# use of the map method

import React from 'react';

function App() {

  const items = [

    { id: 1, name: "Apple" },

    { id: 2, name: "Banana" },

    { id: 3, name: "Cherry" }

  ];

  return (

    <ul>

      {items.map(item => (

        <li key={item.id}>{item.name}</li>

      ))}

    </ul>

  );

}

export default App;

# Passing Data via Props Props in React

import React from 'react';

// Child component

function Header({ title }) {

  return <h1>{title}</h1>;

}

// Parent component

function App() {

  return (

    <div>

      <Header title="My App" />

    </div>

  );

}

export default App;

# Passing Components as Children

import React from 'react';

// Container component

function Container(props) {

  return (

    <div className="container-style">

      {props.children}  {/\* Accessing children prop \*/}

    </div>

  );

}

// App component

function App() {

  return (

    <Container>

      <h1>Welcome to My App</h1>

      <p>This content is passed as children to the Container component.</p>

    </Container>

  );

}

export default App;

# Handling Events

import React, { useState } from 'react';

function App() {

  const [count, setCount] = useState(0);

  // Event handler function

  const handleClick = () => {

    setCount(count + 1);

  };

  return (

    <div>

      <h1>Count: {count}</h1>

      <button onClick={handleClick}>Increment</button>

    </div>

  );

}

export default App;

# Passing functions via props

import React, { useState } from 'react';

// Child component

function Child({ onButtonClick }) {

  return (

    <button onClick={onButtonClick}>Click Me</button>

  );

}

// Parent component

function App() {

  const [message, setMessage] = useState("");

  // Function to handle the button click

  const handleClick = () => {

    setMessage("Button was clicked!");

  };

  return (

    <div>

      <h1>{message}</h1>

      <Child onButtonClick={handleClick} />

    </div>

  );

}

export default App;

# useState

import React, { useState } from 'react';

function Counter() {

  const [count, setCount] = useState(0);

  return (

    <div>

      <h1>Count: {count}</h1>

      <button onClick={() => setCount(count + 1)}>Increment</button>

    </div>

  );

}

export default Counter;

If the Counter component manages its own state (like in the previous example) and nothing in the homepage component depends on the state of the Counter, then only the Counter will re-render. The homepage itself will not re-render unless its own state or props change.

import React, { useState } from 'react';

function App1() {

  const [counter, setCounter] = useState(15); // Initial counter value set to 15

  const addValue = () => {

    setCounter(counter + 1);

    setCounter(counter + 1);

    setCounter(counter + 1);

    setCounter(counter + 1);

  };

  FIXME: OUTPUT=16

  return (

    <div>

      <p>Counter: {counter}</p>

      <button onClick={addValue}>Increment</button>

    </div>

  );

}

export default App1;

import React, { useState } from 'react';

function App2() {

  const [counter, setCounter] = useState(15); // Initial value set to 15

  const addValue = () => {

    setCounter(prevCounter => prevCounter + 1);

    setCounter(prevCounter => prevCounter + 1);

    setCounter(prevCounter => prevCounter + 1);

    setCounter(prevCounter => prevCounter + 1);

  };

  FIXME: OUTPUT=19

  return (

    <div>

      <p>Counter: {counter}</p>

      <button onClick={addValue}>Increment by 4</button>

    </div>

  );

}

module.exports = {

    app1,

    app2

}

import React from 'react';

import Counter from './Counter';

function HomePage() {

  return (

    <div>

      <h1>Welcome to My Homepage</h1>

      <Counter /> {/\* This will re-render independently \*/}

    </div>

  );

}

export default HomePage;

If the Counter is managing its state but the homepage has some dependency on that state (for example, if the homepage passes props to Counter based on its state), then a state change in Counter could cause the homepage to re-render as well.

import React, { useState } from 'react';

import Counter from './Counter';

function HomePage() {

  const [message, setMessage] = useState("Welcome!");

  return (

    <div>

      <h1>{message}</h1>

      <Counter onCounterChange={() => setMessage("Counter updated!")} />

    </div>

  );

}

export default HomePage;

# React Form

In React, when you update the state using the setState function (like setEmail or setName), the component will indeed re-render. However, the values stored in the state (like name and email) do not get lost during this process.

Each piece of state in a functional component is independent. When you call setEmail(newEmail), React updates only the email state. The name state remains unchanged unless you explicitly call setName to change it.

Button in from field is by default is submit type, onclicking button it submits form, if you do not write handleSubmit form by default send collected data to currect server.

So write own handleSubmit and put preventDefault() to stop mentioned default behaviour

import React, { useState } from 'react';

function App() {

  // State management for form inputs

  const [name, setName] = useState('');

  const [email, setEmail] = useState('');

  const [error, setError] = useState('');

  const handleNameChange = (e) => {

    setName(e.target.value);

  };

  const handleEmailChange = (e) => {

    setEmail(e.target.value);

  };

  // Form submission

  const handleSubmit = (e) => {

    e.preventDefault(); // Prevents the default form submission behavior

    if (!name || !email) {

      setError('Both fields are required');

      return;

    }

    setError('');

    console.log('Form submitted with:', { name, email });

    setName('');

    setEmail('');

  };

  return (

    <div>

      <h1>Simple Form Example</h1>

      <form onSubmit={handleSubmit}>

        <div>

          <label>

            Name:

            <input type="text" value={name} onChange={handleNameChange} />

          </label>

        </div>

        <div>

          <label>

            Email:

            <input type="email" value={email} onChange={handleEmailChange} />

          </label>

        </div>

        {error && <p style={{ color: 'red' }}>{error}</p>}

        <button type="submit">Submit</button>

      </form>

    </div>

  );

}

export default App;

# Use of useRef

1. **Access to DOM Elements**: useRef allows you to directly interact with DOM elements by creating a reference that can be attached to the ref attribute.
2. **Retains Mutable Values**: It can hold mutable values (like previous state or prop values) without causing re-renders when those values change.
3. **Non-DOM Values**: useRef is not limited to DOM references; it can hold any type of value, making it useful for storing values that persist between renders.
4. **Passing as Props**: Refs can be passed as props to child components, enabling direct DOM manipulation or access to mutable values from parent components.

import React, { useRef } from 'react';

function App() {

  const inputRef = useRef(null); // Create a ref

  const focusInput = () => {

    inputRef.current.focus(); // Directly focus the input element

  };

  return (

    <div>

      <input ref={inputRef} type="text" placeholder="Click button to focus" />

      <button onClick={focusInput}>Focus Input</button>

    </div>

  );

}

export default App;

Spread operator  
  
const arr1 = [1, 2, 3];

const arr2 = [arr1, 4, 5]; // Result: arr2 = [[1, 2, 3], 4, 5];

const arr2 = [...arr1, 4, 5]; // Result: arr2 = [1, 2, 3, 4, 5];

setItems((prevItems) => [...prevItems, newItem]);

# Context Api

/context/TodoItemContext.jsx

import React, { createContext, useContext, useState } from 'react';

export const TodoItemContext = createContext();

export const useTodo = () => useContext(TodoItemContext);

export const TodoItemProvider = ({ children }) => {

  const [todo, setTodo] = useState([]);

  const addTodo = (item) => {

    setTodo((prev) => [...prev, item]);

  };

  const removeTodo = (itemToDel) => {

    setTodo((prev) => prev.filter((item) => item !== itemToDel));

  };

  return (

    <TodoItemContext.Provider value={{ todo, addTodo, removeTodo }}>

      {children}

    </TodoItemContext.Provider>

  );

};

