`weather` -rtk-query

🚀 ИТОГОВЫЙ ПЛАН ИЗ 4 ЧЁТКИХ ШАГОВ

ШАГ 1 — Создать RTK Query API (новый файл)

Создаём `weatherApi.ts`.

🔥 *Это первый шаг, который ты реально делаешь.*

ШАГ 2 — Подключить его в store

Добавляем reducer + middleware.

ШАГ 3 — Переписать компонент на RTK Query

Заменяем `dispatch(fetchWeather(...))` на хук `useGetWeatherQuery`.

ШАГ 4 — Удалить лишний slice (по желанию)

Он больше не нужен

🟦 НИЖЕ — ЧЁТКИЙ ПЛАН С КОДОМ (копируй и делай)

✅ ШАГ 1 — Создай файл RTK Query: weatherApi.ts

📌 Создай файл:

`src/features/weather/weatherApi.ts`

```
import { createApi, fetchBaseQuery } from "@reduxjs/toolkit/query/react";
import type { WeatherResponse } from "./types/types";

const apiKey = "3603bb385cba1812ea388450e7b58c94";

export const weatherApi = createApi({
  reducerPath: "weatherApi",
  baseQuery: fetchBaseQuery({
    baseUrl: "https://api.openweathermap.org/data/2.5/",
  }),
  endpoints: (builder) => ({
    getWeather: builder.query<WeatherResponse, string>({
      query: (city) =>
        `weather?q=${city}&appid=${apiKey}&units=metric`,
    }),
  }),
});

export const { useGetWeatherQuery } = weatherApi;
```

👉 Теперь у нас есть API + хук для запроса.

✅ ШАГ 2 — Подключи weatherApi в store

Открой `src/app/store.ts` и добавь:

```
import { weatherApi } from "../features/weather/weatherApi";

export const store = configureStore({
  reducer: {
    weather: weatherReducer, // пока оставляем
    [weatherApi.reducerPath]: weatherApi.reducer, // добавляем
  },
```

```
  middleware: (getDefault) ⇒
    getDefault().concat(weatherApi.middleware), // добавляем
});
```

👉 Теперь RTK Query работает в проекте.

✅ ШАГ 3 — Перепиши Weather.tsx на RTK Query

Открой `Weather.tsx` и:

❌ УДАЛИ полностью:

- `useAppDispatch`
- `useAppSelector`
- `fetchWeather`
- `selectWeather`
- `dispatch(fetchWeather(city))`

🔥 Вместо этого вставь:

```
import { useState } from "react";
import { useGetWeatherQuery } from "./weatherApi";
import styles from "./Weather.module.css";

const Weather = () ⇒ {
  const [city, setCity] = useState("");

  const {
    data,
    error,
    isFetching,
    refetch,
  } = useGetWeatherQuery(city, {
```

```
  skip: !city, // не отправляем запрос пока нет города
});

const handleSearch = () => {
  if (!city) {
    alert("Введите город");
    return;
  }
  refetch(); // вручную запускаем запрос
};

return (
  <div className={styles.container}>
    <div className={styles.card}>
      <h1 className={styles.title}>Прогноз погоды (RTK Query)</h1>

      <inputtype="text"
        placeholder="Введите город"
        value={city}
        className={styles.input}
        onChange={(e) => setCity(e.target.value)}
      />

      <button className={styles.button} onClick={handleSearch}>
        Получить погоду
      </button>

      {isFetching && <p>Загрузка...</p>}
      {error && <p>Ошибка: город не найден</p>}

      {data && (
        <div className={styles.weatherBox}>
          <h2>{data.name}, {data.sys.country}</h2>
          <p>Температура: {data.main.temp}°C</p>
          <p>Погода: {data.weather[0].description}</p>
          <p>Влажность: {data.main.humidity}%</p>
```

```
          <p>Ветер: {data.wind.speed} м/с</p>
        </div>
      )}
    </div>
  </div>
 );
};

export default Weather;
```

👉 Теперь компонент полностью работает на RTK Query.

