EGOTECHWORLD.COM

Beginner Tutorial Series - 2026 Edition

MERN Stack CRUD

with Admin & User Registration

A complete step-by-step guide for absolute beginners

MongoDB - Express.js - React.js - Node.js

Local Setup | Free Hosting | JWT Auth | Admin Panel

What you'll build in this tutorial:

- Full MERN CRUD app: Create, Read, Update, Delete operations
- User registration with email & password (with bcrypt hashing)
- Login system using JWT (JSON Web Tokens)
- Admin dashboard - manage all users & posts
- Role-based access (admin vs normal user)
- Folder structures explained step by step
- Run on localhost first, then deploy free in 2026

Table of Contents

How to use this tutorial

Read each chapter in order and type the code yourself - do not copy and paste. Typing helps your brain remember syntax. Every chapter builds on the previous one, so do not skip ahead.

CHAPTER 01

What is the MERN Stack?

MERN is a collection of four JavaScript technologies used to build modern web applications. The big advantage is that you only need to learn **one programming language - JavaScript** - to build both the frontend (what users see) and the backend (the server). This tutorial is written for absolute beginners in 2026, so do not worry if any of these names feel new.

The four parts of MERN

Letter	Tool	What it does
M	**MongoDB**	The database. It stores your data (users, posts, etc.) as flexible JSON-like documents.
E	**Express.js**	A small framework that runs on top of Node.js to make building APIs easy.
R	**React.js**	The frontend library. Builds the user interface with reusable components.
N	**Node.js**	Runs JavaScript outside the browser - this is your backend server engine.

How the parts talk to each other

Imagine ordering food at a restaurant. **React** is the menu and your table - you see it, you tap it. When you place an order, **Express** is the waiter who takes the order to the kitchen. **Node.js** is the kitchen itself, and **MongoDB** is the fridge where the ingredients (your data) are stored. The waiter brings the food back to your table - that is how data flows from database to screen.

What is CRUD?

CRUD stands for **Create, Read, Update, Delete**. These are the four things almost every app does with data. When you post on Facebook, that is Create. When you scroll your feed, that is Read. Editing your profile is Update. Deleting a comment is Delete.

Why MERN in 2026?

Even with new frameworks like Next.js 15 and Bun appearing, MERN is still the most beginner-friendly full-stack you can learn. Tons of tutorials, tons of jobs (especially remote), and the entire stack runs on free tools. Once you understand MERN, picking up Next.js or any other modern stack becomes much easier.

CHAPTER 02

Tools You Need to Install

Before you write a single line of code, you need to install five things on your computer. Take your time with this chapter - if the tools are not set up correctly, nothing else will work. All versions below are the recommended LTS or stable versions for May 2026.

2.1 Node.js (version 22 LTS)

Node.js lets you run JavaScript on your computer (not just in the browser). It also includes **npm**, which downloads packages (other people's code) for you.

How to install:

1. Go to **nodejs.org**
2. Download the **22.x LTS** version for your operating system
3. Run the installer and click Next on every screen
4. Open Command Prompt (Windows) or Terminal (Mac/Linux) and verify:

```
node --version
# Should print: v22.x.x

npm --version
# Should print: 10.x.x or higher
```

Tip - Use nvm if you can

On Mac and Linux, install **nvm** (Node Version Manager) instead of the direct installer. It lets you switch Node versions easily later. On Windows, search for 'nvm-windows' on GitHub.

2.2 MongoDB Community Server (version 7+)

This is the database that runs on your own computer. We will use the free Community Edition. (Later in Chapter 11 you will switch to the free cloud version - MongoDB Atlas - for hosting.)

