

Tanstack Query와 SWR



1. Tanstack Query

❖ Tanstack Query란?

- 웹 애플리케이션에서 서버의 상태에 대한 가져오기, 캐싱, 동기화, 업데이트를 편하게 할 수 있도록 도와주는 라이브러리
- 서버 상태를 다루는 라이브러리

❖ 등장 배경

- 서버 상태가 로컬 상태, 전역상태와는 다른점
 - 원격에 위치하므로 프론트에서 직접 제어할 수 없음
 - 비동기 처리가 필수적임
 - 내가 서버 상태를 fetch해서 조회하는 동안 다른 클라이언트가 상태를 변경할 수 있음
 - 제때에 갱신하지 않으면 오래된 데이터(out of date data)를 사용자가 보게 됨
- 서버 상태 처리와 관련된 어려운 점
 - 캐싱 기능
 - 오래된 캐시 데이터에 대한 업데이트 -> 우선 오래된 데이터인지를 식별할 수 있는 기능 필요(예: fetch한 시점)
 - Paging, Lazy Loading과 같은 성능 최적화 기능
 - 같은 서버 상태에 대한 여러 요청의 중복을 제거하여 단일 요청으로 처리할 수 있는 기능

1.1 Tanstack Query 개요

❖react-query 설치

```
npm install --save @tanstack/react-query  
//devtools설치  
npm install --save @tanstack/react-query-devtools
```

❖Tanstack Query의 핵심 두가지 작업

- Query : 키를 이용해 여러 쿼리에 대한 결과를 관리
 - 쿼리 결과는 다양한 상태를 가짐
 - fresh : 신선한 상태의 최신 데이터
 - stale : 오래된 데이터
 - inactive : 비활성화된 데이터
- Mutation : 추가, 수정, 삭제와 같은 백엔드 API와의 데이터 변경 작업을 관리

1.2 Tanstack Query 프로젝트 구성

❖프로바이더 설정

- src/main.tsx

```
.....(생략)
const queryClient = new QueryClient();

ReactDOM.createRoot(document.getElementById("root")!).render(
  <React.StrictMode>
    <BrowserRouter>
      <QueryClientProvider client={queryClient}>
        <App />
        <ReactQueryDevtools initialIsOpen={false} />
      </QueryClientProvider>
    </BrowserRouter>
  </React.StrictMode>
);
```

- QueryClient 객체를 생성하여 QueryClientProvider 컴포넌트에 전달함
- QueryClientProvider 컴포넌트는 자식 컴포넌트로 QueryClient를 사용할 자식 컴포넌트를 포함함
- 개발자 도구를 이용하고 싶다면 ReactQueryDevtools 컴포넌트를 자식 컴포넌트와 함께 포함시킴

1.3 useQuery

❖Query?

- 비동기 원본 데이터에 대한 queryKey로 연결된 선언적 의존성
- 서버에서 데이터를 가져오기 위해 Promise 기반 메서드와 함께 쿼리를 사용함

❖useQuery?

- 컴포넌트 또는 사용자 정의 훅에서 쿼리를 구독하기 위해 사용하는 React-Query가 제공하는 훅
- 필수 옵션: queryKey, queryFn

❖사용법 형태

```
const queryObject = useQuery({
  queryKey: ["fetchTodoList", owner],
  queryFn: () => fetchTodoList({ owner }),
  ...(기타 선택적인 옵션들)
})
```

1.3 useQuery

❖useQuery 옵션

■ 필수 옵션

- queryKey : 쿼리를 식별하기 위한 고유 키를 전달함. 문자열 또는 배열을 이용함
- queryFn : 쿼리를 수행하는 함수를 지정함. 반드시 Promise를 리턴해야 함

■ 기타 선택적인 옵션들

- staleTime : 오래된 데이터로 간주하는 시간. 기본값-0
- gcTime : inactive 데이터가 메모리에 남아 있는 시간. 이 시간이 지나면 GC가 수행됨. 기본값-5*60*1000
- refetchInterval : refetch를 수행하는 주기. 밀리초 단위로 설정함. 기본값- false
- refetchOnMount : 컴포넌트가 마운트될 때 refetch를 자동으로 수행할 것인지 여부 지정. 기본값-true
- enabled : useQuery가 설정된 후 쿼리를 자동으로 수행하도록 할것인지 여부 지정. 기본값-true
- retry : 쿼리가 실패했을 때 재시도할 것인지, 몇번 할것인지를 지정함. true/false, number. 기본값-3.

1.3 useQuery

❖useQuery 리턴값

▪ queryObject

- status : 쿼리의 상태정보
 - pending : 캐시된 데이터가 없고, 쿼리 시도가 아직 완료되지 않은 상태
 - 이밖에도 error, success, false
 - fetchStatus : 데이터 가져오기 작업의 상태 정보
 - fetching : 이 값이 true이면, queryFn이 실행중임을 의미함. 초기 pending, background refetch도 포함함
 - isSuccess : true/false
 - isPending : true/false, 현재 Pending 상태인지 확인
 - isFetching : true/false
 - isLoading : isFetching & isPending 의 의미
 - isStale : true/false. 이 값이 true이면 현재의 데이터가 stale(오래된) 상태
 - data : 수신 데이터
 - error : 에러발생시 주어지는 에러 객체. null 또는 TError
 - refetch : 데이터를 다시 가져오기 위한 함수
-
- 참조 : <https://tanstack.com/query/latest/docs/framework/react/reference/useQuery#usequery>