✅ ШАГ 4 — (По желанию) удалить weatherSlice

полностью удалить:

- `weatherSlice.ts`
- его reducer из store
- все старые типы ошибок/loading

🎉 ВОТ — ПРАВИЛЬНАЯ ЛОГИЧНАЯ ПОСЛЕДОВАТЕЛЬНОСТЬ

1️⃣ Создать RTK Query API

2️⃣ Подключить в store

3️⃣ Переписать компонент на новый хук

4️⃣ Удалить старый thunk/slice (если не нужен)

Я буду идти по файлам в логическом порядке:

1. `types.ts` — тип ответа от API
2. `weatherApi.ts` — RTK Query API
3. `store.ts` — подключение API
4. `Weather.tsx` — компонент

---

1. `src/features/weather/types/types.ts`

```
export interface WeatherResponse {
  name: string;
  sys: {
    country: string;
  };
  main: {
    temp: number;
    humidity: number;
  };
  weather: {
    description: string;
  }[];
  wind: {
    speed: number;
  };
}
```

Построчно:

- `export interface WeatherResponse {`

  Объявляем TypeScript-интерфейс `WeatherResponse` и экспортируем его, чтобы использовать в других файлах. Это форма данных, которые мы ожидаем получить от API.

- `name: string;`

  Название города. Строка. Например: `"Yerevan"`.

- `sys: {`

  Вложенный объект `sys` — кусок ответа от API.

- `country: string;`

  Внутри `sys` лежит `country` — двухбуквенный код страны, например `"AM"`.

- `main: {`

  Объект `main` — основные параметры погоды.

- `temp: number;`

  Температура в градусах Цельсия (мы запрашиваем `units=metric`).

- `humidity: number;`

  Влажность воздуха в процентах.

- `weather: {`

  Массив `weather` — список погодных условий (обычно один элемент, но вообще это массив).

- `description: string;`

  Текстовое описание погоды, например `"clear sky"`.

- `}[];`

  Квадратные скобки означают, что `weather` — это **массив объектов** с полем `description`.

- `wind: {`

  Объект `wind` — данные о ветре.

- `speed: number;`

  Скорость ветра, число (м/с).

- `};` и `}`

  Закрываем объекты и интерфейс.

---

2. src/features/weather/weatherApi.ts

```
import { createApi, fetchBaseQuery } from "@reduxjs/toolkit/query/react";
import type { WeatherResponse } from "./types/types";

const apiKey = "3603bb385cba1812ea388450e7b58c94";

export const weatherApi = createApi({
  reducerPath: "weatherApi",
  baseQuery: fetchBaseQuery({
    baseUrl: "https://api.openweathermap.org/data/2.5/",
  }),
  endpoints: (builder) => ({
    getWeather: builder.query<WeatherResponse, string>({
      query: (city) =>
        `weather?q=${city}&appid=${apiKey}&units=metric`,
    }),
  }),
});

export const { useGetWeatherQuery } = weatherApi;
```

Построчно:

- `import { createApi, fetchBaseQuery } from "@reduxjs/toolkit/query/react";`

  Импортируем функции RTK Query:

  - `createApi` — создаёт API-слайс
  - `fetchBaseQuery` — базовая функция для HTTP-запросов (обёртка над `fetch`).

- `import type { WeatherResponse } from "./types/types";`

  Импортируем тип `WeatherResponse`, чтобы типизировать данные, которые вернёт запрос.

- `const apiKey = "3603bb385cba1812ea388450e7b58c94";`

Хардкоженный API-ключ OpenWeather. В реале лучше хранить в `.env`, но для учебного примера ок.

- `export const weatherApi = createApi({`

  Создаём и экспортируем RTK Query API-слайс под именем `weatherApi`.

- `reducerPath: "weatherApi",`

  Имя ветки в Redux store, где будет лежать состояние запросов этого API.