/Todo.jsx

import React, { useContext, useState } from 'react';

import {TodoItemContext, useTodo} from './context/TodoItemContext';

const Todo = () => {

    const { todo, addTodo, removeTodo } = useTodo();

    const [newItem, setnewItem] = useState("");

    return (

        <div>

            <input type="text" placeholder='Enter Item to add' value={newItem} onChange={(e) => setnewItem(e.target.value)} />

            <button onClick={()=>{addTodo(newItem)}}>add</button>

            <ul>

                {

                    todo.map((item, index) => {

                        return <li key={index}>{item} <button onClick={()=>{removeTodo(item)}}>delete</button></li>

                    })

                }

            </ul>

        </div>

    )

}

export default Todo

App.jsx

import React from 'react';

import { TodoItemProvider } from './context/TodoItemContext';

import Todo from './Todo';

function App() {

  return (

    <TodoItemProvider>

      <Todo />

    </TodoItemProvider>

  );

}

export default App;

# useReducer

import React, { useReducer } from 'react';

import { cartReducer, initialState } from './cartReducer';

function ShoppingCart() {

  const [state, dispatch] = useReducer(cartReducer, initialState);

  const addItem = (item) => {

    dispatch({ type: 'add', payload: item });

  };

  const removeItem = (item) => {

    dispatch({ type: 'remove', payload: item });

  };

  return (

    <div>

      <h1>Shopping Cart</h1>

      <p>Total Items: {state.totalItems}</p>

      <button onClick={() => addItem({ id: 1, name: 'Apple' })}>Add Apple</button>

      <button onClick={() => addItem({ id: 2, name: 'Banana' })}>Add Banana</button>

      <h2>Items in Cart:</h2>

      <ul>

        {state.items.map(item => (

          <li key={item.id}>

            {item.name}

            <button onClick={() => removeItem({ id: item.id })}>Remove</button>

          </li>

        ))}

      </ul>

    </div>

  );

}

export default ShoppingCart;

// cartReducer.js

export const initialState = {

    items: [],

    totalItems: 0,

  };

  export function cartReducer(state, action) {

    if (action.type === 'add') {

      return {

        ...state,

        items: [...state.items, action.payload],

        totalItems: state.totalItems + 1,

      };

    } else if (action.type === 'remove') {

      const updatedItems = state.items.filter(item => item.id !== action.payload.id);

      return {

        ...state,

        items: updatedItems,

        totalItems: updatedItems.length,

      };

    } else {

      return state;

    }

  }

# Some doubts

 return {

        ...state,

        items: [...state.items, action.payload],

        totalItems: state.totalItems + 1,

      };

So the final returned object will **not** look like this:

{

    items: [], // This would be incorrect as it gets overridden

    totalItems: 0, // This would be incorrect as it also gets overridden

    items: [...state.items, action.payload], // This will override the previous items

    totalItems: state.totalItems + 1, // This will override the previous totalItems

  };

The final returned state would look like this after an item is added:

{

  items: [...], // This includes all previous items plus the new one

  totalItems: 1 // Incremented value based on the previous totalItems

}

You can indeed use the following return statement directly:

return {

    items: [...state.items, action.payload],

    totalItems: state.totalItems + 1,

  };

By using ...state, you ensure that any additional properties in the state object are preserved. This is particularly important if your state object evolves over time. For example, if you later add more properties to the state, such as totalPrice or discount, these properties would be retained without requiring changes in your reducer.

Using items: [...state.items, action.payload] instead of items: [state.items, action.payload] is important

For example, if state.items is ['Apple', 'Banana'] and action.payload is 'Orange', using the spread operator will create:

items: ['Apple', 'Banana', 'Orange']

If you write items: [state.items, action.payload], it creates a new array with two elements:

items: [['Apple', 'Banana'], 'Orange']

# useEffect

import React, { useState, useEffect } from 'react';

function App() {

  const [data, setData] = useState([]);

  useEffect(() => {

    const fetchData = async () => {

      const response = await fetch('https://jsonplaceholder.typicode.com/posts');

      const result = await response.json();

      setData(result);

    };

    fetchData();

  }, []);

  return (

    <div>

      <h1>Posts</h1>

        <ul>

          {data.map(post => (

            <li key={post.id}>{post.title}</li>

          ))}

        </ul>

    </div>

  );

}

export default App;

useEffect(() => {

  console.log('This runs only once on mount');

}, []); // Runs once after initial render

useEffect(() => {

  console.log('This runs after every render');

}); // Runs after every render (not recommended for performance reasons)

useEffect(() => {

  console.log('Count has changed:', count);

}, [count]); // Runs after initial render and whenever `count` changes

# The useEffect Hook Cleanup

**Purpose of Cleanup**: The cleanup function in useEffect is crucial for managing side effects in React components, particularly for actions that require cleanup when the component unmounts or before the effect runs again. This is essential for preventing memory leaks and ensuring proper resource management (like clearing timers or removing event listeners).

import React, { useEffect, useState } from 'react';

function TimerComponent() {

  const [count, setCount] = useState(0);

  useEffect(() => {

    const timerID = setInterval(() => {

      setCount(prevCount => prevCount + 1); // Increment the count every second

    }, 1000); // 1000 ms = 1 second

    // Cleanup function to clear the interval

    return () => {

      clearInterval(timerID);

    };

  }, []); // Empty dependency array to run the effect only once on mount

  return <div>Count: {count}</div>;

}

export default TimerComponent;

# useCallback

If ParentComponent re-renders due to the count state changing, the ChildComponent will not re-render because the reference to handleButtonClick remains the same, preventing unnecessary rendering.

The useCallback hook in React is used to memoize functions, meaning it preserves the function across re-renders.

It ensures that child components receive the same function reference

import React, { useState, useCallback } from 'react';

function ChildComponent({ onButtonClick }) {

  console.log('Child component rendered');

  return <button onClick={onButtonClick}>Click Me</button>;

}

function ParentComponent() {

  const [count, setCount] = useState(0);

  // Memoizing the callback function with useCallback

  const handleButtonClick = useCallback(() => {

    console.log('Button clicked!');

  }, []); // This function will not change between renders

  console.log('Parent component rendered');

  return (

    <div>

      <h1>Count: {count}</h1>

      <button onClick={() => setCount(count + 1)}>Increment Count</button>

      <ChildComponent onButtonClick={handleButtonClick} />

    </div>

  );

}

export default ParentComponent;

# useMemo

useCallback memorizes functions, useMemo memorizes values

useMemo caches the result of expensive calculations so that the same computation is not repeated on every render.

import React, { useState, useMemo } from 'react';

function ExpensiveCalculationComponent() {

  const [count, setCount] = useState(0);

  // An expensive calculation that only runs when `count` changes

  const expensiveCalculation = useMemo(() => {

    console.log('Calculating expensive value...');

    let total = 0;

    for (let i = 0; i < 1000000000; i++) {

      total += i; // Simulating an expensive calculation

    }

    console.log('Completed')

    return total;

  }, [count]); // Only recalculates when `count` changes

  return (

    <div>

      <h1>Count: {count}</h1>

      <button onClick={() => setCount(count + 1)}>Increment Count</button>

      <p>Expensive Calculation Result: {expensiveCalculation}</p>

    </div>

  );

}

export default ExpensiveCalculationComponent;

# Outlet

Outlet is a component that serves as a placeholder for rendering child routes. It allows you to define a nested routing structure where certain components can render based on the current route.

import React from 'react';

import {

  BrowserRouter as Router,

  Routes,

  Route,

  Link,

  Outlet

} from 'react-router-dom';

