COS30043 - 10.2HD - Using TanStack Query for Data Fetching in Vue

Author: Ta Quang Tung - 104222196

This tutorial extends the material taught during week 7 by showing an advanced method to fetch data in Vue.js using the TanStack Query library.

Video link: https://youtu.be/qdutXCjzL-w. Alternatively, you can follow along with this written tutorial.
Link to full GitHub repository: https://github.com/pine04/tanstack-query-demo.

A single page application loads the entire user interface on the initial request. As the user interacts with the app, it needs to dynamically request data from the backend to update the current webpage instead of loading an entirely new page. To achieve this, one could use either the Fetch API native to JavaScript or the Axios library. These technologies alone are adequate in small-scale applications, but as the application scales up and requires more complex data fetching strategies, these libraries by themselves are not enough. They do not handle use cases such as caching or refetching stale data.

TanStack Query is a frontend data fetching library that helps solve these problems. A popular library available across many frontend frameworks such as Vue, React, and Angular, TanStack Query offers comprehensive data fetching solutions that are very intuitive to use. In this tutorial, I will show you how to use this library in Vue.js with a basic example of a social media app.

To follow this tutorial, you will need Node installed on your machine. You can follow the instruction steps here. I will not go into details on the Node installation process to focus on the TanStack Query library. Node comes with the built-in npm package manager, which is the one we are going to use for this tutorial. I am using Node version 20.11.1.

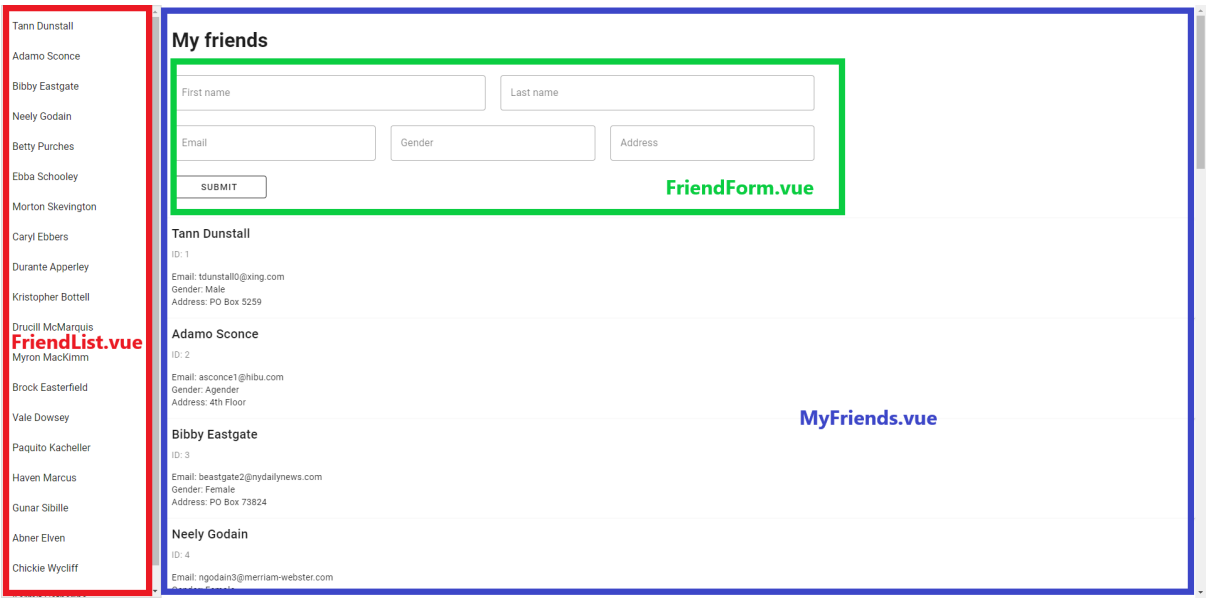
This tutorial utilizes a custom API written in Express. The code for this backend can be found in this repository.

Next, we need to create a Vue 3 app. I am using Vuetify to quickly scaffold my UI, but you are totally free to use whatever UI libraries you want or skip them altogether. I will create my Vuetify app with this terminal command:

```
npm create vuetify@latest
```

This command will then ask for a number of settings related to the project. Please select Barebones for presets, NO for TypeScript, npm for the package installer, and YES for install dependencies.

