main.tsx

```
import React from "react";
import ReactDOM from "react-dom/client";
import App from "./App";
import "./index.css";
import { QueryClient, QueryClientProvider } from "@tanstack/react-query";
import { ReactQueryDevtools } from "@tanstack/react-query-devtools";
import "bootstrap/dist/css/bootstrap.css";

const queryClient = new QueryClient(
/*{
 possible customization of React Query:
  defaultOptions: {
   queries: {
     retry: 3,
     cacheTime: 300_000,
     staleTime: 10 * 1000,
     refetchOnWindowFocus: false,
     refetchOnReconnect: false,
     refetchOnMount: false,
   },
 },
}*/);

ReactDOM.createRoot(document.getElementById("root") as HTMLElement).render(
 <React.StrictMode>
   <QueryClientProvider client={queryClient}>
     <App />
     <ReactQueryDevtools />
   </QueryClientProvider>
 </React.StrictMode>
);

/* QueryClient = core object for managing and caching remote data
 - create QueryClient object
 - wrap app component in QueryClientProvider using that object
 - stale data is automatically refreshed when:
     * the network is reconnected
     * a component is mounted
     * the window is refocused
  React Query Dev Tools will not be included in production */
```

App.tsx

```
import "./App.css";
import PostList from "./react-query/PostList";
import PostListLoad from "./react-query/PostListLoad";
import PostListPagination from "./react-query/PostListPagination";
import TodoForm from "./react-query/TodoForm";
import TodoFormWithHook from "./react-query/TodoFormWithHook";
import TodoList from "./react-query/TodoList";

const App = () => {
 return (
   <>
     <hr className="red-hr" />
     <h2>POST LIST with USER FILTER</h2>
     <PostList />
     <hr className="red-hr" />
     <h2>POST LIST with PAGINATION</h2>
     <PostListPagination />
     <hr className="red-hr" />
     <h2>POST LIST - Load More</h2>
     <PostListLoad />
     <hr className="red-hr" />
     <h2>TODO LIST</h2>
     <TodoForm />
     <TodoList />
     <hr className="red-hr" />
     <h2>TODO LIST w/ FORM HOOK</h2>
     <TodoFormWithHook />
     <TodoList />
   </>
 );
};

export default App;
```

App.css

```
html, body {
 max-width: 1280px;
 padding: 2rem;
 font-family: monospace !important;
 font-size: 13px;
```

```
}

h2 {
 font-weight: bold;
}

button {
 font-weight: bold !important;
}

select {
 font-weight: bold !important;
}

.red-hr {
 border-color: blue;
 border-width: 2px;
 opacity: 0.6;
}
```

apiClient.ts

```
import axios from 'axios'

const axiosInstance = axios.create({
   baseURL: 'https://jsonplaceholder.typicode.com'
})


class APIClient<T> {
   endpoint: string;

   constructor(endpoint: string) {
       this.endpoint = endpoint;
}
   getAll = () => {
       return axiosInstance.get<T[]>(this.endpoint).then(response => response.data)
   }

   post = (data: T) => {
       axiosInstance
       .post<T>(this.endpoint, data)
       .then(response => response.data)
   }
```

```
}

export default APIClient;
```

todoService.ts

```
import APIClient from "./apiClient"

export interface Todo {
 id: number;
 title: string;
 userId: number;
 completed: boolean;
}
const apiClient = new APIClient<Todo>('/todos')

export default new APIClient<Todo>('/todo')
```

constants.ts

```
export const CACHE_KEY_TODOS = ['todos'];
```

useTodos.ts

```
import { useQuery } from '@tanstack/react-query';
import { CACHE_KEY_TODOS } from '../constants';
import APIClient from '../services/apiClient';

const apiClient = new APIClient<Todo>('/todos')

export interface Todo {
 id: number;
 title: string;
 userId: number;
 completed: boolean;
}

const useTodos = () => {
     return (
       useQuery<Todo[], Error>({
   queryKey: CACHE_KEY_TODOS,
   queryFn: apiClient.getAll,
```

```
   staleTime: 10 * 1000,
}));
}

export default useTodos

 /*  useQuery is the React Query hook used for fetching data
       - it takes a configuration object with 2 props
           1) queryKey: unique id for query used internally for caching
               - any data retrieved from the backend, it is stored in the cache
                   and is accessible using this key
               - the key's first value is usually a string that identifies the
                   type of data to be stored
               - it can also take additional values specifying use of the data
           2) queryFn: the function used to fetch data from the backend
               - it returns a promise that resolves the data or throws an error
       - This can be used with axios as above or with any other HTTP library for
           fetching data
   - At runtime, getTodos will be called. When the promise is resolved,
       it will return an array of Todos that will be stored in the cache
       against the query key
   - This configuration allows for multiple retries with react query if the
       call to the server fails at first.
   - It also allows for auto refresh and caching, which improves performance
   - React Query also makes state variables for errors, data, loading, etc
       unnecessary, as the useQuery object contains all these */
```