How to install:

1. Go to **mongodb.com/try/download/community**
2. Choose your OS and download the MSI/PKG installer
3. During install, tick **'Install MongoDB as a service'**
4. Also install **MongoDB Compass** (a free GUI to view your data)

```
# After install, verify it is running:
mongosh
# You should see a prompt like:  test>
# Type 'exit' to leave
```

2.3 Visual Studio Code (your editor)

VS Code is a free code editor by Microsoft. Download from **code.visualstudio.com**. After installing, install these extensions (click the squares icon on the left, search and click Install):

Extension	Why you need it
ES7+ React/Redux snippets	Type 'rafce' to make a React component fast
Prettier - Code formatter	Auto-format your code on save
MongoDB for VS Code	Browse your database inside VS Code
Thunder Client	Test your API endpoints (like Postman, but inside VS Code)
GitHub Copilot (optional)	AI helper - free for students in 2026

2.4 Git (for version control)

Git tracks changes in your code. You will also need it later to push your project to GitHub for free hosting. Install from **git-scm.com** - just click Next on every screen.

```
# Verify after install:
git --version
# Configure your name & email (one time):
git config --global user.name "Your Name"
git config --global user.email "you@email.com"
```

2.5 A modern web browser

Use Google Chrome, Brave, or Firefox. We will be opening the developer tools (press F12) often to check errors and the network tab.

Quick check - is everything ready?

Open a fresh terminal and run all three commands: **node -v**, **npm -v**, **mongosh**. If all three respond without errors, you are ready for Chapter 3. If one fails, fix it before moving on.

CHAPTER 03

Project Folder Structure

Good folder structure is half the battle. We will keep the backend and frontend in **two separate folders** inside one parent project. This is the cleanest, most common pattern in 2026.

Create the parent folder

```
# Open your terminal and run:
mkdir mern-crud-app
cd mern-crud-app
```

Final structure (after all chapters)

```
mern-crud-app/
|
+-- backend/
|   +-- config/
|   |   +-- db.js                 # MongoDB connection
|   +-- controllers/
|   |   +-- authController.js     # register, login logic
|   |   +-- postController.js     # CRUD logic for posts
|   |   +-- userController.js     # admin user management
|   +-- middleware/
|   |   +-- authMiddleware.js     # protect & adminOnly
|   +-- models/
|   |   +-- User.js               # User schema
|   |   +-- Post.js               # Post schema
|   +-- routes/
|   |   +-- authRoutes.js
|   |   +-- postRoutes.js
|   |   +-- userRoutes.js
|   +-- .env                      # secret keys (never commit)
|   +-- .gitignore
|   +-- package.json
|   +-- server.js                 # main entry point
|
+-- frontend/
|   +-- public/
|   +-- src/
|   |   +-- components/
|   |   |   +-- Navbar.jsx
|   |   |   +-- ProtectedRoute.jsx
|   |   +-- pages/
|   |   |   +-- Home.jsx
|   |   |   +-- Login.jsx
|   |   |   +-- Register.jsx
|   |   |   +-- Dashboard.jsx
|   |   |   +-- AdminPanel.jsx
|   |   +-- context/
|   |   |   +-- AuthContext.jsx   # global login state
|   |   +-- api/
|   |   |   +-- axios.js          # axios setup
|   |   +-- App.jsx
|   |   +-- main.jsx
|   +-- .env
|   +-- index.html
|   +-- package.json
|   +-- vite.config.js
|
+-- README.md
```

Why this structure?

Separation of concerns. Each folder has one job. **controllers/** hold the logic. **models/** describe how data looks. **routes/** only decide which URL goes to which controller. When your app grows to 50 files, this discipline saves you hours.

Beginner tip

Do not create all these folders by hand right now. We will create them one by one in the chapters ahead, and you will understand each one as you build it. This page is just a map.

CHAPTER 04

Setting Up MongoDB Locally

In Chapter 2 you installed MongoDB. Now we will check it is running and create a database for our project. MongoDB stores data in **collections** (similar to tables in MySQL) and each row is called a **document** (a JSON object).

Step 1 - Start MongoDB

If you ticked 'Install as a service' during install, MongoDB is already running in the background every time your computer starts. To verify:

```
# In any terminal:
mongosh

# You should see something like:
# Current Mongosh Log ID: ...
# Connecting to: mongodb://127.0.0.1:27017/
# test>
```

Step 2 - Create our database

Inside the **mongosh** prompt, run these commands one by one:

```
# Switch to (or create) our database
use mern_crud_db

# Insert one test document so the DB actually saves
db.test.insertOne({ message: 'Hello MERN' })

# List your databases
show dbs

# Exit
exit
```

Step 3 - Use Compass to view data visually

Open **MongoDB Compass** from your start menu. The connection string is already filled in:

```
mongodb://localhost:27017
```

Click **Connect**. You will see **mern_crud_db** in the list. Click it and you can browse documents like a spreadsheet. This is your visual window into the database while you develop.

