialized state.

See also Skipping queries with TypeScript using `skipToken`

useMutation

Accessing a useMutation hook

```
const useMutationResult = api.endpoints.updatePost.useMutation(options)
// or
const useMutationResult = api.useUpdatePostMutation(options)
```

Signature

```
type UseMutation = (
  options?: UseMutationStateOptions
) => [UseMutationTrigger, UseMutationResult | SelectedUseMutationResult]

type UseMutationStateOptions = {
  // A method to determine the contents of `UseMutationResult`
```

```
  selectFromResult?: (result: UseMutationStateDefaultResult) => any
  // A string used to enable shared results across hook instances which have
the same key
  fixedCacheKey?: string
}

type UseMutationTrigger<T> = (arg: any) => Promise<
  { data: T } | { error: BaseQueryError | SerializedError }
> & {
  requestId: string // A string generated by RTK Query
  abort: () => void // A method to cancel the mutation promise
  unwrap: () => Promise<T> // A method to unwrap the mutation call and
provide the raw response/error
  reset: () => void // A method to manually unsubscribe from the mutation
call and reset the result to the uninitialized state
}

type UseMutationResult<T> = {
  // Base query state
  originalArgs?: unknown // Arguments passed to the latest mutation call. Not
available if using the `fixedCacheKey` option
  data?: T // Returned result if present
  error?: unknown // Error result if present
  endpointName?: string // The name of the given endpoint for the mutation
  fulfilledTimestamp?: number // Timestamp for when the mutation was
completed

  // Derived request status booleans
  isUninitialized: boolean // Mutation has not been fired yet
  isLoading: boolean // Mutation has been fired and is awaiting a response
  isSuccess: boolean // Mutation has data from a successful call
  isError: boolean // Mutation is currently in an "error" state
  startedTimeStamp?: number // Timestamp for when the latest mutation was
initiated

  reset: () => void // A method to manually unsubscribe from the mutation
call and reset the result to the uninitialized state
}
```

TIP

The generated `UseMutation` hook will cause a component to re-render by default after the trigger callback is fired, as it affects the properties of the result. If you want to call the trigger but don't care about subscribing to the result with the hook, you can use the `selectFromResult` option to limit the properties that the hook cares about.

Returning a completely empty object will mean that any individual mutation call will cause only one re-render at most, e.g.

```
selectFromResult: () => ({})
```

- **Parameters**
  - `options`: A set of options that control the subscription behavior of the hook:

        - `selectFromResult`: A callback that can be used to customize the mutation result returned as the second item in the tuple
        - `fixedCacheKey`: An optional string used to enable shared results across hook instances
- **Returns**: A tuple containing:
    - `trigger`: A function that triggers an update to the data based on the provided argument. The trigger function returns a promise with the properties shown above that may be used to handle the behavior of the promise
    - `mutationState`: A query status object containing the current loading state and metadata about the request, or the values returned by the `selectFromResult` option where applicable. Additionally, this object will contain
        - a `reset` method to reset the hook back to it's original state and remove the current result from the cache
        - an `originalArgs` property that contains the argument passed to the last call of the `trigger` function.

Description

A React hook that lets you trigger an update request for a given endpoint, and subscribes the component to read the request status from the Redux store. The component will re-render as the loading status changes.

Features

- Manual control over firing a request to alter data on the server or possibly invalidate the cache
- 'Subscribes' the component to keep cached data in the store, and 'unsubscribes' when the component unmounts
- Returns the latest request status and cached data from the Redux store
- Re-renders as the request status changes and data becomes available

`useQueryState`

Accessing a useQuery hook

```
const useQueryStateResult = api.endpoints.getPosts.useQueryState(arg,
options)
```

Signature

```
type UseQueryState = (
  arg: any | SkipToken,
  options?: UseQueryStateOptions
) => UseQueryStateResult | SelectedQueryStateResult
```

```
type UseQueryStateOptions = {
  skip?: boolean
  selectFromResult?: (result: UseQueryStateDefaultResult) => any
}

type UseQueryStateResult<T> = {
  // Base query state
  originalArgs?: unknown // Arguments passed to the query
  data?: T // The latest returned result regardless of hook arg, if present
  currentData?: T // The latest returned result for the current hook arg, if
present
  error?: unknown // Error result if present
  requestId?: string // A string generated by RTK Query
  endpointName?: string // The name of the given endpoint for the query
  startedTimeStamp?: number // Timestamp for when the query was initiated
  fulfilledTimeStamp?: number // Timestamp for when the query was completed

  isUninitialized: false // Query has not started yet.
  isLoading: false // Query is currently loading for the first time. No data
yet.
  isFetching: false // Query is currently fetching, but might have data from
an earlier request.
  isSuccess: false // Query has data from a successful load.
  isError: false // Query is currently in an "error" state.
}
```