Before we get into the interesting details of TanStack Query, let's quickly scaffold the UI. Our application will look like the following:



The UI will be split into two main sections: the left sidebar (FriendList.vue) and the main content (MyFriends.vue). Inside MyFriends.vue is a form where the user can add a new friend (FriendForm.vue). The following code snippets show the template for each component. Note that I will exclusively be using the Composition API for this tutorial.

FriendList.vue

```
<template>
    <v-navigation-drawer border="md">
        <v-list>
            <v-list-item v-for="friend in friends" :key="friend">
                {{ friend.first_name + " " + friend.last_name }}
            </v-list-item>
        </v-list>
    </v-navigation-drawer>
</template>

<script setup>
import { ref } from "vue";

const friends = ref([
    // Friend data will go inside this.
]);
</script>
```

MyFriends.vue

```
<template>
    <h1 class="mx-4 my-8">My friends</h1>

    <FriendForm></FriendForm>

    <v-card v-for="friend in friends">
        <v-card-title>{{ friend.first_name + " " + friend.last_name
}}</v-card-title>
        <v-card-subtitle>ID: {{ friend.id }}</v-card-subtitle>
        <v-card-text>
            <p>Email: {{ friend.email }}</p>
            <p>Gender: {{ friend.gender }}</p>
            <p>Address: {{ friend.address }}</p>
        </v-card-text>
    </v-card>
</template>

<script setup>
import { ref } from "vue";
import FriendForm from "./FriendForm";

const friends = ref([
    // Friend data will go inside this.
]);
</script>
```

FriendForm.vue

```
<template>
    <v-form variant="outlined" class="mx-4 my-8" style="max-width:
64rem;">
        <v-row>
            <v-col cols="6">
                <v-text-field label="First name"
v-model="form.firstName" variant="outlined" hide-details></v-text-field>
            </v-col>
            <v-col cols="6">
                <v-text-field label="Last name" v-model="form.lastName"
variant="outlined" hide-details></v-text-field>
            </v-col>
        </v-row>
        <v-row>
            <v-col cols="4">
                <v-text-field label="Email" v-model="form.email"
variant="outlined" hide-details></v-text-field>
```

```
            </v-col>
            <v-col cols="4">
                <v-text-field label="Gender" v-model="form.gender"
variant="outlined" hide-details></v-text-field>
            </v-col>
            <v-col cols="4">
                <v-text-field label="Address" v-model="form.address"
variant="outlined" hide-details></v-text-field>
            </v-col>
        </v-row>
        <v-row>
            <v-col cols="2">
                <v-btn type="submit" variant="outlined"
width="100%">Submit</v-btn>
            </v-col>
        </v-row>
    </v-form>
</template>

<script setup>
import { reactive } from "vue";

const form = reactive({
    firstName: "",
    lastName: "",
    email: "",
    gender: "",
    address: ""
});
</script>
```

App.vue

```
<template>
  <v-app>
    <v-main>
      <MyFriends />
    </v-main>

    <FriendList />
  </v-app>
</template>

<script setup>
//
```

```
</script>
```

We can now get started with TanStack Query. First, we will install the library by typing the following command in the terminal:

```
npm i @tanstack/vue-query
```

After installing, we need to add the two highlighted lines in the main.js file:

```
/**
 * main.js
 *
 * Bootstraps Vuetify and other plugins then mounts the App`
 */

// Plugins
import { registerPlugins } from '@/plugins'
import { VueQueryPlugin } from '@tanstack/vue-query' // ADD THIS

// Components
import App from './App.vue'

// Composables
import { createApp } from 'vue'

const app = createApp(App)

registerPlugins(app)
app.use(VueQueryPlugin) // AND THIS

app.mount('#app')
```

We are now ready to start using this library in our code. The simplest, yet most important function from this library is `useQuery()`, which is used to get some data from the server. It is used with GET requests to the server. Navigate to the MyFriends.vue component and add the following lines into `<script setup>`:

```
import { useQuery } from "@tanstack/vue-query";

async function fetchFriends() {
    const response = await fetch("http://localhost:8000/friends");
    if (!response.ok) {
        throw new Error("Could not fetch data.");
    }
```

```
    const data = await response.json();
    return data.friends;
}

const { isPending, isError, data, error } = useQuery({
    queryKey: ["friends"],
    queryFn: fetchFriends
});
```

The function fetchFriend is an asynchronous function that sends a GET request to the server using the Fetch API. This function is very similar to what you may have written if you have used the Fetch API before. It will either throw an error if the request is not successful, or return a Promise that resolves to an array of friends. I am using a custom Express backend that returns a JSON object which has an array of friends for this endpoint. Each friend is an object with an ID, first name, last name, email, gender, and password.

