Create ReactQuery implementaion for login api(mobile based login, otp verification), vendor list, vendor detail, Orderslist, OrderStatusUpdate(in-transit, Delivered) with scalable and easy switch for different environment.

Creating a scalable and flexible React Query implementation for a mobile-based login system, vendor list and details, and order management requires careful planning. The goal is to make it easy to switch between different environments (e.g., development, staging, production). Here's a basic outline of how you can achieve this:

**Step 1: Setup Environment Variables**

First, define environment variables for different environments. Create a .env file for each environment (e.g., .env.development, .env.production) and define your API base URLs:

REACT\_APP\_API\_BASE\_URL=https://api.example.com

**Step 2: Install Required Packages**

Ensure you have the necessary packages installed:

npm install react-query axios

**Step 3: Setup Axios Instance**

Create a reusable Axios instance configured to use the base URL from the environment variables. This allows easy switching between environments.

*// src/api/axiosInstance.js*

import axios from 'axios';

const axiosInstance = axios.create({

baseURL: process.env.REACT\_APP\_API\_BASE\_URL,

headers: {

'Content-Type': 'application/json',

},

});

export default axiosInstance;

**Step 4: Create API Functions**

Define functions to interact with your API endpoints.

*// src/api/api.js*

import axiosInstance from './axiosInstance';

*// Login API*

export const login = async (phoneNumber) => {

const response = await axiosInstance.post('/auth/login', { phoneNumber });

return response.data;

};

*// OTP Verification*

export const verifyOtp = async (phoneNumber, otp) => {

const response = await axiosInstance.post('/auth/verify-otp', { phoneNumber, otp });

return response.data;

};

*// Get Vendor List*

export const fetchVendors = async () => {

const response = await axiosInstance.get('/vendors');

return response.data;

};

*// Get Vendor Detail*

export const fetchVendorDetail = async (vendorId) => {

const response = await axiosInstance.get(`/vendors/${vendorId}`);

return response.data;

};

*// Get Orders List*

export const fetchOrders = async () => {

const response = await axiosInstance.get('/orders');

return response.data;

};

*// Update Order Status*

export const updateOrderStatus = async (orderId, status) => {

const response = await axiosInstance.put(`/orders/${orderId}/status`, { status });

return response.data;

};

**Step 5: Setup React Query Hooks**

Create custom hooks to use these API functions with React Query.

// src/hooks/useAuth.js

import { useMutation } from 'react-query';

import { login, verifyOtp } from '../api/api';

export const useLogin = () => {

return useMutation((phoneNumber) => login(phoneNumber));

};

export const useVerifyOtp = () => {

return useMutation(({ phoneNumber, otp }) => verifyOtp(phoneNumber, otp));

};

// src/hooks/useVendors.js

import { useQuery } from 'react-query';

import { fetchVendors, fetchVendorDetail } from '../api/api';

export const useVendors = () => {

return useQuery('vendors', fetchVendors);

};

export const useVendorDetail = (vendorId) => {

return useQuery(['vendorDetail', vendorId], () => fetchVendorDetail(vendorId), {

enabled: !!vendorId,

});

};

// src/hooks/useOrders.js

import { useQuery, useMutation, useQueryClient } from 'react-query';

import { fetchOrders, updateOrderStatus } from '../api/api';

export const useOrders = () => {

return useQuery('orders', fetchOrders);

};

export const useUpdateOrderStatus = () => {

const queryClient = useQueryClient();

return useMutation(({ orderId, status }) => updateOrderStatus(orderId, status), {

onSuccess: () => {

queryClient.invalidateQueries('orders');

},

});

};

**Step 6: Implement in Components**

Use these hooks in your components to manage the API interactions.