1.4 useQuery 적용 예제

❖useQuery를 이용해 조회기능 작성

■ 미리 주어지는 예제

- 백엔드 : todosvc
 - 두 소유자(owner)의 샘플 데이터 제공 : gdhong, mrlee
 - npm install 실행 후 npm run dev로 서버 기동
 - http://localhost:3000
- 프론트엔드 : react-query-todolist-1
 - 전체적인 틀이 완성되어 있음
 - 백엔드 API 와 통신할 수 있는 API 미리 제공 : src/apis/ToDoAPI.ts
 - react-router를 이용해 3개의 라우팅 페이지 미리 제공
 - » Home, AddTodo, EditTodo
 - main.tsx에 QueryClientProvider 설정 완료
 - Spinner UI로 react-cssspin 사용

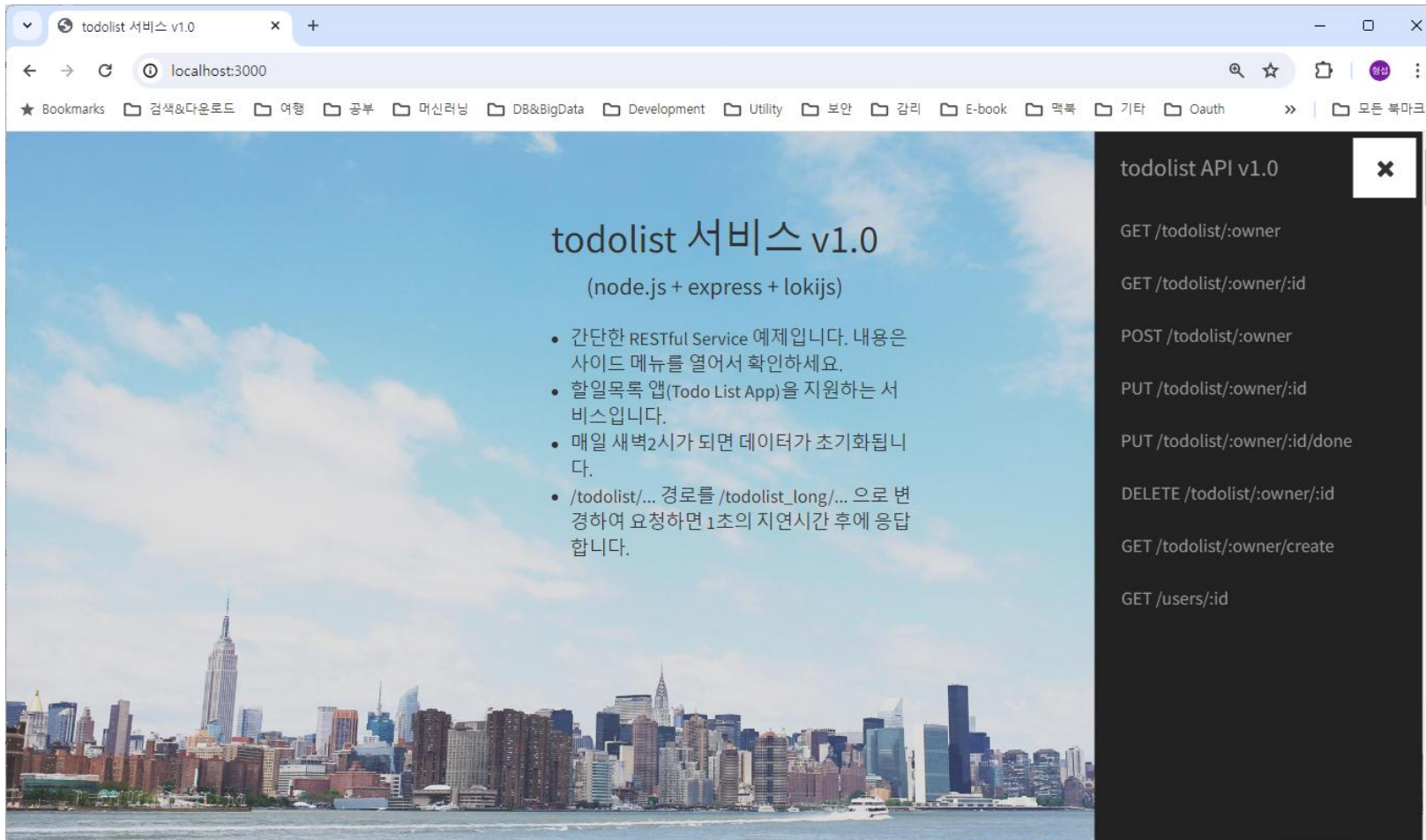
■ 작성할 기능

- TodoList 컴포넌트에 useQuery를 이용해 특정 소유자(owner)의 todoList를 조회함

1.4 useQuery 적용 예제

❖ todosvc 백엔드 API : http://localhost:3000

- 오른쪽 사이드 메뉴에서 제공 API 기능 확인



1.4 useQuery 적용 예제

❖ TodoList 컴포넌트 변경

- src/components/TodoList.tsx 변경


```
.....(생략 - import는 자동생성 기능활용)
type PropsType = { owner: string };
const TodoList = ({ owner }: PropsType) => {
  const { data, error, isFetching } = useQuery({
    queryKey: ["fetchTodoList", owner],
    queryFn: () => fetchTodoList({ owner }),
  });
  return (
    <div>
      {" "}
      <ul>
        {data &&
          data?.map((todoItem: TodoType) => {
            return <TodoListItem key={todoItem.id} todoItem={todoItem} />;
          })
        }
      </ul>
      { isFetching ? <ReactCsspin opacity={0.8} message="로딩중입니다" /> : "" }
      { error ? <h3>에러발생 : {error.message}</h3> : "" }
    </div>
  );
};
export default TodoList;
```

1.4 useQuery 적용 예제


❖ 1단계 적용 결과

[Home](#) | [AddTodo](#)

할일 소유자 :




로딩중입니다



[Home](#) | [AddTodo](#)

할일 소유자 :

- 남원구경 - 고향집에 가봐야합니다. (완료)
- 저녁약속(10.11) - 지인과의 중요한 저녁 약속입니다.
- AWS 밋업 - AWS 밋업에 반드시 참석해야 합니다.
- AAI 모임 - 공인강사들 모임이 있습니다. (완료)



1.5 useMutation

❖Mutation?

