React Query API Structure

how to make good structure of api with react query react ts

To structure a robust API layer in a React TypeScript application using React Query, consider the following steps for clarity, scalability, and maintainability:

1. Set Up API Client

- Use an HTTP client like `axios` to handle requests and centralize error handling, headers, and base URL.
- Create a reusable `apiClient.ts` file for configuring axios.

typescript

```
// src/api/apiClient.ts
import axios from 'axios';

const apiClient = axios.create({
  baseURL: 'https://your-api.com/api',
  headers: { 'Content-Type': 'application/json' },
});

export default apiClient;
```

2. Define API Service Functions

- Organize API calls by domain (e.g., `userService`, `productService`) in separate files within an `api` folder.
- Each service function should handle specific API routes and return data to the query functions.

typescript

```
// src/api/userService.ts
import apiClient from './apiClient';

export const fetchUser = async (userId: string) => {
  const response = await apiClient.get(`/users/${userId}`);
  return response.data;
};

export const updateUser = async (userId: string, data: any) => {
  const response = await apiClient.put(`/users/${userId}`, data);
  return response.data;
};
```

3. Create Custom React Query Hooks

- Define custom hooks for each API operation, which helps in organizing and centralizing query logic.
- Use React Query's `useQuery` for fetching and `useMutation` for updates.

```typescript
// src/hooks/useUserQuery.ts
import { useQuery, useMutation, useQueryClient } from '@tanstack/react-query';
import { fetchUser, updateUser } from '../api/userService';

export const useUserQuery = (userId: string) => {
  return useQuery(['user', userId], () => fetchUser(userId));
};

export const useUpdateUser = () => {
  const queryClient = useQueryClient();
  return useMutation(
    (data: { userId: string; updateData: any }) => updateUser(data.userId,
data.updateData),
    {
      onSuccess: () => {
        queryClient.invalidateQueries(['user']);
      },
    }
  );
};
```

4. Define Types for Data

- Define TypeScript interfaces or types to ensure type safety and better code readability.

typescript

```typescript
// src/types/userTypes.ts
export interface User {
  id: string;
  name: string;
  email: string;
}
```

5. Centralize Queries with Query Keys

- Use query keys to uniquely identify and scope your queries for better cache control and easy refetching.

typescript

```typescript
const userQueryKey = (userId: string) => ['user', userId];
```

6. Implement Global Error Handling

- You can add global error handling in your React Query setup, especially useful for redirecting on errors like 401 (unauthorized) or displaying a toast notification.

typescript

```typescript
import { QueryClient } from '@tanstack/react-query';

export const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      onError: (error) => {
        // Handle error globally, log out if needed, etc.
      },
    },
  },
});
```

7. Wrap Your App with QueryClientProvider

- Initialize React Query by wrapping your app with `QueryClientProvider` and passing in `queryClient`.

typescript

```
import { QueryClientProvider } from '@tanstack/react-query';
import { queryClient } from './api/queryClient';

const App = () => (
  <QueryClientProvider client={queryClient}>
    {/* rest of your app */}
  </QueryClientProvider>
);
```

This structure keeps API calls modular, separates concerns, and leverages React Query's caching and mutation handling, making it easier to extend and maintain the application.

how to make pagination with react ts and server with node example

Implementing pagination in a React TypeScript frontend with a Node.js backend requires coordination between both layers to handle data fetching, page size, and navigation. Here's an example setup:

Backend (Node.js + Express)

In your Node.js backend, set up a paginated endpoint that responds with a subset of data, along with the total number of items and/or pages. Here's a basic example:

1. Define API Route for Pagination

javascript

```
// src/routes/items.ts
import express from 'express';
import { getPaginatedItems } from '../controllers/itemsController';
const router = express.Router();

router.get('/items', getPaginatedItems);

export default router;
```

2. Create Controller Logic

- Accept `page` and `limit` as query parameters to determine the data subset to return.