Connection string explained

mongodb:// = the protocol.
localhost = your own computer.
27017 = the default port MongoDB listens on.
Later, for cloud hosting, this URL will change to something starting with **mongodb+srv://**.

CHAPTER 05

Building the Backend

Time to write code! The backend is a Node.js + Express server that talks to MongoDB and exposes a REST API. The React frontend will call this API later.

Step 1 - Create the backend folder

```
# From inside mern-crud-app/
mkdir backend
cd backend

# Create a package.json (accept all defaults with -y)
npm init -y
```

Step 2 - Install backend packages

```
npm install express mongoose dotenv cors bcryptjs jsonwebtoken
npm install --save-dev nodemon
```

What each package does:

Package	Purpose
express	Web server framework
mongoose	Talks to MongoDB easily
dotenv	Loads secret values from a .env file
cors	Lets the React frontend call this API
bcryptjs	Hashes passwords securely
jsonwebtoken	Creates JWT login tokens
nodemon (dev)	Auto-restarts server when you save

Step 3 - Create the .env file

Create a file named **.env** in the backend folder. Never share this file or commit it to GitHub - it holds secrets.

```
PORT=5000
MONGO_URI=mongodb://localhost:27017/mern_crud_db
JWT_SECRET=this_is_a_long_random_string_change_me_now
NODE_ENV=development
```

Step 4 - Create config/db.js

```
// backend/config/db.js
const mongoose = require('mongoose');

const connectDB = async () => {
  try {
    const conn = await mongoose.connect(process.env.MONGO_URI);
    console.log(`MongoDB connected: ${conn.connection.host}`);
  } catch (error) {
    console.error(`Error: ${error.message}`);
    process.exit(1);
  }
};

module.exports = connectDB;
```

Logic explained: *mongoose.connect()* is an async operation (it takes time to reach the database), so we use **async/await**. If something goes wrong, we log it and stop the server with *process.exit(1)*.

Step 5 - Create models/User.js

A Mongoose model is a blueprint that says what fields each user document must have.

```
// backend/models/User.js
const mongoose = require('mongoose');
const bcrypt = require('bcryptjs');

const userSchema = new mongoose.Schema({
  name:     { type: String, required: true },
  email:    { type: String, required: true, unique: true },
  password: { type: String, required: true },
  role:     {
    type: String,
    enum: ['user', 'admin'],
    default: 'user'
  },
}, { timestamps: true });
```

Password hashing methods (same file)

```
// Hash password BEFORE saving
userSchema.pre('save', async function (next) {
  if (!this.isModified('password')) return next();
  const salt = await bcrypt.genSalt(10);
  this.password = await bcrypt.hash(this.password, salt);
  next();
});

// Method to check password at login
userSchema.methods.matchPassword = async function (entered) {
  return await bcrypt.compare(entered, this.password);
};

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

Why we hash passwords

If your database is ever leaked, plain-text passwords would let hackers log in as your users on other websites too (since people reuse passwords). **bcrypt** turns 'hello123' into something like '$2a$10$abc...' that cannot be reversed. Even you, the admin, cannot see the original password.

Step 6 - Create models/Post.js

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

const postSchema = new mongoose.Schema({
  title:   { type: String, required: true },
  content: { type: String, required: true },
  author:  {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User',
    required: true
  },
}, { timestamps: true });

module.exports = mongoose.model('Post', postSchema);
```

Logic: *author* is a **reference** to a User. We store only the user's ID, not their whole record. Later we can *populate* it to get full author details when we need them.