- 데이터를 생성,업데이트,삭제하거나 서버의 부작용(side effects)을 수행하는 것

❖useMutation?

- Mutation을 처리할 때 사용하는 React Query가 제공하는 훅
- 필수 옵션 : mutationFn

❖사용 방법

```
const mutationObject = useMutation({  
  mutationFn: ({ id }: { id: number }) => deleteTodo({ owner, id }),  
  ...(기타 선택적인 옵션들)  
})
```

1.5 useMutation

❖useMutation의 옵션

■ 필수 옵션

- mutationFn : mutation을 수행하는 함수를 지정. 반드시 Promise를 리턴해야 함

■ 선택적 옵션

- gcTime : inactive mutation의 캐시 데이터가 남아있는 시간. 이 시간 후에 GC가 수행됨
- onMutate
 - mutationFn이 실행되기 전에 이 함수가 실행됨
 - mutationFn과 동일한 인자가 전달됨.
 - 이 함수에서 리턴된 값은 작업 실패시 onError, onSettled 함수에 전달되어 낙관적 업데이트를 롤백하는데 사용될 수 있음
- onSuccess
 - mutation 이 성공하면 실행됨. mutation 수행 결과가 인자로 전달됨
- onError
 - mutation 이 실패하면 실행됨. 오류 정보가 인자로 전달됨
- onSettled
 - Success이든 Error이든 항상 실행됨. mutation 수행결과 또는 오류정보가 인자로 전달됨

1.5 useMutation

❖useMutation 리턴값

▪ mutationObject

- mutate : mutation을 트리거하는 함수. 이 함수를 호출하면 mutationFn이 실행됨
 - mutationFn과 동일한 인자를 이용
- mutateAsync : mutate 와 동일한 기능을 수행하는 함수. 대신 Promise를 리턴함
- status : mutation 의 상태
 - idle : mutationFn이 실행되기 전인 상태,
 - pending : mutation이 현재 실행중인 상태
 - error : mutation 실행 실패
 - success : mutation 실행 성공
- isIdle, isPending, isError, isError() : 각각의 상태를 확인하기 위한 함수
- data : mutation 수행 결과 응답 데이터
- reset : mutation 내부 상태를 초기화하는 함수
- failureCount : mutation 실패 횟수. reset()을 호출하면 0으로 초기화됨
- failureReason : mutation 실패 원인

- 참조: <https://tanstack.com/query/latest/docs/framework/react/reference/useMutation#usemutation>

1.5 useMutation

❖ useMutation을 이용해 추가, 삭제 기능 구현

- 이전 예제에 이어서 코드 변경

❖ AddTodo 컴포넌트 변경 : src/components/AddTodo.tsx

```
import { useMutation } from "@tanstack/react-query";
import { useState } from "react";
import { addTodo } from "../apis/ToDoAPI";
import { useNavigate } from "react-router-dom";
import { ReactCsspin } from "react-csspin";

type PropsType = { owner: string };

const AddTodo = ({ owner }: PropsType) => {
  const [todo, setTodo] = useState<string>("");
  const [desc, setDesc] = useState<string>("");
  const navigate = useNavigate();

  const addTodoMutation = useMutation({
    mutationFn: ({ todo, desc }: { todo: string; desc: string }) => addTodo({ owner, todo, desc }),
    onSuccess: () => {
      navigate("/");
    },
  });
```

1.5 useMutation

❖ AddTodo 컴포넌트 변경 (이어서)

```
const addTodoHandler = () => {
  addTodoMutation.mutate({ todo, desc });
  setTodo("");
  setDesc("");
};

return (
  <div>
    <h2>할일 추가</h2>
    <hr />
    <div>
      Todo : <input type="text" value={todo} onChange={(e) => setTodo(e.target.value)} /> <br />
      Desc : <input type="text" value={desc} onChange={(e) => setDesc(e.target.value)} /> <br />
      <button onClick={addTodoHandler}>추가</button>
    </div>
    {addTodoMutation.isPending ? <ReactCssspin opacity={0.8} message="추가중입니다" /> : ""}
  </div>
);
};

export default AddTodo;
```


1.5 useMutation

❖ AddTodo 기능 작동 여부 확인

[Home](#) | [AddTodo](#)

할일 소유자 :

할일 추가

Todo :

Desc :

[Home](#) | [AddTodo](#)

할일 소유자 :

할일 추가

Todo :

Desc :


- ES6 공부 - ES6공부를 해야 합니다 (완료)
- Vue 학습 - Vue 학습을 해야 합니다
- 놀기 - 노는 것도 중요합니다. (완료)
- 야구장 - 프로야구 경기도 봐야합니다.
- 111 - 1111



[Home](#) | [AddTodo](#)

할일 소유자 :


할일 추가



추가중입니다

Todo :

Desc :