// Main Layout

function Layout() {

  return (

    <div>

      <h1>My Website</h1>

      <nav>

        <Link to="/">Home</Link> | <Link to="/about">About</Link> | <Link to="/contact">Contact</Link>

      </nav>

      <hr />

      <Outlet /> {/\* Renders the matched child route \*/}

    </div>

  );

}

// Home Component

function Home() {

  return <h2>Home Page</h2>;

}

// About Component

function About() {

  return <h2>About Page</h2>;

}

// Contact Component

function Contact() {

  return <h2>Contact Page</h2>;

}

// App Component

function App() {

  return (

    <Router>

      <Routes>

        <Route path="/" element={<Layout />}>

          <Route index element={<Home />} /> {/\* Default child route \*/}

          <Route path="about" element={<About />} />

          <Route path="contact" element={<Contact />} />

        </Route>

      </Routes>

    </Router>

  );

}

export default App;

# React Query Overview

import { useQuery, useMutation, useQueryClient } from "@tanstack/react-query";

function App() {

  // Setting up a query to fetch data

  const { data, error, isLoading } = useQuery({

    queryKey: ["todo"],

    queryFn: () =>

      fetch("https://jsonplaceholder.typicode.com/posts").then((res) =>

        res.json()

      ),

  });

  // Access the query client to manage cache

  const queryClient = useQueryClient();

  // Mutation setup for posting data

  const { mutate, isPending, isError, isSuccess } = useMutation({

    mutationFn: (newPost) =>

      fetch("https://jsonplaceholder.typicode.com/posts", {

        method: "POST",

        body: JSON.stringify(newPost),

        headers: { "Content-type": "application/json; charset=UTF-8" },

      }).then((res) => res.json()),

    onSuccess: () => {

      queryClient.invalidateQueries(["todo"]); // Refreshes data on success

    },

  });

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

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

  return (

    <div>

      <h1>Posts</h1>

      <ul>

        {data?.map((post) => (

          <li key={post.id}>{post.title}</li>

        ))}

      </ul>

      <button onClick={() => mutate({ title: "New Post" })}>Add Post</button>

      {isPending && <p>Adding post...</p>}

      {isSuccess && <p>Post added successfully!</p>}

      {isError && <p>Error adding post.</p>}

    </div>

  );

}

export default App;

It's a library that helps you manage the state of data you fetch from servers, like APIs, in your React applications.  
*One of the most powerful tools for managing server-side state in React.*

**Advantages**

* **Data Fetching Made Easy**: With a simple useQuery hook, fetching data becomes super easy.
* **Built-in Loading and Error States**: No need to write custom code for handling loading, errors, or success states.
* **Automatic Caching**: React Query automatically caches your data.
* **Background Refetching**: If your data gets stale or out of date, TanStack Query can refetch it in the background.
* **Pagination and Infinite Scrolling**: Handling pagination or infinite scrolling? React Query has tools specifically designed for these complex use cases.

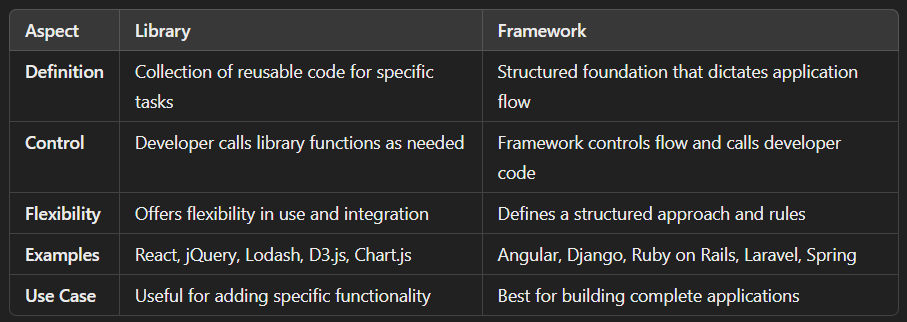
**Core Functions**

* **refetch**: Re-fetches data for a specific query, typically triggered manually to get the latest data (e.g., refetch() in useQuery).
* **invalidateQueries**: Marks queries as "stale" so they automatically refetch next time they're accessed. Useful after data changes (e.g., queryClient.invalidateQueries("todo")).
* **resetQueries**: Resets query state (like loading and error states) to its initial state without refetching data (e.g., queryClient.resetQueries("todo")).
* **removeQueries**: Deletes a query from the cache entirely, clearing its data and state (e.g., queryClient.removeQueries("todo")).

**Configuration Options**

* **staleTime**: Controls how long data is served as "fresh" from the cache without re-fetching.
* **cacheTime**: Controls how long the query data stays in the cache after no longer being used.

**Example Scenario**  
If staleTime is set to 5 minutes, data will be served from the cache for up to 5 minutes without re-fetching.  
If cacheTime is set to 10 minutes, data will stay in the cache for 10 minutes after it’s no longer in use, after which it will be removed from memory.

**ReactJS**

* A popular JavaScript library for building user interfaces, especially single-page applications (SPAs).
* Uses a component-based architecture, where UI elements are created as reusable components.
* Allows developers to manage state, handle events, and render efficiently using a virtual DOM.
* Ideal for creating dynamic, responsive UIs in applications like dashboards or social platforms.

**Next.js**

* A React-based framework that adds features like server-side rendering (SSR), static site generation (SSG), and routing.
* Optimized for performance and SEO, making it a good choice for building websites or apps that require fast loading and better search engine indexing.
* Provides a complete setup for creating full-stack applications with backend API routes alongside frontend React components.

**Key Difference**: ReactJS is a UI library, while Next.js is a framework built on top of React to create more complex, optimized, full-stack applications.

**React Router**

React Router is a standard library for creating dynamic routes and navigation in [React JS](https://www.geeksforgeeks.org/react-tutorial/)

**BrowserRouter** is a component and it uses the HTML5 history API to keep the UI in sync with the URL

<BrowserRouter>

    (/\* Your routes go here \*/}

</BrowserRouter>

<Routes>

    <Route path="/" element={<Home />} />

    <Route path="/about" element={<About />} />

</Routes>

**What are hooks? Why we use them?**

Hooks in React are functions that allow you to use state, lifecycle methods, and other React features in functional components. They simplify code, enhance reusability, and enable managing side effects, context, and more, without needing class components.

In short, hooks make functional components more powerful and reduce the need for class-based components.

**UseState Hook**

useState is a React hook that lets you add state to functional components. It returns a state variable and a function to update that state, making it possible to manage state in simpler, functional components.

How it is better than class components

* **Simpler Syntax**: useState provides a concise, functional approach to managing state, eliminating the need for this.state and this.setState found in class components.
* **Cleaner Code**: With useState, each piece of state is managed individually, making the code more modular and readable compared to managing all state within a single state object in class components.

const [stateVariable, setStateVariable] = useState(initialValue);

useState(initialValue): Initializes the state with initialValue.

[stateVariable, setStateVariable]: The first element is the state variable, and the second is the function used to update the state.

**How to update state based on previous value**?

const increment = () => { setCount(prevCount => prevCount + 1);

**UseEffect Hook**

The useEffect hook in React is used to handle **side effects** in functional components. It allows you to run code after the component has rendered, making it useful for tasks like data fetching, subscriptions, timers, or manually updating the DOM.

useEffect(() => {

    // Side effect logic goes here

    return () => {

        // Cleanup logic (optional)

    };

}, [dependencies]);

* **Effect function**: This is where your side effect code runs.
* **Cleanup function**: This optional return function cleans up side effects like subscriptions or timers when the component unmounts.
* **Dependencies array**: React re-runs the effect if any of the values in this array change.