*// ExampleComponent.js*

import React from 'react';

import { useLogin, useVerifyOtp } from './hooks/useAuth';

import { useVendors, useVendorDetail } from './hooks/useVendors';

import { useOrders, useUpdateOrderStatus } from './hooks/useOrders';

const ExampleComponent = () => {

const { mutate: login } = useLogin();

const { mutate: verifyOtp } = useVerifyOtp();

const { data: vendors } = useVendors();

const { data: orders } = useOrders();

const { mutate: updateOrderStatus } = useUpdateOrderStatus();

*// Example usage*

const handleLogin = () => {

login('1234567890');

};

const handleVerifyOtp = () => {

verifyOtp({ phoneNumber: '1234567890', otp: '123456' });

};

const handleUpdateOrderStatus = (orderId, status) => {

updateOrderStatus({ orderId, status });

};

return (

<div>

{/\* Render UI components and trigger these functions as needed \*/}

</div>

);

};

export default ExampleComponent;

**Conclusion**

This setup provides a scalable and environment-flexible approach to implementing a React Query-based API system. By using environment variables, you can easily switch between different environments without changing your code. The custom hooks make your API interactions clean and reusable across different components.

Create ReactQuery implementaion for login api(mobile based login, otp verification), vendor list, vendor detail, Orderslist, OrderStatusUpdate(in-transit, Delivered) with scalable and easy switch for different environment and store user access token for all service in logout to clear the token

To implement a React Query setup that handles login with mobile-based OTP verification, vendor details, order management, and token management for different environments, follow these steps. We'll ensure that the access token is stored securely and cleared upon logout.

**Step 1: Setup Environment Variables**

Define environment variables for different environments in .env.development, .env.production, etc.

REACT\_APP\_API\_BASE\_URL=https://api.example.com

**Step 2: Install Required Packages**

Install react-query, axios, and optionally js-cookie for handling cookies if you choose to store tokens in cookies.

npm install react-query axios js-cookie

**Step 3: Setup Axios Instance with Token Handling**

Create a reusable Axios instance that includes token handling. The token can be stored in localStorage or cookies.

*// src/api/axiosInstance.js*

import axios from 'axios';

import Cookies from 'js-cookie';

const axiosInstance = axios.create({

baseURL: process.env.REACT\_APP\_API\_BASE\_URL,

headers: {

'Content-Type': 'application/json',

},

});

*// Add a request interceptor to include the token*

axiosInstance.interceptors.request.use(

(config) => {

const token = Cookies.get('accessToken');

if (token) {

config.headers.Authorization = `Bearer ${token}`;

}

return config;

},

(error) => Promise.reject(error)

);

export default axiosInstance;

**Step 4: Create API Functions**

Define functions for your API interactions.

*// src/api/api.js*

import axiosInstance from './axiosInstance';

import Cookies from 'js-cookie';

*// Login API*

export const login = async (phoneNumber) => {

const response = await axiosInstance.post('/auth/login', { phoneNumber });

return response.data;

};

*// OTP Verification*

export const verifyOtp = async (phoneNumber, otp) => {

const response = await axiosInstance.post('/auth/verify-otp', { phoneNumber, otp });

const { accessToken } = response.data;

if (accessToken) {

Cookies.set('accessToken', accessToken);

}

return response.data;

};

*// Logout*

export const logout = async () => {

Cookies.remove('accessToken');

};

*// Get Vendor List*

export const fetchVendors = async () => {

const response = await axiosInstance.get('/vendors');

return response.data;

};

*// Get Vendor Detail*

export const fetchVendorDetail = async (vendorId) => {

const response = await axiosInstance.get(`/vendors/${vendorId}`);

return response.data;

};

*// Get Orders List*

export const fetchOrders = async () => {

const response = await axiosInstance.get('/orders');

return response.data;

};

*// Update Order Status*

export const updateOrderStatus = async (orderId, status) => {

const response = await axiosInstance.put(`/orders/${orderId}/status`, { status });

return response.data;

};

**Step 5: Setup React Query Hooks**

Create custom hooks using React Query.