1.5 useMutation

❖ 삭제 기능 추가

- TodoListItem 컴포넌트에 삭제 버튼을 추가하고 삭제 기능을 구현함
 - 삭제 버튼을 클릭하면 useMutation을 이용해 DELETE /todolist_long/:owner/:id 로 요청함
 - 삭제를 위해 owner 정보를 전달해야 함
 - TodoListItem PropsType과 Props 변경
 - TodoList에서 TodoListItem으로 전달하는 속성에 owner 추가
 - 삭제 후 데이터를 다시 fetch하기 위해 refetch 함수를 사용해야 함
 - 이 함수도 TodoListItem 컴포넌트로 속성을 이용해 전달함
- TodoList 컴포넌트 변경 : src/components/TodoList.tsx

```
.....(생략)
const TodoList = ({ owner }: PropsType) => {
  const { data, error, isFetching, refetch } = useQuery(.....(생략));
  return (
    <div>
      .....(생략)
      return <TodoListItem key={todoItem.id} todoItem={todoItem} refetch={refetch} owner={owner} />;
      .....(생략)
    </div>
  );
};
export default TodoList;
```

1.5 useMutation

- TodoListItem 컴포넌트 변경 : src/components/TodoListItem.tsx

```
import { useMutation } from "@tanstack/react-query";
import { TodoType, deleteTodo } from "../apis/TodoAPI";
import { ReactCspin } from "react-csspin";

type PropsType = { owner: string; todoItem: TodoType; refetch: () => void };
const TodoListItem = ({ owner, todoItem, refetch }: PropsType) => {
  const deleteTodoMutation = useMutation({
    mutationFn: ({ id }: { id: number }) => deleteTodo({ owner, id }),
    onSuccess: () => {
      refetch();
    },
  });
  return (
    <>
      <li>
        <button onClick={() => deleteTodoMutation.mutate({ id: todoItem.id })}>삭제</button>
        {" "}{todoItem.todo} - {todoItem.desc}{" "}{todoItem.done ? "(완료)" : ""}
      </li>
      {deleteTodoMutation.isPending ? <ReactCspin opacity={0.8} message="삭제중입니다" /> : ""}
    </>
  );
};
export default TodoListItem;
```


1.5 useMutation

❖삭제 기능 테스트

[Home](#) | [AddTodo](#)

할일 소유자 :


- ES6 공부 - ES6공부를 해야 합니다 (완료)
- Vue 학습 - Vue 학습을 해야 합니다
- 놀기 - 노는 것도 중요합니다. (완료)
- 야구장 - 프로야구 경기도 봐야합니다.
- 111 - 1111




[Home](#) | [AddTodo](#)

할일 소유자 :

- ES6 공부 - ES6공부를 해야 합니다 (완료)
- Vue 학습 - Vue 학습을 해야 합니다
- 놀기 - 노는 것도 중요합니다. (완료)
- 야구장 - 프로야구 경기도 봐야합니다.
- 111 - 1111


삭제중입니다



1.6 useQueryClient

❖useQueryClient 훅

- 현재의 QueryClient 인스턴스를 리턴함
- useQuery()를 여러번 하더라도 QueryClient 인스턴스는 하나를 사용함
 - useQuery()는 QueryClient 인스턴스를 구독하기 위한 훅임

❖사용법

```
const queryClient = useQueryClient(queryClient?: QueryClient)
```

- 옵션
 - queryClient
 - 사용자 정의 QueryClient 객체
 - 이 옵션을 지정하지 않으면 가장 가까운 컨텍스트의 queryClient 객체가 사용됨
- 리턴값
 - queryClient 객체
- 직접 객체를 생성하여 사용할 수도 있음

1.6 useQueryClient

❖QueryClient 객체의 메서드

- fetchQuery : 쿼리 실행, useQuery와 동일한 인자(queryKey, queryFn 옵션) 사용

```
const data = await queryClient.fetchQuery({ queryKey, queryFn })
```

- invalidateQueries : 지정한 queryKey에 대한 쿼리속성, 상태를 무효화함
 - 무효화 결과 fetch가 다시 일어남
 - refetch가 일어나지 않도록 하려면 refetchType : "none" 옵션을 지정함

```
await queryClient.invalidateQueries({  
  queryKey: ['posts'],  
  refetchType: 'active'  
})
```

- refetchQueries : 조건에 부합하는 쿼리를 재수행함

1.6 useQueryClient

❖몇가지 유용한 훅

- `uselsFetching` : 현재 실행중인 Fetch 작업이 몇개인지 리턴함
- `uselsMutating` : 현재 실행중인 Mutation 작업이 몇개인지 리턴함

❖EditTodo 컴포넌트 변경 : `src/components/EditTodo.tsx`

```
import { uselsFetching, useMutation, useQueryClient } from "@tanstack/react-query";
import { useNavigate, useParams } from "react-router-dom";
import { TodoType, fetchTodoOne, updateTodo } from "../apis/ToDoAPI";
import { ReactCspin } from "react-csspin";
import { useCallback, useEffect, useRef, useState } from "react";
```

```
type PropsType = { owner: string };
```

```
const EditTodo = ({ owner }: PropsType) => {
  const [data, setData] = useState<TodoType>();
  const refTodo = useRef<HTMLInputElement>(null);
  const refDesc = useRef<HTMLInputElement>(null);
  const refDone = useRef<HTMLInputElement>(null);
  const navigate = useNavigate();
  const params = useParams<{ id: string }>();
  const queryClient = useQueryClient();
  const isFetching = uselsFetching();
```

```
// fetch한 한건의 데이터를 저장할 상태  
// 비제어 컴포넌트 방법으로 업데이트하기 위해 useRef 사용
```

```
// 업데이트 후 / 경로로 이동하기 위한 navigate() 함수
```