  В сторе появится `state.weatherApi`.

- `baseQuery: fetchBaseQuery({`

  Указываем, как делать запросы по умолчанию.

- `baseUrl: "https://api.openweathermap.org/data/2.5/",`

  Базовый URL для всех запросов этого API. Всё, что мы укажем в `query`, будет добавляться к этому адресу.

- `endpoints: (builder) => ({`

  Функция, внутри которой мы описываем все эндпоинты (запросы).

  `builder` помогает создавать `query` и `mutation`.

- `getWeather: builder.query<WeatherResponse, string>({`

  Описываем запрос `getWeather`.

  - Первый generic: `WeatherResponse` — тип данных, которые вернёт запрос.
  - Второй generic: `string` — тип аргумента (город).

- `query: (city) =>`

  Функция, которая по аргументу `city` возвращает строку или объект с настройками запроса.

- `` `weather?q=${city}&appid=${apiKey}&units=metric`, ``

  Относительный путь к API. В итоге полный URL будет:

  `https://api.openweathermap.org/data/2.5/weather?q=Город&appid=...&units=metric`.

- `}),`

  Закрываем описание `getWeather`.

- `}),`

  Закрываем объект `endpoints`.

- `});`

  Закрываем `createApi`.

- `export const { useGetWeatherQuery } = weatherApi;`

  Достаём и экспортируем сгенерированный хук `useGetWeatherQuery`.

  Этот хук будем использовать в React-компоненте вместо `dispatch(fetchWeather(...))`.

---

3. Подключение в `store.ts`

Примерно так:

```
import { configureStore } from "@reduxjs/toolkit";
import weatherReducer from "../features/weather/weatherSlice";
import { weatherApi } from "../features/weather/weatherApi";

export const store = configureStore({
  reducer: {
    weather: weatherReducer,
    [weatherApi.reducerPath]: weatherApi.reducer,
  },
  middleware: (getDefault) ⇒
    getDefault().concat(weatherApi.middleware),
});

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

Построчно:

- `import { configureStore } from "@reduxjs/toolkit";`

  Импорт функции для создания Redux store.

- `import weatherReducer from "../features/weather/weatherSlice";`

  Импорт старого редьюсера погоды (createAsyncThunk-версия).

  Можно оставить ради сравнения/демо.

- `import { weatherApi } from "../features/weather/weatherApi";`

  Импортируем наш RTK Query API-слайс.

- `export const store = configureStore({`

  Создаём и экспортируем store.

- `reducer: {`

  Объект со всеми редьюсерами.

- `weather: weatherReducer,`

  Ветка `state.weather` — старый slice. Можно удалить позже.

- `[weatherApi.reducerPath]: weatherApi.reducer,`

  Динамическое имя ключа.

  Это то же самое, что

  `weatherApi: weatherApi.reducer`,

  потому что `reducerPath = "weatherApi"`.

- `middleware: (getDefault) =>`

  Настраиваем middleware.

- `getDefault().concat(weatherApi.middleware),`

  Берём стандартные middleware Redux Toolkit и добавляем RTK Query middleware.

  Оно отвечает за кэширование, рефетчинг и т.д.

- `});`

  Закрываем конфигурацию стора.

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

  Тип всего корневого состояния Redux.

- `export type AppDispatch = typeof store.dispatch;`

Тип диспатча, пригодится в `useAppDispatch`.

4. `src/features/weather/Weather.tsx` (RTK Query версия)

```
import { useState } from "react";
import { useGetWeatherQuery } from "./weatherApi";
import styles from "./Weather.module.css";

const Weather = () => {
  const [city, setCity] = useState("");

  const {
    data,
    error,
    isFetching,
    refetch,
  } = useGetWeatherQuery(city, {
    skip: !city,
  });

  const handleSearch = () => {
    if (!city) {
      alert("Введите город");
      return;
    }
    refetch();
  };

  return (
    <div className={styles.container}>
      <div className={styles.card}>
        <h1 className={styles.title}>Прогноз погоды (RTK Query)</h1>