// src/hooks/useAuth.js

import { useMutation } from 'react-query';

import { login, verifyOtp, logout } from '../api/api';

export const useLogin = () => {

return useMutation((phoneNumber) => login(phoneNumber));

};

export const useVerifyOtp = () => {

return useMutation(({ phoneNumber, otp }) => verifyOtp(phoneNumber, otp));

};

export const useLogout = () => {

return useMutation(() => logout());

};

// src/hooks/useVendors.js

import { useQuery } from 'react-query';

import { fetchVendors, fetchVendorDetail } from '../api/api';

export const useVendors = () => {

return useQuery('vendors', fetchVendors);

};

export const useVendorDetail = (vendorId) => {

return useQuery(['vendorDetail', vendorId], () => fetchVendorDetail(vendorId), {

enabled: !!vendorId,

});

};

// src/hooks/useOrders.js

import { useQuery, useMutation, useQueryClient } from 'react-query';

import { fetchOrders, updateOrderStatus } from '../api/api';

export const useOrders = () => {

return useQuery('orders', fetchOrders);

};

export const useUpdateOrderStatus = () => {

const queryClient = useQueryClient();

return useMutation(({ orderId, status }) => updateOrderStatus(orderId, status), {

onSuccess: () => {

queryClient.invalidateQueries('orders');

},

});

};

**Step 6: Implement in Components**

Use these hooks in your components.

*// ExampleComponent.js*

import React from 'react';

import { useLogin, useVerifyOtp, useLogout } from './hooks/useAuth';

import { useVendors, useVendorDetail } from './hooks/useVendors';

import { useOrders, useUpdateOrderStatus } from './hooks/useOrders';

const ExampleComponent = () => {

const { mutate: login } = useLogin();

const { mutate: verifyOtp } = useVerifyOtp();

const { mutate: logout } = useLogout();

const { data: vendors } = useVendors();

const { data: orders } = useOrders();

const { mutate: updateOrderStatus } = useUpdateOrderStatus();

const handleLogin = () => {

login('1234567890');

};

const handleVerifyOtp = () => {

verifyOtp({ phoneNumber: '1234567890', otp: '123456' });

};

const handleLogout = () => {

logout();

};

const handleUpdateOrderStatus = (orderId, status) => {

updateOrderStatus({ orderId, status });

};

return (

<div>

{/\* Render UI components and trigger these functions as needed \*/}

<button onClick={handleLogin}>Login</button>

<button onClick={handleVerifyOtp}>Verify OTP</button>

<button onClick={handleLogout}>Logout</button>

</div>

);

};

export default ExampleComponent;

**Conclusion**

This setup provides a robust way to manage different environments and handle authentication tokens securely. By using environment variables and a centralized Axios instance, you can easily switch between environments. The custom hooks make API interactions clean and reusable, and token management is handled efficiently with cookies or local storage.

This above ReactQuery implementation keep on calling server api

If your React Query implementation is repeatedly calling the server API, it might be due to several reasons, such as not properly managing the query's lifecycle or configuration. Here are some common causes and solutions to address this issue:

**1. Ensure Query Keys are Stable**

Ensure that the query keys are stable and do not change between renders unless necessary. If the key changes, React Query treats it as a new query and refetches data.

// src/hooks/useVendorDetail.js

export const useVendorDetail = (vendorId) => {

return useQuery(

['vendorDetail', vendorId],

() => fetchVendorDetail(vendorId),

{

enabled: !!vendorId, // Only fetch if vendorId is not null or undefined

}

);

};

**2. Disable Refetch on Window Focus**

React Query, by default, refetches queries when the window regains focus. You can disable this behavior if it's not needed.

*// src/hooks/useVendors.js*

export const useVendors = () => {

return useQuery('vendors', fetchVendors, {

refetchOnWindowFocus: false, *// Disable refetch on window focus*

});

};

**3. Manage Cache Time and Stale Time**

Adjust the cacheTime and staleTime to control how long the data remains fresh. This can help in reducing unnecessary refetches.

* staleTime: Time in milliseconds before the data is considered stale.
* cacheTime: Time in milliseconds to keep unused data in memory.

*// src/hooks/useOrders.js*

export const useOrders = () => {

return useQuery('orders', fetchOrders, {

staleTime: 5 \* 60 \* 1000, *// 5 minutes*

cacheTime: 10 \* 60 \* 1000, *// 10 minutes*

});

};

**4. Check for Unnecessary Re-renders**

Ensure that your components are not re-rendering unnecessarily, which could lead to repeated query executions. Use React.memo or useMemo to optimize component re-renders.