FUNCTION: contains the code to be executed when useEffect triggers.

DEPENDENCY: is an optional parameter, useEffect triggers when the given dependency is changed.

**Controlling side effects in useEffect :**

1. To run useEffect on every render do not pass any dependency

useEffect(()->{

// Example Code

})

2. To run useEffect only once on the first render pass any empty array in the dependecy

useEffect(()->{

// Example Code

}, [] )

3. To run useEffect on change of a particular value. Pass the state and props in the dependency array

useEffect(()->{

// Example Code

}, [props, state] )

**For componentDidMount**

useEffect(() => {

    //You can add your code here for mounting phase of component

    console.log("Mounting in Functional Component")

}, [])

// adding an empty array ensures that the useEffect is only triggered once

// (when the component mounts)

**For componentDidUpdate**

useEffect(() => {

    //You can add your code for updating phase of component

    console.log("Updating in Functional Component")

}, [values])

//values triggers re render whenever they are updated in your program,

//you can add multiple values by separating them by commas

**For componentWillUnmount**

**useEffect(() => {**

    return () => {

        //You can add your code for unmounting phase of component

        console.log("Functional Component Removed ")

    }

}, [])

//Write all the code of unmounting phase only inside the callback function

**useQuery Hook**

The useQuery hook is part of **React Query** (a library for data fetching and state management), used for fetching data in functional components. It simplifies data fetching, caching, synchronization, and state management in React.

**Key Features** are Data Fetching, Automatic Refetching, Error Handling:

const { data, error, isLoading, isError } = useQuery('queryKey', fetchFunction);

* **queryKey**: A unique key that identifies the query, used for caching and refetching.
* **fetchFunction**: A function that returns the data (e.g., an API call).
* **data**: The fetched data.
* **isLoading**: A boolean indicating if the data is still loading.
* **isError**: A boolean indicating if there was an error in fetching.

**Example:** const { data, isLoading, error } = useQuery('posts', fetchPosts);

**useMutation Hook**

The useMutation hook is used for **creating, updating, or deleting data**. It is typically used when interacting with APIs for actions that change data (e.g., POST, PUT, DELETE requests).

**Key Features** are Mutate Data, Error Handling, OnSuccess and OnError

const mutation = useMutation(mutationFunction, {

onSuccess: (data) => { /\* handle success \*/ },

onError: (error) => { /\* handle error \*/ },

onSettled: () => { /\* handle both success and error \*/ }

});

* **mutationFunction**: A function that performs the mutation (e.g., a POST or DELETE request).
* **onSuccess**: A callback that runs if the mutation succeeds.
* **onError**: A callback that runs if the mutation fails.
* **onSettled**: A callback that runs when the mutation either succeeds or fails.

**Example:**

const mutation = useMutation(

    (newUser) => axios.post('/api/users', newUser), // mutation function to POST data,

    onSuccess: (data) => {

        console.log('Data successfully posted:', data);

    },

    onError: (error) => {

        console.error('Error posting data:', error);

    }

});

const handlePost = () => {

    mutation.mutate({ title: 'New Post', body: 'Post content' });

};

**mutate**: Callback-based and doesn't return a Promise. Best for quick operations where you rely on onSuccess and onError.

**mutateAsync**: Promise-based, allowing for more control with async/await. Ideal for complex or sequential operations.

**useQuery**: Used for fetching data. Provides automatic caching, refetching, and state management.

**useMutation**: Used for sending requests that modify data (create, update, delete).

**queryClient**: Provides control over queries (e.g., invalidating, updating, or refetching queries programmatically).

**useLocation**

Hook to get the current URL location object, including pathname, search, and hash.

const location = useLocation();

location.pathname

**useNavigate**

Hook to programmatically navigate to different routes.

const navigate = useNavigate();

navigate('/home');

**UseContext Hook**

The useContext hook is a built-in React hook that allows a functional component to consume values from a context directly. It eliminates the need for passing props through every level of the component tree, making it easier to share data globally across components.

Comparison with Redux:

* useContext: Simpler, best for small to medium apps with basic shared state.
* Redux: More complex, suited for large apps with complex state management and advanced features.

**Creating the Context**:

First, we'll create a context called UserContext.

const UserContext = createContext();

**Creating a Provider Component**:

Next, we'll create a provider component that will supply the context value to its children.

const UserProvider = ({ children }) => {

    const [user, setUser] = useState({ name: 'John Doe', age: 30 });

    return (

        <UserContext.Provider value={{ user, setUser }}>

            {children}

        </UserContext.Provider>

    );

};

**Consuming the Context in a Component**:

Then, we'll create a component that consumes the context value.

const { user } = useContext(UserContext);

**Using the Context in the App Component**:

const App = () => {

    return (

        <UserProvider>

            <UserProfile />

            <UpdateUser />

        </UserProvider>

    );

};

export default App;

* UserProvider is a component that provides the context value (user information and a function to update it) to its children.
* The App component wraps its children with UserProvider, ensuring that all components within App have access to the context.

**useMemo Hook**

useMemo is a React hook that memorizes the result of an expensive computation and recomputes it only when one of its dependencies changes. It helps in optimizing performance by preventing unnecessary recalculations.

const Component = ({ items }) => {

    const sortedItems = useMemo(() => {

        return items.sort(); // expensive computation

    }, [items]); // Recalculates only when 'items' changes

    return <ul>{sortedItems.map(item => <li key={item}>{item}</li>)}</ul>;

};

A **toast** is a small notification that appears briefly on the screen to inform the user about something, like a success or error message

**Axios**: Axios is a popular JavaScript library for making HTTP requests. It simplifies the process of sending asynchronous HTTP requests to REST endpoints and handling responses. It works in both client-side (browser) and server-side (Node.js) environments.

const axios = require('axios');

// Simple POST request with Axios

axios.post('https://jsonplaceholder.typicode.com/posts', {

    title: 'foo',

    body: 'bar',

    userId: 1

})

    .then(response => {

        console.log(response.data);

    })

    .catch(error => {

        console.error('Error:', error);

    });

**Fetch**: fetch() is a built-in JavaScript function for making HTTP requests, introduced with the Fetch API. It’s promise-based, which makes it easier to handle asynchronous requests, and it works directly in the browser without any additional dependencies.

// Simple POST request with Fetch

fetch('https://jsonplaceholder.typicode.com/posts', {

    method: 'POST',

    headers: {

        'Content-Type': 'application/json'

    },

    body: JSON.stringify({

        title: 'foo',

        body: 'bar',

        userId: 1

    })

})

    .then(response => response.json())

    .then(data => {

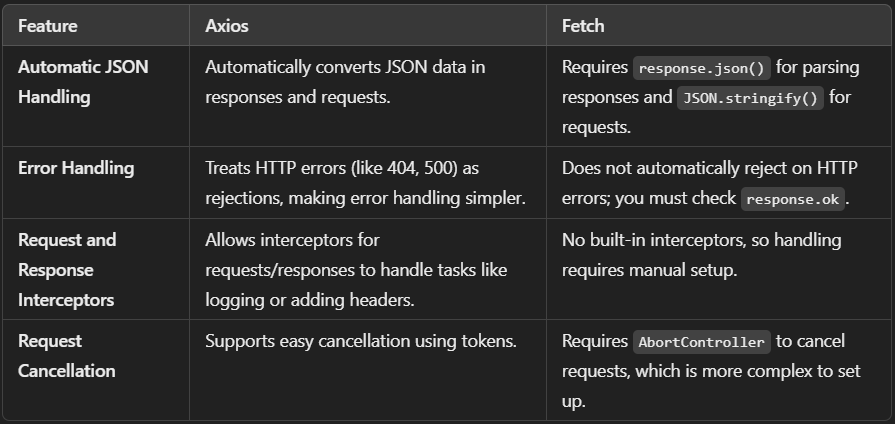
        console.log(data);

    })

    .catch(error => {

        console.error('Error:', error);

    });



**UseDebounce (Custom Hook)**

The useDebounce hook in React delays the update of a value until a specified period has passed without changes. It’s useful for limiting the frequency of actions like API calls during user input, as it only triggers after the user stops typing for the defined delay, reducing load and improving performance.