1.6 useQueryClient

❖ EditTodo 컴포넌트 변경 (이어서)

```
// /edit/:id --> id 값 확인을 위해 useParams 이용
const id: number = parseInt(params.id ? params.id : "");

const updateTodoMutation = useMutation({
  mutationFn: ({ id, todo, desc, done }: TodoType) => updateTodo({ owner, id, todo, desc, done }),
  onSuccess: () => {
    navigate("/"); //업데이트가 성공하면 홈화면으로 이동
  },
});
// queryClient객체를 이용해 쿼리 실행.
// useEffect를 이용해 마운트되는 시점에 실행할 것이므로 순환참조 실행을 방지하기 위해 useCallback 사용
const fetchOne = useCallback(async () => {
  const data = await queryClient.fetchQuery({
    queryKey: ["fetchTodoOne", owner, id],
    queryFn: () => fetchTodoOne({ owner, id }),
  });
  setData(data);
}, [id, owner, queryClient]);

//마운트 될때 편집화면에 띄워줄 데이터 한건 읽어옴
useEffect(() => {
  fetchOne();
}, [fetchOne]);
```


1.6 useQueryClient

❖ EditTodo 컴포넌트 변경 (이어서)

```
return (  
  <div>  
    <h2>할 일 수정</h2>  
    <hr />  
    {data ? (  
      <div>  
        id : {params.id}  
        <br />  
        Todo : <input type="text" defaultValue={data.todo} ref={refTodo} /> <br />  
        Desc : <input type="text" defaultValue={data.desc} ref={refDesc} /> <br />  
        Done : <input type="checkbox" defaultChecked={data.done} ref={refDone} /> <br />  
        <br />  
        <button onClick={updateHandler}>업데이트</button>  
      </div>  
    ) : (  
      ""  
    )}  
    {isFetching > 0 ? <ReactCspin opacity={0.8} message="로딩중입니다" /> : ""}  
    {updateTodoMutation.isPending ? <ReactCspin opacity={0.8} message="업데이트중입니다" /> : ""}  
  </div>  
);  
};  
  
export default EditTodo;
```

1.6 useQueryClient

❖ TodoListItem 컴포넌트 변경 : src/components/TodoListItem.tsx

```
return (  
  <div>  
    <h2>할 일 수정</h2>  
    <hr />  
    {data ? (  
      <div>  
        id : {params.id}  
        <br />  
        Todo : <input type="text" defaultValue={data.todo} ref={refTodo} /> <br />  
        Desc : <input type="text" defaultValue={data.desc} ref={refDesc} /> <br />  
        Done : <input type="checkbox" defaultChecked={data.done} ref={refDone} /> <br />  
        <br />  
        <button onClick={updateHandler}>업데이트</button>  
      </div>  
    ) : (  
      ""  
    )}  
    {isFetching > 0 ? <ReactCspin opacity={0.8} message="로딩중입니다" /> : ""}  
    {updateTodoMutation.isPending ? <ReactCspin opacity={0.8} message="업데이트중입니다" /> : ""}  
  </div>  
);  
};  
  
export default EditTodo;
```


1.6 useQueryClient

❖ 실행 결과

[Home](#) | [AddTodo](#)

할일 소유자 :

할일 수정



로딩중입니다

[Home](#) | [AddTodo](#)

할일 소유자 :

할일 수정

id : 1716882255423

Todo :


Desc :

Done : ☒

[Home](#) | [AddTodo](#)

할일 소유자 :

할일 수정




업데이트중입니다

id : 1716882255423

Todo :

Desc :

Done : ☒



1.7 useQuery 옵션

❖ Tanstack Query의 막강한 기능

- 서버 상태의 캐싱, 동기화 기능
- 이것을 위해 useQuery의 선택적 옵션들을 잘활용할 필요가 있음

❖ 기본 설정값 한번더 확인

- staleTime : 기본값-0
 - 이 값이 0이라는 것은 백엔드 API에서 서버 상태를 가져온 직후에 바로 오래된 데이터로 간주된다는 것
 - 이 값을 10000으로 지정하면 캐시된 데이터가 10초간 fresh 상태로 유지했다가 stale 상태로 변경됨
- gcTime : 기본값-5*60*1000 밀리초, 5분
 - 서버 상태의 캐시가 inactive 데이터가 되어도 5분간은 캐시에 남아있다는 것을 의미함
- refetchInterval : 기본값-false
 - refetch를 주기적으로 실행하지 않음을 의미함
- refetchOnMount : 기본값-true
 - 컴포넌트가 stale상태이고 새롭게 마운트되면 refetch를 자동으로 수행함
- enabled : 기본값-true
 - useQuery가 설정된 후 쿼리를 자동으로 수행하도록 함.

