Karyvio

LEARN | BUILD | GROW | GET HIRED

Better Developers Brighter Futures ☺

Complete MERN Stack

Handwritten Study Guide

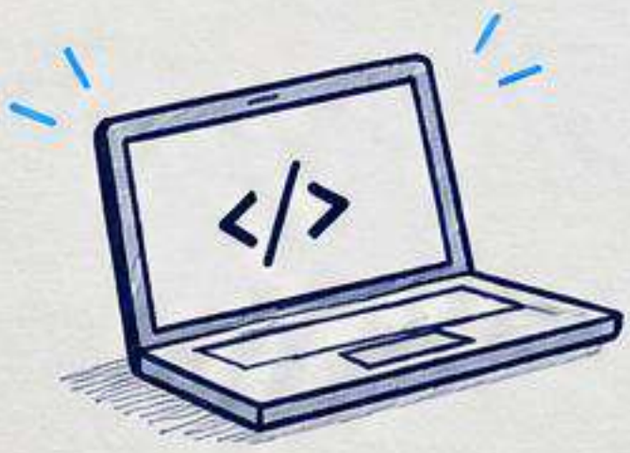
From Basics to Production

MongoDB | Express.js | React.js | Node.js

- ☑ Concepts
- ☑ Code Examples
- ☑ Real-World Projects
- ☑ Interview Questions
- ☑ Cheat Sheets
- ☑ Deployment Guide

"Learn Today Build Tomorrow" ☺

Code
Practice
Get Hired



Karyvio.com

Your Learning Partner in Tech Journey

Small Steps Big Growth

Karyvio
LEARN | BUILD | GROW | GET HIRED

MERN Stack Complete Notes

Beginner to Advanced

A Complete Guide to Build Modern Web Applications

Same Technology Different Dreams Keep Learning :)

1 What is MERN Stack ?

Ans →

- MERN stack is a combination of four powerful technologies used to build modern, full-stack web applications.
- It stands for MongoDB, Express.js, React.js and Node.js.
- It allows you to build both frontend and backend using JavaScript.
- MERN is popular because it is fast, flexible, scalable and has a large community.

MERN Technologies

MongoDB	Express.js	React.js	Node.js
Database (NoSQL)	Backend Framework	Frontend Library	JavaScript Runtime

2 Why Use MERN Stack ?

Ans →

- Uses a single language (JavaScript) for both frontend and backend.
- Large community support.
- Open source and free to use.
- Fast development with modern tools.
- Highly scalable and flexible.
- Great for real-world applications (startups, companies, products).

Key Features :-

- Full-stack with JavaScript
- Component based UI (React)
- RESTful API with Express)
- NoSQL database (MongoDB)
- Fast and developer friendly
- Large ecosystem (npm)
- Highly scalable
- Suitable for modern web apps

Used For :-

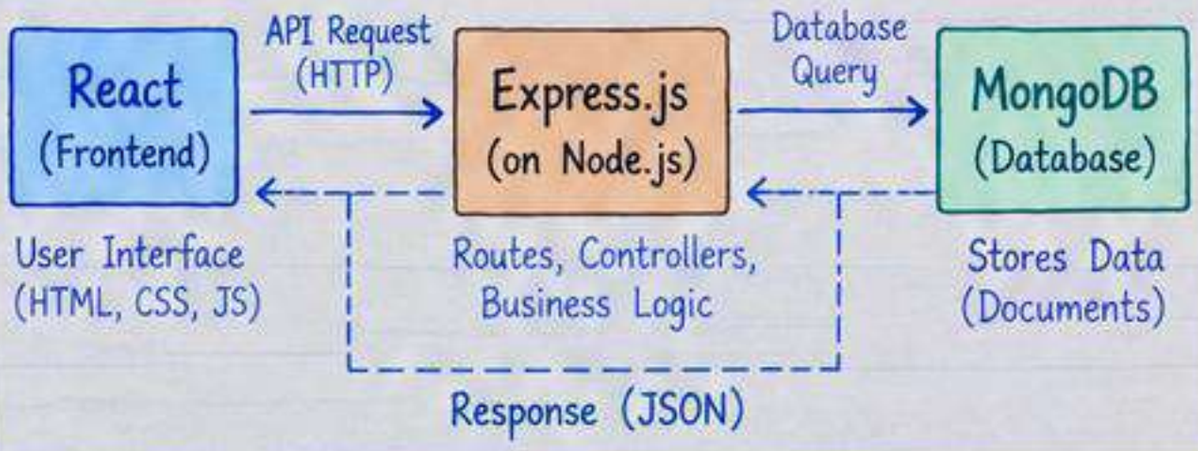
- Web applications
- E-commerce platforms
- Social media apps
- Real-time applications
- Admin dashboards
- Job portals
- Learning platforms
- SaaS products
- and many more ...

3 How MERN Works Together ?

Ans →

- React handles the user interface (frontend).
- It sends API requests to the Express server.
- Express (running on Node.js) handles the requests, business logic and communicates with MongoDB.
- MongoDB stores the data in a flexible format (documents).
- The server sends the response back to React, which updates the UI.

MERN Architecture (Flow)



4 Real World Examples Using MERN

Ans →

- Facebook (uses React)
- Instagram (uses React)
- Netflix (uses React)
- Uber (uses Node.js)
- LinkedIn (uses React)
- Many startups and modern companies use MERN for their products.

Benefits for Developers :-

- Learn once, use everywhere (JavaScript)
- Many learning resources available
- Active community support
- Easy to find jobs and opportunities
- Great for building your portfolio
- Can build real-world projects
- Frequently used in industry

Important Note :-

- MERN is not a full framework, it's a stack of technologies.
- You can replace any technology (e.g. Next.js instead of React) based on project needs.
- MERN is a great choice for beginners and professionals both.

5 What You Will Learn in These Notes ?

Ans →

- Frontend development with React
- Backend development with Node.js and Express
- Database management with MongoDB
- Authentication, authorization and security
- Building real-world MERN projects
- Deployment and industry best practices
- Interview questions and cheat sheets

MERN Learning Roadmap (High Level)

Learn React (Frontend) → Learn Node.js (Runtime) → Learn Express.js (Backend) → Learn MongoDB (Database) → Build Projects & Deploy

"Small Steps Everyday Make You a Better Developer!" :)

Karyvio
LEARN | BUILD | GROW | GET HIRED

React Setup & Project Structure

Set you your first React project and understand the folder structure

Good Setup Leads to Great Projects !

1 Requirements Before You Start ?

Ans →
- A computer (Windows / macOS / Linux)
- Node.js (v18 or higher)
- npm (comes with Node.js)
- A code editor (VS Code recommended)
- Basic knowledge of HTML, CSS and JavaScript

Check Versions :-

```
node -v     // e.g. v20.11.0
npm -v      // e.g. 10.2.4
npx -v      // e.g. 10.2.4
```

Make sure Node.js is installed properly.

2 Create a New React Project (Using Vite)

Ans →
- Vite is a modern and fast build tool for React.
- Open your terminal and run the following command:

```
npm create vite@latest my-react-app -- --template react
```

- Now go to the project folder: `cd my-react-app`
- Install the dependencies: `npm install`
- Start the development server: `npm run dev`

Explanation :-
- **npm create vite@latest** → creates a new React project using Vite.
- **--template react** → uses React template.
- **npm install** → installs all required dependencies.
- **npm run dev** → starts the development server (usually on http://localhost:5173).

3 Project Folder Structure

Ans →
- A clean and organized folder structure helps in better development and maintenance.
- Vite provides a simple and modern structure.
- You can customize it based on your project need.

Folder Structure (Vite + React)

```
my-react-app/
├── node_modules/      # Project dependencies
├── public/            # Static files (images, favicon, etc.)
├── src/               # Source code
│   ├── assets/        # Images, fonts, etc.
│   ├── components/    # Reusable components
│   ├── App.jsx        # Main component
│   └── main.jsx       # Entry point
├── index.html         # HTML template
├── package.json       # Project metadata and scripts
├── vite.config.js     # Vite configuration
└── README.md          # Project documentation
```

4 Important Files Explained

Ans →

File / Folder	Purpose
index.html	Main HTML file (root template)
src/main.jsx	Entry point of the React app
src/App.jsx	Main component (you can modify this)
src/components/	Reusable components
src/assets/	Static assets like images
public/	Files that are directly served
vite.config.js	Vite configuration file
package.json	Project details and dependencies

Useful npm Scripts

```
npm run dev       # start dev server
npm run build     # build for production
npm run preview   # preview build
npm install       # install dependencies
```

5 Run Your First React App

Ans →
- After running npm run dev, open the URL (usually http://localhost:5173) in your browser.
- You will see a default React + Vite page.
- You can now start modifying the code in src/App.jsx.
- Save the file and the browser will automatically reload (Hot Module Replacement).

Default App Output (Vite + React)

Pro Tips :-
- Use VS Code with useful extensions (ES7+ React, Prettier, ESLint)
- Keep components small and reusable.
- Follow a consistent folder structure.
- Use meaningful file and folder names.
- You can use Tailwind CSS for styling (we'll cover it later).

Karyvio
LEARN | BUILD | GROW | GET HIRED

React Fundamentals

Learn the core concepts of React to build UI components

Strong Fundamentals Build Great Projects !

1 What is JSX ?

Ans →
- JSX (JavaScript XML) is a syntax extension for JavaScript.
- It allows us to write HTML-like code inside JavaScript.
- JSX makes it easier to create and visualize UI components.
- It gets transformed into regular JavaScript (using Babel) before running in the browser.

Example :-

```
// JSX Example
const element = (
  <h1 className="title">
    Hello React !
  </h1>
);
```

→ Looks like HTML but it's JavaScript (JSX)

2 React Components

Ans →
- Components are the building blocks of a React app.
- A component is a reusable piece of UI.
- There are two types of components:
  - Function Components (modern, preferred)
  - Class Components (old, rarely used)
- Components help in building modular and maintainable code.

Function Component Example :-

```
function Welcome() {
  return <h2>Welcome to Karyvio!</h2>;
}

export default Welcome;
```

← Simple Function Component

3 Props (Properties)

Ans →
- Props allow us to pass data from a parent component to a child component.
- They are read-only (cannot be modified by the child).
- Props make components reusable and dynamic.

Example :-

```
function UserCard({ name, age }) {
  return (
    <div>
      <h3>{name}</h3>
      <p>Age: {age}</p>
    </div>
  );
}
```

Using the Component :-

```
<UserCard
   name="Sovit"
   age={21} />
```

↳ Passing data via props

4 State

Ans →
- State is used to store and manage data inside a component.
- When state changes, React re-renders the component automatically.
- We use the useState hook to create state in function components.

Example :-

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

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

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>
        Increment
      </button>
    </div>
  );
}
```

Click button to update state and re-render the component

5 Events

Ans →
- Events allow us to handle user interactions like clicks, input changes, form submission etc.
- We use camelCase event names (e.g. onClick, onChange) in React.

Example :-

```
function Button() {
  const handleClick = () => {
    alert("Button Clicked!");
  };

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

→ Handle click event in React

6 Conditional Rendering

Ans →
- Used to render different UI based on conditions.
- We can use if-else, ternary operator, or logical && operator.

Example (Ternary) :-

```
function Message({ isLoggedIn }) {
  return (
    <div>
      {isLoggedIn ? (
        <p>Welcome Back!</p>
      ) : (
        <p>Please Login</p>
      )}
    </div>
  );
}
```

↩ Render different UI based on condition

Common Event List :-
- onClick
- onChange
- onSubmit
- onFocus
- onBlur
- onKeyDown
- onKeyUp
- onMouseEnter
- onMouseLeave
- and many more ...

Karyvio
LEARN | BUILD | GROW | GET HIRED

React Hooks

Use powerful Hooks to manage state, side effects and more

Hooks Make React More Powerful and Flexible !

1 What are React Hooks ?

Ans →
- Hooks are special functions that let us use React features in functional components.
- Introduced in React 16.8.
- They allow us to manage state, side effects and other React features without writing class components.
- Hooks make components simpler, cleaner and more reusable.

Advantages of Hooks :-
- Use state and other features in functional components
- Cleaner and simpler code
- Reusable logic (custom hooks)
- Better code organization
- No need for class components
- Actively maintained by React team

Commonly Used Hooks :-
- useState
- useEffect
- useRef
- useMemo
- useCallback
- useContext
- useReducer
- useLayoutEffect
- useId
- and more ...

2 useState Hook

Ans →
- useState is used to add state (data) to functional components.
- It returns an array with two values:
  - The current state value
  - A function to update the state
- When the state changes, the component re-renders automatically.

Example :-

```
import { useState } from "react";
function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <h2>Count: {count}</h2>
      <hutton onClick<(() => setCount(count + 1)}>
         Increment
      </button>
  )</div>
};
```

Click button to update state and re-render component

3 useEffect Hook

Ans →
- useEffect is used to handle side effects in functional components.
- Side effects include: API calls, timers, DOM manipulation, subscriptions, etc.
- It runs after the component renders.
- We can control when it runs using a dependency array.

Example :-

```
import { useEffect, useState } from "react";
function User() {
  const [data, setData] = useState(null);
  useEffect(() => {
    // Fetch data from API
    fetch("https://jsonplaceholder.typicode.com/users/1")
     .then(res => res.json())
     .then(data => setData(data));
  }, []); // empty array = run only once

  return <div>{data ? <h2>{data.name}</h2> : "Loading..."}</div>;
}
```

Runs once when component mounts

4 useRef Hook

Ans →
- useRef is used to access DOM elements directly or to store mutable values.
- It does not cause re-renders when the value changes.

Example :-

```
import { useRef } from "react";
function InputFocus() {
  const inputRef = useRef(null);
  const handleClick = () => {
    inputRef.current.focus();
  };
  return (
    <div>
      <input ref={inputRef} pleceholder="Type here" />
      <button onClick<(handleClick)>Focus Input</button>
  } </div>
};
```

Focuses input using useRef

Real Use Cases :-
- Access input fields
- Focus elements
- Store previous values
- Integrate with third-party libraries

5 useMemo and useCallback

Ans →
- useMemo is used to memoize expensive calculations (optimizes performance).
- useCallback is used to memoize functions (to prevent unnecessary re-renders).

useMemo Example :-

```
const result = useMemo(() => {
   return a * b;
}, [a, b]);
```

↳ Recalculate only when a or b changes

useCallback Example :-

```
const handleClick = useCallback(() => {
   console.log("Button clicked");
}, []);
```

↳ Same function reference across re-renders

6 Rules of Hooks

Ans →
- Only call Hooks at the top level of your components.
- Do not call Hooks inside loops, conditions or nested functions.
- Only call Hooks from React function components or custom hooks.
- Follow these rules to avoid unexpected behavior.

Important Note :-
- Dooks don't work in regular JavaScript functions.
- You can create your own custom hooks.
- Hooks make your code more modular and reusable.
- Always use the latest version of React.

Karyvio
LEARN | BUILD | GROW | GET HIRED

React Forms & Validation

Handle user input, validate data and buiid robust forms in React

Good Forms Create Better User Experiences !

1 What are Forms in React ?

Ans →
- Forms in React are used to take user input, like text, email, password, etc.
- React uses controlled components to handle form data.
- Form data is stored in state and updated on every change.
- Forms are commonly used for login, registration, contact, search, etc.

Simple Form Example :-

```
import { useState } from "react";
function SimpleForm() {
  const [name, setName] = useState("");
  return (
    <form>
      <input
        type="text"
        value={name}
        onChange={(e) => setName(e.target.value)}
        placeholder="Enter your name"
      />
      <p>Hello, {name || "User"}!</p>
    </form>
  );
}
```

Controlled component (using state)

2 Controlled vs Uncontrolled Components

Feature	Controlled Component	Uncontrolled Component
Data Handling	Handled by React state	Handled by DOM (ref)
Value Access	Using state (value, onChange)	Using useRef
Re-render	Re-renders on every change	Does not re-render
Validation	Easier to validate	Harder to validate
Best For	Most React forms	Simple forms, file inputs

3 Handling Multiple Inputs

Ans →
- Use a single state object to manage multiple form fields.
- Update the state based on the input name.
- This makes code cleaner and scalable.

Example :-

```
import { useState } from "react";
function UserForm() {
  const [formData, setFormData] = useState({
    name: "",
    email: "",
    password: "",
  });
  const handleChange = (e) => {
    const { name, value } = e.target;
    setFormData({ ...formData, [name]: value });
  };
  return (
    <form>
      <input name="name" value={formData.name}
       onChange={handleChange} pleceholder="Name" />
      <input name="email" type="email" value={formData.email}
       onChange={handleChange} placeholder="Email" />
      <input name="password" type="password"
       value={formData.password} onChange={handleChange}
       placeholder="Password" />
    </form>
  );
```

Using single state object for multiple inputs

4 Form Validation Techniques

Ans →
- Validation ensures the user enters correct data.
- We can validate on:
  - On Change (real-time)
  - On Blur (when field loses focus)
  - On Submit (when form is submitted)
- Common validation rules:
  - Required fields
  - Minimum/maximum length
  - Email format
  - Password strength
  - Custom validation (e.g., confirm password)

Example (Basic Validation) :-

```
const [error, setError] = useState("");
const handleSubmit = (e) => {
  e.preventDefault();
  if (!formData.email) {
    setError("Email is required");
    return;
  }
  // submit form data
};
return (
  <form onSubmit={handleSubmit}>
    <input type="email" value={formData.email}
     onChange={handleChange} placeholder="Email" />
    {error && <p className="error">{error}</p>}
    <button type="submit">Submit</button>
  </form>
);
}
```

Validate before submitting

Important Note :-
- Always validate user input.
- Show meaningful error messages.
- Use libraries like Formik + Yup for large forms.
- Keep UI simple and user-friendly.

5 Using a Validation Library (Yup + Formik)

Ans →
- Formik helps to manage form state easily.
- Yup is used for schema-based validation.
- It reduces boilerplate code and is widely used.
- Great for complex forms.

Yup Validation Schema :-

```
import * as Yup from "yup";
const schema = Yup.objoct({
  name: Yup.string().required("Name is required"),
  email: Yup.string().email("Invalid email").required(),
  password: Yup.string().min(6, "Min 6 characters").required();
});
```

Formik Example :-

```
<Formik
  initialValues={{ name: "", email: "", password: "" }}
  validationSchema={schema}
  onSubmit={(values) => console.log(values)}>
>
  {({ handleChange, handleSubmit, errors }) => (
    <form onSubmit={handleSubmit}>
      <input name="name" onChange={handleChange} />
      {errors.name && <p>{errors.name}</p>}
      <button type="submit">Register</button>
    </form>
  )}
</Formik>
```

Common Use Cases :-
- Login / Register form
- Contact form
- Profile update form
- Search form
- Feedback form
- and more ...

Karyvio
LEARN | BUILD | GROW | GET HIRED

React Routing

Navigate between pages in a React application

Good Routing Makes Your App Feel Like a Real App !

1 What is React Router ?

Ans →

- React Router is a library for handling routing in React applications.
- It allows us to navigate between different components (pages) without reloading the page.
- It uses the HTML5 History API.
- Most commonly used version: react-router-dom (since we are using React for web).

Key Features :-

- Client-side routing
- Dynamic routes
- Nested routes
- Route parameters
- Protected routes
- Programmatic navigation
- Lightweight and flexible

Used In :-

- Single Page Applications (SPAs)
- Web applications
- Admin dashboards
- E-commerce websites
- Portfolio websites
- and many more ...

2 Install React Router

Ans →

- Run the following command in your project folder:

```
npm install react-router-dom
```

Check Version :-

```
npm list react-router-dom
```

Make sure it is installed properly.

3 Basic Routing Setup

Ans →

- Wrap your app with <BrowserRouter>.
- Define routes using <Routes> and <Route>.
- Use <Link> for navigation.
- Use <Outlet> for nested routes.

Example :- (App.jsx)

```jsx
import { BrowserRouter, Routes, Route, Link } from "react-router-dom";
import Home from "./pages/Home";
import About from "./pages/About";

function App() {
  return (
    <BrowserRouter>
     <nav>
       <Link to="/">Home</Link> |
       <Link to="/about">About</Link>
     </nav>
     <Routes>
       <Route path="/" element={<Home />} />
       <Route path="/about" element={<About />} />
     </Routes>
    </BrowserRouter>
  );
}
export default App;
```

Basic routing example

4 Important Components

Ans →

Component	Purpose
BrowserRouter	Wraps the entire app (uses HTML5 API)
Routes	Container for all routes
Route	Defines a single route
Link	Navigates to a route (like <a> tag)
NavLink	Like Link but with active styles
Outlet	Renders child routes (for nested routes)
useNavigate	Programmatic navigation
useParams	Access route parameters
useLocation	Get current location info

5 Route Parameters

Ans →

- Used to capture dynamic values from the URL.
- Example: /user/1, /user/2, etc.
- Use the useParams hook to access the value.

Example :-

```jsx
import { useParams } from "react-router-dom";

function User() {
  const { id } = useParams();
  return <h2>User ID: {id}</h2>;
}
```

Access dynamic value from URL

6 Nested Routes

Ans →

- Used to render child routes inside a parent component.
- Use <Outlet> to display child components.

Example :-

```jsx
<Routes>
  <Route path="/dashboard" element={<Dashboard />}>
    <Route path="overview" element={<Overview />} />
    <Route path="settings" element={<Settings />} />
  </Route>
</Routes>
```

Child routes render inside <Outlet />

7 Programmatic Navigation

Ans →

- Use the useNavigate hook to navigate programmatically (e.g., after login).
- Useful for redirects.

Example :-

```jsx
import { useNavigate } from "react-router-dom";

function Login() {
  const navigate = useNavigate();
  const handleLogin = () => {
    // after successful login
    navigate("/dashboard");
  };
},
```

Navigate after an action

8 Protected Routes (Private Routes)

Ans →

- Used to restrict access to certain pages.
- Check if user is authenticated, otherwise redirect.

Example :-

```jsx
function PrivateRoute({ children }) {
  const isAuth = true; // check auth (token, etc.)
  return isAuth ? children : <Navigate to="/login" />;
}
```

Allow access only if authenticated

9 Best Practices

Ans →

- Use meaningful and clean URLs.
- Keep route components small.
- Use lazy loading for better performance.
- Handle 404 (Not Found) page.
- Use protected routes for sensitive pages.

Example: 404 Page

```jsx
<Route path="*" element={<NotFound />} />
```

Catches all unknown routes

Common Issues :-

- Blank page (forgot to wrap with BrowserRouter).
- Route not working (check path and component).
- useNavigate is not a function (wrong import).
- Nested routes not rendering (missing <Outlet />).

Karyvio
LEARN | BUILD | GROW | GET HIRED

React State Management

Manage and share state across components effectively

Good State Management Makes Your App Scalable !

1 What is State Management ?

Ans →
- State management means managing data that changes over time in a React application.
- It helps components share and update data in a predictable way.
- For small apps we can use useState or Context API, and for large apps we use Redux Toolkit or other state management libraries.

Why State Management ?
- Share data between components
- Centralized state (better organization)
- Avoid prop drilling
- Predictable state updates
- Easier debugging
- Better for large and complex apps

Popular Libraries
- useState (built-in)
- useContext (built-in)
- Redux Toolkit (most used)
- Zustand
- Recoil
- Jotai
- MobX
- and more ...