Step 1: Implement useDebounce Hook

import { useState, useEffect } from 'react';

function useDebounce(value, delay) {

  const [debouncedValue, setDebouncedValue] = useState(value);

  useEffect(() => {

    // Set a timer to update the debounced value after the specified delay

    const handler = setTimeout(() => {

      setDebouncedValue(value);

    }, delay);

    // Cleanup function to clear the timeout if value changes before delay

    return () => {

      clearTimeout(handler);

    };

  }, [value, delay]);

  return debouncedValue;

}

export default useDebounce;

**Step 2: Using useDebounce in a Component**

Below is a component that uses the useDebounce hook to handle input changes. The debounced value is used to trigger an API call after the user stops typing for the specified delay.

import React, { useState, useEffect } from 'react';

import useDebounce from './useDebounce';

function SearchComponent() {

    const [searchTerm, setSearchTerm] = useState('');

    const debouncedSearchTerm = useDebounce(searchTerm, 500); // 500ms delay

    useEffect(() => {

        // API call only when debouncedSearchTerm updates

        if (debouncedSearchTerm) {

            console.log(`Searching for: ${debouncedSearchTerm}`);

         // Fetch or process data here (fetchSearchResults(debouncedSearchTerm))

        }

    }, [debouncedSearchTerm]);

    return (

        <div>

            <input

                type="text"

                placeholder="Search..."

                value={searchTerm}

                onChange={(e) => setSearchTerm(e.target.value)}

            />

        </div>

    );

}

export default SearchComponent;

**Nodejs**

Node.js is a runtime environment that allows developers to run JavaScript on the server side, outside of a browser. Built on Chrome's V8 JavaScript engine, it enables building fast and scalable applications.

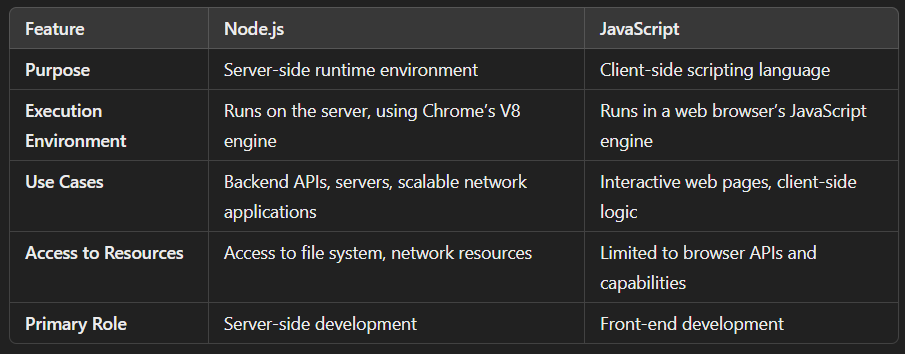
**1.**[**What is Node.js?**](https://www.geeksforgeeks.org/introduction-to-node-js)

[Node.js](https://www.geeksforgeeks.org/node-js-introduction) is a JavaScript engine used for executing JavaScript code outside the browser. It is normally used to build the backend of the application and is highly scalable.

**2.**[**What is the difference between Node.js and JavaScript?**](https://www.geeksforgeeks.org/difference-between-node-js-and-javascript)

JavaScript is a scripting language whereas Node.js is an engine that provides the runtime environment to run JavaScript code.

Here we have [difference table between Node.js and JavaScript](https://www.geeksforgeeks.org/difference-between-node-js-and-javascript)



**3. Is Node.js single-threaded?**

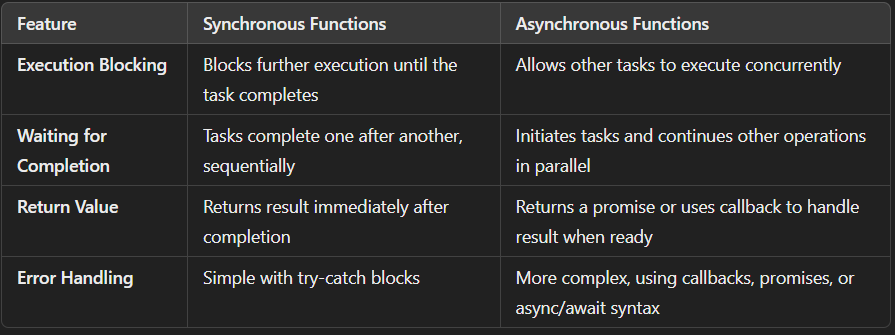
Yes,[Node.js is single-threaded](https://www.geeksforgeeks.org/why-node-js-is-a-single-threaded-language) by default. However, it utilizes event-driven architecture and non-blocking I/O operations to handle multiple concurrent requests efficiently, enabling scalability and high performance in applications.

**4. What kind of API function is supported by Node.js?**

There are two types of [API functions](https://www.geeksforgeeks.org/types-of-api-functions-in-node-js) supported by Node.js:

* **Synchronous:** These API functions are used for blocking code.
* **Asynchronous:** These API functions are used for non-blocking code.

**5. What is the difference between Synchronous and Asynchronous functions?**



**3. For Node.js, why Google uses V8 engine?**

Well, are there any other options available? Yes, of course, we have [Spidermonkey](https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey" \t "_blank) from Firefox, Chakra from Edge but Google’s v8 is the most evolved(since it’s open-source so there’s a huge community helping in developing features and fixing bugs) and fastest(since it’s written in c++) we got till now as a JavaScript and WebAssembly engine. And it is portable to almost every machine known.

**6. What is a module in Node.js?**

In Node.js Application, a [Module](https://www.geeksforgeeks.org/what-are-modules-in-node-js) can be considered as a block of code that provide a simple or complex functionality that can communicate with external application. Modules can be organized in a single file or a collection of multiple files/folders. Modules are useful because of their reusability and ability to reduce the complexity of code into smaller pieces. Some examples of modules are. http, fs, os, path, etc.

**7. What is npm and its advantages?**

npm (Node Package Manager) is the default package manager for Node.js. It allows developers to discover, share, and reuse code packages easily. Its advantages include dependency management, version control, centralized repository, and seamless integration with Node.js projects.

**8. What is middleware?**

[Middleware](https://www.geeksforgeeks.org/middleware-in-express-js) is the function that works between the request and the response cycle. Middleware gets executed after the server receives the request and before the controller sends the response.

**9. How does Node.js handle concurrency even after being single-threaded?**

[Node.js handles concurrency](https://www.geeksforgeeks.org/if-node-js-is-single-threaded-then-how-to-handles-concurrency) by using asynchronous, non-blocking operations. Instead of waiting for one task to complete before starting the next, it can initiate multiple tasks and continue processing while waiting for them to finish, all within a single thread.

**10. What is control flow in Node.js?**

Control flow in Node.js refers to the sequence in which statements and functions are executed. It manages the order of execution, handling asynchronous operations, callbacks, and error handling to ensure smooth program flow.