        <inputtype="text"
          placeholder="Введите город"
```

```
        value={city}
        className={styles.input}
        onChange={(e) => setCity(e.target.value)}
      />

      <button className={styles.button} onClick={handleSearch}>
        Получить погоду
      </button>

      {isFetching && <p className={styles.loading}>Загрузка...</p>}
      {error && <p className={styles.error}>Ошибка: город не найден</p>}

      {data && (
        <div className={styles.weatherBox}>
          <h2>
            {data.name}, {data.sys.country}
          </h2>
          <p>Температура: {data.main.temp}°C</p>
          <p>Погода: {data.weather[0].description}</p>
          <p>Влажность: {data.main.humidity}%</p>
          <p>Ветер: {data.wind.speed} м/с</п>
        </div>
      )}
    </div>
  </div>
 );
};

export default Weather;
```

Построчно:

- `import { useState } from "react";`

  Импортируем React-хук `useState` для управления локальным состоянием города.

- `import { useGetWeatherQuery } from "./weatherApi";`

  Импортируем RTK Query хук, который сгенерировали в `weatherApi.ts`.

- `import styles from "./Weather.module.css";`

  Импорт CSS-модуля со стилями.

- `const Weather = () => {`

  Объявляем функциональный компонент `Weather`.

- `const [city, setCity] = useState("");`

  Создаём локальное состояние `city` (строка) и функцию `setCity` для изменения.

  Начальное значение — пустая строка.

- ```
  const { data, error, isFetching, refetch,
  } = useGetWeatherQuery(city, { skip: !city,
  });
  ```

  Построчно:

  - `const { data, error, isFetching, refetch } =`

    Делаем деструктуризацию результата работы хука.

  - `useGetWeatherQuery(city, {`

    Вызываем запрос с аргументом `city`.

    Второй аргумент — объект настроек.

  - `skip: !city,`

    Если `city` пустая строка → `!city === true` → запрос **не выполняется**.

    То есть, пока пользователь ничего не ввёл, запрос не уйдёт.

  - `});`

    Закрываем вызов хука.

  В итоге у нас есть:

  - `data` — результат запроса (ответ от API)
  - `error` — ошибка, если запрос не удался
  ```

- `isFetching` — флаг загрузки
- `refetch` — функция, чтобы можно было вручную перезапустить запрос

- `const handleSearch = () => {`

  Обработчик клика по кнопке "Получить погоду".

- `if (!city) {`

  Если город не введён...

- `alert("Введите город");`

  Показываем пользователю предупреждение.

- `return;`

  Прерываем выполнение функции.

- `}`

  Конец проверки.

- `refetch();`

  Если город введён — вручную запускаем запрос.

  Хотя хук уже «подписан» на `city`, мы используем `refetch`, чтобы показать студентам явный вызов.

- `};`

  Конец функции `handleSearch`.

Дальше — JSX:

- `<div className={styles.container}>`

  Корневой контейнер компонента с классом из CSS-модуля.

- `<div className={styles.card}>`

  Карточка по центру.

- `<h1 className={styles.title}>Прогноз погоды (RTK Query)</h1>`

  Заголовок страницы. Можно специально подписать, что это RTK Query версия.

Инпут:

```
<input
  type="text"
  placeholder="Введите город"
  value={city}
  className={styles.input}
  onChange={(e) ⇒ setCity(e.target.value)}
/>
```

- `type="text"` — обычное текстовое поле.
- `placeholder="Введите город"` — подсказка внутри.
- `value={city}` — controlled input, значение берётся из состояния.
- `onChange={(e) ⇒ setCity(e.target.value)}` — при вводе обновляем `city`.