2 useState for Local State

Ans →
- useState is used to manage state in a single component.
- Best for simple and small-scale state.
- Syntax: const [state, setState] = useState(initialValue)

Example :-

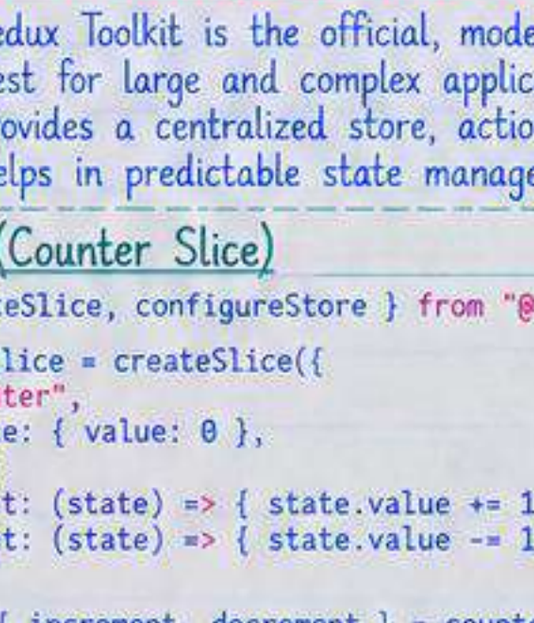
```jsx
import { useState } from "react";

function Counter() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <h2>Count: {count}</h2>
      <button onClick={() => setCount(count + 1)}>
        Increment
      </button>
    </div>
  );
}
```

Local state in a component

3 Prop Drilling Problem

Ans →
- When we need to pass the same data through many components, it becomes difficult.
- This is called prop drilling.
- It makes the code harder to maintain.



Solution:-
Use Context API or Redux to avoid prop drilling.

4 Context API

Ans →
- Context API helps to share state across components without passing props manually.
- Useful for medium-size applications (theme, auth, user info).

Example :-

```jsx
import { createContext, useContext, useState } from "react";

const ThemeContext = createContext();

function ThemeProvider({ children }) {
  const [theme, setTheme] = useState("light");
  return (
    <ThemeContext.Provider value={{ theme, setTheme }}>
      {chiidren}
    </ThemeContext.Provider>
  );
}
// Use in component
function Header() {
  const { theme, setTheme } = useContext(ThemeContext);
  return <h2>Current Theme: {theme}</h2>;
}
```

Sharing state using Context API

5 Redux Toolkit (Advanced State Management)

Ans →
- Redux Toolkit is the official, modern way to use Redux.
- Best for large and complex applications.
- Provides a centralized store, actions and reducers.
- Helps in predictable state management and debugging.

Example :- (Counter Slice)

```jsx
import { createSlice, configureStore } from "@reduxjs/toolkit";

const counterSlice = createSlice({
  name: "counter",
  initialState: { value: 0 },
  reducers: {
    increment: (state) => { state.value += 1 },
    decrement: (state) => { state.value -= 1 }
  }
});
export const { increment, decrement } = counterSlice.actions;
export const store = configureStore({
    reducer: { counter: counterSlice.reducer }
});
```

Create slice and store using Redux Toolkit

6 useSelector and useDispatch

Ans →
- Use useSelector to read data from the store.
- Use useDispatch to trigger actions.

Example :-

```jsx
import { useSelector, useDispatch } from "react-redux";
import { increment } from "./counterSlice";

function Counter() {
  const count = useSelector((state) => state.counter.value);
  const dispatch = useDispatch();

  return (
    <div>
      <h2>Count: {count}</h2>
      <button onClick={() => dispatch(increment())}>
        Increment
      </buttom>
    </div>
  );
}
```

Read state and dispatch actions

7 When to Use What ?

Ans →

Use Case	Recommended Tool
Local component state	useState
Share state between few components	useContext
Large and complex application	Redux Toolkit
Simple global state (small apps)	Zustand / Jotai
Server state (API data)	React Query / TanStack Query

8 Best Practices

Ans →
- Keep state as minimal as possible.
- Use the right tool for the right use case.
- Avoid unnecessary re-renders.
- Keep your state structure clean and organized.
- Use TypeScript for better maintainability.
- Use DevTools (React DevTools, Redux DevTools).
- Follow immutable update patterns.

Important Note :-
- Don't overuse global state.
- Keep components small and focused.
- State management significantly improves code maintainability and scalability.

Karyvio
LEARN | BUILD | GROW | GET HIRED

React API Integration

Connect your React app with backend APIs and handle data efficiently

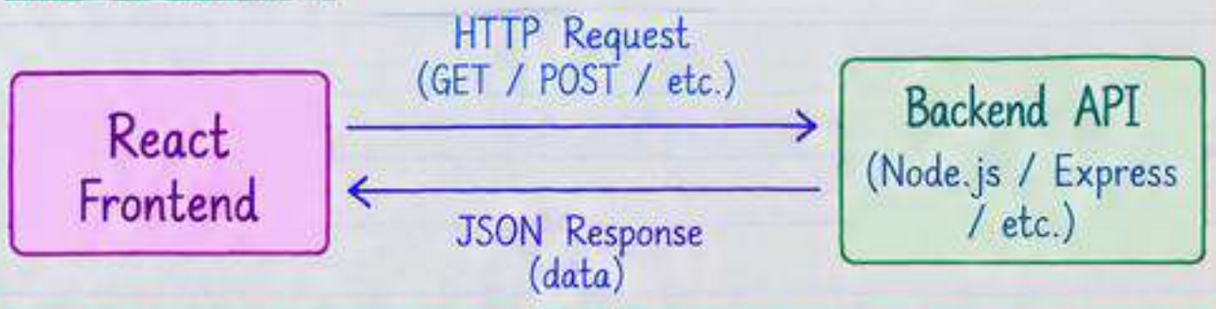
APIs
Bring Real Data
To Your App
Make It Dynamic !

1 What is API Integration ?

Ans →

- API (Application Programming Interface) allows two applications to communicate.
- In React, we integrate APIs to fetch and send data to a backend server.
- We can use Fetch API (built-in) or Axios (third-party library).
- Commonly used for real-time data like users, products, blogs, etc.

How It Works ?



2 Using Fetch API (Built-in)

Ans →

- Fetch is a modern, built-in JavaScript API for making HTTP requests.
- It returns a Promise.
- We need to convert the response to JSON using res.json().

Example :: Fetch Data (GET Request)

```
import { useEffect, useState } from "react";

function Users() {
  const [users, setUsers] = useState([]);

  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/users")
      .then(res => res.json())
      .then(data => setUsers(data))
      .catch(err => console.error("Error:", err));
  }, []);

  return (
    <div>
      <h2>Users:</h2>
      <ul>{users.map(u => <li key={u.id}>{u.name}</li>)}</ul>
    </div>
  );
}
```

Fetch data from a public API and display users

3 Using Axios (Popular Library)

Ans →

- Axios is a promise-based HTTP client (better error handling, interceptors, etc).
- Easier to use than fetch.
- First install Axios:

```
npm install axios
```

Example :: Using Axios (GET Request)

```
import axios from "axios";
import { useEffect, useState } from "react";

function Posts() {
  const [posts, setPosts] = useState([]);

  useEffect(() => {
    axios.get("https://jsonplaceholder.typicode.com/posts")
      .then(res => setPosts(res.data))
      .catch(err => console.error(err));
  }, []);

  return (
    <div>
      <h2>Posts:</h2>
      <ul>{posts.map(p => <li key={p.id}>{p.title}</li>)}</ul>
    </div>
  );
}
```

Using Axios to fetch posts

4 POST, PUT, DELETE Requests

Ans →

- We can send data to the server using POST, update using PUT/PATCH and delete using DELETE.

Example :: POST Request (Create Data)

```
import axios from "axios";
const createPost = async () => {
  try {
    const res = await axios.post("https://jsonplaceholder.typicode.com/posts", {
      title: "New Post",
      body: "This is a new post",
      userId: 1
    });
    console.log("Created:", res.data);
  } catch (err) {
    console.error("Error:", err);
  }
};
```

Send data to the server (using POST)

5 Handling Loading & Error States

Ans →

- Always handle loading and error states for a better user experience.
- Show a loader while data is fetching.
- Show a meaningful error message if the API fails.

Example :: Loading and Error Handling

```
const [loading, setLoading] = useState(tree);
const [error, setError] = useState("");

useEffect(() => {
  axios.get("/api/data")
    .then(res => setData(res.data))
    .catch(err => setError("Failed to load data"))
    .finally(() => setLoading(false));
}, []);
```

Handle loading and error states

UI Example ::

```
if (loading) return <p>Loading...</p>;
if (error) return <p className="error">
    {error}
  </p>;
return <DataComponent data={data} />;
```

6 Create a Reusable API Service

Ans →

- It's a good practice to create a separate file for API calls to keep code clean and reusable.

api.js

```
import axios from "axios";
const API = axios.create({
  baseURL: "https://jsonplaceholder.typicode.com",
  headers: {
    "Content-Type": "application/json"
  }
});
export default API;
```

Create a single Axios instance

Using in a Component

```
import API from "./api";
API.get("/users").then(res => setUsers(res.data));
```

Use API instance

7 Common API Methods (Axios)

Method	Purpose	Example
get()	Fetch data	axios.get('/users')
post()	Create data	axios.post('/users', data)
put()	Update (full)	axios.put('/users/1', data)
patch()	Update (partial)	axios.patch('/users/1', data)
delete()	Delete data	axios.delete('/users/1')

8 Best Practices

Ans →

- Use a separate API service file.
- Handle loading and error states.
- Use environment variables for base URL.
- Use interceptors (for auth tokens).
- Keep API calls atomic and reusable.
- Use meaningful error messages.
- Avoid calling API directly inside components (reuse custom hooks/services).

Real World Use Cases

- Login / Register
- Fetch user profile
- Get product list
- Search data
- Create / update data
- Pagination
- Integrate with third-party APIs (Google, GitHub, etc.)
- and many more ...

Karyvio
LEARN | BUILD | GROW | GET HIRED

React Advanced Patterns

Build reusable, scalable and maintainable React applications

Advanced Patterns Make Your Code Cleaner, Smarter and Scalable !

1 What are Advanced Patterns ?

Ans →
- Advanced patterns are best practices and techniques to write clean, reusable and maintainable React code.
- They help us solve common problems in large applications.
- These patterns improve code structure, reusability, performance and scalability.
- They are widely used in real-world projects.

Common Advanced Patterns :-
- Custom Hooks
- Higher Order Components (HOCs)
- Render Props
- Component Composition
- Compound Components
- Controlled vs Uncontrolled Components
- Code Splitting & Lazy Loading
- Error Boundaries
- State Lifting & Context Patterns
- and more ...

2 Custom Hooks

Ans →
- Custom hooks allow us to reuse stateful logic between components.
- We can extract logic into a separate function starting with use.
- Example: useFetch, useLocalStorage, useToggle, etc.

Example: useFetch (Custom Hook)

```
import { useState, useEffect } from "react";

function useFetch(url) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    fetch(url)
      .then(res => res.json())
      .then(data => setData(data))
      .catch(err => setError(err))
      .finally(() => setLoading(false));
  }, [url]);
  return { data, loading, error };
}
export default useFetch;
```

Reusable logic in a custom hook

3 Higher Order Components (HOCs)

Ans →
- A HOC is a function that takes a component and returns a new component with additional functionality.
- Useful for code reuse (e.g., authentication, logging, etc.).

Example: withAuth (HOC)

```
function withAuth(Component) {
  return function ProtectedComponent(props) {
    const isAuthenticated = true; // check auth

    if (!isAuthenticated) {
      return <h2>Please login to access this page</h2>;
    }
    return <Component {...props} />;
  };
}
export default withAuth;
```

Adds authentication functionality to any component

4 Component Composition

Ans →
- Use composition instead of inheritance.
- Build flexible and reusable components by
- passing children, props or components.
- Makes components more modular and maintainable.

Example: Using children prop

```
function Card({ title, children }) {
  return (
    <div className="card">
      <h2>{title}</h2>
      <div className="card-body">{children}</div>
    </div>
  );
}
// Usage
<Card title="User Info">
  <p>This is reusable content</p>
</Card>
```

Composition makes components flexible

5 React Lazy Loading (Code Splitting)

Ans →
- Lazy loading helps to load components only when needed.
- It reduces the initial bundle size and improves performance.

Example: Lazy Loading a Component

```
import { lazy, Suspense } from "react";

const About = lazy(() => import("./pages/About"));

function App() {
  return (
    <div>
      <h1>My App</h1>
      <Suspense fallback={<p>Loading...</p>}>
        <About />
      </Suspense>
    </div>
  );
}
```

Component is loaded only when needed

6 Error Boundaries

Ans →
- Error boundaries catch JavaScript errors in child components.
- They prevent the entire app from crashing.
- Useful for production apps.

Example: Error Boundary

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

class ErrorBoundary extends Component {
  constroctor(props) {
    super(props);
    this.state = { hasError: false };
  }
  static getDerivedStateFromError() {
    return { hasError, true };
  }
  render() {
    if (this.state.hasError) {
      return <h2>Something went wrong 😢</h2>;
    }
    return this.props.children;
} }
export default ErrorBoundary;
```

Catches errors in child components

7 Best Practices

Ans
- Use custom hooks to reuse logic.
- Keep components small and focused.
- Use lazy loading for better performance.
- Use error boundaries for stability.
- Prefer composition over inheritance.
- Follow consistent folder structure.
- Use meaningful names and comments.
- Keep components reusable and testable.

Real Use Cases :-
- Custom hooks for API calls
- HOCs for authentication
- Lazy loading in large apps
- Error boundaries in production
- Component composition in UI libraries
- Reusable components in design systems

Important Note :-
- Don't overuse patterns (keep it simple).
- Use the right pattern for the right problem.
- Focus on readability, maintainability and performance.
- Advanced patterns become more useful in medium/large applications.
- Keep learning and explore real-world open source projects.

Karyvio
LEARN | BUILD | GROW | GET HIRED

React Performance & Best Practices

Build faster, cleaner and scalable React applications

Better Performance Better Users Happier You !

1 Why Performance Matters ?

Ans →
- Faster loading and smoother interactions.
- Better user experience and higher retention.
- Improves SEO (Core Web Vitals).
- Reduces unnecessary re-renders and memory usage.
- Important for large and data-heavy applications.

Key Metrics (Web Vitals) :-
- LCP (Largest Contentful Paint) - loading performance
- FID (First Input Delay) - interactivity
- CLS (Cumulative Layout Shift) - visual stability

Tools to Measure :-
- React DevTools (Profiler)
- Lighthouse
- Web Vitals Extension
- Chrome Performance Tab
- Why Did You Render
- and more ...

2 Avoid Unnecessary Re-renders

Ans →
- Re-renders can be expensive, especially in large apps.
- Use React.memo, useMemo and useCallback to optimize.
- Keep state as local as possible.
- Avoid inline functions and objects in JSX.

Example :- React.memo

```jsx
import React from "react";

const UserCard = React.memo(({ name }) => {
  console.log("Rendered:", name);
  return <div>{name}</div>;
});

export default UserCard;
```

Component will re-render only if props change

3 Use useMemo for Expensive Calculations

Ans →
- useMemo caches the result of a calculation.
- Useful for expensive computations.
- Re-calculates only when dependencies change.

Example :- useMemo

```jsx
import { useMemo, useState } from "react";

function ExpensiveComponent({ items }) {
  const sortedItems = useMemo(() => {
    console.log("Sorting items...");
    return [...items].sort((a, b) =>
      a.name.localeCompare(b.name));
  }, [items]);

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

Sorting happens only when items change

4 Use useCallback for Stable Functions

Ans →
- useCallback memoizes a function reference.
- Useful when passing functions to child components.
- Prevents unnecessary re-renders in child components.

Example :- useCallback

```jsx
import { useCallback, useState } from "react";

const Button = ({ onClick }) => {
  console.log("Button rendered");
  return <button onClick={onClick}>Click Me</button>;
};

function App() {
  const [count, setCount] = useState(0);
  const handleClick = useCallback(() => {
    setCount(c => c + 1);
  }, []);

  return <Button onClick={handleClick} />;
}
```

Same function reference across renders

5 Code Splitting & Lazy Loading

Ans →
- Load components only when needed.
- Reduces initial bundle size.
- Use React.lazy and Suspense.

Example :- Lazy Loading

```jsx
import { lazy, Suspense } from "react";

const About = lazy(() => import("./pages/About"));
function App() {
  return (
    <div>
      <h1>My App</h1>
      <Suspense fallback={<p>Loading...</p>}>
        <About />
      </Suspense>
    </div>
  );
}
```

Component is loaded only when needed

6 Optimize Images and Assets

Ans →
- Use modern formats (WebP, AVIF).
- Compress images.
- Use lazy loading for images.
- Use CDN for static assets.

Example :- Lazy Image

```jsx
<img
  src="/images/banner.webp"
  alt="banner"
  loading="lazy"
  width={800}
  height={400}
/>
```

Lazy loading improves page load time

Tips :-
- Use WebP format
- Resize images
- Use next-gen formats
- Use a CDN
- Lazy load below the fold
- Remove unused assets

7 Other Best Practices

Ans →
- Keep components small and focused.
- Use proper folder structure.
- Avoid deep prop drilling (use Context or state management).
- Remove unused code and dependencies.
- Use TypeScript for better maintainability.
- Follow consistent naming and code style.
- Use ESLint and Prettier.
- Regularly update dependencies.

Do's and Don'ts :-

Do's
- Use memonization wisely
- Keep state local
- Use lazy loading
- Optimize assets
- Follow modern React patterns
- Monitor performance regularly

Don'ts
- Don't overuse useMemo/useCallback
- Don't put everything in global state
- Don't ignore re-render warnings
- Don't load large assets unnecessarily
- Don't create inline functions in JSX
- Don't sacrifice readability for optimization

Karyvio
LEARN | BUILD | GROW | GET HIRED

Node.js Introduction

JavaScript Runtime for Building Scalable Backend Applications

Same JavaScript Everywhere Frontend to Backend

1 What is Node.js ?

Ans →

- Node.js is an open-source, cross-platform JavaScript runtime environment.
- It allows us to run JavaScript code outside the browser (on the server).
- Built on Chrome's V8 JavaScript engine.
- Used to build fast, scalable, and real-time applications.
- Popular for building APIs, microservices, CLI tools, and more.

Key Features :-

- Fast and efficient (V8 engine)
- Non-blocking, event-driven I/O
- Single-threaded with event loop
- Huge ecosystem (NPM)
- Cross-platform (Windows, Linux, macOS)
- Great for real-time applications
- Supports modern JavaScript (ES6+)

Popular Use Cases :-

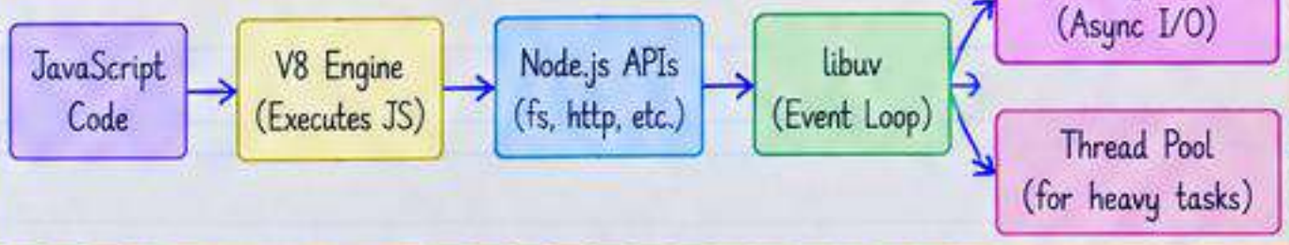
- REST APIs
- Real-time applications (Chat, Notifications)
- Microservices
- Command-line tools (CLI)
- Web servers
- Server-side rendering
- IoT and data streaming
- Build tools and automation

2 How Node.js Works ?

Ans →

- Node.js uses the V8 engine to execute JavaScript.
- It follows an event-driven, non-blocking I/O model.
- Handles multiple requests using a single thread with an event loop.
- Uses libuv (C library) for asynchroncus I/O operations.

Node.js Architecture :-



3 Install Node.js

Ans →

- Download from official website: https://nodejs.org
- Choose LTS (Long Term Support) version.
- After installation, verify using:

```
# Check version
node -v
npm -v
```

Shows installed Node.js and NPM version

4 Run Your First Node.js Program

Ans →

- Create a file named **index.js**
- Write a simple JavaScript code
- Run it using **node** command

index.js

```
console.log("Hello, Node.js!");

// Run in terminal
node index.js
```

Your first Node.js program is ready !

5 Common Built-in Modules

Ans →

- Node.js provides many built-in modules to perform common tasks.
- No need to install them separately.

Module	Purpose
fs	Work with files and directories
http	Create web server
path	Handle file paths
os	Get OS information
events	Work with event emitters
util	Utility functions
url	Parse URL strings
crypto	Cryptography functions

6 Example: HTTP Server

Ans →

- Using built-in **http** module, we can create a simple web server.

server.js

```
const http = require('http');
const server = http.createServer(
  (req, res) => {
    res.writeHead(200, {
      'Content-Type': 'text/plain'
    });
    res.end('Hello from Node.js Server!');
  }
);
server.listen(3000, () => {
  console.log('Server running at
http://localhost:3000');
});
```

Open in browser http://localhost:3000

7 NPM (Node Package Manager)

Ans →

- NPM comes bundled with Node.js.
- Helps to manage third-party packages (libraries).
- Used to install, update and manage dependencies.

Common NPM Commands :-

npm init -y	# Initialize project
npm install <package>	# Install package
npm install	# Install all dependencies
npm uninstall <package>	# Remove package
npm update	# Update packages
npm list	# List installed packages

8 package.json

Ans →

- package.json stores metadata about the project.
- It keeps track of dependencies, scripts, version, etc.
- Created using: **npm init -y**

package.json

```
{
  "name": "my-app",
  "version": "1.0.0",
  "description": "A simple Node.js app",
  "main": "index.js",
  "scripts": {
    "start": "node index.js"
  },
  "dependencies": {}
}
```

Helps to manage your project easily !

9 Advantages of Node.js

Ans →

- Uses JavaScript (same language frontend & backend)
- Fast and lightweight
- Non-blocking and event-driven
- Large community support
- Rich ecosystem (NPM)
- Suitable for real-time and scalable apps
- Good for microservices architecture

Next Topics to Learn :-

- Modules in Node.js
- File System (fs)
- Working with Express.js
- Middleware
- Building REST APIs
- Database Integration (MongoDB / PostgreSQL)
- Authentication & Authorization

10 Best Practices

Ans →

- Use LTS version
- Follow modular structure
- Use environment variables (.env)
- Handle errors properly
- Use async/await instead of callbacks
- Validate input data
- Use third-party libraries wisely
- Keep dependencies updated
- Follow security best practices
- Structure your project (MVC or modular)

Remember :-

- Node.js is just JavaScript running on server.
- Start small, build projects, and keep practicing.
- Explore NPM and open-source packages.
- Focus on building real-world applications.
- Consistency is the key !

Karyvio
LEARN | BUILD | GROW | GET HIRED

Node.js Core Concepts

Understand the core building blocks of Node.js to write efficient and scalable applications

Master Core Concepts Build Better Backends !

1 Global Object and Globals

Ans →
- Node.js provides global objects and variables that are available in all modules.
- Some commonly used globals are:
- These globals help in core functionality like file paths, timers, debugging, etc.