**11. What do you mean by event loop in Node.js?**

The[event loop](https://www.geeksforgeeks.org/node-js-event-loop) in Node.js is a mechanism that allows it to handle multiple asynchronous tasks concurrently within a single thread. It continuously listens for events and executes associated callback functions.

**12. What is the order in which control flow statements get executed?**

The order in which the statements are executed is as follows:

* Execution and queue handling
* Collection of data and storing it
* Handling concurrency
* Executing the next lines of code

**13. What are the main disadvantages of Node.js?**

Here are some main [disadvantages of Node.js](https://www.geeksforgeeks.org/the-pros-and-cons-of-node-js-in-web-development) listed below:

* **Single-threaded nature:** May not fully utilize multi-core CPUs, limiting performance.
* **NoSQL preference:** Relational databases like MySQL aren’t commonly used.
* **Rapid API changes:**Frequent updates can introduce instability and compatibility issues.

**14. What is REPL in Node.js?**

[REPL](https://www.geeksforgeeks.org/node-js-repl-read-eval-print-loop)in Node.js stands for Read, Evaluate, Print, and Loop. It is a computer environment similar to the shell which is useful for writing and debugging code as it executes the code in on go.

**15.**[**How to import a module in Node.js?**](https://www.geeksforgeeks.org/import-and-export-in-node-js)

We use the require module to import the External libraries in Node.js. The result returned by require() is stored in a variable which is used to invoke the functions using the dot notation.

**16. What is the difference between Node.js and AJAX?**

Node.js is a JavaScript runtime environment that runs on the server side whereas AJAX is a client-side programming language that runs on the browser.

**17. What is package.json in Node.js?**

[package.json](https://www.geeksforgeeks.org/node-js-package-json) in Node.js is a metadata file that contains project-specific information such as dependencies, scripts, version, author details, and other configuration settings required for managing and building the project.

**18.**[**How to write hello world using node.js?**](https://www.geeksforgeeks.org/how-to-create-a-simple-server-in-node-js-that-display-hello-world)

JavaScript

**const** http = require('http');

*// Create a server object*

http.createServer(**function** (req, res) {

res.write('Hello World!');

res.end();

}).listen(3000);

Run this program from the command line and see the output in the browser window. This program prints Hello World on the browser when the browser sends a request through http://localhost:3000/.

**19. What is the most popular Node.js framework used these days?**

The most famous Node.js framework used is Express.js as it is highly scalable, efficient, and requires very few lines of code to create an application.

**Intermediate Node.js Interview Questions and Answers**

In this set we will be looking at intermediate Node Interview Question for candidates with over 2 years of experience.

**32.**[**What are the three methods to avoid callback hell?**](https://www.geeksforgeeks.org/how-to-avoid-callback-hell-in-node-js)

The three methods to avoid callback hell are:

* Using async/await()
* Using promises
* Using generators

**33.**[**What is body-parser in Node.js?**](https://www.geeksforgeeks.org/body-parser-middleware-in-node-js)

Body-parser is the Node.js body-parsing middleware. It is responsible for parsing the incoming request bodies in a middleware before you handle it. It is an NPM module that processes data sent in HTTP requests.

**34.**[**What is CORS in Node.js?**](https://www.geeksforgeeks.org/use-of-cors-in-node-js)

The word [CORS](https://www.geeksforgeeks.org/use-of-cors-in-node-js)stands for “Cross-Origin Resource Sharing”. Cross-Origin Resource Sharing is an HTTP-header based mechanism implemented by the browser which allows a server or an API to indicate any origins (different in terms of protocol, hostname, or port) other than its origin from which the unknown origin gets permission to access and load resources.

**7. What are node.js buffers?**

In general, buffers is a temporary memory that is mainly used by stream to hold on to some data until consumed. Buffers are introduced with additional use cases than JavaScript’s Unit8Array and are mainly used to represent a fixed-length sequence of bytes. This also supports legacy encodings like ASCII, utf-8, etc. It is a fixed(non-resizable) allocated memory outside the v8.

**8. What is node.js streams?**

Streams are instances of EventEmitter which can be used to work with streaming data in Node.js. They can be used for handling and manipulating streaming large files(videos, mp3, etc) over the network. They use buffers as their temporary storage.

**Nodejs Code**

const app = express();

Creates an instance of the Express app, app, which will handle requests, define routes, and manage middleware.

Use middleware:

**bodyParser.urlencoded({ extended: true })**:

* This is middleware that helps parse incoming request data when submitted through HTML forms (using the POST method).
* The { extended: true } option allows the parser to handle more complex data structures, like nested objects.

**app.use(bodyParser.urlencoded({ extended: true }));**:

* This line tells Express to use the urlencoded middleware for all incoming requests, so that it can parse data sent through form submissions.

**app.use(express.json());**

* This middleware enables Express to parse JSON-formatted data in the body of incoming requests.
* It’s often used for APIs, allowing data to be sent and received in JSON format, which is widely used for structured data exchange.

Start the server:

app.listen(PORT, () => {

console.log(`Server is running on port ${PORT}`);

});

PORT: Sets the server port, with a default of 3000 if an environment variable isn’t set.

app.listen(): Starts the server and listens for requests on the specified port. The callback logs the server’s startup.

**ExpressJs**

* Express.js is a fast, flexible, and minimalist web framework for Node.js, designed to simplify server-side development with JavaScript. Developed by the Node.js Foundation, it provides robust features for building web applications and APIs.
* Express makes it easy to organize functionality with middleware, manage routing, and handle dynamic HTTP responses, enhancing productivity and streamlining web development.

**Bcrypt**

* **Purpose**: Used to hash passwords securely.
* **How it Works**: Adds a "salt" to each password hash, making identical passwords unique and resistant to attacks.
* **Usage**: During registration, passwords are hashed with bcrypt before storing. For login, bcrypt verifies the hashed password against stored data.

**JWT (JSON Web Token)**

* **Purpose**: Used for stateless, token-based authentication.
* **How it Works**: Encodes user data into a token signed with a secret key. The token is sent to the client, which includes it in requests to prove identity.
* **Usage**: Upon login, the server creates a JWT. For each request, the server verifies the token to confirm user identity.

**Cookies Management**

* **Purpose**: Used to store session data or tokens on the client side.
* **How it Works**: Cookies can store the JWT securely with httpOnly and secure flags, which protect it from JavaScript access and ensure it’s sent only over HTTPS.
* **Usage**: JWT is stored in an HTTP-only cookie, which the client automatically sends with each request to protected routes.

const express = require('express');

const bcrypt = require('bcrypt');

const jwt = require('jsonwebtoken');

const cookieParser = require('cookie-parser');

const app = express();

app.use(express.json());

app.use(cookieParser());

const users = {}; // In-memory database (username: hashedPassword)

const secretKey = "yourSecretKey";

const saltRounds = 10;

// Register Route

app.post('/register', async (req, res) => {

  const { username, password } = req.body;

  try {

    const hashedPassword = await bcrypt.hash(password, saltRounds);

    users[username] = { password: hashedPassword };

    res.status(201).send("User registered successfully");

  } catch (error) {

    res.status(500).send("Error registering user");

  }

});

// Login Route

app.post('/login', async (req, res) => {

  const { username, password } = req.body;

  const user = users[username];

  if (!user) return res.status(400).send("User not found");

  try {

    const isMatch = await bcrypt.compare(password, user.password);

    if (!isMatch) return res.status(400).send("Invalid password");

    // Generate JWT token

    const token = jwt.sign({ username }, secretKey, { expiresIn: "1h" });

    // Set JWT in a secure, HTTP-only cookie

    res.cookie("authToken", token, { httpOnly: true, secure: true });

    res.send("Logged in successfully");

  } catch (error) {

    res.status(500).send("Error logging in");

  }

});

// Protected Route

app.get('/protected', (req, res) => {

  const token = req.cookies.authToken;

  if (!token) return res.status(401).send("Access denied");

  try {

    const verifiedUser = jwt.verify(token, secretKey);

    res.send(`Access granted to ${verifiedUser.username}`);

  } catch (error) {

    res.status(400).send("Invalid token");

  }

});

app.listen(3000, () => console.log("Server running on port 3000"));

MongoDB is a **NoSQL, document-based database** that stores data in JSON-like documents, making it flexible and easy to work with for modern applications. It’s designed to handle complex, evolving data and can scale horizontally across multiple servers, supporting large data volumes and high traffic.