1.7 useQuery 옵션

❖ react-query-todolist 앱에 이 옵션들을 적용해보자

- 1. staleTime을 늘리면?
 - 컴포넌트가 마운트되더라도 fresh 상태라면 refetch하지 않음
 - 불필요한 서버로의 요청을 줄일 수 있음
- 2. gcTime을 조정하면?
 - gcTime은 inactive 상태의 쿼리가 캐시에서 제거되는 시간
 - 쿼리를 사용하는 컴포넌트가 언마운트되어 Observer가 없는 상태가 되는 즉시 inactive 상태로 전환됨
 - 이 시점으로부터 gcTime이 지나면 GC가 수행되어 완전히 사라짐
- gcTime을 staleTime보다 크게 설정하는 것이 바람직함
 - staleTime 이내에 컴포넌트가 언마운트되었다가 다시 마운트되면 캐시를 이용하게 됨
 - 만일 staleTime이 gcTime보다 길다면 데이터가 stale하지 않음에도 캐시가 만료되어 보여줄 데이터가 없는 상태가 발생함
- 3. refetchOnMount를 false로 지정하고 refetchInterval을 20*1000 으로 지정하면
 - 컴포넌트가 마운트되더라도 refetch를 하지 않고 20초간격으로만 refetch를 수행함
 - refetchIntervalBackground을 true로 지정하면 브라우저에 포커스가 있지 않아도 refetch를 주기적으로 실행함
 - refetchOnMount가 'always'이면 fresh 상태인 캐시가 있어도 컴포넌트가 마운트되면 refetch를 수행함

1.7 useQuery 옵션

❖ TodoList 컴포넌트에서의 useQuery 코드를 다음과 같이 변경

```
const { data, error, isFetching, refetch } = useQuery({
  queryKey: ["fetchTodoList", owner],
  queryFn: () => fetchTodoList({ owner }),
  staleTime: 20*1000,
  gcTime: 30*1000,
  refetchOnMount: true,
  refetchInterval: 20*1000,
  refetchIntervalInBackground: true,
});
```

- 다른 종류의 브라우저 두개를 실행한 후 한쪽에서 새로운 할 일을 추가해봄
 - refetchInterval에 의해 데이터가 재조회됨
 - interval 시간에 도달하지 않았더라도 언마운트 - 마운트가 일어나면 refetch 수행
- refetchInterval을 너무 짧게 설정하지 않을 것
 - 백엔드 API 서버로의 지나친 요청으로 애플리케이션 전체 성능에 나쁜영향을 줌

1.7 useQuery 옵션

❖placeholderData 옵션

- 이값이 설정되면 쿼리가 pending 상태일 때 서버에서 fetch해오지 않았을 때 임시로 보여줄 dummy 데이터로 사용됨
- 이 옵션에 설정된 데이터는 캐시로 유지되지 않음
- 이 옵션에 keepPreviousData를 import하여 지정하면
 - 새로운 데이터를 fetch하는 동안 이전 데이터가 화면에 보여짐
- react-query-infinite 예제의 ContactsAPI.ts와 App02 컴포넌트 참조

❖initialData 옵션

- 쿼리 캐시의 초기 데이터 지정
- placeholderData와의 차이점은 캐시로 유지된다는 점

1.8 useInfiniteQuery

❖useInfiniteQuery

- 기존 useQuery에 무한 스크롤 UI를 지원할 수 있는 기능을 추가한 후
- 추가된 옵션
 - initialPageParam :
 - 필수 옵션, 첫번째 페이지 데이터를 fetch할 때 사용할 페이지 번호
 - getNextPageParam
 - 필수 옵션, 다음 페이지의 번호를 알아내기 위한 함수.
 - 이 함수가 null 또는 undefined를 리턴하면 더이상 액세스할 페이지가 없음을 의미함
 - getPreviousPageParam
 - maxPages :
 - 무한 쿼리 데이터에 지정할 최대 페이지 수
 - 최대 페이지 수에 도달한 경우에 추가로 새 페이지를 가져오면 지정된 방향에 따라 첫번째 또는 마지막 페이지가 캐시에서 삭제됨
- react-query-infinite 예제의 ContactsAPI.ts와 App1 컴포넌트 참조

2. Tanstack Query와 SWR 비교

❖SWR

- 데이터 가져오기를 위한 React Hooks
- Stale-While-Revalidate
 - "우선 캐시(stale)로부터 데이터를 반환한 후, fetch 요청(revalidate)을 하고, 최종적으로 최신화된 데이터를 가져오는 전략"
 - HTTP 캐시 무효 전략의 이름이자 라이브러리 이름
 - React Query도 이 전략을 사용함

❖공통적인 용어

- staleTime
- cacheTime : React Query에서는 gcTime

❖차이점

- SWR의 mutate 는 revalidation의 의미
 - React Query 의 invalidateQueries() 또는 refetch()와 같은 의미
- SWR의 useSWRMutation 혹은 == React Query의 useMutation

2.1 Data Fetching

❖사용 방법 비교

- swr 예제 : swr-todolist
- 데이터 가져오기

```
const fetchTodoList = (url: string) => {  
  const fetchUrl = `${BASEURL}${url}`;  
  return axios.get<TodoType[]>(fetchUrl).then((response) => response.data);  
};  
  
//useSWR  
//기본 사용법 : const { data, error, isLoading, isValidating, mutate } = useSWR(key, fetcher, options)  
//--- useSWR 예시  
const { data, error, isLoading, isValidating } = useSWR("/todolist_long/gdhong", fetchTodoList);  
  
//useQuery와 비교  
//--- useQuery 예시  
const { data, error, isLoading } = useQuery({ queryKey: ['todos'], queryFn: getTodos })
```

2.1 Data Fetching

❖useSWR 옵션

- suspense : 기본값 false
 - <React.Suspense>를 함께 사용하려면 이 옵션을 true로 지정.
- fetcher : fetcher 함수
- revalidateIfStale : 기본값 true
 - 컴포넌트가 다시 마운트되었을 때 오래된 데이터가 있는 경우 갱신(revalidate)할지를 지정
- revalidateOnMount : 마운트되었을 때 자동 갱신할지를 지정. 데이터의 stale 여부와는 상관없음
- revalidateOnFocus : 실행중인 브라우저 창에 포커싱되었을 때 자동 갱신할지 여부 지정
- focusThrottleInterval : 기본값 5000, 이 시간 범위 동안 단 한 번만 갱신