useAddTodos.ts

```
import { useMutation, useQueryClient } from "@tanstack/react-query";
import todoService from "../services/todoService";
import APIClient from "../services/apiClient";
import { CACHE_KEY_TODOS } from "../constants";
import { Todo } from "../services/todoService";

const apiClient = new APIClient<Todo>("/todos");

interface AddTodoContext {
 previousTodos: Todo[];
}

const useAddTodo = (onAdd: () => void) => {
   const queryClient = useQueryClient();
 return useMutation<Todo, Error, Todo, AddTodoContext>({
```

```
    mutationFn: apiClient.post,
    onMutate: (newTodo) => {
      const previousTodos = queryClient.getQueryData<Todo[]>(CACHE_KEY_TODOS) || [];
      queryClient.setQueryData<Todo[]>(CACHE_KEY_TODOS, (todos = []) => [
        newTodo,
        ...todos,
      ]);

      onAdd();

      return { previousTodos };
    },

    onSuccess: (savedTodo, newTodo) => {
      queryClient.setQueryData<Todo[]>(CACHE_KEY_TODOS, (todos) =>
        todos?.map((todo) => (todo === newTodo ? savedTodo : todo))
      );
    },
    onError: (error, newTodo, context) => {
      if (!context) return;
      queryClient.setQueryData<Todo[]>(CACHE_KEY_TODOS, context.previousTodos);
    },
  });

}

export default useAddTodo;
```

TodoList.tsx

```
import useTodos from "./hooks/useTodos";

const TodoList = () => {
  const { data: todos, error, isLoading } = useTodos();

  if (isLoading) return <p>Loading...</p>;
  if (error) return <p>{error.message}</p>;

  return (
    <ul className="list-group">
      {/* - making todos optional in the case that the call to the backend fails
    and todos is undefined */}
      {todos?.map((todo) => (
        <li key={todo.id} className="list-group-item">
```

```
        {todo.title}
      </li>
    ))}
  </ul>
 );
};

export default TodoList;
```

TodoForm.tsx

```
import { useRef } from "react";

import { useMutation, useQueryClient } from "@tanstack/react-query";
import axios from "axios";
import { Todo } from "./services/todoService";

/* The useMutation and useQueryClient hooks are used for managing mutations and
accessing the query client respectively.
- The useQueryClient hook is called to obtain the query client instance. It is
used to manage and interact with the cache.
- The mutationFn property specifies the actual function that will be called when
the mutation is triggered. In this case, it sends a POST request to
"https://jsonplaceholder.typicode.com/todos" with the todo object as the
request payload. The response is then extracted and returned.*/

const TodoForm = () => {
 const queryClient = useQueryClient();
 const addTodo = useMutation<Todo, Error, Todo>({
   mutationFn: (todo: Todo) =>
     axios
       /* The Todo type represents the structure of a todo item.*/
       .post<Todo>("https://jsonplaceholder.typicode.com/todos", todo)
       .then((response) => response.data),

   onSuccess: (savedTodo, newTodo) => {
     /* Approach 1: invalidating the cache (does not work with JSONplaceholder)
        queryClient.invalidateQueries({
          queryKey: ['todos']
        })
     Approach 2: Updating the data in the cache directly using setQueryData and
                    passing the queryKey and an updating function */

     queryClient.setQueryData<Todo[]>(["todos"], (todos) => [
```

```
      savedTodo,
      ...(todos || []),
    ]);

    // clearing the form field if the value is already being added
    if (ref.current) ref.current.value = "";

    /* The onSuccess property defines a callback function that will be executed
      if the mutation succeeds. It receives the savedTodo (the response data from
      the server) and newTodo (the todo object passed to addTodo.mutate(), which
      is not used in this case).
      - Inside the onSuccess callback, the cache is updated by using the setQueryData
      function of the query client. It sets the new data for the "todos" query key
      in the cache. The new data is an array that includes the savedTodo as the first
      item, followed by the existing todos (if any). */
  },
});

/* The useRef hook is used to create a reference to the input element in the form.
   It is initialized with a value of null. */
const ref = useRef<HTMLInputElement>(null);

return (
  <>
    {addTodo.error && (
      <div className="alert alert-danger">{addTodo.error.message}</div>
    )}
    <form
      className="row mb-3"
      onSubmit={(event) => {
        event.preventDefault();
        // insuring a user inputs a value before sending a post request
        if (ref.current && ref.current.value)
          addTodo.mutate({
            id: 0,
            title: ref.current?.value,
            completed: false,
            userId: 1,
          });
      }}
    >
      <div className="col">
        <input ref={ref} type="text" className="form-control" />
      </div>
      <div className="col">
        <button disabled={addTodo.isLoading} className="btn btn-primary">
```

```
          {addTodo.isLoading ? "Adding..." : "Add"}
        </button>
      </div>
    </form>
  </>
 );
};

export default TodoForm;
```