**Key Features:**

* **Schema Flexibility**: No fixed structure; fields can vary across documents.
* **Horizontal Scalability**: Distributes data across multiple servers for performance.
* **Rich Query Language**: Supports advanced querying, filtering, and aggregation.

const mongoose = require('mongoose');

const mongoURI = "mongodb://localhost:27017/mydatabase";

mongoose.connect(mongoURI, {

    useNewUrlParser: true,

    useUnifiedTopology: true,

}).then(() => {

    console.log("MongoDB connected");

}).catch(err => console.error("Connection error", err));

const userSchema = new mongoose.Schema({

    name: { type: String, required: true },

    email: { type: String, required: true, unique: true },

    age: Number,

    createdAt: { type: Date, default: Date.now },

});

const User = mongoose.model('User', userSchema);

const createUser = async () => {

    const newUser = new User({

        name: "John Doe",

        email: "john@example.com",

        age: 30,

    });

    await newUser.save();

    console.log("User created:", newUser);

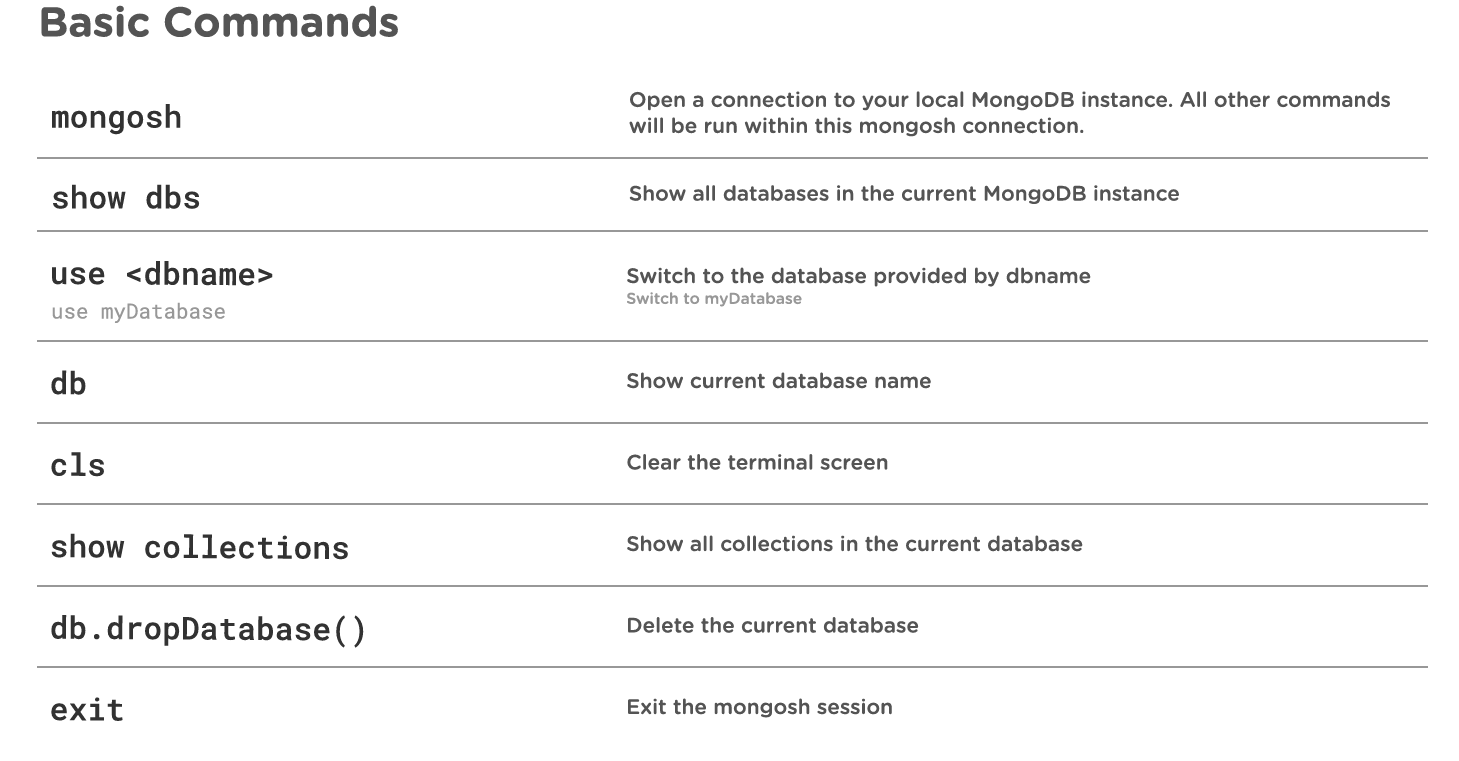
};

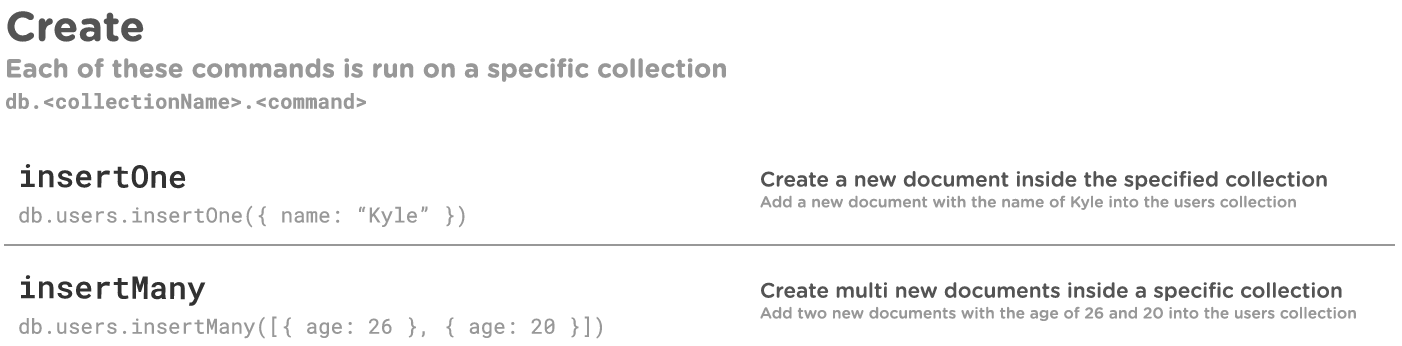
const getUsers = async () => {

    const users = await User.find();