2.1 Data Fetching

❖useSWR 옵션(이어서)

- `refreshInterval` : 기본값 0
 - number로 설정하는 경우. Polling 주기를 ms 로 지정
 - function으로 설정하는 경우. 함수가 최신 데이터를 인자로 받은 후 interval 반환

```
const { data } = useSWR('/api/data', fetcher, {  
  refreshInterval: (latestData) => {  
    // 데이터에 따라 다른 간격 반환  
    if (!latestData) return 1000; // 데이터가 없으면 1초마다  
    if (latestData.isActive) return 20000; // 활성 상태면 20초마다  
    return 60000; // 비활성 상태면 60초마다  
  }  
})
```

- `refreshWhenHidden` : 기본값 false. 브라우저 창이 보이지 않을때 Polling
- `refreshWhenOffline` : 기본값 false. `navigator.onlin`이 false일 때 Polling

2.1 Data Fetching

❖useSWR 옵션(이어서)

- `shouldRetryOnError` : 기본값 `true`. `fetcher` 실행시 오류가 있으면 재시도할지 여부 지정
- `errorRetryInterval` : 기본값 `5000`. 에러 재시도 Interval
- `errorRetryCount` : 최대 에러 재시도 수
- `fallback` : 첫 렌더링 시 로딩 상태 없이 바로 데이터를 표시할 수 있도록 부여하는 초기 데이터
 - 주로 전역 설정에서 키값 별로 지정함

```
<SWRConfig value={{
  fallback: {
    '/api/user': { name: 'John', id: 1 },
    '/api/posts': [
      { id: 1, title: 'First Post' },
      { id: 2, title: 'Second Post' }
    ]
  }
}}>
  <App />
</SWRConfig>
```

- `keepPreviousData` : 기본값 `false`. 새 데이터가 로드될때까지 이전 키의 데이터를 반환
 - Paging할 때 `true`로 지정하면 효과적

2.1 Data Fetching

❖useSWR 옵션(이어서)

- onSuccess(data, key, config) : 비동기 처리 성공시 실행할 콜백함수
- onError(err, key, config) : 비동기 처리 실패시 실행할 콜백함수
- onErrorRetry(err, key, config, revalidate, revalidateOps) : 에러 재시도시 실행할 콜백함수
- onDiscarded(key) : 경합 상태로 인해 요청이 무시될 경우 실행할 콜백함수
- onLoadingSlow(key, config) : 요청을 로드하는데 너무 오래 걸리는 경우 실행할 콜백함수
 - loadingTimeout 옵션(기본값 3000) 과 연결지어 수행됨
 - 용도
 - 사용자에게 로딩이 지연되고 있음을 알려거나 추가적인 UI를 표시할 때 사용함.
 - 또는 사용자에게 최소 기능을 제공해 사용자 경험 개선

2.2 SWR 전역 설정

❖ 전역 설정 방법

- `<SWRConfig />` 컨텍스트를 이용하여 전역 설정을 수행
- 앞절에서의 옵션을 지정함

```
ReactDOM.createRoot(document.getElementById('root')).render(  
  <React.StrictMode>  
    <SWRConfig  
      value={{  
        refreshInterval: 3000,  
        revalidateOnFocus: false,  
        dedupingInterval: 2000,  
      }}  
    >  
      <App />  
    </SWRConfig>  
  </React.StrictMode>,  
)
```

2.3 Mutation

❖ useSWRMutation() 혹은 원격 데이터 변경

- swr의 mutation이 아닌 react-query에서의 mutation

//useSWRMutation() 사용

```
const addTodo = (url: string, { arg }: { arg: AddTodoType }) => {  
  const addUrl = `${BASEURL}${url}`;  
  return axios.post(addUrl, { todo: arg.todo, desc: arg.desc });  
};
```

```
const { trigger, isMutating } = useSWRMutation("/todolist_long/gdhong", addTodo);
```

.....

//updater함수의 arg 인자형식으로 인자 전달

```
trigger({ todo, desc });
```

//Tanstack Query와의 비교

```
const addTodoMutation = useMutation({  
  mutationFn: ({ todo, desc }: { todo: string; desc: string }) => addTodo({ owner, todo, desc }),  
  onSuccess: () => {  
    navigate("/");  
  },  
});
```

.....

```
addTodoMutation.mutate({ todo, desc });
```


2.3 Mutation

❖ backend의 데이터를 업데이트 후 프론트엔드 캐시 데이터 변경

- useSWR() 혹은 호출 후 리턴된 mutate 함수를 이용해 명시적으로 캐시 갱신

```
const { data: todos = [], mutate } = useSWR('/api/todos', fetcher);

const { trigger: createTrigger, isMutating: isCreating } = useSWRMutation('/api/todos', createTodo);

const handleAddTodo = async (todo:string) => {
  try {
    //명시적으로 원격 데이터 수정후 즉시 캐시 갱신
    const newTodo = await createTrigger({ todo, completed: false })
    mutate([...todos, newTodo], false)
  } catch (error) {
    console.error('Todo 추가 실패:', error)
  }
}
```

2.4 Pagination

❖페이징 처리는 useSWR() 만으로 가능

- keepPreviousData 옵션을 true로 지정하여 페이지 이동시 이전 페이지 데이터가 유지되도록 함
- sw-contacts-8 예제의 App1.tsx 참조