TodoFormOptimistic.tsx

```
import { useRef } from "react";

import { useMutation, useQueryClient } from "@tanstack/react-query";
import axios from "axios";
import { Todo } from "./services/todoService";

interface AddTodoContext {
 previousTodos: Todo[];
}

const TodoForm = () => {
 const queryClient = useQueryClient();
 const addTodo = useMutation<Todo, Error, Todo, AddTodoContext>({
   mutationFn: (todo: Todo) =>
     axios
       /* The Todo type represents the structure of a todo item.*/
       .post<Todo>("https://jsonplaceholder.typicode.com/todos", todo)
       .then((response) => response.data),

   // Implementing optimistic update, step one = onMutate:
   onMutate: (newTodo) => {
     // creating a context object that holds all todos before mutating
     // if it is undefined, we will return an empty array
     const previousTodos = queryClient.getQueryData<Todo[]>(["todos"]) || [];
     queryClient.setQueryData<Todo[]>(["todos"], (todos) => [
       newTodo,
       ...(todos || []),
     ]);

     if (ref.current) ref.current.value = "";

     return { previousTodos };
```

```
  },

  /* Handling success scenario:
  - if the request is successful, we should replace the new todo with the one we
    get from the backend, because the new todo does not have an id. We have set
    it to 0.
  - we will replace it with a saved todo from the backend instead
  - we will get a todo object from the backend if the input todo is the same as
    the saved todo. We will then replace the new todo with the saved todo.
  - otherwise, we return the same todo object
  */

  onSuccess: (savedTodo, newTodo) => {
    queryClient.setQueryData<Todo[]>(["todos"], (todos) =>
      todos?.map((todo) => (todo === newTodo ? savedTodo : todo))
    );
  },
  /* Handling an error scenario:
  - if the request fails, we should roll back and restore the UI to previous state
  - the third parameter below is context. This object is used to pass data between
  our callbacks
  - Here we need a context object that includes the previous todos before we
  updated the cache, which we created above, previousTodos.
  */

  onError: (error, newTodo, context) => {
    if (!context) return;
    queryClient.setQueryData<Todo[]>(["todos"], context.previousTodos);
  },
});

const ref = useRef<HTMLInputElement>(null);

return (
  <>
    {addTodo.error && (
      <div className="alert alert-danger">{addTodo.error.message}</div>
    )}
    <form
      className="row mb-3"
      onSubmit={(event) => {
        event.preventDefault();
        // insuring a user inputs a value before sending a post request
        if (ref.current && ref.current.value)
          addTodo.mutate({
            id: 0,
```

```
            title: ref.current?.value,
            completed: false,
            userId: 1,
          });
      }}
    >
      <div className="col">
        <input ref={ref} type="text" className="form-control" />
      </div>
      <div className="col">
        <button className="btn btn-primary">Add</button>
      </div>
    </form>
  </>
 );
};

export default TodoForm;
```

TodoFormWithHook.tsx

```
import { useRef } from "react";
import useAddTodo from "./hooks/useAddTodo";

const TodoFormWithHook = () => {
 const ref = useRef<HTMLInputElement>(null);
 const addTodo = useAddTodo(() => {
   if (ref.current) ref.current.value = "";
 });

 return (
   <>
     {addTodo.error && (
       <div className="alert alert-danger">{addTodo.error.message}</div>
     )}
     <form
       className="row mb-3"
       onSubmit={(event) => {
         event.preventDefault();
         // insuring a user inputs a value before sending a post request
         if (ref.current && ref.current.value)
           addTodo.mutate({
             id: 0,
             title: ref.current?.value,
```

```
            completed: false,
            userId: 1,
          });
      }}
    >
      <div className="col">
        <input ref={ref} type="text" className="form-control" />
      </div>
      <div className="col">
        <button disabled={addTodo.isLoading} className="btn btn-primary">
          {addTodo.isLoading ? "Adding..." : "Add"}
        </button>
      </div>
    </form>
  </>
 );
};

export default TodoFormWithHook;
```