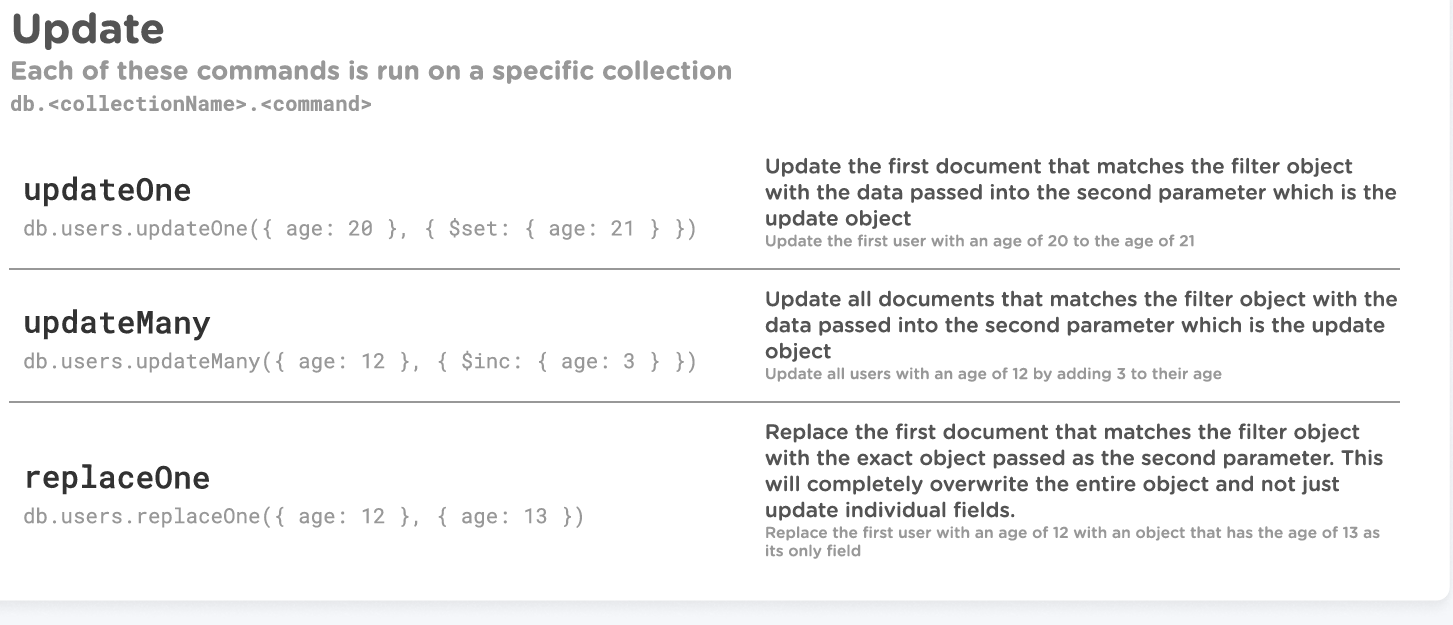
    console.log("All Users:", users);

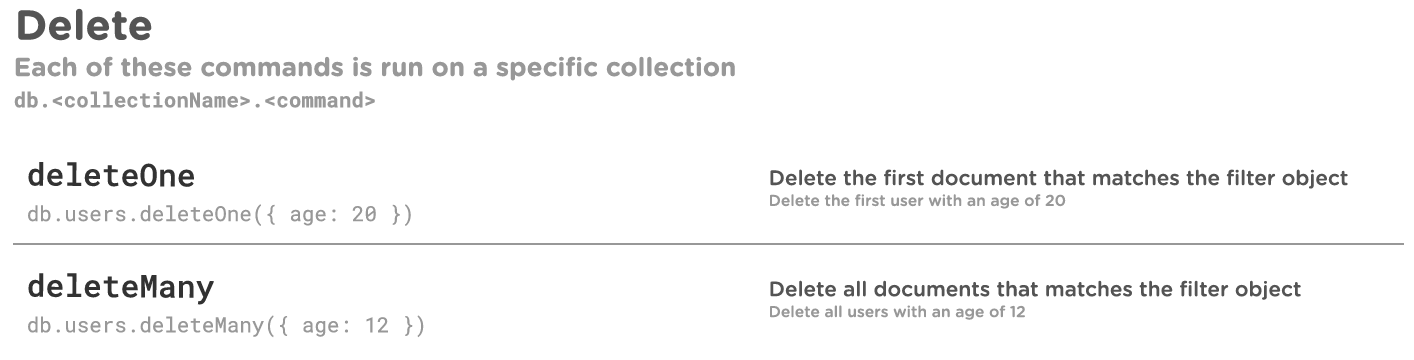
};

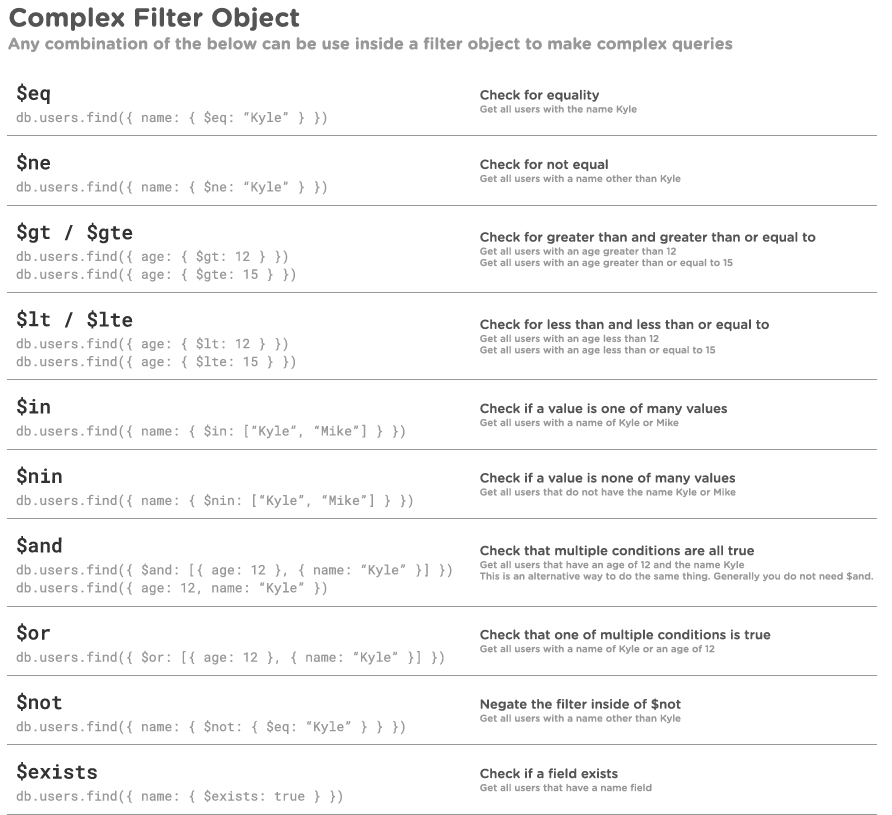


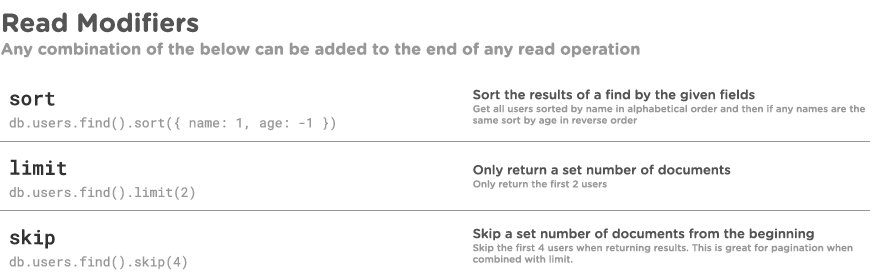












**HTTP status code**s are responses given by a server to a client's request on the web, indicating the result of the request.

