React Query

Terms

Caching
Mutations
Infinite queries
Paginated queries
Parameterized queries
Optimistic updates
Query key
Query function

Summary

- React Query is a library that simplifies fetching, caching, and updating data in React applications.
- It provides two primary hooks for fetching and updating data: **useQuery** and **useMutation**.
- To create a query object, we provide a **queryKey** that identifies the query and a **queryFn** that fetches the data.
- Query objects have three important properties: **data**, **error**, and **isLoading**.
- To provide better separation of concerns and reuse opportunities, we can encapsulate our queries in custom hooks.
- React Query provides a set of DevTools that can be used to inspect and debug your queries.
- React Query provides a number of options that can be used to customize the behavior of our queries, such as caching options and query retry behavior. These options can be set globally or on a per-query basis.

- React Query supports parameterized queries, where query parameters can be passed as arguments to the **useQuery** hook.
- React Query provides built-in support for paginated and infinite queries.
- To create a mutation object, we provide a **mutationFn** that mutates the data.
- Mutation objects have callbacks such as **onSuccess**, **onError** and **onMutate** (used for optimistic updates).

CREATING A QUERY

```
const { data, error, isLoading } = useQuery<Todo[], Error>({
  queryKey: ['todos'],
  queryFn: () => axios.get('').then((res) => res.data),
});
```

CUSTOMIZING QUERY SETTINGS

```
// Globally
const client = new QueryClient({
  defaultOptions: {
    queries: {
      retry: 3,
      cacheTime: 300_000, // 5m
      staleTime: 10 * 1000, // 10s
    },
  },
});
```

```
// Per query
const { data } = useQuery({
  staleTime: 10 * 1000, // 10s
});
```

PARAMETERIZED QUERIES

```
const { data } = useQuery<Todo[], Error>({
  queryKey: ['users', userId, 'todos'],
});
```

CREATING A MUTATION

```
const queryClient = useQueryClient();

const { error, isLoading } = useMutation<Todo, Error, Todo>({
  mutationFn: () => axios.post('').then((res) => res.data),
  onSuccess: (savedTodo, newTodo) => {
    // Approach 1: Invalidate the cache
    queryClient.invalidateQueries({ queryKey: ['todos'] });

    // Approach 2: Update the cache
    queryClient.setQueryData<Todo[]>(['todos'], (todos = []) => [
      savedTodo,
      ...todos,
    ]);
  },
});
```

Global State Management

Terms

Client state	Reducer
Context	Server state
Prop drilling	State management libraries
Provider	

Summary

- Most React applications have some state to be managed, which can be server or client state.
- React Query helps us manage and cache server state in a simple, elegant way.
- We can define local state in our components using the state or reducer hooks.
- If state management logic becomes too complex and scattered, we can consolidate it using a reducer, which is a pure function that takes the current state and an action, and returns the new state.
- To share state between components, we can lift the state up to the closest parent and pass it down as props, but this can lead to prop drilling in larger, more complex trees.
- Context allows components to share data without having to pass it down manually through each level of the component tree.
- Providers are components that wrap a group of child components and provide them with access to a specific context object.
- We can use custom hooks to access context in a more readable and reusable way.

- When an object contained by a context is updated, all components using that context will re-render.

- To minimize unnecessary re-renders, we should split up a context into smaller, more focused ones with a single responsibility. However, too finely grained contexts can result in complex, hard to maintain component trees.

- If splitting up a context doesn't make sense and we need more control over state updates and component rendering, we can use a state management library.

- There are many state management libraries for React apps. Examples are Redux, MobX, recoil, xState, Zustand, and more.

- These days, React Query and Zustand can replace the need for Redux in many applications.

CONSOLIDATING STATE UPDATES WITH A REDUCER

```
interface Action {
  type: 'INCREMENT' | 'RESET';
}

const counterReducer = (state: number, action: Action): number => {
  if (action.type === 'INCREMENT') return state + 1;
  if (action.type === 'RESET') return 0;
  return state;
}

const Counter = () => {
  const [value, dispatch] = useReducer(counterReducer, 0);

  return (
    <>
      <p>{value}</p>
      <button onClick={() => dispatch({ type: 'INCREMENT' })}>
        Increment
      </button>
    </>
  );
};
```

ACTIONS WITH A PAYLOAD

```
interface Task {
  id: number;
  title: string;
}

interface AddTask {
  type: 'ADD';
  task: Task;
}

interface DeleteTask {
  type: 'DELETE';
  taskId: number;
}

type TaskAction = AddTask | DeleteTask;

const tasksReducer = (tasks: Task[], action: TaskAction): Task[] => {
  switch (action.type) {
    case 'ADD':
        return [action.task, ...tasks];
    case 'DELETE':
        return tasks.filter(t => t.id !== action.taskId);
  }
}
```