usePosts.ts

```
import { useQuery } from '@tanstack/react-query';
import axios from 'axios';

interface Post {
 id: number;
 title: string;
 body: string;
 userId: number;
}

const usePosts = (userId: number | undefined) => useQuery<Post[], Error>({
   /* Heirarchical structure that represents the relationship of objects
      left to right data gets more specific
      If a userId is selected, the array contains all data, otherwise,
      it contains all the posts only */
   queryKey: userId ? ["users", userId, "posts"] : ["posts"],
   queryFn: () =>
   axios
   .get("https://jsonplaceholder.typicode.com/posts", {
       params: {
           userId
       }
```

```
        })
      .then((response) => response.data),
    staleTime: 10 * 1000,
});

export default usePosts
```

usePostsLoad.ts

```
import { useInfiniteQuery } from '@tanstack/react-query';
import axios from 'axios';

interface Post {
 id: number;
 title: string;
 body: string;
 userId: number;
}

interface PostQuery {
   pageSize: number,
}

const usePostsLoad = (query: PostQuery
 ) => useInfiniteQuery<Post[], Error>({
   queryKey: ["posts", query],
   // initialized to 1 to get the data for first page initially
   queryFn: ({ pageParam = 1 }) =>
   axios
   .get("https://jsonplaceholder.typicode.com/posts", {
       params: {
           _start: (pageParam- 1) * query.pageSize,
           _limit: query.pageSize
       }
       })
      .then((response) => response.data),
   staleTime: 10 * 1000,
   // keeps previous page's data until next page is loaded to avoid
   // jumping visuals on the page
   keepPreviousData: true,
   // allPages contains the data for all of the pages
   // this is called when the Load More button is clicked
   getNextPageParam: (lastPage, allPages) => {
       // return the next page number if we have not reached the end
```

```
        // the API used here requires this method, because it does not tell
        // the total number of pages ahead of time so we can calculate
        return lastPage.length > 0 ? allPages.length + 1 : undefined;
    }
});

export default usePostsLoad
```

usePostsPagination.ts

```
import { useQuery } from '@tanstack/react-query';
import axios from 'axios';

interface Post {
 id: number;
 title: string;
 body: string;
 userId: number;
}

interface PostQuery {
   page: number,
   pageSize: number,
}

/*  The usePostsPagination function takes a query parameter of type PostQuery.
   - Inside the function, the useQuery hook is used to define a data fetching query.
   - The queryKey property specifies the unique key for the query. In this case,
     it is an array containing the string "posts" and the query object.
   - The queryFn property specifies the function that will be called to fetch the
     data. It sends a GET request to "https://jsonplaceholder.typicode.com/posts"
     with query parameters _start and _limit based on the provided query.page and
     query.pageSize values. It then extracts and returns the response data.
   - The staleTime property sets the time (in milliseconds) for how long the data
     is considered fresh without re-fetching.
   - The keepPreviousData property is set to true to keep the previous page's data
     until the next page is loaded, preventing jumping visuals on the page. */


const usePostsPagination = (query: PostQuery
 ) => useQuery<Post[], Error>({
   queryKey: ["posts", query],
   queryFn: () =>
   axios
```

```
    .get("https://jsonplaceholder.typicode.com/posts", {
        params: {
            _start: (query.page - 1) * query.pageSize,
            _limit: query.pageSize
        }
        })
      .then((response) => response.data),
    staleTime: 10 * 1000,
    // keeps previous page's data until next page is loaded to avoid
    // jumping visuals on the page
    keepPreviousData: true,
});

export default usePostsPagination
```