The useQuery function takes an object as its parameters. This object accepts a number of values, but two required ones are queryKey and queryFn. QueryKey is an array of values that determine the caching behavior of the query. When queryKey changes, the query will be refetched from the server. In our case, the queryKey has one constant value, but in more complex cases it can take objects and variables. QueryFn is the actual function that does the data fetching. In our case it is fetchFriend. A function supplied to queryFn must either throw an error or return a resolved or rejected Promise.

useQuery returns an object with various values. In our sample code, we have destructured this object into four variables. isPending is a boolean value that will be true if there is no cached data and the query is not finished. isError is a boolean value that will be true if queryFn throws an error or returns a rejected Promise. data is the data returned from the query. error is the error object produced when the query fails. All of these values are returned as refs, which means that the UI will respond to their changes.

To use these values in our UI, we will first remove the friends ref we created earlier in <script setup>:

```
// REMOVE THESE LINES
const friends = ref([
    // Friend data will go inside this.
]);
```

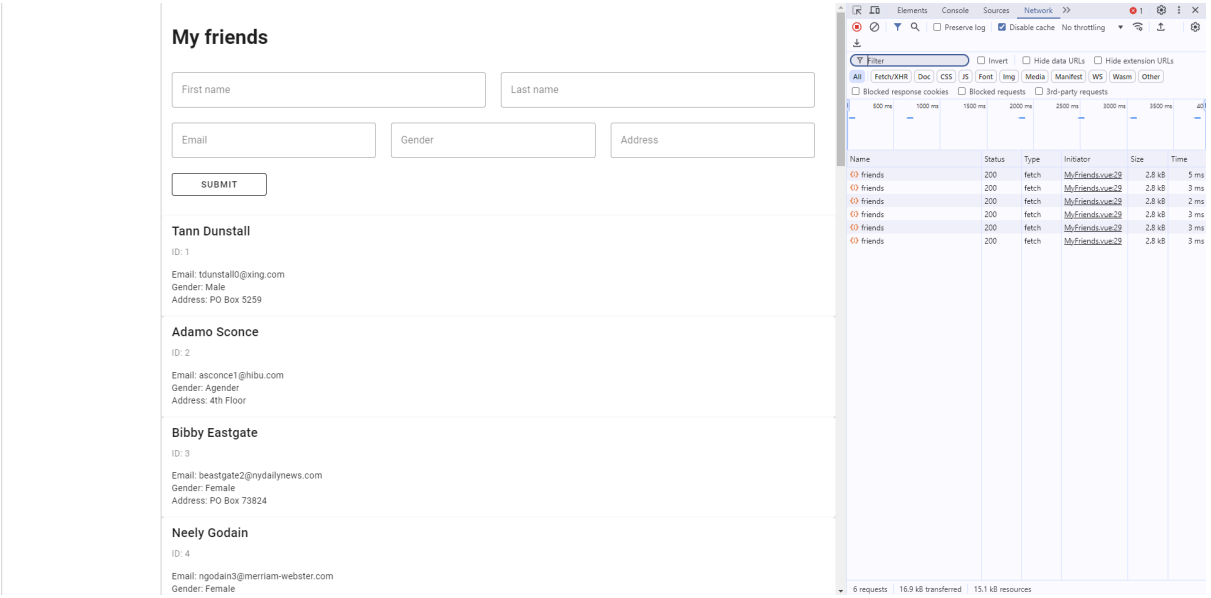
In the <template> markup, we will update the code for rendering the list of friends as follows:

```
<p v-if="isPending">Loading...</p>
<p v-else-if="isError">{{ error }}</p>
<v-card v-else v-for="friend in data">
    <v-card-title>{{ friend.first_name + " " + friend.last_name
```

```
}}</v-card-title>
    <v-card-subtitle>ID: {{ friend.id }}</v-card-subtitle>
    <v-card-text>
        <p>Email: {{ friend.email }}</p>
        <p>Gender: {{ friend.gender }}</p>
        <p>Address: {{ friend.address }}</p>
    </v-card-text>
</v-card>
```

This code uses conditional rendering to reflect the status of the query. If the query is pending, a Loading message will be shown. If an error happens, the error message will be displayed. Otherwise, the query has completed and we can render the data.