Global	Description
global	The global namespace object
__dirname	Directory name of the current module
__filename	Full path of the current module file
process	Information about the current process
require()	Function to import modules
module	Information about the current module
exports	Used to export from a module

2 Modules in Node.js

Ans →
- Node.js uses a modular architecture (CommonJS by default).
- We can create, import and export modules using require() and module.exports.
- Helps in code organization and reusability.

Example :- module1.js

```
// Exporting from a module
const greet = (name) => {
  return `Hello, ${name}!`;
};
module.exports = { greet };
```

Example :- app.js

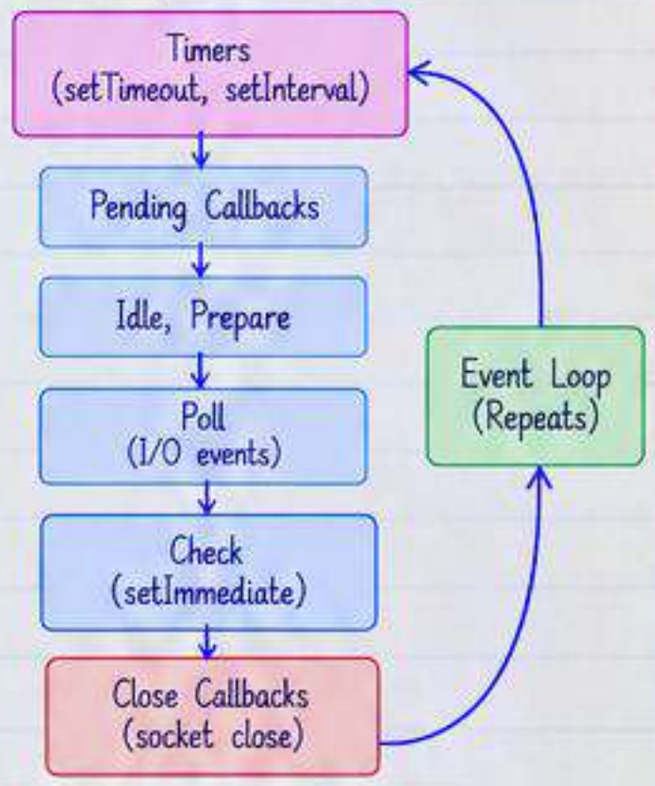
```
// Importing the module
const { greet } = require('./module1');

console.log(greet('Sovit'));
// Output: Hello, Sovit!
```

3 The Event Loop

Ans →
- Node.js is non-blocking and uses an event-driven architecture.
- The event loop handles asynchronous operations.
- It allows Node.js to perform non-blocking I/O using a single thread.
- Time-consuming tasks are offloaded to the libuv thread pool.

Event Loop Flow :-



4 Asynchronous Programming

Ans →
- Node.js uses non-blocking I/O to handle multiple operations concurrently.
- We can handle async code using:
  - Callbacks
  - Promises
  - async/await
  - Event Emitters

5 Callbacks

Ans →
- Callbacks are functions passed as arguments to be executed later.
- They follow the error-first pattern (err, data).

Example :- Callback

```
fs.readFile('data.txt', 'utf8', (err, data) => {
  if (err) {
    console.error('Error:', err);
    return;
  }
  console.log('File Content:', data);
});
```

6 Promises

Ans →
- Promises make async code cleaner and more manageable.
- They have three states: Pending, Fulfilled, Rejected.

Example :- Promise

```
const fs = require('fs').promises;

fs.readFile('data.txt', 'utf8')
  .then(data => console.log(data))
  .catch(err => console.error(err));
```

7 async / await

Ans →
- async/await is syntactic sugar over Promises.
- It makes asynchronous code look synchronous and easier to read.

Example :- async / await

```
const fs = require('fs').promises;

async function readFile() {
  try {
    const data = await fs.readFile('data.txt', 'utf8');
    console.log(data);
  } catch (err) {
    console.error('Error:', err);
  }
}
readFile();
```

8 EventEmitter

Ans →
- EventEmitter allows us to create and handle custom events.
- Widely used in Node.js core modules like HTTP, Streams, etc.

Example :- EventEmitter

```
const EventEmitter = require('events');

const emitter = new EventEmitter();

emitter.on('greet', (name) => {
  console.log('Hello, ${name}!');
});

emitter.emit('greet', 'Sovit');
// Output: Hello, Sovit!
```

9 Process Object

Ans →
- The process object provides information about the current Node.js process.
- Useful for environment variables, arguments, exiting the process, etc.

Example :- Process

```
console.log(process.env.NODE_ENV);
console.log(process.argv);
console.log(process.platform);
// Exit the process
process.exit(0);
```

10 Streams (Brief Overview)

Ans →
- Streams are used to handle large amounts of data efficiently (e.g., files, network data).
- Types: Readable, Writable, Duplex, Transform.
- Helps in processing data in chunks instead of loading entire data into memory.

Key Takeaways :-
- Understand how Node.js works under the hood.
- Use appropriate async patterns (callbacks, promises, async/await).
- Leverage modules and core APIS effectively.
- Event loop and non-blocking I/O make Node.js fast and scalable.
- Practice with real-world examples to strengthen concepts.

Good Code Better Opportunities !

Karyvio
LEARN | BUILD | GROW | GET HIRED

Node.js Built-in Modules

Powerful built-in modules to handle common tasks without extra installations

Use
Built-in Modules
Write Efficient Code
Keep It Simple

1 What are Built-in Modules ?

Ans →
- Node.js comes with a set of built-in modules that provide core functionality.
- These modules are part of Node.js itself, so no need to install them using npm.
- They help us perform common tasks like file handling, paths, HTTP server, OS info, etc.
- We can use them directly with require() or import (in ES modules).

2 How to Use Built-in Modules ?

Ans →
- Using CommonJS (default in Node.js)

```
const fs = require('fs');
```

- Using ES Modules (Node.js 14+)

```
import fs from 'fs';
```

List of Common Built-in Modules

- fs (File System)
- path (File & Directory Paths)
- http (Create HTTP Server)
- os (Operating System Info)
- events (Event Handling)
- url (URL Utilities)
- querystring (Parse Query Strings)
- crypto (Cryptography)
- util (Utility Functions)
- stream (Working with Streams)
- zlib (Compression)
- and more ...

3 fs Module (File System)

Ans →
- Used to read, write, update and delete files.
- Supports both synchronous and asynchronous methods.

Example :- Read and Write File

```
const fs = require('fs');

fs.writeFile('hello.txt', 'Hello Node.js!', (err) => {
  if (err) throw err;
  console.log('File written!');
});

fs.readFile('hello.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log('File content:', data);
});
```

Create a file and read its content

4 path Module

Ans →
- Used to work with file and directory paths.
- Helps to handle paths in a cross-platform way.

Useful for handling file paths safely

Example :- Path Utilities

```
const path = require('path');

console.log(path.basename('/user/docs/file.txt')); // file.txt
console.log(path.dirname('/user/docs/file.txt'));  // /user/docs
console.log(path.extname('file.txt'));             // .txt
console.log(path.join('folder', 'sub', 'file.txt')); // folder/sub/file.txt
```

5 http Module

Ans →
- Used to create a simple HTTP server.
- Helps in building web servers and APIs.

Creates a basic web server

Example :- Simple HTTP Server

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

const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello from Node.js Server!');
});
server.listen(3000, () => {
  console.log('Server running at http://localhost:3000');
});
```

6 os Module

Ans →
- Provides information about the operating system.
- Useful to get system details like platform, CPU, memory, etc.

Get information about your system

Example :- OS Information

```
const os = require('os');

console.log('Platform:', os.platform());
console.log('CPU Cores:', os.cpus().length);
console.log('Total Memory:', os.totalmem() / (1024 ** 3), 'GB');
console.log('Free Memory:', os.freemem() / (1024 ** 3), 'GB');
```

7 events Module

Ans →
- Provides the EventEmitter class.
- Helps to handlle custom events in Node.js.

Example :- Event Emitter

```
const EventEmitter = require('events');

const emitter = new EventEmitter();

emitter.on('greet', (name) => {
  console.log('Hello,', name);
});

emitter.emit('greet', 'Sovit');
```

Create and listen to custom events

8 url Module

Ans →
- Used to parse and work with URLs.
- Helps to extract parts like query, pathnane, etc.

Example :- Parse URL

```
const url = require('url');

const myUrl = new URL(
  'https://karyvio.com/blog?category=tech&id=1'
);
console.log(myUrl.hostname); // karyvio.com
console.log(myUrl.pathname); // /blog
console.log(myUrl.searchParams.get('category')); // tech
console.log(myUrl.searchParams.get('id')); // 1
```

9 crypto Module

Ans →
- Used for cryptography-related fhrctionalities.
- Helps in hashing, encryption, etc.

Example :- Hash a Password

```
const crypto = require('crypto');

const hash = crypto
  .createHash('sha256')
  .update('myPassword')
  .digest('hex');

console.log('Hash:', hash);
```

Create hash for storing passwords

10 zlib Module (Compression)

Ans →
- Used to compress and decompress data.
- Useful for reducing file size and improving performance.

Example :- Compress a String

```
const zlib = require('zlib');

const text = 'Hello Node.js! Hello Node.js!';

zlib.gzip(text, (err, compressed) => {
  if (err) throw err;
  console.log('Compressed Size:', compressed.length, 'bytes');
});
```

Compress data using gzip

Some More Built-in Modules :-

Module	Purpose
util	Utility functions (promisify, inspect, etc.)
stream	Work with streaming data
querystring	Parse and stringify query strings
child_process	Run system commands
cluster	Create multiple Node.js processes
readline	Read input from command line
tty	Terminal utilities
net	Create TCP/IPC servers
dns	Perform DNS lookups

Best Practices :-

- Use built-in modules instead of third-party packages when possible.
- Prefer asynchronous methods (to avoid blocking).
- Handle errors properly.
- Use path module for cross-platform compatibility.
- Keep your code modular and clean.
- Explore the official Node.js documentation.

Key Takeaway :-

Node.js built-in modules give you powerful tools out of the box. Use them wisely to build efficient, scalable and secure applications !

Express.js Introduction

A minimal and flexible Node.js web application framework

Simple
Routing
Powerful Middleware
Build Real APIs !

1 What is Express.js ?

Ans →

- Express.js is a minimal and flexible web application framework for Node.js.
- It provides a simple and powerful set of features to build web servers, APIs, and web applications.
- It is built on top of Node.js HTTP module.
- Widely used in production by companies like Netflix, Uber, IBM, etc.
- Helps to handle routing, middleware, request/response, and more easily.

Why Express.js ?

- Simple and easy to learn
- Minimal and unopinionated
- Large community support
- Flexible and extensible
- Great for REST APIs
- Works well with databases (MongoDB, MySQL, etc.)
- Used in real-world applications

2 Features of Express.js

Ans →

- Simple routing
- Middleware support
- Request and response handling
- Template engine integration
- Error handling
- Static file serving
- Easy integration with databases
- Lightweight and minimal
- Highly customizable

3 Install Express.js

Ans →

- Make sure you have Node.js installed.
- Create a new project folder.
- Initialize npm and install Express.

Commands :-

```
# Create project folder
mkdir my-express-app
cd my-express-app

# Initialize npm
npm init -y

# Install Express
npm install express
```

This will download Express and save it in node_modules

4 Your First Express.js App

Ans →

- Create a file named index.js.
- Import express, create an app, define a route and start the server.

index.js :-

```
const express = require('express');
const app = express();

app.get('/', (req, res) => {
   res.send('Hello from Express.js!');
});

const PORT = 3000;
app.listen(PORT, () => {
  console.log('Server running at http://localhost:${PORT}');
});
```

Run the server :-
node index.js

Open in browser :-
http://localhost:3000

5 Understanding the Code

Ans →

- require('express') - imports Express.
- express() - creates an Express application.
- app.get() - defines a route for GET request.
- req - request object (contains data from client).
- res - response object (used to send data back).
- app.listen() - starts the server.

6 Basic Routing

Ans →

- Express provides methods for different HTTP methods like GET, POST, PUT, DELETE, etc.
- Routes define how your app responds to a client request.

Example Routes :-

```
app.get('/', (req, res) => {
  res.send('Home Page');
});

app.get('/about', (req, res) => {
  res.send('About Page');
});

app.post('/user', (req, res) => {
  res.send('User created');
});
```

7 Request and Response

Ans →

- req object: contains information about the incoming request.
- res object: used to send a response to the client.

Commonly Used Properties :-

Property	Description
req.params	Route parameters
req.query	Query string parameters
req.body	Request body data
req.headers	Request headers
res.send()	Send response (string, object, etc.)
res.json()	Send JSON response
res.status()	Set HTTP status code

8 Middleware in Express.js

Ans →

- Middleware functions run between the request and the response.
- Used for logging, authentication, error handling, parsing data, etc.

Example :-

```
const express = require('express');
const app = express();

// Logger middleware
app.use((req, res, next) => {
  console.log('${req.method} ${req.url}');
  next();
});

app.get('/', (req, res) => {
  res.send('Middleware example');
});
```

9 Serving Static Files

Ans →

- Express can serve static files like HTML, CSS, JS, images, etc.

Example :-

```
app.use(express.static('public'));
// Now files inside 'public' folder
// can be accessed directly
```

Folder Structure :-

```
my-express-app/
├── node_modules/
├── public/
│   ├── index.html
│   └── style.css
├── index.js
└── package.json
```

Access
http://localhost:3000/index.html

10 Next Steps

Ans →

- Learn about Express Router for modular routing.
- Use middleware like express.json() (to parse JSON data).
- Connect with a database (MongoDB / MySQL).
- Implement error handling.
- Build real-world projects (e.g., REST API).
- Explore popular middlewares (cors, morgan, etc).
- Deploy your app (Render, Vercel, etc).

Key Takeaway :-

- Express.js makes web development with Node.js easier and faster.
- It is minimal, flexible, and widely used.
- It is the foundation for building powerful APIs and web applications.
- Start small, practice routing and middleware, and gradually build bigger projects !

Express.js Routing & Middleware

Handle requests with routes and middleware to build powerful APIs

Routes
Control Flow
Middleware Adds Power
Build Better
APIs !

1 What is Routing ?

Ans →

- Routing refers to defining how an application responds to a client request for a specific URL and HTTP method (GET, POST, etc.).
- Each route can have a callback function to handle the request.
- Express provides a simple and flexible routing system.

2 What is Middleware ?

Ans →

- Middleware functions are functions that run during the request-response cycle.
- They have access to req, res, and next().
- Can perform tasks like logging, authentication, validation, error handling, etc.
- Can run globally, for specific routes, or for specific HTTP methods.

3 Middleware Function Signature

Ans →

```
function middleware(req, res, next) {
  // do something
  next(); // pass control
}
```

If you don't call next(), the request will be stuck and won't reach the next middleware or route handler.

4 Defining Routes

Ans →

- Use HTTP methods like get(), post(), put(), delete(), etc.
- Routes can be defined directly in app.js or in separate route files.

Example :- Basic Routes

```
const express = require('express');
const app = express();

app.get('/', (req, res) => {
  res.send('Home Page');
});

app.get('/about', (req, res) => {
  res.send('About Page');
});

app.listen(3000, () => {
  console.log('Server started on port 3000');
});
```

5 Route Parameters

Ans →

- Use : to define dynamic parameters in the route.

Example :- Route Parameter

```
app.get('/user/:id', (req, res) => {
  const userId = req.params.id;
  res.send('User ID: ${userId}');
});
```

Example URL:-

```
GET /user/123
Response: User ID: 123
```

6 Query Parameters

Ans →

- Use req.query to access query parameters.

Example :- Query Parameter

```
app.get('/search', (req, res) => {
  const keyword = req.query.q;
  res.send('Search: ${keyword}');
});
```

Example URL:-

```
GET /search?q=react
Response: Search: react
```

7 Using Multiple Middleware

Ans →

- We can use multiple middleware functions for a single route.

Example :- Multiple Middleware

```
app.get('/dashboard', authMiddleware,
  logMiddleware, (req, res) => {
  res.send('Dashboard');
});
```

8 Types of Middleware

Ans →

Type	Description
Application-level	Used globally or for specific routes.
Router-level	Used with express.Router().
Built-in	Provided by Express (e.g. express.json()).
Third-party	Installed via npm (e.g. cors, morgan).
Error-handling	Handles errors (4 agruments).

9 Built-in Middleware

Ans →

- Express provides some built-in middleware functions.

Example :- Built-in Middleware

```
// Parse JSON data
app.use(express.json());

// Parse URL encoded data
app.use(express.urlencoded({
    extended: true
}));

// Serve static files
app.use(express.static('public'));
```

10 Custom Middleware Example

Ans →

- We can create our own middleware for tasks like logging, authentication, etc.

Example :- Logger Middleware

```
function logger(req, res, next) {
  console.log('${new Date().toISOString()} -
   ${req.method} ${req.url}');
  next();
}

app.use(logger);

app.get('/', (req, res) => {
  res.send('Hello from Express with Middleware');
});
```

Example :- Authentication Middleware

```
function auth(req, res, next) {
  const token = req.headers['authorization'];

  if (!token) {
    return res.status(401).send('Access Denied');
  }

  // verify token (dummy check)
  if (token === 'secret') {
    next();
  } else {
    res.status(403).send('Invalid Token');
  }
}

app.get('/profile', auth, (req, res) => {
  res.send('Protected Profile');
});
```

Key Takeaways :-

- Routing defines the endpoints for handling client requests.
- Middleware functions run before route handlers.
- Use next() to pass control to the next function.
- Express has built-in middleware like express.json() and express.static().
- We can create custom middleware for logging, authentication, validation, etc.
- Organize routes using express.Router() for better structure.

Next Topics :-

- Express Router and Modular Route Files
- Error Handling in Express
- Connecting Express with Databases (MongoDB)
- Building RESTful APIs
- Authentication & Authorization (JWT)

Karyvio
LEARN | BUILD | GROW | GET HIRED

REST API Development

Build Real-World APIs using Node.js and Express

Good APIs Power Great Products Build, Document and Scale !

1 What is a REST API ?

Ans →

- REST (Representational State Transfer) is an architectural style for building web APIs.
- It uses standard HTTP methods, is stateless, and works with resources (usually JSON).
- REST APIs allow different systems (frontend, mobile apps, third-party services) to communicate over HTTP.
- Widely used in modern web and mobile applications.

2 Principles of REST

Ans →

- Client-Server architecture
- Stateless (each request is independent)
- Resource-based (use nouns for endpoints)
- Use standard HTTP methods (GET, POST, PUT, DELETE)
- Use standard status codes
- Use JSON for request and response data
- Cacheable
- Layered system
- Uniform interface

3 HTTP Methods

Ans →

Method	Purpose
GET	Read / Get a resource
POST	Create a new resource
PUT	Update an existing resource (replace)
PATCH	Partially update a resource
DELETE	Delete a resource

4 Project Setup

Ans →

- Initialize a new Node.js project and install Express.
- We will also use some useful packages.

Commands :-

```
mkdir rest-api
cd rest-api
npm init -y
npm install express
npm install nodemon --save-dev

# (optional) environment variables
npm install dotenv
```

Creates a new project and installs Express

5 Project Structure

Ans →

```
rest-api/
├── src/
│   ├── routes/          # route files
│   ├── controllers/     # business logic
│   ├── models/          # database models (optional)
│   ├── middlewares/     # custom middleware
│   ├── utils/           # helper functions
│   └── app.js           # express app setup
├── server.js            # start server
├── .env                 # environment variables
└── package.json
```

6 Basic Server Setup

Ans →

src/server.js

```
require('dotenv').config();
const app = require('./app');

const PORT = process.env.PORT || 3000;

app.listen(PORT, () => {
  console.log(`Server running at
http://localhost:${PORT}`);
});
```

7 Express App Setup

Ans →

src/app.js

```
const express = require('express');
const app = express();

app.use(express.json()); // parse JSON
app.use(express.urlencoded({ extended: true }));

app.get('/', (req, res) => {
  res.send('REST API is running!');
});

module.exports = app;
```

Basic Express configuration with JSON parsing

8 Creating a Resource (Example: Users)

Ans →

1. Route (src/routes/userRoutes.js)

```
const express = require('express');
const router = express.Router();
const userController = require('../controllers/
userController');

router.get('/', userController.getUsers);
router.get('/:id', userController.getUserById);
router.post('/', userController.createUser);
router.put('/:id', userController.updateUser);
router.delete('/:id',userController.deleteUser);

module.exports = router;
```

2. Controller (src/controllers/userController.js)

```
let users = [];
let id = 1;
exports.getUsers = (req, res) => {
  res.json(users);
};
exports.getUserById = (req, res) => {
  const user = users.find(u => u.id == req.params.id);
if (!user) return res.status(404).json({
    message: 'User not found'
  });
  res.json(user);
};
exports.createUser = (req, res) => {
  const newUser = { id: id++, ...req.body };
  users.push(newUser);
  res.status(201).json(newUser);
// Similarly: updateUser and deleteUser ...
```

9 Use Routes in App

Ans →

src/app.js

```
const express = require('express');
const app = express();
const userRoutes = require('./routes/userRoutes');

app.use(express.json());
app.use('/api/users',userRoutes);// prefix routes

app.get('/', (req, res) => {
  res.send('API is running...');
});

module.exports = app;
```

10 Test Your API

Ans →

- Start the server: npm run dev
- Use Postman, Thunder Client (VS Code), or curl to test.
- Example requests:

Method	Endpoint	Description
GET	/api/users	Get all users
GET	/api/users/1	Get user by ID
POST	/api/users	Create a user
PUT	/api/users/1	Update user
DELETE	/api/users/1	Delete user

Base URL: http://localhost:3000

11 Best Practices

Ans →

- Use meaningful route names and HTTP methods
- Validate request data (e.g., using express-validator or Joi)
- Use environment variables (dotenv)
- Handle errors properly
- Use a consistent response format
- Follow MVC structure (routes, controllers, models)
- Add authentication (JWT) for secured APIs
- Use logging (morgan)
- Document your API (Swagger / OpenAPI)
- Write tests (Jest, Supertest)
- Follow REST naming conventions (use nouns)
- Use proper status codes
- Keep code modular and clean

12 Next Topics

Ans →

- Connecting to a database (MongoDB / PostgreSQL)
- Input validation and error handling
- Authentication and Authorization (JWT)
- API Documentation (Swagger)
- Deploying REST APIs (Render, Vercel, etc.)

Karyvio
LEARN | BUILD | GROW | GET HIRED

API Error Handling in Express.js

Build reliable and user-friendly APIs with proper error handling

Good Error Handling Makes Your API Stable, Secure and Developer Friendly !

1 Why Error Handling ?

Ans →
- Prevents server crashes
- Provides meaningful error messages
- Improves user experience
- Helps in debugging and monitoring
- Makes APIs more secure and reliable

2 Types of Errors

Ans →
- Synchronous errors (try...catch)
- Asynchronous errors (promises, async/await)
- Validation errors (invalid input)
- Not found errors (resource not found)
- Authentication/Authorization errors
- Database errors (e.g. MongoDB, PostgreSQL)
- Custom application errors

3 HTTP Status Codes

Ans →

Code	Meaning
200	OK (Success)
400	Bad Request (Invalid input)
401	Unauthorized (Invalid token)
403	Forbidden (No permission)
404	Not Found (Resource not found)
409	Conflict (Already exists)
500	Internal Server Error (Server issue)

4 Basic Try-Catch (Synchronous)

Ans →

```
app.get('/sync-error', (req, res) => {
  try {
    const x = 10 / 0;
    res.send('Result: ' + x);
  } catch (error) {
    res.status(500).json({
      success: false,
      message: 'Something went wrong',
      error: error.message
    });
  }
});
```

5 Handling Async Errors

Ans →

```
app.get('/async-error', async (req, res) => {
  try {
    const user = await User.findById(req.params.id);
    if (!user) {
      return res.status(404).json({
        success: false,
        message: 'User not found'
      });
    }
    res.json({ success: true, data: user });
  } catch (error) {
    res.status(500).json({
      success: false,
      message: 'Server error',
      error: error.message
    });
  }
});
```

6 Async Handler (Cleaner)

Ans → We can create a wrapper to catch async errors automatically.

// utils/asyncHandler.js

```
const asyncHandler = (fn) => (req, res, next) => {
  Promise.resolve(fn(req, res, next)).catch(next);
};

// Usage
const asyncHandler = require('./utils/asyncHandler');

app.get('/users/:id', asyncHandler(async (req, res) => {
  const user = await User.findById(req.params.id);
  if (!user) return res.status(404).json({
    message: 'User not found' });
  res.json(user);
}));
```

7 Global Error Handling Middleware

Ans →

```
// Place this after all routes
app.use((err, req, res, next) => {
  console.error(err.stack);

  const status = err.statusCode || 500;
  const message = err.message || 'Internal Server Error';

  res.status(status).json({
    success: false,
    message,
    stack: process.env.NODE_ENV === 'development'
        ? err.stack
        : undefined
  });
});
```

8 Custom Error Class

Ans → Create a reusable error class for consistent errors.

// utils/AppError.js

```
class AppError extends Error {
 constructor(message, statusCode = 500, isOperational = true) {
   super(message);
   this.statusCode = statusCode;
   this.isOperational = isOperational;
   Error.captureStackTrace(this, this.constructor);
 }

module.exports = AppError;
```

// Usage in route

```
const AppError = require('./utils/AppError');
if (!user) {
  throw new AppError('User not found', 404);
}
```

9 Handling Validation Errors

Ans → Using express-validator (or any validation library).