*// ExampleComponent.js*

import React, { useMemo } from 'react';

import { useVendors } from './hooks/useVendors';

const ExampleComponent = React.memo(() => {

const { data: vendors } = useVendors();

const vendorList = useMemo(() => {

return vendors?.map((vendor) => <div key={vendor.id}>{vendor.name}</div>);

}, [vendors]);

return <div>{vendorList}</div>;

});

export default ExampleComponent;

**5. Check for Enabled Queries**

Ensure that queries are enabled only when necessary. If using conditional queries, make sure the conditions are set correctly.

// src/hooks/useVendorDetail.js

export const useVendorDetail = (vendorId) => {

return useQuery(

['vendorDetail', vendorId],

() => fetchVendorDetail(vendorId),

{

enabled: !!vendorId, // Ensure this is only true when vendorId is valid

}

);

};

**6. Review Query Client Configuration**

Make sure your React Query client is configured correctly, especially if you have set global defaults.

*// src/index.js or App.js*

import { QueryClient, QueryClientProvider } from 'react-query';

const queryClient = new QueryClient({

defaultOptions: {

queries: {

refetchOnWindowFocus: false,

staleTime: 5 \* 60 \* 1000, *// 5 minutes*

cacheTime: 10 \* 60 \* 1000, *// 10 minutes*

},

},

});

const App = () => (

<QueryClientProvider client={queryClient}>

{/\* Your app components \*/}

</QueryClientProvider>

);

export default App;

**Conclusion**

By ensuring stable query keys, managing query lifecycle options such as refetchOnWindowFocus, staleTime, and cacheTime, and optimizing component re-renders, you can prevent unnecessary API calls with React Query. Adjust these settings based on your application's specific needs and usage patterns.

Create ReactQuery implementaion for login api(mobile based login, otp verification), vendor list, vendor detail, Orderslist, OrderStatusUpdate(in-transit, Delivered) with scalable and easy switch for different environment and store user access token for all service in logout to clear the token an avoid unnessary api call whenever change happen in server table only need call the api

To create a React Query implementation that handles mobile-based login with OTP verification, vendor details, and order management while ensuring scalability, environment flexibility, and efficient API calls, follow these steps. We'll also manage the user access token and avoid unnecessary API calls by using server-side updates.

**Step 1: Setup Environment Variables**

Define environment variables for different environments in .env.development, .env.production, etc.

REACT\_APP\_API\_BASE\_URL=https://api.example.com

**Step 2: Install Required Packages**

Install react-query, axios, and optionally js-cookie for handling cookies if you choose to store tokens in cookies.

npm install react-query axios js-cookie

**Step 3: Setup Axios Instance with Token Handling**

Create a reusable Axios instance that includes token handling. The token can be stored in localStorage or cookies.

*// src/api/axiosInstance.js*

import axios from 'axios';

import Cookies from 'js-cookie';

const axiosInstance = axios.create({

baseURL: process.env.REACT\_APP\_API\_BASE\_URL,

headers: {

'Content-Type': 'application/json',

},

});

*// Add a request interceptor to include the token*

axiosInstance.interceptors.request.use(

(config) => {

const token = Cookies.get('accessToken');

if (token) {

config.headers.Authorization = `Bearer ${token}`;

}

return config;

},

(error) => Promise.reject(error)

);

export default axiosInstance;

**Step 4: Create API Functions**

Define functions for your API interactions.