Step 7 - Create server.js (the entry point)

```
// backend/server.js
require('dotenv').config();
const express = require('express');
const cors = require('cors');
const connectDB = require('./config/db');

const app = express();

// Connect to MongoDB
connectDB();

// Middleware
app.use(cors());
app.use(express.json());            // parse JSON request bodies

// Routes (we will add these next)
app.use('/api/auth', require('./routes/authRoutes'));
app.use('/api/posts', require('./routes/postRoutes'));
app.use('/api/users', require('./routes/userRoutes'));

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

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

Step 8 - Update package.json scripts

Open **package.json** and replace the *scripts* section:

```
"scripts": {
  "start": "node server.js",
  "dev": "nodemon server.js"
}
```

CHAPTER 06

Registration & Login

Now we will build the most important part of any real app: letting people sign up and log in. We will use **JWT (JSON Web Tokens)** - the standard way in 2026 - so users stay logged in across page refreshes.

How JWT works (simple version)

1. User sends email + password to **/api/auth/login**.
2. Server checks them. If correct, server creates a **token** - a long string of letters - and sends it back.
3. The browser stores this token in *localStorage*.
4. Every time the user wants to do something protected (like delete a post), the browser sends the token along with the request.
5. The server checks the token is valid before allowing the action.

Step 1 - controllers/authController.js (register)

```
// backend/controllers/authController.js
const User = require('../models/User');
const jwt = require('jsonwebtoken');

// Helper - create a token
const generateToken = (id) => {
  return jwt.sign({ id }, process.env.JWT_SECRET,
                  { expiresIn: '7d' });
};

// @desc   Register new user
// @route  POST /api/auth/register
exports.registerUser = async (req, res) => {
  try {
    const { name, email, password } = req.body;

    const exists = await User.findOne({ email });
    if (exists)
      return res.status(400).json({
        message: 'Email already used'
      });

    const user = await User.create({ name, email, password });
    res.status(201).json({
      _id: user._id,
      name: user.name,
      email: user.email,
      role: user.role,
      token: generateToken(user._id),
    });
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
};
```

Login function (same file, continued)

```
// @desc   Login user
// @route  POST /api/auth/login
exports.loginUser = async (req, res) => {
  try {
    const { email, password } = req.body;
    const user = await User.findOne({ email });

    if (user && (await user.matchPassword(password))) {
      res.json({
        _id: user._id,
        name: user.name,
        email: user.email,
        role: user.role,
        token: generateToken(user._id),
      });
    } else {
      res.status(401).json({
        message: 'Invalid email or password'
      });
    }
  } catch (err) {
    res.status(500).json({ message: err.message });
  }
};
```

Step 2 - routes/authRoutes.js

```
// backend/routes/authRoutes.js
const express = require('express');
const router = express.Router();
const { registerUser, loginUser } = require('../controllers/authController');

router.post('/register', registerUser);
router.post('/login', loginUser);

module.exports = router;
```

Step 3 - middleware/authMiddleware.js

Middleware = a function that runs before a route handler. We use it to read the token and check who the user is.

```
// backend/middleware/authMiddleware.js
const jwt = require('jsonwebtoken');
const User = require('../models/User');

// Logged-in users only
exports.protect = async (req, res, next) => {
  let token;
  if (req.headers.authorization &&
      req.headers.authorization.startsWith('Bearer')) {
    try {
      token = req.headers.authorization.split(' ')[1];
      const decoded = jwt.verify(token, process.env.JWT_SECRET);
      req.user = await User.findById(decoded.id).select('-password');
      next();
    } catch (err) {
      res.status(401).json({ message: 'Token failed' });
    }
  } else {
    res.status(401).json({ message: 'No token, not authorized' });
  }
};

// Admins only
exports.adminOnly = (req, res, next) => {
  if (req.user && req.user.role === 'admin') return next();
  res.status(403).json({ message: 'Admins only' });
};
```

Step 4 - Test it with Thunder Client

Run **npm run dev** in the backend folder. Open Thunder Client in VS Code, click **New Request**, set method to **POST**, URL to **http://localhost:5000/api/auth/register**, go to **Body** -> **JSON**, paste:

```
{
  "name": "Test User",
  "email": "test@email.com",
  "password": "123456"
}
```

Click **Send**. You should get back the user object with a token. If yes - your auth system works!

> **Common error**
>
> If you get a CORS error or 'cannot connect', check that **app.use(cors())** is in server.js BEFORE the routes, and that **npm run dev** is still running without errors.

CHAPTER 07

Building the Frontend

Now we shift to the frontend. We will use **Vite** (the modern replacement for create-react-app) to scaffold a React 19 project. Vite is much faster.

Step 1 - Create the React app

```
# Go BACK to the parent folder mern-crud-app/
cd ..