Although TanStack Query caches your fetched data, by default it considers all cached data stale, even if it has just been fetched. Also, by default, stale data is refetched when you leave and click back to the browser tab. To see this in action, open Inspect > Network and click away then come back to the webpage a few times. You will see several new network requests appearing in the list.



In our case, refetching like this is unnecessary and wasteful. We can disable this behavior by setting the refetchOnWindowFocus option to false in useQuery:

```
const { isPending, isError, data, error } = useQuery({
    queryKey: ["friends"],
    queryFn: fetchFriends,
    refetchOnWindowFocus: false
});
```

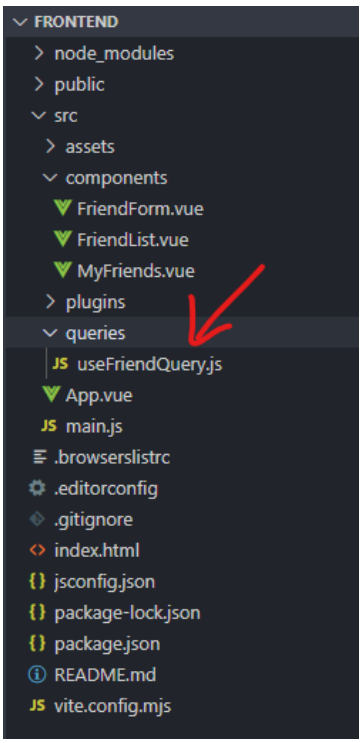
We can now do the same for our FriendList.vue sidebar. Copy and paste the code over to

that file and remove unnecessary code. The result should look like this.

```
<template>
    <v-navigation-drawer border="md">
        <v-list>
            <p v-if="isPending">Loading...</p>
            <p v-else-if="isError">{{ error }}</p>
            <v-list-item v-else v-for="friend in data" :key="friend">
                {{ friend.first_name + " " + friend.last_name }}
            </v-list-item>
        </v-list>
    </v-navigation-drawer>
</template>

<script setup>
import { useQuery } from "@tanstack/vue-query";

async function fetchFriends() {
    const response = await fetch("http://localhost:8000/friends");
    if (!response.ok) {
        throw new Error("Could not fetch data.");
    }
    const data = await response.json();
    return data.friends;
}

const { isPending, isError, data, error } = useQuery({
    queryKey: ["friends"],
    queryFn: fetchFriends,
    refetchOnWindowFocus: false
});
</script>
```

But notice that our MyFriends.vue and FriendList.vue files have duplicated query code. More specifically, the function fetchFriends and the useQuery invocation have been repeated. For better readability and reusability, we can extract this commonality into another file.

In the src directory, create a new directory called queries (you can call it whatever you want though) and create a file named useFriendQuery.js.



Inside this query, copy, paste, and modify the code to fetch friends like this.

```
import { useQuery } from "@tanstack/vue-query";

async function fetchFriends() {
    const response = await fetch("http://localhost:8000/friends");
    if (!response.ok) {
        throw new Error("Could not fetch data.");
    }
    const data = await response.json();
    return data.friends;
}

export default function useFriendQuery() {
    const queryResult = useQuery({
        queryKey: ["friends"],
        queryFn: fetchFriends,
        refetchOnWindowFocus: false
    });

    return queryResult;
}
```

Here, we created a new function called useFriendQuery that calls useQuery and returns the result. Notice that we don’t immediately destructure the result value. Instead, we will do this in the components that use this query.

Next, update the <script setup> code for MyFriends.vue and FriendList.vue like this:

MyFriends.vue

```
<script setup>
import useFriendQuery from "../queries/useFriendQuery";
import FriendForm from "./FriendForm";

const { isPending, isError, data, error } = useFriendQuery();
</script>
```

FriendList.vue

```
<script setup>
import useFriendQuery from "../queries/useFriendQuery";

const { isPending, isError, data, error } = useFriendQuery();
</script>
```

Now our code works and looks much cleaner.

At this point, you might be wondering if the same could be done for POST, PUT, or DELETE requests. The answer is no as these requests are the result of user interactions and modify data on the server. They should not and cannot follow the same behaviors as GET requests (imagine how disastrous it would be if a DELETE request was sent every time the user navigated back to the browser tab!). Requests like these use a different mechanism called useMutation().