SHARING STATE WITH CONTEXT

```
interface TasksContextType {
  tasks: Task[];
  dispatch: Dispatch<TaskAction>;
}

const TasksContext = React.createContext<TasksContextType>(
  {} as TasksContextType
);


function App() {
  const [tasks, dispatch] = useReducer(tasksReducer, []);

  return (
      <TasksContext.Provider value={{ tasks, dispatch }}>
        <NavBar />
        <HomePage />
      </TasksContext.Provider>
  );
}
```

CREATING A PROVIDER

```
interface Props {
  children: ReactNode;
}

const TasksProvider = ({ children }: Props) => {
  const [tasks, dispatch] = useReducer(tasksReducer, []);

  return (
    <TasksContext.Provider value={{ tasks, dispatch }}>
      {children}
    </TasksContext.Provider>
  );
};
```

CREATING A CUSTOM HOOK TO ACCESS CONTEXT

```
const useTasks = () => useContext(TasksContext);
```

MANAGING STATE WITH ZUSTAND

```
import { create } from 'zustand';

interface CounterStore {
  value: number;
  increment: () => void;
}

const useCounterStore = create<CounterStore>((set) => ({
  value: 0,
  increment: () => set((store) => ({ value: store.value + 1 })),
}));


const Counter = () => {
  const { value, increment } = useCounterStore();

  return (
    <>
      <p>{value}</p>
      <button onClick={() => increment()}>Increment</button>
    </>
  );
};
```

Routing

Terms

Index routes	Route parameters
Layout routes	Query string parameters
Route	

Summary

- React Router is a popular routing library for React that allows us to handle navigation and rendering of different components based on the URL.

- To create a router, we use the **createBrowserRouter()** function and provide an array of route objects.

- Each route has two properties: **path** and **element**, where path is a string that defines the URL pattern and element is the component to be rendered when the path matches the current URL.

- Route parameters are defined using a colon (e.g., /users/:userId) in the path of a route object, allowing us to capture dynamic values from the URL and pass them to our components.

- The **useParams()** hook is used to retrieve the route parameters in the current route.

- The **useSearchParams()** hook is used to retrieve and update the query string parameters of the current URL.

- The **<Link>** component is used to create links between different routes.

- The **<NavLink>** component works like the **<Link>** component, but it adds an active class to the current link by default. The name of this class can be customized by passing a function to the **className** prop.

- The **useNavigate()** hook is used to redirect the user programmatically.

- The **errorElement** property of the root route can be used to specify a component to render when a route is not found or an error occurs during rendering.

- The **<Outlet>** component is used as a placeholder to render child component when working with nested routes.

- Layout routes are useful to enforce a common layout or business rules across a group of routes. A layout route is a route without a path.

- An index route represents the default component to render in the outlet of a parent route when the current URL matches the path of the parent route.

SETTING UP ROUTES

```
const router = createBrowserRouter([
  { path: '/', element: <HomePage /> },
  { path: '/users', element: <UserListPage /> },
]);

<React.StrictMode>
  <RouterProvider router={router} />
</React.StrictMode>
```

NAVIGATION

```
<Link to='/users'>Users</Link>

// Adds the "active" class if the route
// matches the current URL.
<NavLink to='/users'>Users</NavLink>
```

PROGRAMMATIC NAVIGATION

```
const navigate = useNavigate();

const handleSubmit = () => {
  navigate('/');
}
```

ROUTE PARAMETERS

```
const router = createBrowserRouter([
  { path: '/', element: <HomePage /> },
  { path: '/users', element: <UserListPage /> },
  { path: '/users/:id', element: <UserDetailPage /> },
]);
```

```
// Get route parameters
const { id } = useParams();

// Get/update query string parameters
const [searchParams, setSearchParams] = useSearchParams();

// Get the current location
const location = useLocation();
```

HANDLING ERRORS

```
const router = createBrowserRouter([
  {
    path: '/',
    element: <HomePage />,
    errorElement: <ErrorPage />,
  },
]);
```

NESTED ROUTES

```
const router = createBrowserRouter([
  {
    path: '/',
    element: <HomePage />,
    children: [{
      path: 'users', element: <UserListPage />
    }],
  },
]);

const HomePage = () => {
  return (
    <>
      <header>Nav</header>
      <Outlet />
    </>
  );
};
```

LAYOUT ROUTES

```
const router = createBrowserRouter([
  {
    // No path
    element: <Layout />,
    children: []
  },
]);
```