- Calculate `skip` (items to skip) and `limit` to fetch the correct set of items.

```javascript
// src/controllers/itemsController.ts
import { Request, Response } from 'express';

export const getPaginatedItems = async (req: Request, res: Response) => {
  const page = parseInt(req.query.page as string) || 1;
  const limit = parseInt(req.query.limit as string) || 10;

  const skip = (page - 1) * limit;

  // Sample data, replace with actual DB logic
  const items = [
    { id: 1, name: 'Item 1' },
    { id: 2, name: 'Item 2' },
    // Assume more items here...
  ];

  const paginatedItems = items.slice(skip, skip + limit);
  const totalItems = items.length;

  res.json({
    data: paginatedItems,
    page,
    totalItems,
    totalPages: Math.ceil(totalItems / limit),
  });
};
```

Frontend (React + TypeScript + React Query)

On the React side, use React Query to fetch paginated data and manage the state of the current page.

1. Create a Custom Hook for Fetching Paginated Data

- Use React Query's `useQuery` to fetch paginated data based on the current page.
- Create a fetch function that takes `page` and `limit` as parameters.

typescript

```
// src/api/itemService.ts
import apiClient from './apiClient';

export const fetchItems = async (page: number, limit: number) => {
  const response = await apiClient.get('/items', { params: { page, limit } });
  return response.data;
};
```

2. Create Pagination Hook with React Query

- This hook will handle fetching the paginated items and allow pagination state management.

typescript

```
// src/hooks/usePaginatedItems.ts
import { useQuery } from '@tanstack/react-query';
import { fetchItems } from '../api/itemService';

export const usePaginatedItems = (page: number, limit: number) => {
  return useQuery(['items', page], () => fetchItems(page, limit), {
    keepPreviousData: true, // Allows smooth transitions between pages
  });
};
```

3. Implement Pagination UI Component

- Use a basic UI with "Previous" and "Next" buttons, and display the list of items.
- Manage the current page with React state and update it when navigating between pages.

typescript

```
// src/components/ItemList.tsx
import React, { useState } from 'react';
import { usePaginatedItems } from '../hooks/usePaginatedItems';

const ItemList: React.FC = () => {
  const [page, setPage] = useState(1);
  const limit = 10; // Define how many items per page
  const { data, isLoading, isError } = usePaginatedItems(page, limit);

  const handleNextPage = () => {
    if (data && page < data.totalPages) setPage((prev) => prev + 1);
  };

  const handlePrevPage = () => {
    if (page > 1) setPage((prev) => prev - 1);
  };

  if (isLoading) return <div>Loading...</div>;
  if (isError) return <div>Error loading items</div>;

  return (
    <div>
      <ul>
        {data?.data.map((item: any) => (
          <li key={item.id}>{item.name}</li>
        ))}
      </ul>
      <div>
        <button onClick={handlePrevPage} disabled={page === 1}>
          Previous
        </button>
        <span>Page {page} of {data?.totalPages}</span>
        <button onClick={handleNextPage} disabled={page === data?.totalPages}>
          Next
        </button>
      </div>
    </div>
  );
};

export default ItemList;
```

Summary

1. **Backend**: Define a paginated API route, and calculate items to skip based on the page and limit.
2. **Frontend**: Use React Query to fetch paginated data and a custom hook to handle pagination. Then, use buttons to navigate between pages.

This setup allows you to scale easily, as you only fetch the necessary data per page, keeping the app responsive and efficient.

what is mean data?.data.map((item: any)

In the line `data?.data.map((item: any) => ( ... ))`, the `?.` syntax is known as the **optional chaining** operator. Here's a breakdown of what each part does:

1. `data?`: The optional chaining (`?.`) is used to safely access properties of `data` without causing an error if `data` is `null` or `undefined`. This is especially useful in asynchronous operations where `data` may not be immediately available (like when data is still loading).
2. `data?.data`: Assuming `data` exists, the second `.data` refers to a property on the `data` object. In this example, the `data` object from `React Query`'s API response is expected to contain a `data` field with the actual array of items. The API response might look something like:

   ```json
   {
     "data": [
       { "id": 1, "name": "Item 1" },
       { "id": 2, "name": "Item 2" }
     ],
     "page": 1,
     "totalItems": 20,
     "totalPages": 2
   }
   ```

   So, `data?.data` is the array of items we want to display.
3. `.map((item: any) => ( ... ))`: Assuming `data?.data` is an array, `.map` is called to iterate over each item in the array, allowing us to render each item in the UI. Here, `(item: any)` is the TypeScript type annotation, with `item` typed as `any` (ideally, you'd replace `any` with the actual type).