```
const { body, validationResult } =
  require('express-validator');

app.post('/users',
 body('name').notEmpty().withMessage('Name is required'),
  (req, res, next) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.status(400).json({
        success: false,
        message: 'Validation failed',
        errors: errors.array()
      });
    }
    // create user...
  }
);
```

10 Best Practices

Ans →
- Use meaningful error messages.
- Use correct HTTP status codes.
- Create a global error handler.
- Use a custom error class.
- Handle both sync and async errors.
- Do not expose sensitive error details in production.
- Log errors (use tools like Winston, Morgan, etc.).
- Validate input data.
- Keep error response format consistent.
- Test error scenarios.

Example Error Response

```
{
  "success": false,
  "message": "User not found",
  "error": "User with this ID does not exist"
}
```

Next Topics :-
- Logging with Winston / Morgan
- Input Validation (Joi / express-validator)
- Authentication & Authorization Errors
- Database Error Handling (MongoDB / PostgreSQL)
- API Response Formatting
- Monitoring (Sentry, Logs, APM)
- Building a Complete Error Handling Module

Karyvio
LEARN | BUILD | GROW | GET HIRED

API Validation & Security

Make your APIs secure, reliable and production ready

Validate Input
Secure Your APIs
Prevent Attacks
Build Production
Ready Backends

1 Why Validation & Security ?

Ans →
- Prevents invalid or malicious input.
- Protects your application from attacks (e.g., SQL injection, XSS, NoSQL injection).
- Improves data quality and reliability.
- Builds trust and makes your API production ready.
- Essential for real-world applications.

2 What to Validate ?

Ans →
- Request body (POST/PUT/PATCH)
- Query parameters (e.g., ?page=1)
- Route parameters (e.g., /user/:id)
- Headers (e.g., Authorization)
- File uploads (type, size)
- Use validation libraries like express-validator or Joi.

3 Security Best Practices

Ans →
- Validate all inputs.
- Use authentication (JWT, OAuth).
- Use authorization (role-based access).
- Protect against common attacks (XSS, SQLi).
- Use environment variables for secrets.
- Enable HTTPS in production.
- Use security middleware (helmet, cors, rate-limit).
- Sanitize inputs and escape output.
- Keep dependencies updated.

4 Install Validation Library

Ans → • We will use express-validator.

Install package :-

```
npm install express-validator
```

5 Validate Request Data

Ans → • Example: Validate user registration data.

routes/userRoutes.js

```
const { body, validationResult } =
  require('express-validator');
app.post('/register', [
  body('name').isLength({ min: 3 })
    .withMessage('Name must be at least 3 characters'),
  body('email').isEmail().withMessage('Invalid email'),
  body('password').isLength({ min: 6 })
    .withMessage('Password must be at least 6 characters')
], (req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }
  res.json({ message: 'Validation passed!' });
});
```

6 Example Error Response

Ans →

```
{
  "errors": [
    { "msg": "Name must be at least 3
      characters", "param": "name" },
    { "msg": "Invalid email", "param": "email" }
  ]
}
```

Tip :-
- Always return clear and user-friendly error messages.
- Use consistent error response format.

7 Sanitize Input (Prevent XSS)

Ans →
- Use express-mongo-sanitize or dompurify (for frontend) to remove malicious input.

Install package :-

```
npm install express-mongo-sanitize
```

Use in app.js :-

```
const mongoSanitize = require('express-mongo-sanitize');
app.use(mongoSanitize());
```

8 Authentication with JWT

Ans →
- JWT (JSON Web Token) is used for secure authentication.
- Client sends token in headers for protected routes.

Generate Token (Login) :-

```
const jwt = require('jsronaktoken');

const token = jwt.sign(
  { id: user.id, role: user.role },
   process.env.JWT_SECRET,
  { expiresIn: '1d' }
);

res.json({ token });
```

9 Protect Routes (Middleware)

Ans → • Create a middleware to verify JWT.

middleware/auth.js

```
const jwt = require('jsronaktoken');

module.exports = (req, res, next) => {
  const token = req.headers.authorization?split(' ')[1];
  if (!token) return res.status(401).json({
    message: 'No token provided'
  });
  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded;
    next();
  } catch (err) {
    return res.status(401).json({
      message: 'Invalid token';
  } );
};
```

10 Authorization (Role-Based Access)

Ans →
- Restrict access based on user roles.

middleware/role.js

```
module.exports = (role) => {
  return (req, res, next) => {
    if (req.user.role !== role) {
      return res.status(403).json({
        message: 'Access denied'
      })
    }
    next();
} };
```

Use in route :-

```
app.get('/admin', auth, role('admin'), (req, res) => {
  res.send('Admin Dashboard');
});
```

11 Security Middleware

Ans →
- Use helmet, cors and rate limiting for extra security.

Install packages :-

```
npm install helmet cors express-rate-limit
```

Use in app.js :-

```
const helmet = require('helmet');
const cors = require('cors');
const rateLimit = require('express-rate-limit');

app.use(helmet());
app.use(cors());

const limter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100 // limit each IP to 100 requests
});
app.use(limiter);
```

12 Common Security Threats

Ans →

Threat	Prevention
SQL Injection	Use ORM (Prisma/Drizzle), validate input
NoSQL Injection	Sanitize input (express-mongo-sanitize)
XSS (Cross-site Scripting)	Escape output, sanitize input
CSRF	Use CSRF tokens (for web apps)
Brute Force	Use rate limiting
Data Exposure	Use environment variables, .gitignore
Insecure Dependencies	Keep packages updated

Key Takeaways :-
- Always validate and sanitize user input.
- Use authentication (JWT) and authorization.
- Enable security middleware (helmet, cors, rate-limit).
- Handle errors properly without exposing sensitive details.
- Follow security best practices for production.
- Keep your dependencies and environment secure.

Karyvio
LEARN | BUILD | GROW | GET HIRED

MongoDB Introduction

A flexible, scalable, NoSQL database for modern applications

Flexible Schema
High Performance
Easy to Scale
Perfect for Modern Apps

1 What is MongoDB ?

Ans →

- MongoDB is a NoSQL, document-oriented database.
- Stores data in flexible, JSON-like documents (BSON format).
- Designed for high performance, scalability, and ease of development.
- Widely used in modern web and mobile applications.
- Ideal for applications with changing data structures.

2 Key Features

Ans →

- Document-oriented (JSON-like)
- Flexible schema
- High performance
- Horizontal scalability (sharding)
- Built-in replication (high availability)
- Rich query language
- Supports indexing, aggregation, and geospatial queries
- Works great with Node.js, Express, and other modern frameworks

3 SQL vs MongoDB

Ans

Feature	SQL (Relational)	MongoDB (NoSQL)
Data Model	Tables	Collections (Documents)
Schema	Fixed	Flexible
Query Language	SQL	MongoDB Query (JSON)
Relationships	Joins	Embedded Documents (or References)
Scaling	Vertical	Horizontal (Sharding)
Best For	Structured data	Dynamic / unstructured data

4 Install and Run MongoDB

Ans →

Using MongoDB Community Server (Local)

```
# Download from https://www.mongodb.com/try/download/community
# or use package manager (example for Ubuntu)
sudo apt update
sudo apt install -y mongodb

# Start MongoDB service
sudo systemctl start mongodb
sudo systemctl enable mongodb

# Check status
sudo systemctl status mongodb
```

5 Basic Concepts

Ans →

- **Database** – Container for collections.
- **Collection** – Group of documents (similar to a table).
- **Document** – A single record (JSON-like object).
- **Field** – Key-value pair inside a document.
- **_id** – Unique identifier for each document (automatically generated).

6 Example Document

Ans →

```
{
  "_id": ObjectId("64f8c2e5b1a7d3e9c8a1b2c3"),
  "name": "Sovit Lenka",
  "email": "sovit@example.com",
  "age": 21,
  "skills": ["JavaScript", "Node.js", "MongoDB"],
  "isActive": true,
  "createdAt": ISODate("2026-09-16T10:00:00Z")
}
```

7 Using MongoDB Shell

Ans →

```
# Start MongoDB shell
mongosh

# Show databases
show dbs

# Create/switch database
use karyvio

# Show collections
show collections

# Insert a document
db.users.insertOne({ name: "Sovit", age: 21 })

# Find documents
db.users.find()

db.users.find({ age: { $gt: 18 } })
```

8 Common MongoDB Operations

Ans →

```
// Insert
db.users.insertOne({ name: "Smruti", age: 22 })

// Find
db.users.find({ age: { $gt: 18 } })

// Update
db.users.updateOne(
  { name: "Smruti" },
  { $set: { age: 23 } }
)

// Delete
db.users.deleteOne({ name: "Smruti" })
```

9 MongoDB with Node.js (Mongoose)

Ans →

```
// Install mongoose
npm install mongoose

// models/User.js
const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({
  name: String,
  email: String,
  age: Number,
  skills: [String]
});

module.exports = mongoose.model('User', userSchema);

// Usage in app.js
const User = require('./models/User');

const newUser = await User.create({
  name: "Sovit",
  email: "sovit@example.com",
}) age: 21
```

10 Best Practices

Ans →

- Use proper indexing for faster queries.
- Validate data using Mongoose schemas.
- Handle errors gracefully.
- Use environment variables for connection strings.
- Keep documents small and focused.
- Use aggregation for complex queries.
- Enable authentication in production.
- Take regular backups.

11 Use Cases

Ans →

- Web and mobile applications
- Real-time data (chat, notifications)
- Content management systems (CMS)
- E-commerce platforms
- IoT and sensor data
- User management and authentication
- Any application with dynamic or rapidly changing data

12 Next Topics

Ans →

- MongoDB Query Operators
- Indexing and Performance
- Aggregation Pipeline
- Relationships (Embedding vs Referencing)
- MongoDB Atlas (Cloud)
- Backup and Restore
- Real-world Project with MongoDB

MongoDB.

Karyvio
LEARN | BUILD | GROW | GET HIRED

MongoDB CRUD Operations

Perform Create, Read, Update and Delete operations in MongoDB

CRUD
Create Data
Read Data
Update Data
Delete Data
Powerful Applications !

1 What is CRUD ?

Ans →
- CRUD stands for Create, Read, Update and Delete.
- These are the basic operations to manage data in a database.
- MongoDB provides powerful methods for CRUD using its query language and drivers (like the Node.js driver).
- Used to build real-world applications like blogs, e-commerce, and more.

2 Setup & Connect

Ans →
- Install MongoDB locally (or use MongoDB Atlas).
- Use the official MongoDB Node.js driver or Mongoose (ODM).
- Create a connection to your database.

Example :- Connect using Mongoose

```
const mongoose = require('mongoose');

mongoose.connect('mongodb://localhost:27017/
karyvio', {
  useNewUrlParser: true,
  useUnifiedTopology: true
})
.then(() => console.log('✅ Connected to MongoDB'))
.catch(err => console.error('❌ Connection error:', err));
```

3 Sample Data Model

Ans → Let's create a simple User model.

models/User.js

```
const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({
  name: String,
  email: { type: String, unique: true },
  age: Number,
  createdAt: { type: Date, default: Date.now }
});

module.exports = mongoose.model('User', userSchema);
```

4 Create (Insert)

Ans → Insert a new document using create() or new Model().save().

Example :- Create a new user

```
const User = require('./models/User');

// Using create()
const newUser = await User.create({
  name: 'Sovit Lenka',
  email: 'sovit@example.com',
  age: 22
});

console.log('User created:', newUser);
```

Inserted Document (Output)

```
{
  _id: ObjectId("68c1a8f3e1b2c3d4e5f67890"),
  name: "Sovit Lenka",
  email: "sovit@example.com",
  age: 22,
  createdAt: "2026-09-16T10:00:00.000Z"
}
```

5 Read (Find)

Ans → Retrieve documents using find(), findOne(), etc.

Examples :- Read data

```
const User = require('./models/User');

// Find all users
const users = await User.find();
console.log(users);

// Find one user by ID
const user = await User.findById('68c1a8f3e1b2c3d');
console.log(user);

// Find user by condition
const youngUsers = await User.find({ age: { $lt: 25 } });
console.log(youngUsers);

// Find with specific fields
const usersWithName = await User.find({}, {
 name: 1, email: 1 });
console.log(usersWithName);
```

6 Update

Ans → Update existing documents using updateOne(), updateMany() or findByIdAndUpdate().

Examples :- Update data

```
const User = require('./models/User');

// Update one document
await User.updateOne(
  { _id: '68c1a8f3e1b2c3d' },
  { $set: { age: 23 } }
);

// Or using findByIdAndUpdate (returns updated doc)
const updatedUser = await User.findByIdAndUpdate(
  '68c1a8f3e1b2c3d',
  { $set: { name: 'Sovit S. Lenka' } },
  { new: true }
);
console.log('Updated User:', updatedUser);
```

Update Operators :-

$set	→ Set field value
$inc	→ Increment number
$unset	→ Remove field

7 Delete

Ans → Delete documents using deleteOne(), deleteMany() or findByIdAndDelete().

Examples :- Delete data

```
const User = require('./models/User');

// Delete one document
await User.deleteOne({ _id: '68c1a8f3e1b2c3d' });

// Delete by condition
await User.deleteMany({ age: { $lt: 18 }});

// Or using findByIdAndDelete (returns deleted doc)
const deletedUser = await User.findByIdAndDelete(
  '68c1a8f3e1b2c3d'
);
console.log('Deleted User:', deletedUser);
```

8 Common Query Operators

Ans → Use MongoDB query operators for powerful queries.

Operator	Description	Example
$eq	Equal to	{ age: { $eq: 22 } }
$ne	Not equal	{ age: { $ne: 22 } }
$gt	Greater than	{ age: { $gt: 18 } }
$lt	Less than	{ age: { $lt: 30 } }
$in	In array	{ age: { $in: [20, 22] } }
$regex	Pattern match	{ name: { $regex: 'Sovit' } }
$and	Logical AND	{ $and: [{ age: 22 }, { name: 'Sovit' }] }
$or	Logical OR	{ $or: [{ age: 18 }, { age: 22 }] }

9 Example: Complete CRUD Flow

Ans →
1. Create → Add a new user
2. Read → Fetch the user
3. Update → Modify user details
4. Delete → Remove the user

10 Best Practices

Ans →
- Use validation (Mongoose schema validation).
- Handle errors with try-catch.
- Use proper indexes for better performance.
- Avoid fetching unnecessary fields.
- Use pagination for large datasets.
- Sanitize input to prevent injection attacks.
- Use environment variables for DB connection.
- Keep your database schema organized.

11 Next Topics

Ans →
- Mongoose Advanced Queries
- Aggregation Pipeline
- Indexing and Performance
- Relationships (References & Embedding)
- Transactions in MongoDB
- MongoDB Atlas (Cloud)
- Building a REST API with MongoDB

Karyvio
LEARN | BUILD | GROW | GET HIRED

MongoDB Data Modeling

Design efficient and scalable data models for real-world applications

Good Data Model Makes Your App Faster, Scalable and Easy to Maintain !

1 What is Data Modeling ?

Ans →
- Data modeling is the process of designing the structure of data in a database.
- It defines how data is stored, related, and accessed.
- In MongoDB, we design schemas (usually in application code) to represent data structure.
- A good data model improves performance, scalability, and maintainability.

2 Why is it Important ?

Ans →
- Helps in organizing data efficiently.
- Reduces data duplication.
- Improves query performance.
- Makes the application scalable.
- Simplifies maintenance.
- Supports real-world use cases (e.g., users, products, orders, etc.).

3 Key Principles

Ans →
- Understand your application requirements.
- Model for your query patterns.
- Balance between normalization and denormalization.
- Avoid over-complicating the schema.
- Plan for scalability and future growth.
- Keep data consistent and efficient.

4 Types of Relationships

Ans →

Type	Description	Example
One-to-One	One document related to one document	User ↔ Profile
One-to-Many	One document related to many documents	User ↔ Posts
Many-to-Many	Many documents related to many documents	Users ↔ Courses

5 Embedding vs Referencing

Ans →

Aspect	Embedding	Referencing
Data Storage	Store related data in the same document	Store a reference (ObjectId) to another document
Performance	Faster (single query)	Slower (requires lookup)
Use Case	When related data is frequently accessed together	When data is large or used separately
Example	User with address embedded	User with orders referenced

6 Example: Embedded Model

Ans →

```
// User with embedded address
const user = {
  _id: new ObjectId(),
  name: "Sovit Lenka",
  email: "sovit@example.com",
  address: {
    street: "123 Main Road",
    city: "Bhubaneswar",
    state: "Odisha",
    pincode: "751001"
  }
};
```

7 Example: Referenced Model

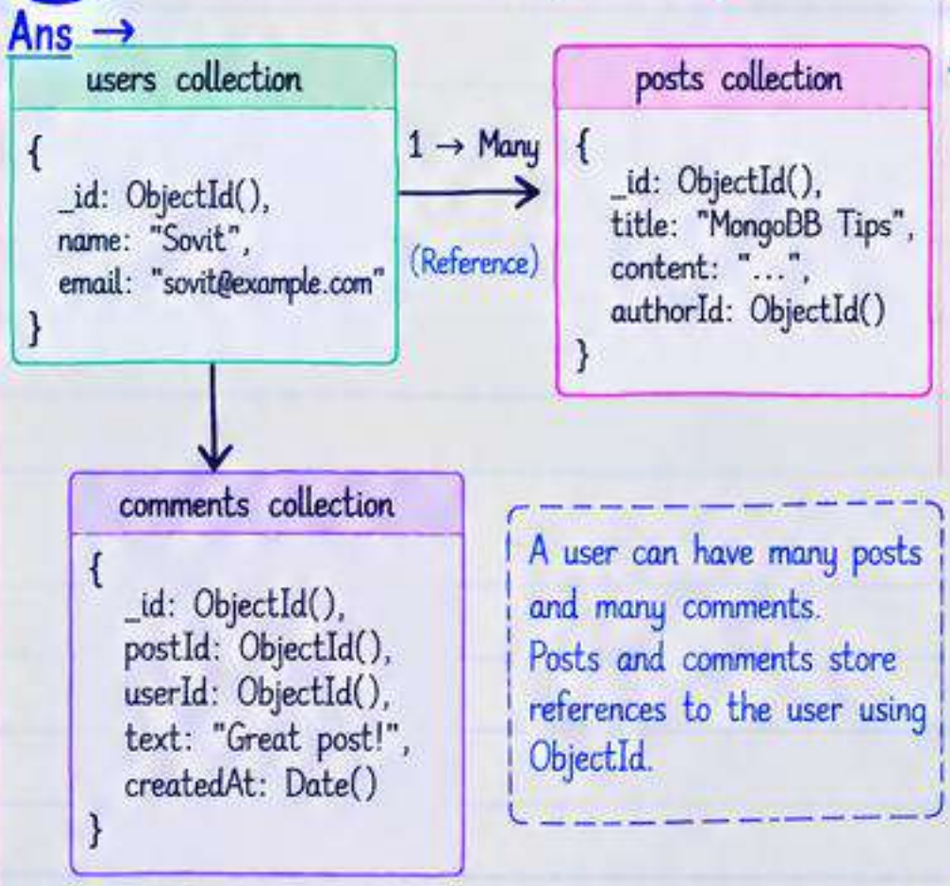
Ans →

```
// User document
const user = {
  _id: new ObjectId(),
  name: "Sovit Lenka",
  email: "sovit@example.com",
  profileId: new ObjectId("64f8c2e5b1a7d3e9"),
};
```

```
// Profile document
const profile = {
  _id: new ObjectId("64f8c2e5b1a7d3e9"),
  bio: "Full Stack Developer",
  skills: ["JavaScript", "Node.js", "MongoDB"],
  socialLinks: {
    github: "https://github.com/sovit8917",
    linkedin: "https://linkedin.com/in/sovit"
  }
};
```

8 Real-World Example: Blog

Ans →



9 Best Practices

Ans →
- Model based on query patterns.
- Use embedding for small, frequently accessed related data.
- Use referencing for large or rarely accessed data.
- Avoid deep nesting.
- Keep document size under 16MB (limit per document).
- Use indexes on commonly queried fields.
- Consider data growth and future requirements.
- Use consistent naming conventions.
- Validate data using Mongoose schemas.

10 Common Pitfalls

Ans →
- Over-embedding large data (leads to huge documents).
- Excessive referencing (causes many lookups).
- Not using indexes.
- Ignoring document size limit (16MB).
- Poor field naming and inconsistent schema.
- Not planning for scalability.
- Storing unstructured or unnecessary data.

11 When to Embed vs Reference ?

Ans →

Use Case	Embed	Reference
Small related data	✓	-
Large related data	-	✓
Frequently accessed together	✓	-
Rarely accessed data	-	✓
One-to-One	Either	Either
One-to-Many / Many-to-Many	Usually	Usually

12 Next Topics

Ans →
- Mongoose Schema Design
- Validation in Mongoose
- Indexing in MongoDB
- Aggregation Pipeline
- Relationship Modeling (Advanced)
- Polymorphic Associations
- Multi-tenant Data Modeling
- Real-world Project Design

MongoDB.

Karyvio
LEARN | BUILD | GROW | GET HIRED

Mongoose — MongoDB ODM for Node.js

A powerful and elegant way to work with MongoDB in Node.js applications

Schemas
Models
Validation
Middleware
and more...
with Mongoose !

1 What is Mongoose ?

Ans →

- Mongoose is an ODM (Object Data Modeling) library for MongoDB and Node.js.
- It provides a schema-based solution to model application data.
- Handles validation, type casting, relationships, middleware, and more.
- Makes it easier to work with MongoDB using JavaScript/TypeScript.

2 Why Use Mongoose ?

Ans →

- Schema-based data modeling
- Built-in validation
- Type casting (automatically converts data types).
- Middleware (pre/post hooks).
- Query helpers and instance methods
- Better error handling
- Active community and widely used in production

3 Install Mongoose

Ans → Install using npm :