# Create the React app
npm create vite@latest frontend -- --template react

cd frontend
npm install
npm install axios react-router-dom
```

Why these packages?

axios = library to call our backend API.
react-router-dom = lets us have multiple pages (/login, /dashboard, etc.) inside one React app.

Step 2 - Set the API base URL

Create **frontend/.env** with this single line:

```
VITE_API_URL=http://localhost:5000/api
```

Step 3 - Create src/api/axios.js

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

const api = axios.create({
  baseURL: import.meta.env.VITE_API_URL,
});

// Attach token to every request automatically
api.interceptors.request.use((config) => {
  const user = JSON.parse(localStorage.getItem('user'));
  if (user?.token) {
    config.headers.Authorization = `Bearer ${user.token}`;
  }
  return config;
});

export default api;
```

Logic: Instead of writing the token by hand on every request, this *interceptor* automatically attaches it. Write less, do more.

Step 4 - Create src/context/AuthContext.jsx

React Context lets us share login state across all pages without passing props through every component.

```
// frontend/src/context/AuthContext.jsx
import { createContext, useState, useContext } from 'react';
import api from '../api/axios';

const AuthContext = createContext();

export const AuthProvider = ({ children }) => {
  const [user, setUser] = useState(
    JSON.parse(localStorage.getItem('user'))
  );

  const login = async (email, password) => {
    const { data } = await api.post('/auth/login',
                                    { email, password });
    localStorage.setItem('user', JSON.stringify(data));
    setUser(data);
  };

  const register = async (name, email, password) => {
    const { data } = await api.post('/auth/register',
                                    { name, email, password });
    localStorage.setItem('user', JSON.stringify(data));
    setUser(data);
  };

  const logout = () => {
    localStorage.removeItem('user');
    setUser(null);
  };

  return (
    <AuthContext.Provider
      value={{ user, login, register, logout }}>
      {children}
    </AuthContext.Provider>
  );
};

export const useAuth = () => useContext(AuthContext);
```

Step 5 - Register page (src/pages/Register.jsx)

```
// frontend/src/pages/Register.jsx
import { useState } from 'react';
import { useNavigate, Link } from 'react-router-dom';
import { useAuth } from '../context/AuthContext';

export default function Register() {
  const [form, setForm] = useState({ name:'', email:'', password:'' });
  const [error, setError] = useState('');
  const { register } = useAuth();
  const navigate = useNavigate();

  const handleChange = (e) =>
    setForm({ ...form, [e.target.name]: e.target.value });

  const handleSubmit = async (e) => {
    e.preventDefault();
    try {
      await register(form.name, form.email, form.password);
      navigate('/dashboard');
    } catch (err) {
      setError(err.response?.data?.message || 'Failed');
    }
  };

  return (
    <div className='form-card'>
      <h2>Create Account</h2>
      {error && <p className='err'>{error}</p>}
      <form onSubmit={handleSubmit}>
        <input name='name' placeholder='Name'
               value={form.name} onChange={handleChange} required />
        <input name='email' type='email' placeholder='Email'
               value={form.email} onChange={handleChange} required />
        <input name='password' type='password' placeholder='Password'
               value={form.password} onChange={handleChange} required />
        <button type='submit'>Register</button>
      </form>
      <p>Have an account? <Link to='/login'>Login</Link></p>
    </div>
  );
}
```

Step 6 - App.jsx with routing

```
// frontend/src/App.jsx
import { BrowserRouter, Routes, Route } from 'react-router-dom';
import { AuthProvider } from './context/AuthContext';
import Navbar from './components/Navbar';
import Home from './pages/Home';
import Login from './pages/Login';
import Register from './pages/Register';
import Dashboard from './pages/Dashboard';
import AdminPanel from './pages/AdminPanel';
import ProtectedRoute from './components/ProtectedRoute';

export default function App() {
  return (
    <AuthProvider>
      <BrowserRouter>
        <Navbar />
        <Routes>
          <Route path='/' element={<Home />} />
          <Route path='/login' element={<Login />} />
          <Route path='/register' element={<Register />} />
          <Route path='/dashboard' element={
            <ProtectedRoute><Dashboard /></ProtectedRoute>} />
          <Route path='/admin' element={
            <ProtectedRoute admin><AdminPanel /></ProtectedRoute>} />
        </Routes>
      </BrowserRouter>
    </AuthProvider>
  );
}
```