Full Example in Context

If `data` exists and `data.data` is an array, `data?.data.map` will iterate over each item. If `data` is `undefined` or `null`, `data?.data` returns `undefined`, and `.map` will not be called, preventing any runtime errors.

give me exmaple for paginaiton in react ts with calling api with react query with good structre and with scss

Here’s an example of setting up pagination in a React TypeScript application with React Query and styling with SCSS. We’ll use a good structure to separate concerns, keeping the API, hooks, and components organized.

1. API Setup

Define an API client and a function to fetch paginated data.

`src/api/apiClient.ts`

This sets up a base API client, like Axios, for making HTTP requests.

```typescript
// src/api/apiClient.ts
import axios from 'axios';

const apiClient = axios.create({
  baseURL: 'https://your-api.com/api',
  headers: { 'Content-Type': 'application/json' },
});

export default apiClient;
```

`src/api/itemService.ts`

This function fetches paginated data from the API and accepts `page` and `limit` parameters.

```typescript
// src/api/itemService.ts
import apiClient from './apiClient';

export const fetchItems = async (page: number, limit: number) => {
  const response = await apiClient.get('/items', { params: { page, limit } });
  return response.data;
};
```

2. Define TypeScript Types

Define types for better type safety.

`src/types/Item.ts`

```typescript
// src/types/Item.ts
export interface Item {
  id: number;
  name: string;
}

export interface PaginatedResponse<T> {
  data: T[];
  page: number;
  totalItems: number;
  totalPages: number;
}
```

3. React Query Hook

Create a custom hook that handles the pagination logic with React Query.

`src/hooks/usePaginatedItems.ts`

```typescript
// src/hooks/usePaginatedItems.ts
import { useQuery } from '@tanstack/react-query';
import { fetchItems } from '../api/itemService';
import { PaginatedResponse, Item } from '../types/Item';

export const usePaginatedItems = (page: number, limit: number) => {
  return useQuery<PaginatedResponse<Item>>(['items', page], () => fetchItems(page, limit), {
    keepPreviousData: true, // Enables smooth page transitions
  });
};
```

4. Pagination Component

This component uses the `usePaginatedItems` hook and includes the UI for pagination.

`src/components/ItemList.tsx`

```typescript
```

```
// src/components/ItemList.tsx
import React, { useState } from 'react';
import { usePaginatedItems } from '../hooks/usePaginatedItems';
import '../styles/ItemList.scss';

const ItemList: React.FC = () => {
  const [page, setPage] = useState(1);
  const limit = 10; // Items per page

  const { data, isLoading, isError } = usePaginatedItems(page, limit);

  const handleNextPage = () => {
    if (data && page < data.totalPages) setPage((prev) => prev + 1);
  };

  const handlePrevPage = () => {
    if (page > 1) setPage((prev) => prev - 1);
  };

  if (isLoading) return <div>Loading...</div>;
  if (isError) return <div>Error loading items</div>;

  return (
    <div className="item-list">
      <ul>
        {data?.data.map((item) => (
          <li key={item.id} className="item">
            {item.name}
          </li>
        ))}
      </ul>
      <div className="pagination-controls">
        <button onClick={handlePrevPage} disabled={page === 1}>
          Previous
        </button>
        <span>
          Page {page} of {data?.totalPages}
        </span>
        <button onClick={handleNextPage} disabled={page === data?.totalPages}>
          Next
        </button>
      </div>
    </div>
  );
};

export default ItemList;
```

5. Styling with SCSS

Create styles for the list and pagination controls.

`src/styles/ItemList.scss`

```scss
// src/styles/ItemList.scss
.item-list {
  max-width: 600px;
  margin: 0 auto;
  padding: 20px;
  font-family: Arial, sans-serif;

  .item {
    padding: 10px;
    border-bottom: 1px solid #ddd;
    &:last-child {
      border-bottom: none;
    }
  }

  .pagination-controls {
    display: flex;
    justify-content: space-between;
    align-items: center;
    margin-top: 20px;

    button {
      padding: 8px 16px;
      background-color: #007bff;
      color: white;
      border: none;
      border-radius: 4px;
      cursor: pointer;
      &:disabled {
        background-color: #ccc;
        cursor: not-allowed;
      }
    }

    span {
      font-weight: bold;
    }
  }
}
```

6. Main App Component

Integrate `ItemList` into your main app component.