Кнопка:

```
<button className={styles.button} onClick={handleSearch}>
  Получить погоду
</button>
```

- `onClick={handleSearch}` — при клике вызываем функцию, которая проверяет, введён ли город, и вызывает `refetch()`.

Состояния загрузки/ошибки:

```
{isFetching && <p className={styles.loading}>Загрузка...</p>}
{error && <p className={styles.error}>Ошибка: город не найден</p>}
```

- `isFetching && ...` — рендерим текст "Загрузка...", если запрос в процессе.

- `error && ...` — рендерим "Ошибка: город не найден", если была ошибка.

Вывод данных:

```
{data && (
  <div className={styles.weatherBox}>
    <h2>
      {data.name}, {data.sys.country}
    </h2>
    <p>Температура: {data.main.temp}°C</p>
    <p>Погода: {data.weather[0].description}</p>
    <p>Влажность: {data.main.humidity}%</p>
    <p>Ветер: {data.wind.speed} м/с</p>
  </div>
)}
```

- `data && (...)` — блок отрисуется только если `data` не `null`.
- `data.name` — город.
- `data.sys.country` — страна.
- `data.main.temp` — температура.
- `data.weather[0].description` — текст описания погоды из первого элемента массива.
- `data.main.humidity` — влажность.
- `data.wind.speed` — скорость ветра.

- `export default Weather;`

  Экспортируем компонент по умолчанию.

🟦 Откуда появляется хук `useGetWeatherQuery`?

Он **автоматически создаётся RTK Query**, когда ты вызываешь:

```
export const weatherApi = createApi({
  ...
  endpoints: (builder) ⇒ ({
    getWeather: builder.query({...})
  })
});
```

Вот что важно понять:

🧠 💡 RTK Query сам генерирует хуки для каждого эндпоинта

Если у тебя есть эндпоинт:

```
getWeather: builder.query(...)
```

То RTK Query автоматически создаёт хук с названием:

```
use + название эндпоинта + Query
```

то есть:

```
useGetWeatherQuery
```

И ты можешь достать его вот так:

```
export const { useGetWeatherQuery } = weatherApi;
```

🔍 Давай разберём построчно, *как именно он появляется*:

1. Мы создаём API:

```
export const weatherApi = createApi({ ... })
```

В этот момент RTK Query внутри себя строит объект API, который:

- хранит редьюсер
- создаёт middleware
- готовит кеш
- генерирует хуки (ВАЖНО!)

---

2. Мы описываем эндпоинты:

```
endpoints: (builder) ⇒ ({
  getWeather: builder.query(...)
})
```

Каждый эндпоинт должен иметь **имя**. В нашем случае это:

```
getWeather
```

---

3. RTK Query автоматически создаёт хук:

Для queries:

```
use<ИмяЭндпоинта>Query
```

Получается:

```
useGetWeatherQuery
```

---

4. Мы вытаскиваем хук из API:

```
export const { useGetWeatherQuery } = weatherApi;
```

Это называется **auto-generated hooks API** — стандартная фишка RTK Query.

📌 Итого в одной строке

🔵 **Хук `useGetWeatherQuery` появляется автоматически, потому что мы создали эндпоинт `getWeather` в `createApi`**

RTK Query сам генерирует хуки для всех запросов — тебе не нужно писать их вручную.

```
builder.query → auto hook generator → useGetWeatherQuery()
```

Middleware

1. **Что такое middleware вообще (простыми словами)**
2. **Что делает конкретно эта строка:**

```
middleware: (getDefault) ⇒
  getDefault().concat(weatherApi.middleware)
```

🧠 1. Что такое middleware (простое объяснение)

Представь, что в Redux у нас есть цепочка, через которую проходит каждый `dispatch`.

```
Компонент → dispatch(action) → редьюсер
```

Middleware — это «перехватчик», который стоит между dispatch и редьюсером.

Он получает каждое действие (action), может:

- посмотреть на него,
- изменить его,
- отправить дополнительный запрос,
- остановить его,
- или пропустить дальше.