```
npm install mongoose
```

Or using yarn :

```
yarn add mongoose
```

4 Connect to MongoDB

Ans →

```
const mongoose = require('mongoose');

const connectDB = async () => {
  try {
    await mongoose.connect(
      'mongodb://localhost:27017/karyvio',
      {
        useNewUrlParser: true,
        useUnifiedTopology: true,
      }
    );
    console.log('✅ MongoDB Connected');
  } catch (error) {
    console.error('❌ Connection Error:',
    error.message);
};
module.exports = connectDB;
```

5 Define a Schema

Ans →

```
const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true,
  unique: true },
  age: { type: Number, min: 0 },
  isActive: { type: Boolean, default: true },
}, {
  timestamps: true // adds createdAt, updatedAt
});
```

6 Create a Model

Ans →

```
const mongoose = require('mongoose');
const userSchema = require('./userSchema');

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

module.exports = User;
```

Model Syntax :-

`mongoose.model('ModelName', schema)`

- Model name is usually singular (e.g., User).
- Mongoose automatically creates the collection name in lowercase and plural (users).

7 Create a Document (Insert)

Ans →

```
const User = require('./models/User');

const createUser = async () => {
  try {
    const user = await User.create({
      name: 'Sovit Lenka',
      email: 'sovit@example.com',
      age: 22
    });
    console.log('User created:', user);
  } catch (error) {
    console.error('Error:', error.message);
  }
```

8 Read Documents (Find)

Ans →

```
const User = require('./models/User');

// Find all users
const users = await User.find();

// Find by ID
const user = await User.findById('68c1a8f3e1b2c3d');

// Find with conditions
const activeUsers = await User.find({ isActive: true })
  .select('name email')
  .sort({ createdAt: -1 });
```

9 Update Documents

Ans →

```
// Update by ID
const updatedUser = await User.findByIdAndUpdate(
  '68c1a8f3e1b2c3d',
  { age: 23 },
  { new: true, runValidators: true }
));

// Update many
await User.updateMany(
  { isActive: false },
  { isActive: true }
);
```

10 Delete Documents

Ans →

```
// Delete by ID
await User.findByIdAndDelete('68c1a8f3e1b2c3d');

// Delete many
await User.deleteMany({ isActive: false });

// Delete with condition
await User.deleteOne({ email: 'test@example.com' });
```

11 Validation in Mongoose

Ans →

```
const userSchema = new mongoose.Schema({
  name: { type: String, required: [true, 'Name
  is required'], millength: 3 },
  email: { type: String, required: true,
    unique: true, match: /^[\w.-]+@[\w.-]+\.[a
  -z]{2,3}$/ },
  age: { type: Number, min: [18, 'Must be at
  least 18'], max: 100 }
});
```

12 Middleware (Hooks)

Ans →

```
// Pre-save middleware
userSchema.pre('save', function (next) {
  console.log('Saving user:', this.name);
  next();
});
// Post-save middleware
userSchema.post('save', function (doc) {
  console.log('User saved:', doc._id);
});
```

13 Instance Methods

Ans →

```
userSchema.methods.getProfile = function () {
  return `${this.name} (${this.email})`;
};
```

14 Static Methods

Ans →

```
userSchema.statics.findByEmail = function (email) {
  return this.findOne({ email });
};
```

15 Best Practices

Ans →

- Use schemas with proper validation.
- Handle errors with try-catch.
- Use indexes for frequently queried fields.
- Keep schema definitions modular.
- Use environment variables for DB connection.
- Follow consistent naming conventions.
- Use TypeScript for better type safety.
- Keep your models organized in a separate folder.
- Use middleware for logging, hashing, etc.

Karyvio
LEARN | BUILD | GROW | GET HIRED

MongoDB Advanced Queries

Powerful query techniques to handle real-world data efficiently

Query Smarter
Handle Large Data
Build Scalable Apps
with MongoDB

1 What are Advanced Queries ?

Ans → Advanced queries go beyond basic CRUD to perform complex data retrieval, filtering, transformation and analysis.

- Used for real-world applications with large and unstructured data.
- Helps in building search, analytics, reporting and scalable features.

2 Query Operators Overview

Ans →

Category	Operators	Purpose
Comparison	$eq, $ne, $gt, $gte, $lt, $lte	Compare values
Logical	$and, $or, $not	Combine conditions
Element	$in, $nin	Match in array
Evaluation	$exists, $type	Check field existence/type
Array	$all, $elemMatch, $size	Work with arrays
Text	$text	Full-text search
Geospatial	$geoNear, $geoWithin	Location based queries

3 Comparison Operators

Ans →

```
// Find users older than 22
db.users.find({ age: { $gt: 22 } })

// Find users with age between 18 and 25
db.users.find({
   age: { $gte: 18, $lte: 25 }
})
```

Common Comparison Operators:

- $eq → Equal
- $ne → Not equal
- $gt → Greater than
- $gte → Greater than or equal
- $lt → Less than
- $lte → Less than or equal
- $in → In an array
- $nin → Not in an array

4 Logical Operators

Ans →

```
// Users age > 18 AND skill = Node.js
db.users.find({
  $and: [
    { age: { $gt: 18 } },
    { "skills": "Node.js" }
  ]
})

// Users from Odisha OR remote location
db.users.find({
  $or: [
    { state: "Odisha" },
    { workMode: "Remote" }
  ]
})
```

5 Querying Arrays

Ans →

```
// Users with a specific skill (element in array)
db.users.find({ skills: "JavaScript" })

// Users with multiple skills
db.users.find({
  skills: { $all: ["JavaScript", "Node.js"] }
})

// Using $elemMatch for complex array query
db.users.find({
  skills: {
    $elemMatch: { name: "MongoDB", level: "Advanced" }
  }
})

// Find users with exactly 3 skills
db.users.find({ skills: { $size: 3 } })
```

6 Projection (Select Fields)

Ans →

```
// Return only name and email
db.users.find(
  {},
  { name: 1, email: 1, _id: 0 }
)

// Exclude sensitive fields
db.users.find(
  {},
  { password: 0, refreshToken: 0 }
)

// Rename a field using aggregation (see next)
// Projection keeps responses smaller and faster.
```

Tip :
- 1 = include field
- 0 = exclude field
  (You cannot mix 1 and 0, except _id)

7 Sorting, Limiting & Skipping

Ans →

```
// Get latest 10 users
db.users.find()
  .sort({ createdAt: -1 }) // -1 desc, 1 asc
  .limit(10)

// Skip first 20 and get next 10
db.users.find()
  .sort({ createdAt: -1 })
  .skip(20)
  .limit(10)
```

8 Text Search

Ans →

```
// Create text index (run once)
db.posts.createIndex({
  title: "text",
  content: "text"
})
// Search posts
db.posts.find({
  $text: { $search: "mongodb tutorial" }
})
// Text search with projection and sort
db.posts.find(
  { $text: { $search: "node mongodb" } },
  { score: { $meta: "textScore" }
).sort({ score: { $meta: "textScore" } })
```

9 Aggregation Pipeline

Ans →

```
// Count users by skill
db.users.aggregate([
  { $unwind: "$skills" },
  { $group: { _id: "$skills", count: { $sum: 1 } } },
  { $sort: { count: -1 } }
])

// Get average age by country
db.users.aggregate([
  { $group: { _id: "$country", avgAge: { $avg: "$age" } } },
  { $sort: { avgAge: -1 } }
])
```

10 Geospatial Queries

Ans →

```
// Find users within 5 km of a location
db.users.find({
  location: {
    $near: {
      $geometry: { type: "Point", coordinates: [85.8245, 20.2961] },
      $maxDistance: 5000
    }
  }
})
```

11 Real-World Use Case

Ans →

Example: Job Search

- Filter by location, skills, experience range using advanced queries.
- Use text search for job title/description.
- Apply sorting, pagination and projection for better performance.
- Use aggregation for analytics (e.g., top skills, jobs by company, etc.).

12 Best Practices

Ans →

- Use indexes for frequently queried fields.
- Use projection to fetch only required data.
- Avoid deep nesting in queries.
- Use aggregation for complex analytics.
- Validate input to prevent NoSQL injection.
- Keep queries efficient and optimized.
- Use explain() to analyze query performance.
- Follow schema design best practices.
- Test queries with realistic data.

Karyvio.com

Better Queries. Bigger Possibilities !

MongoDB.
LEARN | BUILD | GROW | GET HIRED

Karyvio
LEARN | BUILD | GROW | GET HIRED

MongoDB Indexing & Performance

Speed up your queries with indexing and performance optimization techniques

Better Indexes
Faster Queries
Scalable Applications
with MongoDB

1 Why Indexing is Important ?

Ans →

- Indexes improve query performance by reducing the number of documents MongoDB needs to scan.
- Helps in faster search, sorting, and filtering.
- Essential for large datasets.
- Improves overall application responsiveness.
- Without indexes, MongoDB performs a collection scan (slow).

2 How Indexes Work ?

Ans →

- Indexes in MongoDB are stored as a **B-Tree** data structure.
- The index stores a reference to the document (not the full document).
- MongoDB uses the index to quickly locate matching documents.
- Similar to an index in a book — it helps find data without reading every page.

3 Create an Index

Ans →

```
// Single field index (ascending)
db.users.createIndex({ email: 1 })

// Single field index (descending)
db.users.createIndex({ createdAt: -1 })

// Compound index
db.users.createIndex({ name: 1, age: -1 })

// Unique index
db.users.createIndex({ email: 1 }, { unique: true })
```

4 Types of Indexes

Ans →

Index Type	Description
Single Field	Index on a single field
Compound	Index on multiple fields
Unique	Ensures unique values
Sparse	Index only documents with the field
TTL (Time to Live)	Automatically delete documents after a specified time
Text	For full-text search
Geospatial	For location-based queries (2dsphere, 2d)
Hashed	Uses a hash of the field value

5 Unique Index Example

Ans →

```
// Ensure each email is unique
db.users.createIndex(
  { email: 1 },
  { unique: true }
)

// Try inserting duplicate
db.users.insertOne({
  name: "Sovit",
  email: "sovit@example.com"
})
// Duplicate email will throw an error.
```

6 TTL Index Example

Ans →

```
// Automatically delete documents
  after 7 days
db.logs.createIndex(
  { createdAt: 1 },
  { expireAfterSeconds: 7 * 24 * 60 * 60 }
)
```

Use Case :

- Temporary data (logs, sessions)
- OTP records
- Cache data

7 Text Index Example

Ans →

```
// Create a text index on title and content
db.articles.createIndex(
  { title: "text", content: "text" }
)

// Search using text index
db.articles.find({
  $text: { $search: "mongodb performance" }
})
```

8 Check & Manage Indexes

Ans →

```
// List all indexes in a collection
db.users.getIndexes()

// Drop a specific index
db.users.dropIndex("email_1")

// Drop all indexes (except _id)
db.users.dropIndexes()
```

9 Analyze Query Performance

Ans →

```
// Use explain() to see how MongoDB
  executes a query
db.users.find( { age: { $gt: 18 } }))
  .explain("executionStats")

// Look for:
// - whether an index was used
// - number of documents scanned
// - execution time
```

10 Common Performance Tips

Ans →

- Create indexes on frequently queried fields.
- Use compound indexes for multi-field queries.
- Avoid over-indexing (too many indexes can slow down write operations).
- Use appropriate index types (text, geospatial, etc.).
- Monitor query performance using explain().
- Avoid large documents and unbounded arrays.
- Use projection to fetch only required fields.
- Regularly review and optimize indexes.

11 Indexing Best Practices

Ans →

- ✅ Index fields used in queries, sorting, and filtering.
- ✅ Use compound indexes when querying multiple fields.
- ✅ Keep index size small (avoid unnecessary fields).
- ✅ Use unique indexes for fields that must be unique.
- ✅ Monitor index usage and remove unused indexes.
- ✅ Use covered queries (query can be satisfied using only the index).
- ✅ Test with explain() and analyze execution plans.

12 Real-World Example: User Search

Ans →

```
// Create compound index for search
db.users.createIndex({
  name: 1,
  skills: 1,
  location: 1
})
// Search with filters
db.users.find({
  skills: "Node.js",
  location: "Bhubaneswar"
}).sort({ name: 1 }).limit(10)
```

Karyvio.com

Optimize Today, Scale for Tomorrow ! ☺

mongoDB.
LEARN | BUILD | GROW | GET HIRED

Karyvio
LEARN | BUILD | GROW | GET HIRED

Express + MongoDB Integration

Build powerful backend applications by combining Express.js with MongoDB

Express + MongoDB a perfect combo for modern web applications !

1 Why Use Express + MongoDB ?

Ans →

- JavaScript everywhere (frontend and backend).
- Flexible and schema-less database.
- Fast development with Express.js.
- Great for REST APIs and real-time apps.
- Scales easily for growing applications.
- Large community and ecosystem.

2 Project Setup

Ans → Create a new project and install dependencies.

```
# Create project folder
mkdir express-mongo-app
cd express-mongo-app

# Initialize npm
npm init -y

# Install dependencies
npm install express mongoose dotenv

# For development (optional)
npm install -D nodemon
```

3 Project Structure

Ans → A clean and modular folder structure.

```
express-mongo-app/
├── config/
│   └── db.js
├── models/
│   └── User.js
├── routes/
│   └── userRoutes.js
├── controllers/
│   └── userController.js
├── .env
├── app.js
└── package.json
```

Keep your project modular and organized for better scalability !

4 MongoDB Connection

Ans → config/db.js

```
const mongoose = require('mongoose');

const connectDB = async () => {
  try {
    await mongoose.connect(process.env.MONGO_URI);
    console.log('✅ MongoDB Connected');
  } catch (error) {
    console.error('❌ Connection Error:',
    error.message);
    process.exit(1);
  }
};
module.exports = connectDB;
```

5 Environment Variables

Ans → Create a .env file (keep it safe).

```
PORT=5000
MONGO_URI=mongodb://localhost:27017/karyvio
NODE_ENV=development
```

Tip :

- Never commit your .env file to GitHub.
- Use different URIs for development and production.
- You can use MongoDB Atlas for cloud database.

6 Define a Model

Ans → models/User.js

```
const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  age: { type: Number, default: 0 },
  createdAt: { type: Date, default: Date.now }
});

module.exports = mongoose.model('User', userSchema);
```

7 Create Routes

Ans → routes/userRoutes.js

```
const exppross = require('expreoss');
const router = express.Router();

const { getUsers, createUser } =
  require('../controllers/userController');

router.get('/', getUsers);
router.post('/', createUser);

module.exports = router;
```

8 Create Controller

Ans → controllers/userController.js

```
const User = require('../models/User');
// Get all users
exports.getUsers = async (req, res) => {
  try {
    const users = await User.find();
    res.json(users);
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
};

// Create a new user
exports.createUser = async (req, res) => {
  try {
    const user = new User(req.body);
    await user.save();
    res.status(201).json(user);
  } catch (error) {
    res.status(400).json({ message: error.message });
  }
};
```

9 Setup Express App

Ans → app.js

```
require('dotenv').config();
const express = require('express');
const connectDB = require('./config/db');
const userRoutes = require('./routes/userRoutes');

const app = express();

// Middleware
app.use(express.json());

// Connect to DB
connectDB();

// Routes
app.use('/api/users', userRoutes);

// Start server
const PORT = process.env.PORT || 5000;
app.listen(PORT, () => {
  console.log('Server running on port ${PORT}');
});
```

10 Test the API

Ans → Run the server and test with Postman or curl.

```
# Start the server
npm run dev  # (if using nodemon)

# Test GET request
curl http://localhost:5000/api/users

# Test POST request
curl -X POST http://localhost:5000/api/users \
-H "Content-Type: application/json" \
-d '{"name":"Sovit","email":"sovit@example.com","age":22}'
```

11 Common Errors & Fixes

Ans →

Error	Solution
Mongoose connection error	Check your MongoDB URI and network connection.
Connot find module	Run npm install again.
CORS error	Use cors middleware if calling from frontend.
Duplicate key error	Check for existing data (e.g., unique email).
Validation error	Add proper schema validation.

12 Best Practices

Ans →

- Use environment variables.
- Structure your code in modules (models, routes, controllers).
- Use proper error handling and env validation.
- Validate input data (e.g., using Joi or express-validator).
- Use MongoDB Atlas for production.
- Follow RESTful API conventions.
- Add logging (e.g., Morgan, Winston).
- Handle async errors properly (use try-catch or async middleware).
- Keep your dependencies updated.

Karyvio
LEARN | BUILD | GROW | GET HIRED

Authentication Fundamentals

Secure Users
Build Trust
Create Better Apps
with Authentication

Learn the core concepts of authentication and build secure user authentication in modern web applications.

1 What is Authentication ?

Ans →

- Authentication is the process of verifying who a user is.
- It ensures that only authorized users can access protected resources.
- Common methods include passwords, OTP, tokens, and biometrics.
- It is different from authorization (which decides what a user can do).

2 Why is Authentication Important ?

Ans →

Protects sensitive user data.

- Prevents unauthorized access.
- Builds trust in your application.
- Enables personalized user experience.
- Required for most real-world applications (e-commerce, banking, social media, etc.).

3 Common Authentication Methods

Ans →

Method	Description
Password	Traditional username + password
OTP (Email/SMS)	One-time password for verification
Token-based (JWT)	Uses a token for stateless auth
OAuth (Social Login)	Login with Google, GitHub, etc.
Biometric	Fingerprint / Face ID
Multi-Factor (MFA)	Two or more verification methods

4 How Authentication Works ?

Ans →

1. User submits credentials (e.g., email & password)
2. Server verifies the credentials (e.g., check in database)
3. If valid, server creates a session or issues a token (e.g., JWT)
4. Client stores the token (in cookie or localStorage)
5. Token is sent with each request to access protected routes
6. Server verifies the token and allows access if valid

5 Password Hashing

Ans

- Never store plain text passwords.
- Use a strong hashing algorithm like bcrypt.
- Hashing converts the password into a fixed-length string.
- Even if the database is leaked, passwords remain safe.

```
// Hash password using bcrypt
const bcrypt = require('bcrypt');

const saltRounds = 10;
const plainPassword = "mySecurePassword";

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

console.log(hashedPassword);
```

6 JSON Web Tokens (JWT)

Ans

- JWT is a token-based authentication method.
- It contains user information in a signed token.
- Stateless (no server-side session needed).
- Consists of 3 parts: Header, Payload, Signature.

Header . Payload . Signature

- Header → Algorithm & Token Type
- Payload → User Data (e.g., userId, role)
- Signature → Used to verify token integrity

```
// Create JWT token
const jwt = require('jsonwebtoken');

const token = jwt.sign(
  { userId: 1, email: "user@example.com" },
  process.env.JWT_SECRET,
  { expiresIn: "1h" }
);
console.log(token);
```

7 Token Verification

Ans →

```
// Verify JWT token
const jwt = require('jsonwebtoken');
try {
  const decoded = jwt.verify(
    token,
    process.env.JWT_SECRET
  );
  console.log('Valid Token:', decoded);
} catch (error) {
  console.error('Invalid Token:',
    error.message);
}
```

8 Protecting Routes (Middleware)

Ans →

```
// Express middleware to protect routes
const jwt = require('jsonwebtoken');

const protect = (req, res, next) => {
  const token = req.headers.authorization;

  if (!token) {
    return res.status(401).json({
      message: "No token provided"
    });
  }
  try {
    const decoded = jwt.verify(
      token.split(' ')[1],
      process.env.JWT_SECRET
    );
    req.user = decoded;
    next();
  } catch (error) {
    res.status(400).json({
      message: "Invalid token"
    });
  }
};
module.exports = protect;
```

9 Best Practices

Ans →

- ✅ Always hash passwords (use bcrypt).
- ✅ Use environment variables for secrets.
- ✅ Set token expiration time.
- ✅ Use HTTPS in production.
- ✅ Implement refresh tokens for better security.
- ✅ Use role-based access control (RBAC).
- ✅ Protect sensitive routes with middleware.
- ✅ Validate input data (e.g., using Joi).
- ✅ Handle authentication errors properly.
- ✅ Consider multi-factor authentication (MFA).

10 Real-World Example (Login)

Ans →

```
// Login route example
app.post('/api/login', async (req, res) => {
  const { email, password } = req.body;
  const user = await User.findOne({ email });
  if (!user) {
    return res.status(400).json({
      message: "User not found"
    });
  }
  const isMatch = await bcrypt.compare(
    password,
    user.password
  );
  if (!isMatch) {
    return res.status(400).json({
      message: "Invalid credentials"
    });
  }
  const token = jwt.sign(
    { userId: user._id, email: user.email },
    process.env.JWT_SECRET,
    { expiresIn: "1h" }
  );
  res.json({ token, user: { id: user._id, email: user.email } });
});
```

11 Next Topics

Ans →

- Refresh Tokens
- Role-Based Access Control (RBAC)
- OAuth (Google/GitHub Login)
- Multi-Factor Authentication (MFA)
- Session Management
- Password Reset (Forgot Password)
- Email Verification
- Security Best Practices

Authentication secures your app today and your users' trust tomorrow ! ☺

12 Key Takeaways

Ans →

- ✅ Authentication verifies who the user is.
- ✅ Use hashed passwords and secure tokens.
- ✅ JWT is a popular token-based method.
- ✅ Protect your routes with middleware.
- ✅ Follow best practices to build secure apps.

"A secure application is a trusted application."

Karyvio
LEARN | BUILD | GROW | GET HIRED

JWT Authentication

Secure, stateless and scalable authentication for modern web applications

Stateless Auth
Secure APIs
Scalable Applications
With JWT

1 What is JWT ?

Ans →

- JWT (JSON Web Token) is an open standard (RFC 7519) for securely transmitting information between parties as a JSON object.
- It is used for authentication and authorization in stateless applications.
- The token contains user information (payload) and is digitally signed to prevent tampering.
- Commonly used in modern web apps, mobile apps, and APIs.

2 Structure of JWT

Ans →

header.payload.signature

Header	Payload	Signature
{ "alg": "HS256", "typ": "JWT" }	{ "userId": "123", "email": "user@example.com", "role": "user", "iat": 1710000000 }	HMACSHA256(base64Url(header) + "." + base64Url(payload), secret)

Example JWT Token (decoded):

```
{
  "userId": "123",
  "email": "user@example.com",
  "role": "user",
  "iat": 1710000000,
} "exp": 1710003600
```

3 Install Required Packages

Ans →

```
npm install jsonwektoben bcryptjs
@types/jsonwektoben

# For TypeScript (optional)
npm install -D @types/bcryptjs
```

Package Purpose:

- jsonwektoben → create & verify tokens
- bcryptjs → hash passwords
- @types/jsonwebtoben → TypeScript types
- @types/bcryptjs → TypeScript types

4 Generate a JWT Token

Ans →