We will demonstrate the use of useMutation with our FriendForm.vue component. Update the code for this component as highlighted in red:

```
<template>
    <v-form variant="outlined" class="mx-4 my-8" style="max-width:
64rem;" @submit="handleFormSubmit">
        // ... LIKE BEFORE
    </v-form>
</template>

<script setup>
import { useMutation } from "@tanstack/vue-query";
import { reactive } from "vue";

const form = reactive({
    first_name: "",
```

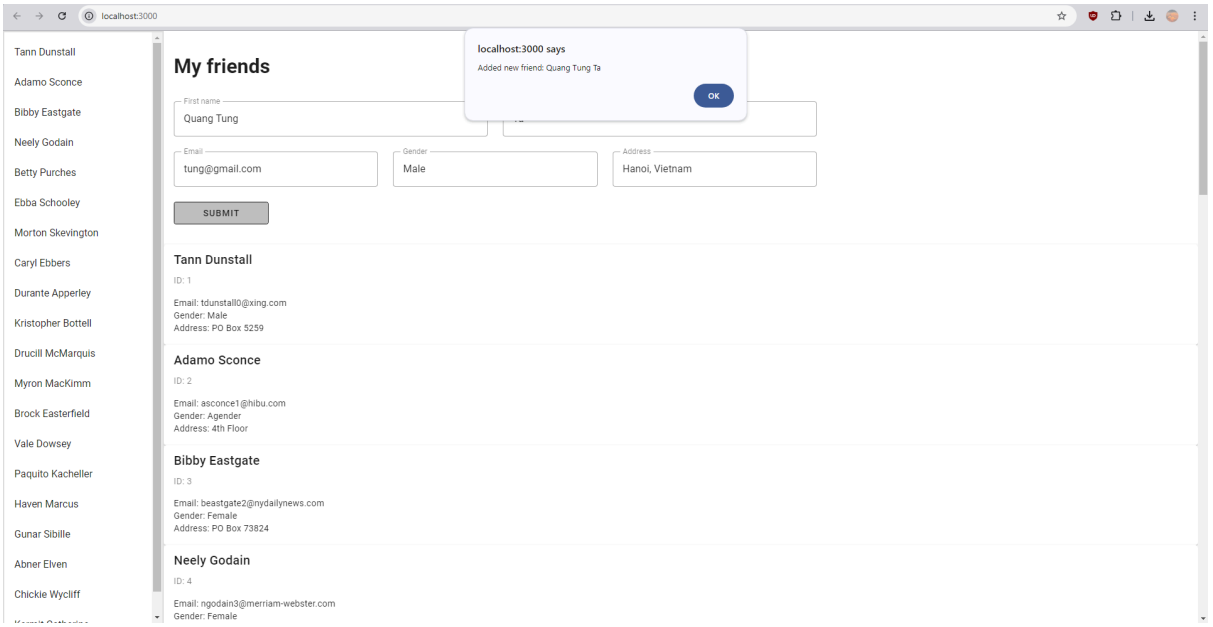
```
    last_name: "",
    email: "",
    gender: "",
    address: ""
});

function handleFormSubmit(e) {
    e.preventDefault();
    mutate();
}

async function addFriend() {
    const options = {
        method: "POST",
        headers: {
            "Content-Type": "application/json"
        },
        body: JSON.stringify(form)
    }
    const response = await fetch("http://localhost:8000/friends",
options);
    if (!response.ok) {
        throw new Error("Could not add friend.");
    }
    const data = await response.json();
    return data.friend;
}

const { mutate } = useMutation({
    mutationFn: addFriend,
    onSuccess: (newFriend) => alert(`Added new friend:
${newFriend.first_name} ${newFriend.last_name}`)
});
</script>
```

addFriend is an asynchronous function that sends a POST request to the server containing the form data to create a new friend. It is very similar to the fetchFriends function above, except it makes a POST request instead of GET. It either throws an error or returns a Promise that resolves with the newly created friend object.

The code below shows how useMutation is used. This function takes an object as its parameter, inside which is a required mutationFn. mutationFn in this case is the addFriend function above. This example also provides useMutation with a callback function that is called when the request succeeds. This callback takes the data returned from mutationFn and alerts the user with the name of the newly created friend. useMutation returns a number of values, however the most important one is mutate. It is a function that must be called to

trigger the request to the server. In our case, we call mutate in the handleFormSubmit function, which is attached to the submit event of the form.