Иными словами:

🔵 **Middleware — это как охранник на проходной:

он проверяет каждое действие перед тем, как оно попадёт в редьюсер.**

Пример аналогии:

- Ты заходишь в здание (dispatch)
- Охрана проверяет тебя (middleware)
- Ты идёшь дальше в кабинет (редьюсер)

Без middleware — ты проходишь сразу.

🟩 2. Зачем RTK Query нужен middleware?

RTK Query делает много магии:

- обновляет кэш
- следит, когда нужно перезапустить запрос
- отменяет запросы
- ищет одинаковые запросы и не дублирует их
- обновляет данные при refetch
- подписывается на данные

Чтобы эта магия работала, RTK Query нужен middleware.

Он должен перехватывать actions типа:

- `getWeather/started`
- `getWeather/success`
- `getWeather/error`
- `getWeather/refetch`

Поэтому мы обязаны добавить его в store.

🟦 3. Что означает конкретная строка в store

```
middleware: (getDefault) ⇒
  getDefault().concat(weatherApi.middleware)
```

Разбор построчно:

✔ `middleware:`

Мы настраиваем middleware в Redux store.

✔ `(getDefault) =>`

Redux Toolkit сам добавляет несколько стандартных middleware:

например, для обработки ошибок, проверки типов и т.д.

Эта функция получает массив этих стандартных middleware через `getDefault()`.

✔ `getDefault()`

Это массив **обычных middleware Redux Toolkit**.

Мы их НЕ хотим удалять — они важные.

Примерно как:

```
[reduxThunk, serializableCheck, immutableCheck]
```

✔ `.concat(weatherApi.middleware)`

`concat` — это как добавление элемента в конец массива.

Мы берём все стандартные middleware и **добавляем RTK Query middleware**.

То есть итоговый список выглядит так:

```
[
  ...defaultMiddleware,
  weatherApi.middleware
```

```
]
```

🟧✔️ Простое объяснение для студентов

👉 Эта строка означает:

🥤 **«Возьми стандартные настройки Redux и добавь к ним RTK Query механизм запросов.»**

Или совсем просто:

🐱 **«Мы подключаем в Redux специальный модуль, который управляет запросами RTK Query.»**

🟦 Полнейшая аналогия для студентов:

Представьте:

- defaultMiddleware — это обычный смартфон
- weatherApi.middleware — это камера, которую мы к нему прикручиваем

Эта строка — это момент, когда мы добавляем новое устройство:

```
телефон + камера
```

То есть Redux начинает уметь работать с API-запросами RTK Query.

Чем отличается вызов запроса автоматически от ручного `refetch()` ?

🟦 RTK Query делает два типа запросов:

👉 1. Автоматический запрос

(когда значение `city` изменилось)

👉 2. Ручной запрос через `refetch()`

И это **разные механизмы**.

Смотри.

🟩 1. Автоматический запрос (без refetch)

Хук:

```
useGetWeatherQuery(city)
```

всегда работает как подписка:

🔵 «Если изменилось значение `city` → сделай новый запрос».

То есть:

1. Пользователь вводит `"Moscow"`
2. `city` становится `"Moscow"`
3. Хук видит это изменение
4. И делает API-запрос

Это поведение — автоматическое ⛳

Ты ничего не вызываешь вручную.

🟧 2. Ручной запрос — `refetch()`

Когда ты вызываешь:

```
refetch();
```

Ты принудительно запускаешь **новый запрос**, даже если:

- `city` НЕ изменился
- запрос уже был ранее
- данные закешированы
- хук считает, что данные актуальны

То есть `refetch()` — это:

🔥 «Сделай запрос прямо сейчас, игнорируя все условия».