```
import jwt from "jsonwektoben";

const generateToken = (user: any) => {
    const payload = {
      userId: user.id,
      email: user.email,
      role: user.role,
    };
    return jwt.sign(
      payload,
      process.env.JWT_SECRET!,
      { expiresIn: "1h" }
    );
};
```

5 Verify a JWT Token

Ans →

```
import jwt from "jsonwektoben";

export const verifyToken = (token: string) => {
  try {
    const decoded = jwt.verify(
      token,
      process.env.JWT_SECRET!
    );
    return decoded;
  } catch (error) {
    throw new Error("Invalid or expired token");
  }
};
```

6 Protect Routes (Middleware)

Ans →

```
import { Request, Response, NextFunction }
  from "express";
import { verifyToken } from "../utils/jwt";

export const authenticate = (
  req: Request,
  res: Response,
  next: NextFunction
) => {
  const token = req.headers.authorization
    ?.split(" ")[1]; // Bearer <token>
  if (!token) return res.status(401).json({
    message: "No token provided"
  });
  try {
    const decoded = verifyToken(token);
    (req as any).user = decoded;
    next();
  } catch (error) {
    return res.status(401).json({
      message: "Invalid token"
    });
  }
};
```

7 Use in a Protected Route

Ans →

```
import { Router } from "express";
import { authenticate } from "../middleware/auth";

const router = Router();

router.get("/profile", authenticate, (req, res) => {
  res.json({
    message: "Protected route",
    user: (req as any).user,
  });
});
```

8 Token in Client (Frontend)

Ans →

1. Store the token in localStorage or httpOnly cookies (cookies are more secure).
2. Send the token in the Authorization header with each request.
3. Example using fetch:

```
fetch("/api/profile", {
  headers: {
    Authorization: `Bearer ${token}`
  }
});
```

9 Token Expiration & Refresh

Ans →

- Set a short expiration time (e.g., 15m, 1h).
- Use refresh tokens to get a new access token.
- Store refresh tokens securely (httpOnly cookies).
- Implement token rotation for better security.

Common Expiry Times:

- Access Token: 15 minutes – 1 hour
- Refresh Token: 7 days – 30 days

10 Best Practices

Ans

- ✅ Use strong secrets (store in environment variables).
- ✅ Set appropriate token expiration times.
- ✅ Use httpOnly cookies for better security.
- ✅ Implement refresh tokens.
- ✅ Validate tokens on every protected route.
- ✅ Do not store sensitive data in token payload.
- ✅ Use HTTPS in production.
- ✅ Handle token errors properly.
- ✅ Consider role-based access control (RBAC).

11 Real-World Use Case

Ans →

- User login (generate JWT)
- Protect API routes (middleware)
- Fetch user profile
- Refresh expired token
- Logout (clear token)
- Role-based access (admin/user)
- Used in MERN stack applications

12 Next Topics

Ans →

- Refresh Token Implementation
- Role-Based Authorization (RBAC)
- Logout & Token Blacklisting
- OAuth 2.0 & Social Login
- Security Best Practices

MongoDB.

Karyvio.com

Secure Authentication, Build Scalable Apps ! ☺

LEARN | BUILD | GROW | GET HIRED

Karyvio
LEARN | BUILD | GROW | GET HIRED

Authorization & Protected APIs

Control what authenticated users can do and protect your backend routes.

Right Users
Right Access
More Secure Apps
with Proper
Authorization

1 What is Authorication ?

Ans →

- Authorization determines what authenticated users are allowed to do (permissions).
- It controls access to resources based on roles, permissions, or policies.
- Comes after authentication (i.e., after verifying the user).
- Helps protect sensitive data and APIs from unauthorized access.

2 Common Authorization Models

Ans →

Model	Description
RBAC	Role-Based Access Control (roles like admin, user, moderator)
ABAC	Attribute-Based Access Control (based on user attributes)
ACL	Access Control List (specific permissions per resource)
PBAC	Policy-Based Access Control (uses rules/policies)

3 Role Example (MongoDB)

Ans →

```
// User schema with roles
const userSchema = new mongoose.Schema({
  name: String,
  email: { type: String, unique: true },
  password: String,
  role: {
    type: String,
    enum: ["user", "admin", "moderator"],
    default: "user"
  }
});
```

4 Role-Based Middleware

Ans →

```
// middleware/authorize.js
export const authorize = (...roles) => {
  return (req, res, next) => {
    if (!req.user) {
      return res.status(401).json({
        message: "Unauthorized"
      });
    }

    if (!roles.includes(req.user.role)) {
      return res.status(403).json({
        message: "Forbidden: Insufficient
        permissions"
      });
    }
    next();
  };
};
```

5 Protecting Routes

Ans →

```
import express from "express";
import { authenticate } from "./auth.js";
import { authorize } from "./authorize.js";

const router = express.Router();

// Any authenticated user
router.get("/profile", authenticate, (req,
res) => {
  res.json({ message: "User profile",
    user: req.user });
});
// Only admin can access
router.get("/admin", authenticate,
  authorize("admin"), (req, res) => {
  res.json({ message: "Admin data" });
});
```

6 Resource Ownership Check

Ans →

```
// Allow users to access only their own data
router.get("/users/:id", authenticate,
 async (req, res) => {
  if (req.user.role !== "admin" &&
    req.user.id !== req.params.id) {
    return res.status(403).json({
      message: "Forbidden: You can only
      access your own data"
    });
  }

  const user = await User.findById(req.params.id)
    .select("-password");
  res.json(user);
});
```

7 Permission-Based Example

Ans →

```
// Check specific permission
const hasPermission = (permission) => {
  return (req, res, next) => {
    if (!req.user.permissions?.includes(
    (permission)) {
      return res.status(403).json({
        message: "Forbidden: Missing permission"
      });
    }
    next();
  };
};
// Usage
couter.delete("/posts/:id", authenticate,
  hasPermission("delete:post"), (req, res) => {
  // delete post
});
```

8 JWT Payload with Role/Permissions

Ans →

```
// While generating token
const token = jwt.sign(
  {
    userId: user._id,
    role: user.role,
    permissions: user.permissions
  },
  process.env.JWT_SECRET,
  { expiresIn: "1h" }
);

// Access in protected route
console.log(req.user.role);
console.log(req.user.permissions);
```

9 Best Practices

Ans →

- ✅ Always authenticate before authorizing.
- ✅ Use roles/permissions instead of hardcoding user IDs.
- ✅ Follow the principle of least privilege.
- ✅ Validate resource ownership (e.g., user can only access their own data).
- ✅ Keep authorization logic in middleware.
- ✅ Use environment variables for secrets.
- ✅ Log and monitor unauthorized access attempts.
- ✅ Return proper HTTP status codes (401, 403).
- ✅ Test all protected routes.

10 Real-World Use Case

Ans →

- Only admins can manage users.
- Only the post owner can edit/delete their post.
- Moderators can approve/reject content.
- Regular users can view and create content.
- Protect premium features for paid users.

11 Common Status Codes

Ans →

Status Code	Meaning
401	Unauthorized (not logged in)
403	Forbidden (not allowed)
404	Not Found (resource doesn't exist)
429	Too Many Requests
500	Internal Server Error

12 Next Topics

Ans →

- Fine-grained permissions (ABAC)
- Policy-based authorization (PBAC)
- API rate limiting
- Refresh token rotation
- OAuth 2.0 & OpenID Connect
- Audit logging for access control

Karyvio.com

Secure APIs Today, Scalable Applications Tomorrow ! ☺

MongoDB.
LEARN | BUILD | GROW | GET HIRED

Karyvio
LEARN | BUILD | GROW | GET HIRED

MERN Security

Build safe, secure and production-ready MERN applications

Secure Code
Secure Data
Secure Users
Build a Safer Web
with MERN

1 Why Security is Important ?

Ans →

- Protects user data and privacy.
- Prevents unauthorized access.
- Builds trust in your application.
- Avoids data breaches and legal issues.
- Essential for production and real users.
- Common attacks: XSS, CSRF, SQL/NoSQL Injection, DDoS, Brute Force, etc.

2 Common Security Threats

Ans →

Threat	Description
XSS	Inject malicious scripts into web pages
CSRF	Trick users into performing actions
NoSQL Injection	Malicious queries to manipulate DB
Brute Force	Guessing passwords repeatedly
Sensitive Data Leak	Exposing keys, tokens, or user data
DDoS	Overload the server with traffic
Insecure Dependencies	Vulnerabilities in third-party packages

3 Secure Authentication

Ans →

- ✅ Use strong passwords (min 8 chars).
- ✅ Hash passwords with bcrypt.
- ✅ Use JWT with short expiry.
- ✅ Store tokens in HttpOnly cookies (in production).
- ✅ Implement refresh tokens.
- ✅ Use multi-factor authentication (MFA) if required.
- ✅ Protect login routes from brute force (rate limiting).

4 Password Hashing (bcrypt)

Ans →

```
const bcrypt = require('bcrypt');

// Hash password
const hashedPassword = await bcrypt.hash(
   'MySecurePassword123',
   10  // salt rounds
);

// Compare password
const isMatch = await bcrypt.compare(
   'MySecurePassword123',
   hashedPassword
);
console.log('Match:', isMatch);
```

5 Secure JWT Setup

Ans →

```
const jwt = require('jsonmaktoken');

// Generate token
const token = jwt.sign(
   { userId: user._id, role: user.role },
   process.env.JMT_SECRET,
   { expiresIn: '1h' }
);

// Verify token (middleware)
const decoded = jwt.verify(token,
   process.env.JMT_SECRET)
);
console.log(decoded);
```

6 Protect Routes (Middleware)

Ans →

```
const jwt = require('jsonmaktoken');

const protect = (req, res, next) => {
  const token = req.cookies.token ||
  req.headers.authorization?.split(' ')[1];
  if (!token) {
    return res.status(401).json({
      message: "No token provided"
  } );
  }
  try {
    const decoded = jwt.verify(token,
     process.env.JMT_SECRET);
    req.user = decoded;
    next();
  } catch (error) {
    return res.status(401).json({
      message: "Invalid token"
  } );
};
module.exports = protect;
```

7 Input Validation & Sanitization

Ans →

```
const { body, validationResult } =
  require('express-validator');
const sanitizeHtml = require('sanitize-html");

app.post('/register', [
 body('email').isEmail().normalizeEmail(),
 body('name').trim().escape()
], (req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({errors: errors.array() });
  }
  //Sanitize user input
  const cleanName = sanitizeHtml(req.body.name);
  // ... save to DB
});
```

8 Secure Headers (Helmet)

Ans →

```
const helmet = require('helmet');
const express = require('express');
const app = express();

app.use(helmet()); // sets secure HTTP headers

// OR customize
app.use(helmet({
   contentSecurityPolicy: true,
   crossOriginEmbedderPolicy: false
}));
```

What Helmet Does ?

- Sets X-Content-Type-Options
- Prevents clickjacking (X-Frame-Options)
- Enables Content-Security-Policy (CSP)
- Adds other important security headers

9 Rate Limiting & Brute Force Protection

Ans →

```
const rateLimit = require('express-rate-limit');

const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100, // limit each IP to 100 requests
  message: "Too many requests, try again later."
});

app.use("/api/", limiter);
```

10 Environment Variables & Secrets

Ans →

- ✅ Store sensitive data in .env file.
- ✅ Never commit .env to Git (add to .gitignore).
- ✅ Use strong, unique secret keys.
- ✅ Use different keys for development and production.
- ✅ Rotate keys regularly.
- ✅ Use secret managers (e.g., Render, AWS, Doppler).

11 Security Checklist for MERN Apps

Ans →

- ✅ Use HTTPS (SSL/TLS).
- ✅ Hash passwords (bcrypt).
- ✅ Use JWT with proper expiry.
- ✅ Validate & sanitize all inputs.
- ✅ Use Helmet for secure headers.
- ✅ Implement rate limiting.
- ✅ Protect routes with middleware.
- ✅ Keep dependencies updated (npm audit).
- ✅ Use environment variables for secrets.
- ✅ Regularly monitor logs and suspicious activity.

12 Next Topics

Ans →

- Advanced security headers (CSP)
- OAuth 2.0 & Social Login Security
- Secure File Uploads (Multer)
- Security Testing Tools (OWASP ZAP)
- Preventing NoSQL Injection
- Implementing 2FA / MFA
- Security Best Practices for Deployment

MongoDB.

Karyvio.com

Security today, a better tomorrow ! ☺

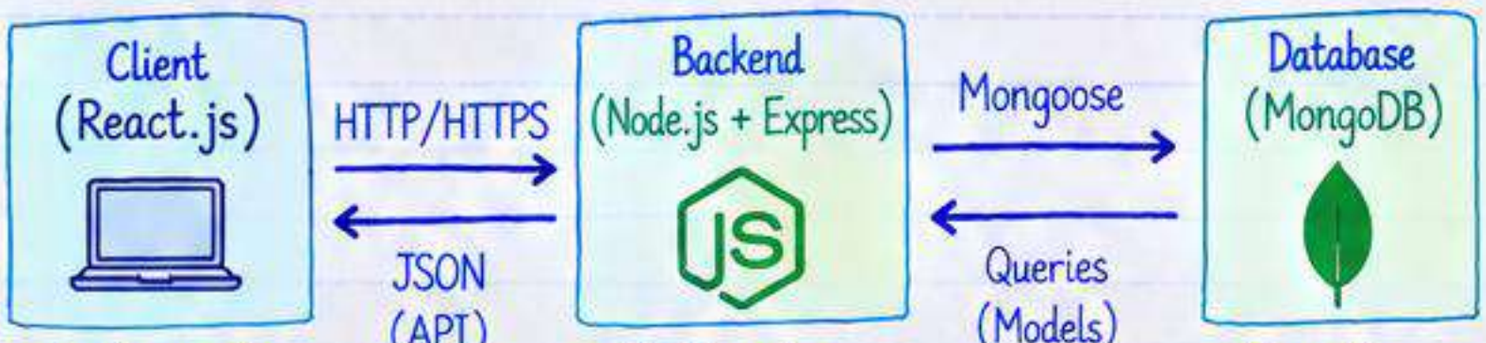
Karyvio
LEARN | BUILD | GROW | GET HIRED

Complete MERN API Architecture

Plan
Build
Secure
Scale
Your MERN APIs !

Design, build and understand a complete MERN backend architecture step by step !

1 High Level Architecture



- Send requests
- Receive responses
- Handle UI state

- Handle routes
- Business logic
- Authentication
- Validation
- Database operations

- Store data
- Flexible schema
- High performance
- Scalable

2 Project Structure

```
mern-api/
├── backend/
│   └── src/
│       ├── config/
│       ├── controllers/
│       ├── models/
│       ├── routes/
│       ├── middlewares/
│       └── utils/
│   ├── .env
│   ├── server.js
│   └── package.json
└── frontend/
    ├── src/
    │   ├── components/
    │   ├── pages/
    │   └── services/ (API calls)
    └── package.json
```

Keep your project modular and organized for better maintainability !

3 Request Flow (End-to-End)

1. User sends request from React (e.g. Login)
2. Request goes to Express Route (/api/auth/login)
3. Route → Controller (business logic)
4. Controller uses Model (Mongoose) to query DB
5. DB returns data to Controller
6. Controller sends response to Client (JSON)
7. React handles the response and updates UI

4 Example Route Setup (Express)

```
// routes/userRoutes.js
const express = require('express');
const router = express.Router();
const { registerUser, loginUser } =
      require('../controllers/userController');
const { protect } = require('../middlewares/auth');

router.post('/register', registerUser);
router.post('/login', loginUser);
router.get('/profile', protect, getProfile);

module.exports = router;
```

5 Mongoose Model (User)

```
// models/User.js
const mongoose = require('ewngoose');

const userSchema = new mongoose.Schema({
  name: String,
  email: { type: String, unique: true },
  password: String,
  role: { type: String, enum: ['user','admin'],
          default: 'user' };
  createdAt: { type: Date, default: Date.now }
});
module.exports = mongoose.model('User',
  userSchema);
```

6 Controller Example

```
// controllers/userController.js
const User = require('../models/User');
const bcrypt = require('bcrypt');
exports.registerUser = async (req, res)=> {
  try {
    const { name, email, password } = req.body;
    const hashedPassword = await bcrypt.hash
      (password, 10);
    const user = await User.create({
      name, email, password: hashedPassword
    });
    res.status(201).json({ message: 'User
      registered successfully', user });
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
};
```

7 Authentication Middleware

```
// middlewares/auth.js
const jwt = require('jsomaketben');

const protect = (req, res, next) => {
  const token = req.haaders.authorization
    ?.split(' ')[1];
  if (!token) return res.status(401).json({
    message: 'No token provided' });
  try {
    const decoded = jwt.verify(token,
      process.env.JWT_SECRET);
    req.user = decoded;
    next();
  } catch (error) {
    res.status(401).json({ message: 'Invalid
     token' });
  }
};
module.exports = { protect };
```

8 Environment Variables (.env)

```
PORT=5000
MONGO_URI=mongodb://localhost:27017/mern
JWT_SECRET=your_jwt_secret_key
NODE_ENV=development
```

Never commit your .env file to GitHub.
Use different vales for development and production.

9 API Response Format

```
{
    "success": true,
    "message": "Data fetched successfully",
    "data": { ... }
}
```

10 Security & Best Practices

- ✅ Use HTTPS in production.
- ✅ Validate input data (express-validator / Joi).
- ✅ Hash passwords (bcrypt).
- ✅ Use JWT with proper expiry.
- ✅ Implement role-based authorization (RBAC).
- ✅ Set CORS, Helmet, rate limiting.
- ✅ Handle errors properly.
- ✅ Keep dependencies updated.

11 Deployment Overview

Frontend (React) — Vercel / Netlify → Backend (Node.js + Express) — Render / Railway → Database (MongoDB) — MongoDB Atlas

- Use environment variables and production configs for secure deployment.

12 Key Takeaways

- Understand the complete flow (Client → Server → Database).
- Keep your code modular.
- Use authentication and authorization.
- Follow security best practices.
- Deploy and monitor your APIs.
- Build scalable, production-ready MERN applications !

Karyvio
LEARN | BUILD | GROW | GET HIRED

Connecting React with Express

Frontend + Backend = Real Projects !!

Connect your React frontend with Express backend and build a full-stack MERN application

1 Overview

React runs in the browser (frontend) and Express runs on a server (backend). They communicate using HTTP requests (e.g. GET, POST) and usually exchange data in JSON format.

Frontend (React) ⇄ HTTP (JSON) ⇉ Backend (Express)

Different ports during development.

2 Architecture Diagram

3 Setup CORS in Express

Since React and Express run on different ports, we need to enable CORS (Cross-Origin Resource Sharing).

```
// server.js
const express = require('express');
const cors = require('cors');

const app = express();

app.use(cors({
    origin: 'http://localhost:5173',
    credentials: true
}));
app.use(express.json());
```

↳ Allow requests from your React app

4 Create a Simple API (Express)

```
// routes/user.js
const express = require('express');
const router = express.Router();

router.get('/api/users', (req, res) => {
    res.json([
      { id: 1, name: 'Sovit' },
      { id: 2, name: 'Smruti' }
    ]);
});

module.exports = router;
```

5 Use the API in React

You can use fetch (built-in) or axios to call your Express API.

Using Fetch:

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

function App() {
 const [users, setUsers] = useState([]);

 useEffect(() => {
  fetch('http://localhost:5000/api/users')
    .then(res => res.json())
    .then(data => setUsers(data))
    .catch(err => console.error(err));
 }, []);

 return (
   <div>
     <h1>Users</h1>
     <ul>
       {users.map(user => (
         <li key={user.id}>{user.name}</li>
       ))}
     </ul>
   </div>
 );
};
export default App;
```

6 Using Axios (Optional)

```
import axios from 'axios';
import { useEffect, useState } from 'react';

function App() {
 const [users, setUsers] = useState([]);

 useEffect(() => {
  axios.get('http://localhost:5000/api/users')
    .then(res => setUsers(res.data))
    .catch(err => console.error(err));
 }, []);

 return (
   <div>
     <h1>Users</h1>
     <ul>
       {users.map(user => (
         <li key={user.id}>{user.name}</li>
       ))}
     </ul>
   </div>
 );
};
export default App;
```

7 Project Structure

```
mern-app/
├── backend/
│   ├── node_modules/
│   ├── routes/
│   ├── server.js
│   └── package.json
└── frontend/
    ├── node_modules/
    ├── src/
    ├── public/
    ├── package.json
    └── vite.config.js
```

Keep frontend and backend in separate folders for better organization. !!

8 Run the Application

Start Backend	Start Frontend
cd backend npm install node server.js	cd frontend npm install npm run dev
Backend runs on: http://localhost:5000	Frontend runs on: http://localhost:5173

9 Test the Connection

- Start both servers.
- Open your React app in browser.
- Check if data is fetched from Express API.
- Use browser DevTools → Network tab to debug requests.
- Handle errors properly.

10 Common Issues & Fixes

- CORS error → Enable cors() in Express.
- Network error → Check if backend server is running.
- Wrong port → Verify API URL.
- Empty data → Check API route and response.
- Mixed content (HTTPS) → Use correct protocol.
- Use .env file for API URL in production.

11 Best Practices

- Use environment variables for API URLs.
- Keep API calls in a separate service file (e.g. api.js).
- Handle loading and error states in UI.
- Use **axios interceptors** for auth tokens (advanced).
- Follow RESTful API standards.
- Use HTTPS in production.

12 Next Topics

- Environment Variables (.env)
- Authentication (JWT)
- Protected Routes in React
- State Management (Context/Redux)
- Deploying MERN Stack
- Real Project: Full-Stack App

"Frontend talks to Backend, and together they build amazing things." !!

Karyvio
LEARN | BUILD | GROW | GET HIRED

MERN Authentication Integration

Integrate authentication in your MERN stack application (Frontend + Backend)

Secure
Authenticate
Authorize
Build Real Projects
with MERN

1 Overview

- Build a complete authentication flow in MERN (MongoDB, Express, React, Node.js).
- Implement user registration, login, logout, and protected routes.
- Use JWT (JSON Web Token) for authentication.
- Store passwords securely using bcrypt.
- Manage user sessions on frontend (using context / Redux / state).