`src/App.tsx`

typescript

```typescript
// src/App.tsx
import React from 'react';
import { QueryClient, QueryClientProvider } from '@tanstack/react-query';
import ItemList from './components/ItemList';

const queryClient = new QueryClient();

const App: React.FC = () => (
  <QueryClientProvider client={queryClient}>
    <div className="App">
      <h1>Paginated Item List</h1>
      <ItemList />
    </div>
  </QueryClientProvider>
);

export default App;
```

7. Optional: Global SCSS Setup

If you’re using SCSS for the whole app, set up a global stylesheet for basic styling.

`src/styles/global.scss`

```scss
// src/styles/global.scss
body {
  margin: 0;
  font-family: Arial, sans-serif;
  background-color: #f4f4f9;
  color: #333;
}

.App {
  text-align: center;
  padding: 20px;
}
```

Then, import the global SCSS file in the main entry point (e.g., `index.tsx`).

```typescript
// src/index.tsx
import React from 'react';
import ReactDOM from 'react-dom/client';
import './styles/global.scss';
import App from './App';

ReactDOM.createRoot(document.getElementById('root')!).render(<App />);
```

Summary

1. **API Layer**: Use `axios` to create an API client and a service function to fetch paginated data.
2. **Custom Hook**: Use React Query to create a hook that handles pagination.
3. **Pagination Component**: Display items and pagination controls, with SCSS for styling.
4. **Global Styling**: Set up global SCSS for a consistent look.

This structure keeps the code organized, maintains type safety, and ensures separation of concerns, making it easy to extend and maintain.

this example but with button as the count paginte

To add numbered pagination buttons instead of "Next" and "Previous," we'll display buttons for each page based on the total number of pages. This approach will use React Query for fetching paginated data and a layout with pagination buttons for navigation.

1. API Setup

We'll use the same API setup as before. If the backend returns a total number of pages, we can use that to dynamically render pagination buttons.

`src/api/apiClient.ts`

```typescript
// src/api/apiClient.ts
import axios from 'axios';

const apiClient = axios.create({
  baseURL
```

2/2

how to do good structre paginataion for pages if we have100 items and we need to each page 20 items with react ts and react query and scss with good structre

To implement pagination with a good structure in React TypeScript using React Query and SCSS, where we have 100 items with 20 items per page, we can break down the setup as follows:

1. **API Client and Service Layer**: Use Axios to call a paginated endpoint.
2. **Custom React Query Hook**: Use React Query to handle the pagination, fetching based on the current page and limit.
3. **Pagination Component**: Display paginated items and numbered pagination buttons.
4. **SCSS Styling**: Style the pagination component and buttons for a user-friendly interface.

Project Structure

```plaintext
src/
├── api/
│   ├── apiClient.ts
│   └── itemService.ts
├── components/
│   └── ItemList.tsx
├── hooks/
│   └── usePaginatedItems.ts
├── types/
│   └── Item.ts
├── styles/
│   ├── ItemList.scss
│   └── Pagination.scss
└── App.tsx
```

1. API Setup

Set up an API client and a function to fetch paginated data.

`src/api/apiClient.ts`

```typescript
import axios from 'axios';

const apiClient = axios.create({
  baseURL: 'https://your-api.com/api',
  headers: { 'Content-Type': 'application/json' },
});

export default apiClient;
```

`src/api/itemService.ts`

This function fetches paginated items from the API and takes `page` and `limit` parameters.

```typescript
import apiClient from './apiClient';

export const fetchItems = async (page: number, limit: number) => {
  const response = await apiClient.get('/items', { params: { page, limit } });
  return response.data;
};
```

2. Define TypeScript Types

Define types for our API response.

`src/types/Item.ts`

```typescript
export interface Item {
  id: number;
  name: string;
}

export interface PaginatedResponse<T> {
  data: T[];
  page: number;
  totalItems: number;
  totalPages: number;
}
```

3. React Query Hook

Create a custom hook that handles pagination logic using React Query.