🟦 Сравнение: автоматический запрос vs refetch()

Механизм	Когда происходит	Что вызывает
Автоматический запрос	Когда изменилось значение аргумента (`city`)	React: изменение состояния
refetch()	Когда мы САМИ его вызвали	Принудительно запускаем новый запрос

🟫 Что важно: `refetch()` игнорирует кеш

Обычный запрос в RTK Query работает так:

```
если в кеше уже есть данные → не делай новый запрос
```

А `refetch()` принудительно заставит RTK Query сделать новый запрос **даже если в кеше есть данные**.

То есть:

🧽 `refetch()` → «обнови данные обязательно».

🟣 Важное резюме (для студентов)

✔️ `fetch automatically` (обычное поведение хука)

Срабатывает, когда меняется аргумент (`city`).

✔️ `refetch()` (ручной вызов)

Принудительно запускает новый запрос сейчас же, независимо от кеша.

🟢 Простой пример-аналогия

Автоматический запрос

Как датчик движения — ты прошёл → он включился.

refetch()

Как выключатель — ты сам включил.

🟣 Вспомогательная фраза для запоминания

> refetch — это кнопка "Обновить данные" вручную.

Если убрать кнопку и НЕ использовать `skip:` правильно.

Но самое главное — у тебя сейчас есть путаница из-за того, *как работает `skip` и аргумент хука*.

Давай разберём **предельно чётко** 👇

🟥 ВАЖНО: RTK Query делает запрос каждый раз, когда меняется аргумент хука

Если написать:

```
useGetWeatherQuery(city)
```

и `city` меняется по мере ввода:

```
"M" → "Mo" → "Mos" → "Mosc" → "Moscow"
```

ТО:

❗ Хук отправит 5 запросов

потому что **аргумент изменился 5 раз**.

Никакой `refetch()` тут не нужен — он вообще не участвует.

🟧 Почему мы передаём `city`, если skip=true?

Потому что:

- `refetch()` делает запрос с последним переданным аргументом
- этот аргумент — `city`

То есть мы передаём аргумент, но НЕ разрешаем автоматический запуск.

🟥 Если `skip: true`, то `refetch()` НЕ работает без дополнительной настройки.

Так работает сам RTK Query.

Чтобы `refetch()` работал — нужно выполнить одно из двух условий:

✅ Вариант 1 — Использовать `refetch` правильно (*и оставить skip: true*)

Если `skip: true`, то RTK Query не делает авто-запросы.

НО!

`refetch()` сработает ТОЛЬКО если хук уже однажды был инициализирован.

А у нас:

```
skip: true → запрос НЕ инициализируется → refetch() нечего перезапускат
ь
```

То есть refetch вызывает *повторный* запрос, но не *первый*.

🟩 Правильное решение: использовать lazy query

(это идеальный вариант для режима “запрос только по кнопке”)

Вместо:

```
useGetWeatherQuery(...)
```

нужно использовать:

```
useLazyGetWeatherQuery
```

Этот хук создан специально для таких случаев.

⚡ Делаем рабочий код (запрос только по кнопке)

1. Импортируем lazy-хук:

```
import { useLazyGetWeatherQuery } from "./weatherApi";
```

2. Получаем функцию-runner

```
const [trigger, { data, error, isFetching }] = useLazyGetWeatherQuery();
```

3. Вызываем запрос по кнопке:

```
const handleSearch = () ⇒ {
  if (!city) {
    alert("Введите город");
    return;
  }
  trigger(city);
};
```