Filling out the form and clicking the submit button, we get the following result:



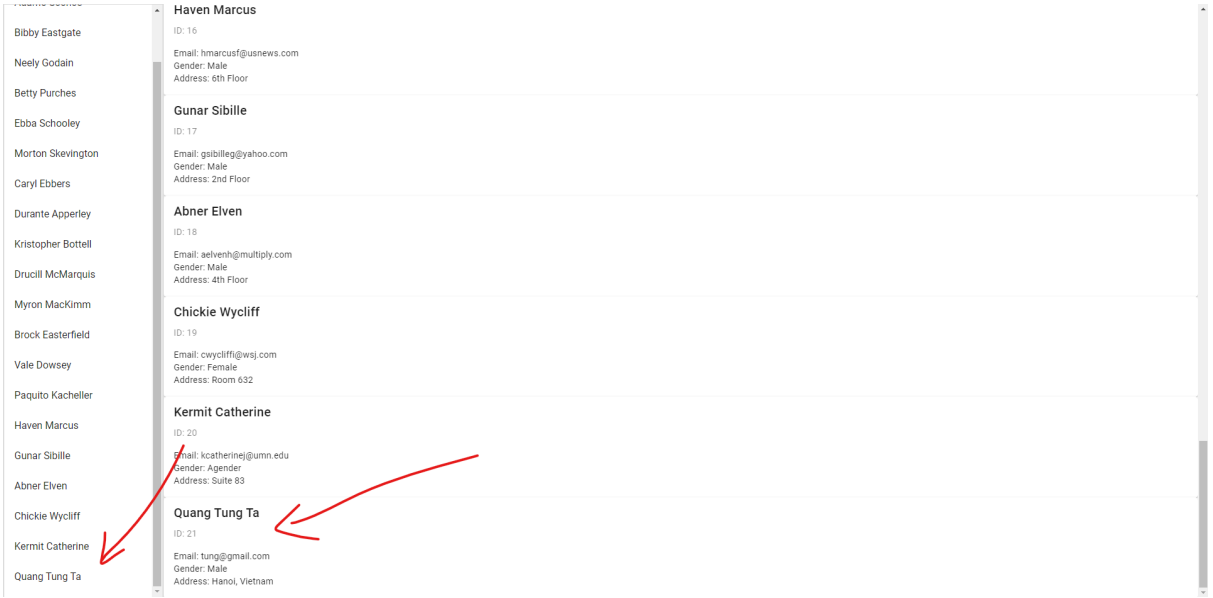
Refreshing the page and scrolling to the bottom of the sidebar and main content, we see that the new friend has indeed been added.

So far, everything is great. But there is one more thing we can improve. It would be really nice if every time we added a friend, the list of friends would automatically update. Thankfully, TanStack Query has a very easy way to do this. Inside <script setup> of FriendForm.vue, add the following highlighted parts:

```
<script setup>
import { useMutation, useQueryClient } from "@tanstack/vue-query";
import { reactive } from "vue";

// ...SAME AS BEFORE

const queryClient = useQueryClient();

const { mutate } = useMutation({
    mutationFn: addFriend,
    onSuccess: (newFriend) => {
        alert(`Added new friend: ${newFriend.first_name}
${newFriend.last_name}`);
        queryClient.setQueryData(["friends"], (oldData) => [...oldData,
newFriend]);
    }
})
</script>
```

Additions to this code includes a call to useQueryClient. This simple function returns the current query client, which is responsible for interacting with the query cache. More important is the inclusion of the setQueryData call inside onSuccess. This function has several variants but the one we use takes two parameters, the key of the query whose data we want to update and an updater function which is invoked to update the data. When called, this updater function will be passed the old data, which is an array of friends. The new data needs to be a different array of friends with the new friend inserted, and we use the spread syntax to conveniently achieve this.

Run the app again and you will see that the list is automatically updated every time the user adds a friend:



This concludes the tutorial for TanStack Query. What I have gone through is only the tip of the iceberg and there is much more to explore from this awesome library. I highly recommend checking out the library's official documentation and examples to understand how to better use it for your use case. I hope this tutorial has been useful to you. Thank you for reading.