`src/hooks/usePaginatedItems.ts`

```typescript
import { useQuery } from '@tanstack/react-query';
import { fetchItems } from '../api/itemService';
import { PaginatedResponse, Item } from '../types/Item';

export const usePaginatedItems = (page: number, limit: number) => {
  return useQuery<PaginatedResponse<Item>>(['items', page], () => fetchItems(page, limit), {
    keepPreviousData: true,
  });
};
```

4. Pagination Component

This component uses `usePaginatedItems` to display items and generates numbered pagination buttons.

`src/components/ItemList.tsx`

typescript

```typescript
import React, { useState } from 'react';
import { usePaginatedItems } from '../hooks/usePaginatedItems';
import '../styles/ItemList.scss';
import '../styles/Pagination.scss';

const ItemList: React.FC = () => {
  const [page, setPage] = useState(1);
  const limit = 20; // 20 items per page

  const { data, isLoading, isError } = usePaginatedItems(page, limit);

  const totalPages = data?.totalPages || 1;

  const handlePageClick = (pageNumber: number) => {
    setPage(pageNumber);
  };

  if (isLoading) return <div>Loading...</div>;
  if (isError) return <div>Error loading items</div>;

  return (
    <div className="item-list">
      <ul>
        {data?.data.map((item) => (
          <li key={item.id} className="item">
            {item.name}
          </li>
        ))}
      </ul>
      <div className="pagination">
        {[...Array(totalPages)].map((_, index) => (
          <button
            key={index}
            onClick={() => handlePageClick(index + 1)}
            className={`pagination-button ${page === index + 1 ? 'active' : ''}`}
          >
            {index + 1}
          </button>
        ))}
      </div>
    </div>
  );
};

export default ItemList;
```

5. Styling with SCSS

`src/styles/ItemList.scss`

Style the list of items:

```scss
.item-list {
  max-width: 600px;
  margin: 0 auto;
  padding: 20px;

  .item {
    padding: 10px;
    border-bottom: 1px solid #ddd;
    &:last-child {
      border-bottom: none;
    }
  }
}
```

`src/styles/Pagination.scss`

Style pagination buttons:

```scss
.pagination {
  display: flex;
  justify-content: center;
  margin-top: 20px;

  .pagination-button {
    padding: 8px 12px;
    margin: 0 5px;
    border: none;
    background-color: #007bff;
    color: white;
    border-radius: 4px;
    cursor: pointer;
    transition: background-color 0.3s;

    &.active {
      background-color: #0056b3;
    }

    &:hover {
      background-color: #0056b3;
    }

    &:disabled {
      background-color: #ccc;
      cursor: not-allowed;
    }
  }
}
```

6. Main App Component

Add `ItemList` to your main app component.

`src/App.tsx`

```typescript
import React from 'react';
import { QueryClient, QueryClientProvider } from '@tanstack/react-query';
import ItemList from './components/ItemList';

const queryClient = new QueryClient();

const App: React.FC = () => (
  <QueryClientProvider client={queryClient}>
    <div className="App">
      <h1>Paginated Item List</h1>
      <ItemList />
    </div>
  </QueryClientProvider>
);
```

```
export default App;
```

7. Optional: Global SCSS Setup

If you want global styling, set it up in `src/styles/global.scss` and import it in `index.tsx`.

`src/styles/global.scss`

```scss
body {
  margin: 0;
  font-family: Arial, sans-serif;
  background-color: #f4f4f9;
  color: #333;
}

.App {
  text-align: center;
  padding: 20px;
}
```

`src/index.tsx`

```typescript
import React from 'react';
import ReactDOM from 'react-dom/client';
import './styles/global.scss';
import App from './App';

ReactDOM.createRoot(document.getElementById('root')!).render(<App />);
```

Summary

1. **API Layer**: Set up Axios with an API function to fetch paginated data.
2. **Custom Hook**: Use React Query for fetching and caching paginated data.
3. **Pagination Component**: Display items and page buttons to navigate.
4. **Styling**: Use SCSS for styling items and pagination buttons.