2 Authentication Flow

1. User submits credentials (Register/Login)
2. Backend validates and creates/verifies user
3. JWT token is generated and sent to frontend
4. Frontend stores token (localStorage / cookies)
5. Token is sent in headers for protected routes
6. Backend verifies token and gives access to resources

3 Install Required Packages

Backend (Express)

```
$ npm install express mongoese bcrypt
  jsonmaktoken cors dotenv

$ npm install --save-dev nodemon
```

Frontend (React)

```
$ npm install axios react-router-dom
$ npm install jwt-decode
```

4 User Model (MongoDB)

/models/User.js

```
const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, unique: true },
  password: { type: String, required: true
}, { timestamps: true });

module.exports = mongoose.model('User',
  userSchema);
```

5 Register Route (Express)

/routes/auth.js

```
const express = require('express');
const router = express.Router();
const User = require('../models/User');
const bcrypt = require('bcrypt');
const jwt = require('jsomaktoken');

router.post('/register', async (req, res) => {
  const { name, email, password } = req.body;
  const hashedPassword = await bcrypt.hash(
    password, 10)
  const user = await User.create({ name, email,
    password: hashedPassword });
  res.status(201).json({ message: 'User
    registered successfully' });
});
module.exports = router;
```

6 Login Route (Express)

/routes/auth.js

```
router.post('/login', async (req, res) => {
  const { email, password } = req.body;
  const user = await User.findOne({ email });
  if (!user) return res.status(401).json({
    message: 'Invalid credentials' });
  const isMatch = await bcrypt.compare(
    password, user.password);
  if (!isMatch) return res.status(401).json({
    message: 'Invalid credentials' });
  const token = jwt.sign(
    { id: user._id, email: user.email },
    process.env.JWT_SECRET,
    { expiresIn: '7d' }
  );
  res.json({ token, user: { id: user._id,
    name: user.name, email: user.email } });
```

7 Protected Route Middleware

/middleware/auth.js

```
const jwt = require('jsomaktoken');
const protect = (req, res, next) => {
  const token = req.haeders.authorization
    ?.split(' ')[1];
  if (!token) return res.status(401).json({
    message: 'No token provided' });
  try {
    const decoded = jwt.verify(token,
      process.env.JWT_SECRET);
    req.user = decoded;
    next();
  } catch (error) {
    return res.status(401).json({
      message: 'Invalid token' });
} };
module.exports = { protect };
```

8 Using Protected API

/routes/user.js

```
const express = require('express');
const router = express.Router();
const { protect } = require('../middleware/auth');

router.get('/profile', protect, async (req, res) =>{
  const user = await User.findById(req.user.id).select(
    '-password')
  res.json(user);
});
module.exports = router;
```

9 Frontend: Login with Axios

/src/api/auth.js

```
import axios from 'axios';

export const login = async (email, password) => {
  const res = await axios.post(
    'http://localhost:5000/api/auth/login',
    { email, password }
  localStorage.setItem('token', res.data.token);
  return res.data;
};
```

10 Frontend: Protected Route (React)

```
import { Navigate } from 'react-router-dom';
import jwtDecode from 'jwt-decode';

const ProtectedRoute = ({ children }) => {
  const token = localStorage.getItem('token');
  if (!token) return <Navigate to='/login' />;
  try {
    jwtDecde(token); // check if token is valid
    return children;
  } catch (error) {
    localStorage.removeItem('token');
    return <Navigate to=//login' />;
};
export default ProtectedRoute;
```

11 Environment Variables

- Create a .env file in backend:
- MONGO_URI=your_mongoodb_connection_string
- JWT_SECRET=your_secret_key
- PORT=5000

12 Best Practices

- Use strong passwords & hash with bcrypt.
- Store JWT in HttpOnly cookies (more secure).
- Set proper token expiration.
- Validate user input (e.g., express-validator).
- Use environment variables for secrets.
- Implement refresh tokens (optional).
- Protect sensitive routes on both frontend and backend.
- Handle errors properly.

13 Next Topics

- Refresh Token Implementation
- Logout Functionality
- Role-Based Access Control (RBAC)
- OAuth (Google, GitHub) Integration
- Password Reset (Email)
- Email Verification
- Two-Factor Authentication (2FA)
- Deploy MERN Auth System

Karyvio
LEARN | BUILD | GROW | GET HIRED

File Uploads & Images

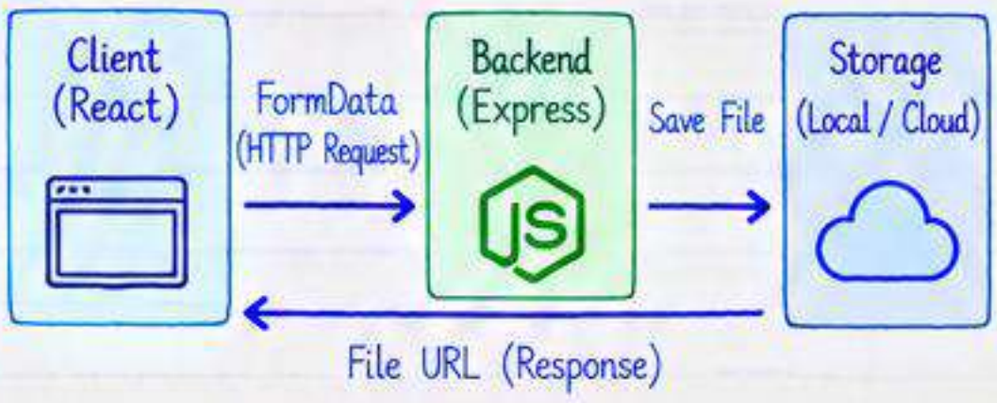
Upload
Process
Store
Serve
Use in App

Handle file uploads, store images and manage media in MERN applications

1 Why File Uploads ?

- Allow users to upload images, documents and other files.
- Common in profiles, blogs, e-commerce, resumes, etc.
- Need proper validation, storage and security.
- Can store files locally or in cloud (Cloudinary, AWS S3, etc.).

2 Upload Flow (Overview)



3 Install Required Packages

Backend (Express)

```
$ npm install multer
$ npm install cloudinary
$ npm install dotenv
```

Frontend (React)

```
$ npm install axios
```

4 Configure Multer (Local Storage)

```
// backend/config/multer.js
const multer = require('multer');
const path = require('path');

const storage = multer.diskStorage({
   destination: (req, file, cb) => {
     cb(null, 'uploads');
   },
   filename: (req, file, cb) => {
     const uniqueName = Date.now() +
        path.extname(file.originalname);
     cb(null, uniqueName);
   }
});

const upload = multer({
   storage,
   limits: { fileSize: 5 * 1024 * 1024 }
});

module.exports = upload;
```

5 Upload Route (Express)

```
// backend/routes/upload.js
const express = require('express');
const upload = require('../config/multer');
const router = express.Router();

router.post('/upload',
  upload.single('image'), (req, res) => {
  if (!req.file) {
     return res.status(400).json({
        message: 'No file uploaded' });
  }

  const fileUrl = `/uploads/${req.file.filename}`;
  res.status(200).json({
     message: 'File uploaded successfully',
     fileUrl
  });
});

module.exports = router;
```

6 Upload to Cloudinary (Cloud Storage)

```
// backend/config/cloudinary.js
const cloudinary = require('cloudinary').v2;
require('dotenv').config();

cloudinary.config({
   cloud_name: process.env.CLOUDINARY_CLOUD,
   api_key: process.env.CLOUDINARY_KEY,
   api_secret: process.env.CLOUDINARY_SECRET
});

module.exports = cloudinary;
```

```
// backend/routes/uploadCloud.js
const router = require('express').Router();
const upload = require('../config/multer');
const cloudinary = require('../config/cloudinary');

router.post('/upload', upload.single('image'),
  async (req, res) => {
    const result = await cloudinary.uploader.upload(
      req.file.path, { folder: 'karyvio' })
    res.json({ url: result.secure_url });
} });
module.exports = router;
```

7 Frontend: Upload Image (React)

```
// src/components/UploadImage.jsx
import { useState } from 'react';
import axios from 'axios';

function UploadImage() {
  const [file, setFile] = useState(null);
  const [url, setUrl] = useState('');

  const handleUpload = async () => {
     const formData = new FormData();
     formData.append('image', file);
     const res = await axios.post(
       'http://localhost:5000/upload', formData,
       { headers: { 'Content-Type': 'multipart/form-data' } }
     );
     setUrl(res.data.fileUrl);
  };

  return (
    <div>
      <input type="file" onChange={(e) => setFile(e.target.files[0])} />
      <button onClick={handleUpload}>Upload</button>
       {url && <img src={url} alt="Uploaded" width="200" />}
    </div>
  );
}
export default UploadImage;
```

8 Store File Info in MongoDB

```
// models/File.js
const mongoose = require('mongoose');

const fileSchema = new mongoose.Schema({
   filename: String,
   url: String,
   uploadedBy: { type: mongoose.Schema.Types.ObjectId,
                 ref: 'User' },
   createdAt: { type: Date, default: Date.now }
});

module.exports = mongoose.model('File', fileSchema);
```

9 Validation & Security

- ✔ Check file type (only images, PDF, etc.).
- ✔ Set file size limit.
- ✔ Use unique file names.
- ✔ Validate on both frontend and backend.
- ✔ Use cloud storage for production.
- ✔ Prevent direct access to sensitive files.
- ✔ Sanitize file names.

10 Best Practices

- ✔ Use cloud storage (Cloudinary / AWS S3) for scalability.
- ✔ Optimize images (resize/compress).
- ✔ Store only file URL in database.
- ✔ Use environment variables for keys.
- ✔ Restrict file types and size.
- ✔ Add authentication (only logged-in users).
- ✔ Delete unused files.
- ✔ Use CDN for faster delivery.

11 Next Topics

- Image Optimization
- Multiple File Uploads
- Drag & Drop Upload UI
- Progress Bar (Upload)
- Delete / Update Files
- Secure File Access (Signed URLs)
- AWS S3 Integration
- Handling Other File Types (PDF, Docs, etc.)

Karyvio
LEARN | BUILD | GROW | GET HIRED

Search, Filter, Sort & Pagination

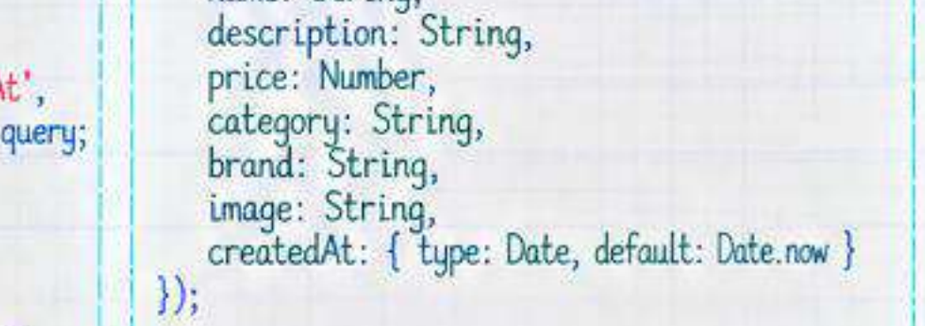
Implement powerful data querying features in MERN applications

Better Search
Better Users
Better Experience
Build Real Projects
with MERN

1 Why It Matters ?

- Helps users find relevant data quickly.
- Improves user experience.
- Essential for real-world applications (e.g., blogs, e-commerce, job portals).
- Reduces load by retrieving only required data.
- Combines Search + Filter + Sort + Pagination for scalable APIs.

2 Flow Overview



Example Request URL:
/api/products?search=watch&category=electronics&sort=price&page=1&limit=10

3 Query Parameters

Param	Description
search	Search by keyword (title, name, etc.)
filter	Filter by fields (category, brand, etc.)
sort	Sort by field (price, createdAt, etc.)
order	asc or desc (default: asc)
page	Page number (default: 1)
limit	Items per page (default: 10)

4 Backend: Build Query (Express)

```
// routes/products.js
const express = require('express');
const router = express.Router();
const Product = require('../models/Product');

router.get('/', async (req, res) => {
  try {
    const { search, category, sort = 'createdAt',
    order = 'asc', page = 1, limit = 10 } = req.query;
  // Build filter
  const filter = {};
  if (search) {
    filter.$or = [
      { name: { $regex: search, $options: 'i' } },
      { description: { $regex: search, $options: 'i' } }
    ];
  }
 if (category) filter.category = category;

// Build sort
const sortOrder = order === 'desc' ? -1 : 1;
const sortOptions = { [sort]: sortOrder };

// Pagination
const pageNum = parseInt(page);
const limitNum = parseInt(limit);
const skip = (pageNum - 1) * limitNum;

const products = await Product.find(filter)
   .sort(sortOptions)
   .skip(skip)
   .limit(limitNum);

 const total = await Product.countDocuments(filter);

 res.json({
    success: true,
    total,
    page: pageNum,
    limit: limitNum,
    data: products
  });
 } catch (error) {
  res.status(500).json({ success: false, message: error.message });
 }
};
module.exports = router;
```

5 Mongoose Model (Example)

```
// models/Product.js
const mongoose = require('mongoose');

const productSchema = new mongoose.Schema({
   name: String,
   description: String,
   price: Number,
   category: String,
   brand: String,
   image: String,
   createdAt: { type: Date, default: Date.now }
});

module.exports = mongoose.model('Product', productSchema);
```

6 Frontend: Fetch with Params

```
// src/services/api.js
import axios from 'axios';

export const getProducts = (params) => {
   return axios.get('/api/products', {
      params
   });
};
// Example usage
 getProducts({
      search: 'watch',
      category: 'electronics',
      sort: 'price',
      order: 'asc',
      page: 1,
      limit: 10
```

7 Frontend: Display Data

```
// src/components/ProductList.jsx
import { useEffect, useState } from 'react';
import { getProducts } from '../services/api';

function ProductList({ params }) {
 const [products, setProducts] = useState([]);
 const [loading, setLoading] = useState(false);

 useEffect(() => {
   const fetchData = async () => {
     setLoading(true);
     const res = await getProducts(params);
     setProducts(res.data.data);
     setLoading(false);
   };
   fetchData();
 }, [params]);
 if (loading) return <p>Loading...</p>;

 return (
   <div>
     {products.map(p => (
       <div key={p._id} className="card">
         <img src={p.image} alt={p.name} />
         <h3 {p.name}</p>
        <p> ₹{p.price}
       ))
   </div>
 );
export default ProductList;
```

8 Pagination UI (React)

```
// Simple Pagination Component
function Pagination({ page, total, limit,
                      onPageChange }) {
 const totalPages = Math.ceil(total / limit);

 return (
   <div className="pagination">
     <button
       disabled={page === 1}
       onClick={() => onPageChange(page - 1)}
     >
       ← Prev
       </button>

       <span> Page {page} of {totalPages} </span>

       <button
         disabled={page === totalPages}
         onClick={() => onPageChange(page + 1)}
       >
         Next →
       </button>
   </div>
 );
};
```

9 Best Practices

- Use indexes for search & filter fields.
- Sanitize and validate query parameters.
- Set default values (page = 1, limit = 10).
- Use text indexes for full-text search.
- Limit maximum page size (e.g., 100).
- Return total count for pagination.
- Optimize queries for performance.

10 Common Use Cases

- Blog posts search by title/category.
- E-commerce product filtering.
- Job listings search by role/location.
- User management (search by name/email).
- Sort by date, price, popularity, etc.
- Pagination for large datasets.

11 Next Topics

- Advanced Search (Full-text)
- Multiple Filters (Range, Date)
- Sorting by Multiple Fields
- Infinite Scroll
- Search with Aggregation (MongoDB)
- Elastissearch Integration
- Real-world Project Implementation

12 Key Takeaway

"Good search and filter make data useful, and good pagination makes it scalable."

Karyvio
LEARN | BUILD | GROW | GET HIRED

Real-Time MERN Apps

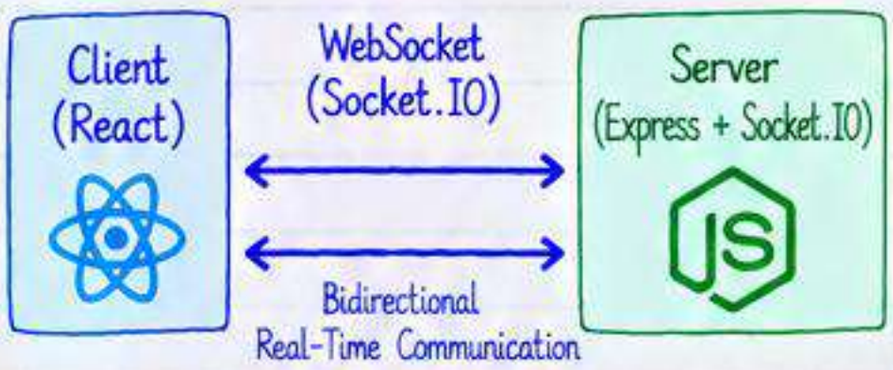
Build real-time features in MERN using WebSockets (Socket.IO)

Real-Time
Real Users
Real Projects
with MERN

1 What is Real-Time ?

- Real-time apps allow instant communication between clients and server.
- Used in chat apps, notifications, live updates, collaboration tools, online gaming, etc.
- In MERN, we use Socket.IO (WebSockets) to achieve real-time communication.
- WebSockets keep a persistent connection between client and server (unlike HTTP).

2 How Real-Time Works ?



Client and server can send and receive data instantly in both directions.

3 Install Required Packages

Backend (Express)

```
npm install socket.io cors
# optional (for TS)
npm install -D @types/socket.io
```

Frontend (React)

```
npm install socket.io-client
```

4 Setup Socket.IO Server (Express)

```
// server.js
const express = require('express');
const http = require('http');
const cors = require('cors');
const { Server } = require('socket.io');

const app = express();
app.use(cors({ origin: 'http://localhost:5173',
               credentials: true }));

const server = http.createServer(app);
const io = new Server(server, {
  cors: { origin: 'http://localhost:5173',
          methods: ['GET', 'POST'] }
});

io.on('connection', (socket) => {
    console.log('User connected:', socket.id);
    socket.on('sendMessage', (data) => {
      io.emit('receiveMessage',data); // broadcast
    });
    socket.on('disconnect', () => {
      console.log('User disconnected:', socket.id);
    });
});

server.listen(5000, () => console.log('Server
running on port 5000'));
```

5 Frontend: Connect to Socket.IO

```
// src/socket.js
import { io } from 'socket.io-client';

const socket = io('http://localhost:5000');

export default socket;
```

```
// In a React component
import { useEffect, useState } from 'react';
import socket from './socket';

function ChatApp() {
  const [message, setMessage] = useState('');
  const [messages, setMessages] = useState([]);

  useEffect(() => {
    socket.on('receiveMessage', (data) => {
      setMessages((prev) => [...prev, data]);
    });
    return () => socket.off('receiveMessage');
  }, []);

  const sendMessage = () => {
    socket.emit('sendMessage', {
      text: message,
      user: 'Sovit'
    });
    setMessage('');
  };
  // JSX for UI (input + button + message list)
}
```

6 Example: Simple Chat UI

7 Real-Time Use Cases

- Chat Applications
- Live Notifications (e.g. new message)
- Live User Presence (online/offline)
- Real-Time Collaboration (Docs, Whiteboard)
- Live Updates (Orders, Tracking, Dashboard)
- Online Games
- Video / Voice Chat (with WebRTC)
- Live Stock / Crypto Price Updates
- Real-time Comments / Likes
- Admin Monitoring (live logs, users, etc.)

8 Best Practices

- Use authentication with Socket.IO (for private rooms).
- Validate and sanitize all incoming data.
- Use rooms for group communication.
- Handle disconnects and reconnections.
- Implement rate limiting to prevent abuse.
- Use a message queue (Redis) for scaling (multiple server instances).
- Store chat history in MongoDB (if needed).
- Handle errors gracefully.

9 Advanced Features

- User joining/leaving notifications
- Typing indicators ("User is typing...")
- Private messaging (1-to-1 chat)
- File sharing (images, docs)
- Read receipts (seen/unseen)
- Online user list
- Push notifications (browser/mobile)
- Use Redis Adapter for scaling
- Deploy with Docker / Cloud (Render, AWS, etc.)

10 Next Topics

- Socket.IO with Authentication (JWT)
- Private Rooms / Group Chat
- Real-Time Notifications System
- Chat App with MongoDB (store messages)
- Scaling with Redis Adapter
- Deploy Real-Time MERN App
- WebRTC (Video / Voice Chat)

11 Key Takeaway

"Real-time features make your application more interactive, engaging and user-friendly."

Karyvio
LEARN | BUILD | GROW | GET HIRED

Testing MERN Applications

Write reliable code with proper testing - Frontend, Backend & Database

Test
Detect
Fix
Ship
:)

1 Why Testing Matters?

- Find bugs early and save time.
- Ensure features work as expected.
- Build confidence before deployment.
- Helps in refactoring and scaling.
- Improves code quality and stability.

2 Types of Testing

Unit Testing	- Test individual functions or components.
Integration Testing	- Test interaction between modules (frontend + backend).
End-to-End (E2E) Testing	- Test complete user flow (real browser).
API Testing	- Test REST APIs (request/response).
Database Testing	- Test data operations and DB rules.

3 Testing Tools Stack

- Jest + React Testing Library (Frontend - React)
- Jest + Supertest (Backend - Express)
- Jest + Mongo Memory Server (Database - MongoDB)
- Postman / Newman (API Testing)
- Cypress / Playwright (E2E Testing)

4 Frontend Testing (React)

```
// Example: Testing a component
import { render, screen } from '@testing-library/react';
import UserCard from "./UserCard';

test('renders user name', () => {
   render(<UserCard name="Sovit" />);
   expect(screen.getByText('Sovit')).toBeInTheDocument();
});
```

Key Points:

- Use React Testing Library (user-centric).
- Test UI behavior, not implementation details.
- Use fireEvent / userEvent for interactions.
- Keep tests isolated and reusable.

5 Backend Testing (Express)

```
// Example: Testing an API route
const request = require('supertest');
const app = require('../app');

describe('GET /api/users', () => {
   it('should return users', async () => {
      const res = await request(app).get('/api/users');
      expect(res.statusCode).toBe(200);
      expect(res.body).toHaveProperty('users');
   });
});
```

Key Points:

- Use Supertest for API testing.
- Mock database with Mongo Memory Server.
- Test status codes, response body and errors.
- Keep routes modular for easier testing.

6 Database Testing (MongoDB)

```
// Example: Using Mongo Memory Server
const { MongoMemoryServer } =
            (mongodb-memory-server');
const mongoose = require('mongoose');

let mongoServer;
let uri;

beforeAll(async () => {
   mongoServer = await MongoMemoryServer.create();
   uri = mongoServer.getUri();
   await mongoose.connect(uri);
});

afterAll(async () => {
   await mongoose.disconnect();
   await mongoServer.stop();
});
```

Key Points:

- Use in-memory DB for tests.
- Seed test data before running tests.
- Clean up data after tests.

7 End-to-End Testing (Cypress)

```
// Example: E2E test

describe('User Login Flow', () => {
   it('should login and redirect', () => {
      cy.visit('/login');
      cy.get('[data-testid="email"]').type('user@test.com');
      cy.get('[data-testid="password"]').type('123456');
      cy.get('button[type="submit"]').click();
      cy.url().should('include', '/dashboard');
   });
});
```

Key Points:

- Test real user flows in browser.
- Use data-testid for stable selectors.
- Check UI, redirects, and session handling.

8 Mocking & Stubbing

```
// Example: Mocking external API
jest.mock('axios');

const axios = require('axios');

it('should fetch data', async () => {
    axios.get.mockResolvedValue({
       data: { user: { name: 'Sovit' } }
    });

    const data = await fetchUser();
    expect(axios.get).toHaveBeenCalledWith('/api/user');
    expect(data.name).toBe('Sovit');
});
```

Key Points:

- Mock external APIs (e.g., payment, email, etc.).
- Use jest.fn() or jest.mock().
- Keep tests fast and independent.

9 Test Scripts & CI/CD

package.json

```
{
   "scripts": {
      "test": "jest",
      "test:watch": "jest --watch",
      "test:coverage": "jest --coverage",
      "e2e": "cypress run"
   }
}
```

GitLab CI / GitHub Actions

- Run tests on every push / PR.
- Generate coverage reports.
- Fail build if tests fail.
- Integrate with deployment pipeline.