PostList.tsx

```
import { useState } from "react";
import usePosts from "./hooks/usePosts";

/* The usePosts is responsible for fetching the posts data based on
  the selected user. */

const PostList = () => {
 /* userId is a state variable created using the useState hook.
    It is initialized without a value (undefined) because no user is
    selected initially.
    - setUserId is a function that can be used to update the value of userId. */

 const [userId, setUserId] = useState<number>();

 /* The usePosts hook is called with the userId value as a parameter.
      It returns an object with the data, error, and isLoading properties.
      - data represents the fetched posts data.
      - error holds any error that occurred during the data fetching process.
      - isLoading indicates whether the data is currently being fetched. */

 const { data, error, isLoading } = usePosts(userId);

 /* If isLoading is true, a loading message is displayed.
    If error exists, an error message is displayed using the error.message
    property. */

 if (isLoading) return <p>Loading...</p>;
```

```
 if (error) return <p>{error.message}</p>;

 /* If no loading or error state is present, the user selection dropdown
    and post list are rendered.
    - The select element represents a dropdown menu for selecting a user.
      The onChange event handler is triggered whenever the selected value changes.
      It updates the userId state with the parsed integer value of the selected option.
    - The selected value is set to userId, so the dropdown maintains the selected user.
    - The data array is mapped over using the map method. Each post is rendered as
      an <li> element with its title displayed. */

 return (
   <>
     <select
       onChange={(event) => setUserId(parseInt(event.target.value))}
       value={userId}
       className="form-select mb-3"
     >
       <option value=""></option>
       <option value="1">User 1</option>
       <option value="2">User 2</option>
       <option value="3">User 3</option>
     </select>
     <ul className="list-group">
       {data?.map((post) => (
         <li key={post.id} className="list-group-item">
           {post.title}
         </li>
       ))}
     </ul>
   </>
 );
};

export default PostList;
```

PostListLoad.tsx

```
import React from "react";
import usePostsLoad from "./hooks/usePostsLoad";

const PostListLoad = () => {
 const pageSize = 10;
 const { data, error, isLoading, fetchNextPage, isFetchingNextPage } =
```

```
  usePostsLoad({
    pageSize,
  });

 if (isLoading) return <p>Loading...</p>;
 if (error) return <p>{error.message}</p>;

 return (
   <>
     <ul className="list-group">
       {/* go to data.pages, and map each page to a React fragment */}
       {data.pages.map((page, index) => (
         <React.Fragment key={index}>
           {/* inside the fragment, map the contents of each page, an array
           of posts, to list items */}
           {page.map((post) => (
             <li key={post.id} className="list-group-item">
               {post.title}
             </li>
           ))}
         </React.Fragment>
       ))}
     </ul>
     {/* Previous and Next buttons */}
     <button
       className="btn btn-primary my-3 ms-1"
       disabled={isFetchingNextPage}
       onClick={() => fetchNextPage()}
     >
       {isFetchingNextPage ? "Loading..." : "Load More"}
     </button>
   </>
 );
};

export default PostListLoad;
```

PostListPagination.tsx

```
import { useState } from "react";
import usePostsPagination from "./hooks/usePostsPagination";

const PostListPagination = () => {
 /* Initializing the page size and page state variables:
```

```
  pageSize is set to 10, representing the number of posts to be displayed per page.
  page is a state variable created using the useState hook. It is initialized with
  a value of 1, indicating the current page.
  */
const pageSize = 10;
const [page, setPage] = useState(1);

/* The usePostsPagination hook is is responsible for fetching the paginated
   posts data and is called with an object containing the page and
   pageSize values as parameters. It returns an object with the data, error, and
   isLoading properties.
  - data represents the fetched posts data.
  - error holds any error that occurred during the data fetching process.
  - isLoading indicates whether the data is currently being fetched. */

const { data, error, isLoading } = usePostsPagination({ page, pageSize });

/* If isLoading is true, a loading message is displayed.
   If error exists, an error message is displayed using the error.message
   property. */

if (isLoading) return <p>Loading...</p>;
if (error) return <p>{error.message}</p>;

/* If no loading or error state is present, the paginated post list is rendered.
  - The posts data is mapped over using the map method. Each post is rendered
    as an <li> element with its title displayed.
  - Two buttons are included for pagination: "Previous" and "Next". The "Previous"
    button is disabled when the current page is the first page (page === 1).
  - Clicking the "Previous" button triggers the setPage function with the current
    page decremented by 1.
  - Clicking the "Next" button triggers the setPage function with the current page
    incremented by 1. */

return (
  <>
    <ul className="list-group">
      {data?.map((post) => (
        <li key={post.id} className="list-group-item">
          {post.title}
        </li>
      ))}
    </ul>
    {/* Previous and Next buttons */}
    <button
      disabled={page === 1}
```

```
        className="btn btn-primary my-3"
        onClick={() => setPage(page - 1)}
      >
        Previous
      </button>
      <button
        className="btn btn-primary my-3 ms-1"
        onClick={() => setPage(page + 1)}
      >
        Next
      </button>
    </>
  );
};

export default PostListPagination;
```