*// src/api/api.js*

import axiosInstance from './axiosInstance';

import Cookies from 'js-cookie';

*// Login API*

export const login = async (phoneNumber) => {

const response = await axiosInstance.post('/auth/login', { phoneNumber });

return response.data;

};

*// OTP Verification*

export const verifyOtp = async (phoneNumber, otp) => {

const response = await axiosInstance.post('/auth/verify-otp', { phoneNumber, otp });

const { accessToken } = response.data;

if (accessToken) {

Cookies.set('accessToken', accessToken);

}

return response.data;

};

*// Logout*

export const logout = async () => {

Cookies.remove('accessToken');

};

*// Get Vendor List*

export const fetchVendors = async () => {

const response = await axiosInstance.get('/vendors');

return response.data;

};

*// Get Vendor Detail*

export const fetchVendorDetail = async (vendorId) => {

const response = await axiosInstance.get(`/vendors/${vendorId}`);

return response.data;

};

*// Get Orders List*

export const fetchOrders = async () => {

const response = await axiosInstance.get('/orders');

return response.data;

};

*// Update Order Status*

export const updateOrderStatus = async (orderId, status) => {

const response = await axiosInstance.put(`/orders/${orderId}/status`, { status });

return response.data;

};

**Step 5: Setup React Query Hooks**

Create custom hooks using React Query.

// src/hooks/useAuth.js

import { useMutation } from 'react-query';

import { login, verifyOtp, logout } from '../api/api';

export const useLogin = () => {

return useMutation((phoneNumber) => login(phoneNumber));

};

export const useVerifyOtp = () => {

return useMutation(({ phoneNumber, otp }) => verifyOtp(phoneNumber, otp));

};

export const useLogout = () => {

return useMutation(() => logout());

};

// src/hooks/useVendors.js

import { useQuery } from 'react-query';

import { fetchVendors, fetchVendorDetail } from '../api/api';

export const useVendors = () => {

return useQuery('vendors', fetchVendors, {

refetchOnWindowFocus: false,

refetchOnReconnect: false,

staleTime: Infinity, // Ensures data is considered fresh

});

};

export const useVendorDetail = (vendorId) => {

return useQuery(['vendorDetail', vendorId], () => fetchVendorDetail(vendorId), {

enabled: !!vendorId,

refetchOnWindowFocus: false,

refetchOnReconnect: false,

staleTime: Infinity,

});

};

// src/hooks/useOrders.js

import { useQuery, useMutation, useQueryClient } from 'react-query';

import { fetchOrders, updateOrderStatus } from '../api/api';

export const useOrders = () => {

return useQuery('orders', fetchOrders, {

refetchOnWindowFocus: false,

refetchOnReconnect: false,

staleTime: Infinity,

});

};

export const useUpdateOrderStatus = () => {

const queryClient = useQueryClient();

return useMutation(({ orderId, status }) => updateOrderStatus(orderId, status), {

onSuccess: () => {

queryClient.invalidateQueries('orders');

},

});

};

**Step 6: Implement in Components**

Use these hooks in your components to manage API interactions efficiently.

*// ExampleComponent.js*

import React from 'react';

import { useLogin, useVerifyOtp, useLogout } from './hooks/useAuth';

import { useVendors } from './hooks/useVendors';

import { useOrders, useUpdateOrderStatus } from './hooks/useOrders';

const ExampleComponent = () => {

const { mutate: login } = useLogin();

const { mutate: verifyOtp } = useVerifyOtp();

const { mutate: logout } = useLogout();

const { data: vendors } = useVendors();

const { data: orders } = useOrders();

const { mutate: updateOrderStatus } = useUpdateOrderStatus();

const handleLogin = () => {

login('1234567890');

};

const handleVerifyOtp = () => {

verifyOtp({ phoneNumber: '1234567890', otp: '123456' });

};

const handleLogout = () => {

logout();

};

const handleUpdateOrderStatus = (orderId, status) => {

updateOrderStatus({ orderId, status });

};

return (

<div>

<button onClick={handleLogin}>Login</button>

<button onClick={handleVerifyOtp}>Verify OTP</button>

<button onClick={handleLogout}>Logout</button>

{/\* Render vendors and orders \*/}

</div>

);

};

export default ExampleComponent;

**Conclusion**

This setup ensures that your API calls are efficient and only happen when necessary. By using staleTime: Infinity, data is considered always fresh unless explicitly invalidated, reducing unnecessary API calls. The token management is handled securely using cookies, and environment switching is easy with environment variables. Adjust the refetch options based on your application's specific needs to optimize performance.