10 Best Practices

- Write meaningful test cases (not just coverage).
- Keep tests simple and focused.
- Use proper mocks and fixtures.
- Maintain test data separately.
- Run tests in CI/CD pipeline.
- Track coverage and fix gaps.

11 Common Issues & Fixes

- Tests fail due to async → use await / findBy*.
- MongoDB connection error → check URI & DB status.
- Mock not working → check jest.mock() path.
- Flaky E2E tests → use waitFor / retry.
- Coverage low → add more meaningful tests.
- Build fails in CI → check environment variables.

"Good tests don't just find bugs, they build better software." :)

Same Code Bigger Opportunities ☺

MERN Deployment & Environment Configuration

Karyvio
LEARN | BUILD | GROW | GET HIRED

Take your MERN app from local to the world

Code Deploy Scale Grow ☺

1 Deployment Options

You can deploy MERN apps on many platforms. Choose based on your needs.

- ✔ Frontend → Vercel, Netlify, Cloudflare Pages
- ✔ Backend → Render, Railway, Fly.io, AWS, DigitalOcean
- ✔ Database → MongoDB Atlas (cloud)
- ✔ File Storage → Cloudinary, AWS S3, or similar
- ✔ Domain & SSL → Namecheap, GoDaddy, Cloudflare
- ✔ CI/CD → GitHub Actions, GitLab CI

2 Environment Variables

Never hardcode sensitive information. Use .env files for local development and set environment variables in the platform dashboard for production.

Never commit .env files to GitHub!

Backend (.env)

```
PORT=5000
MONGO_URI=mongodb+srv://user:pass@cluster.mongodb.net/db
JWT_SECRET=your_secret_key
CLOUDINARY_CLOUD_NAME=xxx
CLOUDINARY_API_KEY=xxx
CLOUDINARY_API_SECRET=xxx
NODE_ENV=production
```

3 Deploy Frontend (Vercel)

1. Push your code to GitHub.
2. Go to https://vercel.com and import your repo.
3. Set environment variables (e.g. VITE_API_URL).
4. Click Deploy.
5. Your app will be live at a .vercel.app domain (or custom domain).

Frontend Environment Variable (.env)

```
VITE_API_URL=https://api.yourdomain.com
```

4 Deploy Backend (Render)

1. Push your code to GitHub.
2. Go to https://render.com and create a new Web Service.
3. Select your repo and set build command and start command.
4. Add environment variables (same as .env).
5. Deploy and get your backend URL.

Render Settings (Example)

```
Build Command : npm install
Start Command : npm start
Environment   : Node.js
Instance Type : Free (or Paid)
```

You can also use Railway, Fly.io or AWS EC2

5 MongoDB Atlas Setup

1. Go to https://www.mongodb.com/atlas
2. Create a free cluster.
3. Whitelist your IP (or allow 0.0.0.0/0).
4. Create a database user.
5. Get the connection string and use it in MONGO_URI.

MongoDB Atlas

6 File Uploads (Cloudinary)

1. Create an account at https://cloudinary.com
2. Get Cloud Name, API Key and API Secret.
3. Add them to your backend environment variables.
4. Use the SDK to upload images in production.

Cloudinary

```
import { v2 as cloudinary } from 'cloudinary';
cloudinary.config({
   cloud_name:  process.env.CLOUDINARY_CLOUD_NAME,
   api_key:     process.env.CLOUDINARY_API_KEY,
   api_secret:  process.env.CLOUDINARY_API_SECRET,
});
```

7 Domain & HTTPS

- ✔ Buy a domain from Namecheap / GoDaddy.
- ✔ Point your domain to Vercel (frontend) and Render (backend).
- ✔ Enable HTTPS (automatic in most platforms).
- ✔ Use a custom domain for professional look.

www.yourdomain.com → Frontend (Vercel)

api.yourdomain.com → Backend (Render)

8 CI/CD with GitHub Actions (Optional)

- ✔ Automatically deploy on every push.
- ✔ Example workflow for backend (Render deployment).

```
# .github/workflows/deploy.yml
name: Deploy Backend
on:
  push:
    branches: [ main ]
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
     - uses: actions/checkout@v4
     - name: Deploy to Render
     - run: curl -X POST ${{ secrets.RENDER_DEPLOY_HOOK }}
```

Drear Build Deploy Repeat ☺

Dream Build Deploy (Netext ☺

Karyvio.com

Build → Deploy → Scale → Get Hired ☺

Karyvio
LEARN | BUILD | GROW | GET HIRED

MERN Production Architecture

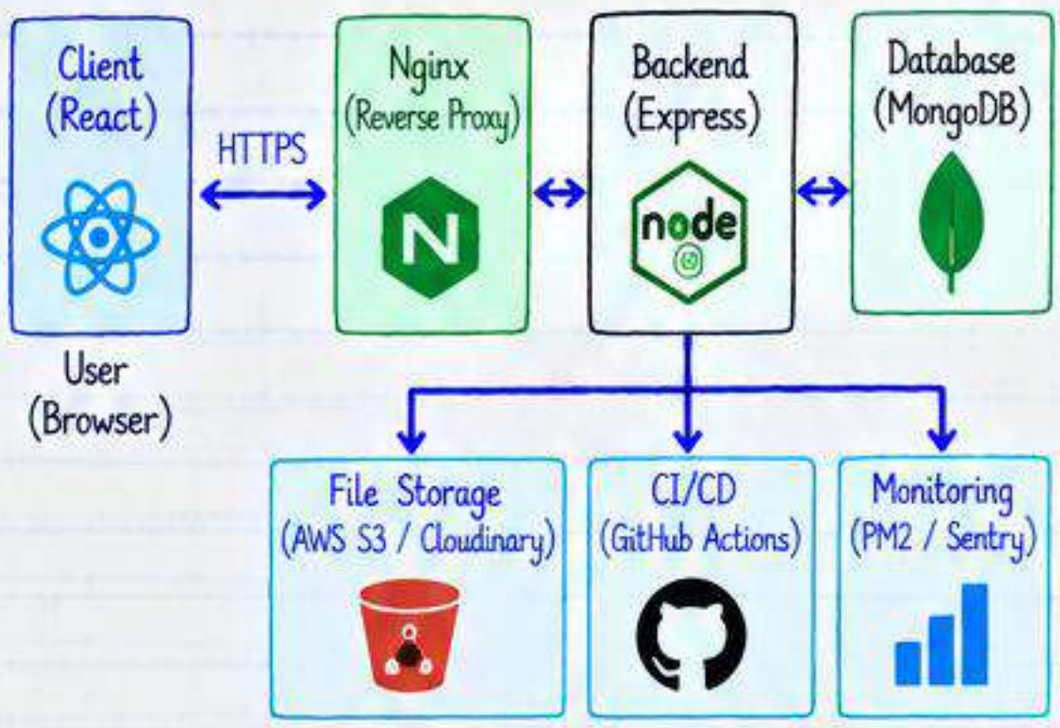
Deploy and scale MERN applications for real-world use

Code
Deploy
Scale
Grow
with MERN

1 Why Production Architecture?

- Make the app available to real users.
- Ensure scalability, reliability and security.
- Use cloud services for better performance.
- Handle environment configurations and domain setup.
- Monitor logs, errors and performance.
- Prepare for high traffic and growth.

2 High Level Architecture



3 Tech Stack (Production)

Frontend
- React.js (Vite / CRA)
- Vercel / Netlify (Hosting)
- HTTPS + Custom Domain

Backend
- Node.js + Express
- PM2 (Process Management)
- Nginx (Reverse Proxy)
- Docker (Optional)

Database
- MongoDB Atlas (Cloud)
- Connection pooling
- Auto backups

Other Services
- AWS S3 / Cloudinary (File Storage)
- GitHub Actions (CI/CD)
- Sentry (Error Tracking)
- UptimeRobot (Uptime Monitoring)

4 Environment Setup

Backend (.env)

```
PORT=5000
MONGO_URI=mongodb+srv://...
JWT_SECRET=your_secret
CLOUDINARY_CLOUD_NAME=xxx
CLOUDINARY_API_KEY=xxx
CLOUDINARY_API_SECRET=xxx
NODE_ENV=production
```

Frontend (.env)

```
VITE_API_URL=https://api.yourdomain.com
VITE_APP_NAME=YourApp
VITE_CLOUDINARY_CLOUD_NAME=xxx
```

⚠ Never expose sensitive keys in frontend. Use environment variables.

5 Deployment Steps

1. Set up production database (MongoDB Atlas).
2. Configure environment variables.
3. Build frontend (npm run build).
4. Deploy frontend (Vercel / Netlify).
5. Deploy backend (Render / Railway / AWS EC2).
6. Use Nginx as reverse proxy.
7. Set up custom domain (e.g. www.yourapp.com).
8. Enable HTTPS (SSL Certificate – Let's Encrypt).
9. Configure file storage (AWS S3 / Cloudinary).
10. Set up CI/CD (GitHub Actions).
11. Configure monitoring (Sentry, UptimeRobot).
12. Test everything (APIs, frontend, uploads, auth).

6 Nginx Configuration (Reverse Proxy)

```
server {
    listen 80;
    server_name yourdomain.com;

    location / {
      proxy_pass http://localhost:5173;  # Frontend
      proxy_http_version 1.1;
      proxy_set_header Upgrade $http_upgrade;
      proxy_set_header Connection "upgrade";
      proxy_set_header Host $host;
    }

    location /api {
      proxy_pass http://localhost:5000;  # Backend
      proxy_set_header Host $host;
      proxy_set_header X-Real-IP $remete_addr;
    }
}
```

7 CI/CD with GitHub Actions

```
# .githuk/workflows/deploy.yml
name: Deploy Backend
on:
   push:
     branches: [ main]
jobs:
   deploy:
     runs-on: ubuntu-latest
     steps:
       - uses: actions/checkout@v4
       - name: Install dependencies
         run: npm install
       - name: Build and Deploy
         run: npm run build
# Add deployment steps (Render, AWS, etc.)
```

8 Monitoring & Logs

- Use PM2 to keep the backend running.
- Monitor errors with Sentry.
- Use UptimeRobot to check uptime.
- View logs with PM2 logs.
- Set alerts for downtime.
- Monitor database performance (MongoDB Atlas).
- Track API response times.
- Use analytics (e.g. Plausible / Google Analytics).

9 Best Practices

- Use environment variables (never hardcode secrets).
- Enable HTTPS always.
- Use a CDN for static assets.
- Optimize images and files.
- Implement rate limiting.
- Use proper error handling and logging.
- Keep dependencies updated.
- Take regular database backups.
- Use monitoring and alerts.
- Test production build before going live.

10 Common Issues & Fixes

- ⚠ CORS errors → Check allowed origins.
- ⚠ Build fails → Check Node.js version and dependencies.
- ⚠ Database connection fails → Verify IP whitelist (MongoDB Atlas).
- ⚠ 502 Bad Gateway → Check backend process (PM2 logs).
- ⚠ File upload issues → Verify Cloudinary/AWS credentials.
- ⚠ Environment variables not loading → Check .env and redeploy.

11 Next Topics

- Dockerize MERN application
- Deploy with Docker Compose
- Kubernetes for MERN
- Serverless MERN (AWS Lambda)
- Scaling strategies (Load Balancer)
- Advanced monitoring (Prometheus + Grafana)

12 Key Takeaway

"A well-deployed MERN app is not just code – it's a product ready for the world."

Complete MERN Project

Build a Real-World Full Stack Application from Scratch

Ideas
Code
Build
Deploy
Grow

1 Project Idea

Let's build a real-world project:

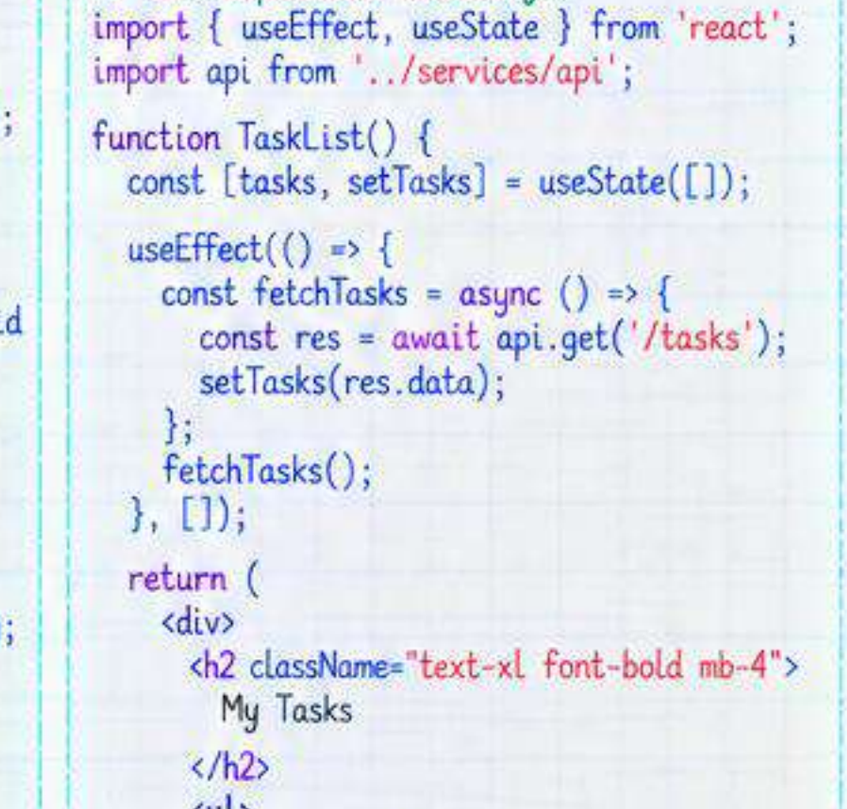
Task Management App

- User Authentication
- Create / Read / Update / Delete Tasks
- Categories & Due Dates
- Search, Filter & Pagination
- Image Upload (optional)
- Real-Time Updates (optional)
- Deploy to production

2 Tech Stack

Frontend	React (Vite)
Backend	Express.js
Database	MongoDB
Auth	JWT (JSON Web Token)
File Upload	Cloudinary / Multer
Real-Time	Socket.IO (optional)
Deployment	Vercel (Frontend)
	Render / Railway (Backend)

3 Project Architecture



4 Project Folder Structure

```
task-manager/
├── client/   # React frontend
│   ├── src/
│   │   ├── components/
│   │   ├── pages/
│   │   ├── services/
│   │   └── ...
├── server/   # Express backend
│   ├── src/
│   │   ├── controllers/
│   │   ├── routes/
│   │   ├── models/
│   │   ├── middleware/
│   │   └── ...
├── .env
├── package.json
└── README.md
```

5 Backend: Sample Route (Express)

```
// routes/tasks.js
const express = require('express');
const router = express.Router();
const { protect } = require('../middleware/auth');
const Task = require('../models/Task');

// Get all tasks
router.get('/', protect, async (req, res) => {
  const tasks = await Task.find({ user: req.user.id
  }).sort({ createdAt: -1 });
  res.json(tasks);
});

// Create a task
router.post('/', protect, async (req, res) => {
  const { title, description, dueDate } = req.body;
  const task = new Task({
    title,
    description,
    dueDate,
    user: req.user.id
  });
  await task.save();
  res.status(201).json(task);
});

module.exports = router;
```

6 Frontend: Sample Component

```
// src/components/TaskList.jsx
import { useEffect, useState } from 'react';
import api from '../services/api';

function TaskList() {
  const [tasks, setTasks] = useState([]);

  useEffect(() => {
    const fetchTasks = async () => {
      const res = await api.get('/tasks');
      setTasks(res.data);
    };
    fetchTasks();
  }, []);

  return (
    <div>
      <h2 className="text-xl font-bold mb-4">
        My Tasks
      </h2>
      <ul>
        {tasks.map(task => (
          <li key={task._id} className="p-2 border
            {task.title}
          </li>
        ))}
      </ul>
    </div>
  );
}
export default TaskList;
```

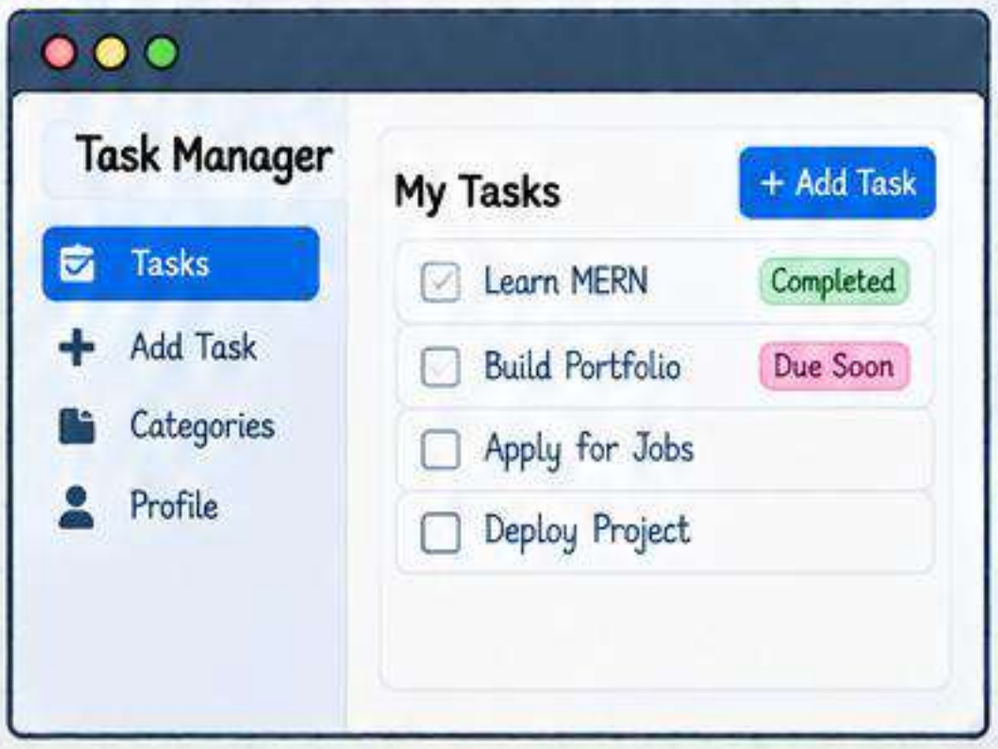
7 Key Features to Implement

- User Registration & Login (JWT)
- Create, Edit, Delete Tasks
- Mark task as complete
- Add categories & due dates
- Search, filter and pagination
- Upload images / attachments
- Real-time updates with Socket.IO
- Responsive UI (Tailwind CSS)
- Deploy frontend & backend
- Add error handling & validations

8 Deployment Steps

- Push code to GitHub.
- Deploy backend on Render / Railway.
- Deploy frontend on Vercel.
- Set environment variables.
- Use MongoDB Atlas (cloud database).
- Configure custom domain (optional).
- Test everything in production.

9 Screenshot (Example UI)



10 Next Improvements

- User profile & settings
- Email notifications
- Drag & drop task sorting
- Team collaboration (multi-user)
- Mobile app (React Native)
- Advanced analytics & charts
- Payment integration (for premium)

11 Key Takeaway

A complete MERN project teaches you real-world development, problem-solving and deployment skills.
Build. Learn. Improve.

Karyvio
LEARN | BUILD | GROW | GET HIRED

MERN Interview + Cheat Sheet

Prepare
Practice
Perform
Get Hired

Quick revision notes, important interview questions and handy code snippets for MERN Stack (MongoDB, Express.js, React, Node.js)

1 General MERN Questions

- What is MERN Stack?
- Why choose MERN over other stacks?
- What are the advantages & disadvantages?
- Explain the flow from frontend to database.
- How do JWT and cookies work?
- How do you handle environment variables?
- How do you deploy a MERN app?

2 MongoDB Questions

- What is MongoDB?
- Explain collections and documents.
- What are indexes?
- Explain aggregation framework.
- How to perform CRUD operations?
- What is the difference between find() and findOne()?
- How do you model relationships (one-to-one, one-to-many)?

3 Express.js Questions

- What is Express.js?
- How do you create a route?
- What is middleware?
- Explain error handling in Express.
- How do you validate request data?
- What is the purpose of next()?
- How do you structure a large Express application?

4 Node.js Questions

- What is Node.js?
- Explain the event loop.
- What are streams?
- What is the difference between process.nextTick() and setImmediate()?
- How do you handle uncaught exceptions?
- What is clustering in Node.js?
- How do you optimize performance in Node.js apps?

5 React.js Questions

- What is React?
- What are components (functional vs class)?
- What is the virtual DOM?
- Explain useState and useEffect.
- What is lifting state up?
- What are props and how do they work?
- What is the difference between controlled and uncontrolled components?
- How do you optimize React performance?

6 Authentication & Security

- How does JWT authentication work?
- How do you implement login/signup in MERN?
- How do you protect routes (frontend & backend)?
- What is password hashing (bcrypt)?
- How do you handle refresh tokens?
- What are common security best practices?
- How do you prevent XSS and CSRF?
- How do you manage user roles and permissions?

7 Common Coding Questions

- Reverse a string
- Check if a number is prime
- Find the largest element in an array
- Remove duplicates from an array
- Merge two sorted arrays
- Implement debounce function
- Write a function for deep clone
- Explain map(), filter(), reduce() with examples

8 System Design / Architecture

- Design a MERN application architecture.
- How will you scale a MERN app?
- How do you handle real-time features (e.g. chat)?
- How do you manage file uploads?
- How will you implement caching (Redis)?
- How do you handle high traffic?
- Design a notification system.
- Explain CI/CD for a MERN app.

9 DevOps & Deployment

- How do you deploy frontend (Vercel/Netlify)?
- How do you deploy backend (Render/Railway)?
- How do you set environment variables?
- How do you use Docker for MERN?
- How do you manage database in production?
- How do you monitor logs and errors?
- What is the role of Nginx in deployment?
- How do you set up CI/CD with GitHub Actions?

10 Useful Code Snippets

Express Route

```
const express = require('express');
const router = express.Router();
router.get('/api/hello', (req, res) => {
  res.json({ message: 'Hello MERN!' });
});
```

Mongoose Model

```
const mongoose = require('mongoose');
const UserSchema = new mongoose.Schema({
   name: String,
   email: String,
   password: String
});
module.exports = mongoose.model('User', UserSchema);
```

11 React Hooks Example

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

function Example() {
  const [data, setData] = useState([]);

  useEffect(() => {
     fetch('/api/data')
       .then(res => res.json())
       .then(setData);
  }, []);

  return <div>Data: {data.length}</div>;
}
```

JWT Verification (Express)

```
const jwt = require('jsonwebtoken');
function auth(req, res, next) {
  const token = req.headers.authorization;
  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded;
    next();
  } catch (err) {
    res.status(401).json({ message: 'Invalid token' });
  }
}
```

12 Quick Cheat Sheet

Task	Command / Code
Create React App	npm create vite@latest
Install Express	npm install express
Install Mongoose	npm install mongoose
Run Backend	node server.js
Run Frontend	npm run dev
Connect MongoDB	mongoose.connect(uri)
Hash Password	bcrypt.hash(password, 10)
Generate JWT	jwt.sign(payload, secret)
Verify JWT	jwt.verify(token, secret)
Deploy Frontend	Vercel / Netlify
Deploy Backend	Render / Railway
Database (Prod)	MongoDB Atlas

"Practice turns knowledge into opportunity."