❖App1.tsx

```
import { useState } from "react";
import ContactList1 from "../ContactList1";
import useSWR from "swr";
import { fetchContactList } from "../apis/ContactAPI";
import { ReactCspin } from "react-csspin";

const App1 = () => {
  const [pageNo, setPageNo] = useState(1);
  const { data, error, isLoading } = useSWR(`/contacts_long?pageno=${pageNo}`, fetchContactList);
```

2.4 Pagination

❖App1.tsx(이어서)

```
const prev = () => {
  if (data && data.pageno > 1) setPageNo(pageNo - 1);
};
const next = () => {
  if (data && data.pageno < Math.floor((data.totalcount - 1) / data.pagesize) + 1) setPageNo(pageNo + 1);
};

return (
  <div>
    <ContactList1 data={data} />
    <button onClick={prev}>이전</button>
    {data ? <span>{" "}{data.pageno} / {Math.floor((data.totalcount - 1) / data.pagesize) + 1}{" "}</span>
      : <span>0 / 0 </span> }
    <button onClick={next}>다음</button>
    {error ? <div>로딩 에러 : {error.message}</div> : ""}
    {isLoading ? <ReactCssspin opacity={0.8} message="로딩 중" /> : ""}
  </div>
);
};

export default App1;
```

2.5 무한 스크롤 지원

❖useSWRInfinite()

- 무한 스크롤을 지원하는 훅
- 사용법

```
const { data, error, size, setSize, isValidating, mutate } =  
  useSWRInfinite(  
    getKey,    // 키 생성 함수  
    fetcher,   // 데이터 페칭 함수  
    options    // 옵션 (선택사항)  
  )
```

❖예제 참조

- swr-contacts-8 예제의 App2.tsx

2.5 무한 스크롤 지원

❖App2.tsx

```
import ContactList2 from "../ContactList2";
import { fetchContactList, type ContactItemType, type ContactListType } from "../apis/ContactAPI";
import useSWRInfinite from "swr/infinite";

const App2 = () => {
  const getKey = (pageIndex: number, previousPageData: ContactListType | null): string | null => {
    // 첫 번째 페이지 (pageIndex가 0일 때 pageno=1로 요청)
    if (pageIndex === 0) return `/contacts_long?pageno=1`;
    // 이전 페이지 데이터가 null이면 더 이상 로드하지 않음
    if (!previousPageData) return null;
    // 이전 페이지에 연락처가 없으면 더 이상 로드하지 않음
    if (previousPageData.contacts.length === 0) return null;
    // 현재까지 로드된 총 연락처 수 계산
    const totalLoaded = pageIndex * previousPageData.pagesize + previousPageData.contacts.length;
    // 모든 데이터를 로드했는지 확인
    if (totalLoaded >= previousPageData.totalcount) return null;
    // 다음 페이지 번호 계산 (pageIndex + 2, 왜냐하면 pageIndex는 0부터 시작하고 pageno는 1부터 시작)
    return `/contacts_long?pageno=${pageIndex + 2}`;
  };
};
```

2.5 무한 스크롤 지원

❖App2.tsx(이어서)

```
const { data, error, size, setSize, isLoading } = useSWRInfinite(getKey, fetchContactList, {
  revalidateFirstPage: false, // 첫 페이지 재검증 방지
  revalidateAll: false, // 모든 페이지 재검증 방지
});

// 모든 페이지의 연락처를 하나의 배열로 합치기
const contacts: ContactItemType[] = data ? data.flatMap((pageData) => pageData.contacts) : [];

// 로딩 상태 계산
const isLoadingMore = isLoading || (size > 0 && data && typeof data[size - 1] === "undefined");
const isEmpty = data?.[0]?.contacts?.length === 0;
const isReachingEnd = isEmpty || (data && data.length > 0 && contacts.length >= data[0].totalcount);

const loadMore = () => {
  if (!isLoadingMore && !isReachingEnd) {
    setSize(size + 1);
  }
};
```

2.5 무한 스크롤 지원

❖App2.tsx(이어서)

```
return (  
  <div>  
    <ContactList2 contacts={contacts} />  
    { /* 상태 정보 표시 */ }  
    {data && data[0] && <div>총 {data[0].totalcount}개 중 {contacts.length}개 로드됨</div>}  
    { /* 더 보기 버튼 */ }  
    { !isReachingEnd && (  
      <div>  
        <button onClick={loadMore} disabled={isLoadingMore}>  
          {isLoadingMore ? "로딩 중..." : "다음 읽어오기"}  
        </button>  
      </div>  
    ) }  
    { /* 끝에 도달했을 때 메시지 */ }  
    { isReachingEnd && !isEmpty && <div style={{ textAlign: "center", margin: "20px", color: "#666" }}>전체 조회 완료</div> }  
    { error ? <div>로딩 에러 : {error.message}</div> : "" }  
    { isLoading ? <ReactCspin opacity={0.8} message="로딩중" /> : "" }  
  </div>  
};  
export default App2;
```

3. Tanstack Query와 SWR 비교

❖ Devtool 지원여부

- Tanstack Query : 지원
- SWR : 미지원

❖ 핵심 철학

- SWR 전략에 기반해 간단하고 직관적인 API 제공
- SWR에 비해 좀 더 복잡하고 다양한 캐싱 전략 구현

❖ 번들 사이즈 크기

- Tanstack Query > SWR

❖ 선택 기준

- 번들 크기 최소화 : SWR
- 강력한 캐싱 관리 기능 필요 : Tanstack Query
- SWR은 devtools이 없으므로 전역 설정에서 onError, onSuccess와 같은 콜백함수를 이용해 콘솔에 메시지를 출력하도록 설정할 것