✔️ Итоговый компонент

```
import { useState } from "react";
import { useLazyGetWeatherQuery } from "./weatherApi";
import styles from "./Weather.module.css";

const Weather = () ⇒ {
  const [city, setCity] = useState("");
```

```
const [trigger, { data, error, isFetching }] = useLazyGetWeatherQuery();

const handleSearch = () ⇒ {
  if (!city) {
    alert("Введите город");
    return;
  }
  trigger(city); // отправляем запрос вручную
};

return (
  <div className={styles.container}>
    <div className={styles.card}>
      <h1 className={styles.title}>Прогноз погоды (RTK Query)</h1>

      <inputtype="text"
        placeholder="Введите город"
        value={city}
        onChange={(e) ⇒ setCity(e.target.value)}
        className={styles.input}
      />

      <button className={styles.button} onClick={handleSearch}>
        Получить погоду
      </button>

      {isFetching && <p>Загрузка...</p>}
      {error && <p>Город не найден</p>}

      {data && (
        <div className={styles.weatherBox}>
          <h2>{data.name}, {data.sys.country}</h2>
          <p>Температура: {data.main.temp}°C</p>
          <p>Погода: {data.weather[0].description}</p>
          <p>Влажность: {data.main.humidity}%</p>
```

```
          <p>Ветер: {data.wind.speed} м/с</p>
        </div>
      )}
    </div>
  </div>
 );
};

export default Weather;
```

🟦 Почему этот вариант правильный ?

Потому что:

✔️ не делает запросы при вводе каждой буквы

✔️ работает строго по кнопке

✔️ ВИДНО разницу между:

- `useGetWeatherQuery` (автозапрос)
- `useLazyGetWeatherQuery` (ручной запуск)

И это прямо по смыслу похоже на:

```
axios.get()        ← вручную
useEffect(fetch)   ← автоматически
```

🟦 ИТОГ: Разница между обычным Redux Toolkit (RTK) и RTK Query

🟩 1. RTK (Redux Toolkit) — *когда пишем всё вручную*

Используем:

- `createSlice`
- `createAsyncThunk`
- `extraReducers`
- `dispatch`
- `useSelector`

Как работает:

1. Создаём asyncThunk для запроса
2. Делаем `axios.get` внутри
3. Прописываем loading / error / data вручную
4. Создаём slice и редьюсер
5. Описываем extraReducers
6. Вызываем `dispatch(fetchWeather())`
7. Читаем данные из Redux через `useSelector`

Минусы:

- 🚫 Много кода
- 🚫 Нужно вручную описывать стейты загрузки, ошибок
- 🚫 Нужно самому управлять кешем
- 🚫 Каждое действие — extraReducers
- 🚫 Нужно вручную вызывать dispatch

Плюсы:

- ✔️ Полный контроль
- ✔️ Понятен новичкам (пошаговая логика)
- ✔️ Подходит для локальных не-API задач

🟦 2. RTK Query — *когда всё делает библиотека*

Используем:

- `createApi`
- `fetchBaseQuery`
- `builder.query`
- автогенерируемые хуки: `useGetWeatherQuery`, `useLazyGetWeatherQuery`

Как работает:

1. Описываем API внутри `createApi`
2. RTK Query сам:
    - делает запрос
    - кэширует результат
    - выдает статус загрузки
    - сохраняет ошибку
    - обновляет данные
3. Из `weatherApi` автоматически создаётся хук для компонента

В компоненте:

```
const { data, error, isLoading } = useGetWeatherQuery(city);
```

Или вручную по кнопке:

```
const [trigger, result] = useLazyGetWeatherQuery();
trigger(city);
```

Плюсы:

- ✔️ На 70–90% меньше кода
- ✔️ Нет slice, thunk, extraReducers
- ✔️ Автогенерация хуков
- ✔️ Автокэширование
- ✔️ Автоматический рефетч
- ✔️ Управление запросами встроено в библиотеку
- ✔️ Меньше ошибок

Минусы:

- ⚠️ Студенты сначала путаются в `skip`, `refetch`, `lazy`
- ⚠️ Нужно объяснить middleware
- ⚠️ Нельзя использовать без Redux store

🟧 Главная идея

🔵 RTK = обычный Redux, облегчающий разработку

Но всё равно **ты сам пишешь всю логику запросов**.

🟣 RTK Query = полностью автоматизированный инструмент для работы с API

Он **убирает 80% ручной работы**.

🟩 Итог в одной строке

> RTK — инструмент для работы со стейтом.
> RTK Query — инструмент для работы с сервером.