- **Parameters**
  - `arg`: The argument passed to the query defined in the endpoint. You can also pass in `skipToken` here as an alternative way of skipping the selection, see skipToken
  - `options`: A set of options that control the return value for the hook
- **Returns**
  - A query result object containing the current loading state, the actual data or error returned from the API call and metadata about the request. Can be customized with `selectFromResult`

Description

A React hook that reads the request status and cached data from the Redux store. The component will re-render as the loading status changes and the data becomes available.

Note that this hook does not trigger fetching new data. For that use-case, see `useQuery` or `useQuerySubscription`.

Features

- Returns the latest request status and cached data from the Redux store
- Re-renders as the request status changes and data becomes available

`useQuerySubscription`

Accessing a useQuerySubscription hook

```
const { refetch } = api.endpoints.getPosts.useQuerySubscription(arg, options)
```

Signature

```
type UseQuerySubscription = (
  arg: any | SkipToken,
  options?: UseQuerySubscriptionOptions
) => UseQuerySubscriptionResult

type UseQuerySubscriptionOptions = {
  skip?: boolean
  refetchOnMountOrArgChange?: boolean | number
  pollingInterval?: number
  refetchOnReconnect?: boolean
  refetchOnFocus?: boolean
}

type UseQuerySubscriptionResult = {
  refetch: () => void // A function to force refetch the query
}
```

- **Parameters**
  - `arg`: The argument passed to the query defined in the endpoint. You can also pass in `skipToken` here as an alternative way of skipping the query, see skipToken
  - `options`: A set of options that control the fetching behaviour of the hook
- **Returns**
  - An object containing a function to `refetch` the data

Description

A React hook that automatically triggers fetches of data from an endpoint, and 'subscribes' the component to the cached data.

The query arg is used as a cache key. Changing the query arg will tell the hook to re-fetch the data if it does not exist in the cache already.

Note that this hook does not return a request status or cached data. For that use-case, see `useQuery` or `useQueryState`.

Features

- Automatically triggers requests to retrieve data based on the hook argument and whether cached data exists by default
- 'Subscribes' the component to keep cached data in the store, and 'unsubscribes' when the component unmounts

- Accepts polling/re-fetching options to trigger automatic re-fetches when the corresponding criteria is met

useLazyQuery

Accessing a useLazyQuery hook

```
const [trigger, result, lastPromiseInfo] =
  api.endpoints.getPosts.useLazyQuery(options)
// or
const [trigger, result, lastPromiseInfo] = api.useLazyGetPostsQuery(options)
```

Signature

```
type UseLazyQuery = (
  options?: UseLazyQueryOptions
) => [UseLazyQueryTrigger, UseQueryStateResult, UseLazyQueryLastPromiseInfo]

type UseLazyQueryOptions = {
  pollingInterval?: number
  refetchOnReconnect?: boolean
  refetchOnFocus?: boolean
  selectFromResult?: (result: UseQueryStateDefaultResult) => any
}

type UseLazyQueryTrigger<T> = (arg: any, preferCacheValue?: boolean) =>
Promise<
  QueryResultSelectorResult
> & {
  arg: unknown // Whatever argument was provided to the query
  requestId: string // A string generated by RTK Query
  subscriptionOptions: SubscriptionOptions // The values used for the query
subscription
  abort: () => void // A method to cancel the query promise
  unwrap: () => Promise<T> // A method to unwrap the query call and provide
the raw response/error
  unsubscribe: () => void // A method used to manually unsubscribe from the
query results
  refetch: () => void // A method used to re-run the query. In most cases
when using a lazy query, you will never use this and should prefer to call
the trigger again.
  updateSubscriptionOptions: (options: SubscriptionOptions) () => void // A
method used to update the subscription options (eg. pollingInterval)
}

type UseQueryStateResult<T> = {
  // Base query state
  originalArgs?: unknown // Arguments passed to the query
  data?: T // The latest returned result regardless of trigger arg, if
present
  currentData?: T // The latest returned result for the trigger arg, if
present
  error?: unknown // Error result if present
  requestId?: string // A string generated by RTK Query
  endpointName?: string // The name of the given endpoint for the query
  startedTimeStamp?: number // Timestamp for when the query was initiated
  fulfilledTimeStamp?: number // Timestamp for when the query was completed

  isUninitialized: false // Query has not started yet.
  isLoading: false // Query is currently loading for the first time. No data
```

```
yet.
  isFetching: false // Query is currently fetching, but might have data from
an earlier request.
  isSuccess: false // Query has data from a successful load.
  isError: false // Query is currently in an "error" state.
}

type UseLazyQueryLastPromiseInfo = {
  lastArg: any
}
```

- **Parameters**
  - `options`: A set of options that control the fetching behavior and returned result value of the hook. Options affecting fetching behavior will only have an effect after the lazy query has been triggered at least once.
- **Returns**: A tuple containing:
  - `trigger`: A function that fetches the corresponding data for the endpoint when called
  - `result`: A query result object containing the current loading state, the actual data or error returned from the API call and metadata about the request. Can be customized with `selectFromResult`
  - `lastPromiseInfo`: An object containing the last argument used to call the trigger function

Description

A React hook similar to `useQuery`, but with manual control over when the data fetching occurs.

This hook includes the functionality of `useLazyQuerySubscription`.

Features

- Manual control over firing a request to retrieve data
- 'Subscribes' the component to keep cached data in the store, and 'unsubscribes' when the component unmounts
- Returns the latest request status and cached data from the Redux store
- Re-renders as the request status changes and data becomes available
- Accepts polling/re-fetching options to trigger automatic re-fetches when the corresponding criteria is met and the fetch has been manually called at least once

Note

When the trigger function returned from a LazyQuery is called, it always initiates a new request to the server even if there is cached data. Set `preferCacheValue`(the second argument to the function) as `true` if you want it to immediately return a cached value if one exists.

`useLazyQuerySubscription`

Accessing a useLazyQuerySubscription hook

```
const [trigger, lastArg] =
  api.endpoints.getPosts.useLazyQuerySubscription(options)
```

Signature

```
type UseLazyQuerySubscription = (
  options?: UseLazyQuerySubscriptionOptions
) => [UseLazyQuerySubscriptionTrigger, LastArg]

type UseLazyQuerySubscriptionOptions = {
  pollingInterval?: number
  refetchOnReconnect?: boolean
  refetchOnFocus?: boolean
}

type UseLazyQuerySubscriptionTrigger = (
  arg: any,
  preferCacheValue?: boolean
) => void
```

- **Parameters**
  - `options`: A set of options that control the fetching behavior of the hook. The options will only have an effect after the lazy query has been triggered at least once.
- **Returns**: A tuple containing:
  - `trigger`: A function that fetches the corresponding data for the endpoint when called
  - `lastArg`: The last argument used to call the trigger function

Description

A React hook similar to `useQuerySubscription`, but with manual control over when the data fetching occurs.

Note that this hook does not return a request status or cached data. For that use-case, see `useLazyQuery`.

Features

- Manual control over firing a request to retrieve data

- 'Subscribes' the component to keep cached data in the store, and 'unsubscribes' when the component unmounts
- Accepts polling/re-fetching options to trigger automatic re-fetches when the corresponding criteria is met and the fetch has been manually called at least once

`usePrefetch`

Accessing a usePrefetch hook

```
const prefetchCallback = api.usePrefetch(endpointName, options)
```

Signature

```
type UsePrefetch = (
  endpointName: string,
  options?: UsePrefetchOptions
) => PrefetchCallback

type UsePrefetchOptions =
  | {
      // If specified, only runs the query if the difference between `new
Date()` and the last
      // `fulfilledTimeStamp` is greater than the given value (in seconds)
      ifOlderThan?: false | number
    }
  | {
      // If `force: true`, it will ignore the `ifOlderThan` value if it is
set and the query
      // will be run even if it exists in the cache.
      force?: boolean
    }

type PrefetchCallback = (arg: any, options?: UsePrefetchOptions) => void
```

- **Parameters**
  - `endpointName`: The name of the endpoint to prefetch data for
  - `options`: A set of options that control whether the prefetch request should occur
- **Returns**
  - A `prefetch` callback that when called, will initiate fetching the data for the provided endpoint

Description

A React hook which can be used to initiate fetching data ahead of time.

Features

- Manual control over firing a request to retrieve data