CHAPTER 08

CRUD Operations

Now we will let logged-in users **Create, Read, Update, Delete** their own posts. This is the heart of CRUD.

Step 1 - Backend post controller

```
// backend/controllers/postController.js
const Post = require('../models/Post');

// CREATE
exports.createPost = async (req, res) => {
  const { title, content } = req.body;
  const post = await Post.create({
    title, content, author: req.user._id,
  });
  res.status(201).json(post);
};

// READ ALL (public)
exports.getPosts = async (req, res) => {
  const posts = await Post.find()
    .populate('author', 'name email');
  res.json(posts);
};

// READ MINE
exports.getMyPosts = async (req, res) => {
  const posts = await Post.find({ author: req.user._id });
  res.json(posts);
};
```

Update and Delete (same file, continued)

```
// UPDATE
exports.updatePost = async (req, res) => {
  const post = await Post.findById(req.params.id);
  if (!post) return res.status(404).json({ message: 'Not found' });

  // Only owner OR admin can edit
  const isOwner = post.author.toString() === req.user._id.toString();
  if (!isOwner && req.user.role !== 'admin')
    return res.status(403).json({ message: 'Not your post' });

  post.title = req.body.title || post.title;
  post.content = req.body.content || post.content;
  const saved = await post.save();
  res.json(saved);
};

// DELETE
exports.deletePost = async (req, res) => {
  const post = await Post.findById(req.params.id);
  if (!post) return res.status(404).json({ message: 'Not found' });

  const isOwner = post.author.toString() === req.user._id.toString();
  if (!isOwner && req.user.role !== 'admin')
    return res.status(403).json({ message: 'Not your post' });

  await post.deleteOne();
  res.json({ message: 'Post deleted' });
};
```

Step 2 - Post routes

```
// backend/routes/postRoutes.js
const express = require('express');
const router = express.Router();
const { protect } = require('../middleware/authMiddleware');
const c = require('../controllers/postController');

router.get('/', c.getPosts);
router.get('/mine', protect, c.getMyPosts);
router.post('/', protect, c.createPost);
router.put('/:id', protect, c.updatePost);
router.delete('/:id', protect, c.deletePost);

module.exports = router;
```

Step 3 - Frontend Dashboard (state & handlers)

```
// frontend/src/pages/Dashboard.jsx
import { useEffect, useState } from 'react';
import api from '../api/axios';

export default function Dashboard() {
  const [posts, setPosts] = useState([]);
  const [form, setForm] = useState({ title: '', content: '' });
  const [editId, setEditId] = useState(null);

  // READ
  const loadPosts = async () => {
    const { data } = await api.get('/posts/mine');
    setPosts(data);
  };
  useEffect(() => { loadPosts(); }, []);

  // CREATE or UPDATE (same form, decided by editId)
  const submit = async (e) => {
    e.preventDefault();
    if (editId) {
      await api.put(`/posts/${editId}`, form);
    } else {
      await api.post('/posts', form);
    }
    setForm({ title: '', content: '' });
    setEditId(null);
    loadPosts();
  };

  // DELETE
  const remove = async (id) => {
    if (!confirm('Delete this post?')) return;
    await api.delete(`/posts/${id}`);
    loadPosts();
  };

  // Start editing - load a post into the form
  const startEdit = (post) => {
    setEditId(post._id);
    setForm({ title: post.title, content: post.content });
  };
```

Dashboard JSX (return part)

```
  return (
    <div className='container'>
      <h2>My Posts</h2>

      <form onSubmit={submit}>
        <input value={form.title}
          onChange={(e)=>setForm({...form, title:e.target.value})}
          placeholder='Title' required />
        <textarea value={form.content}
          onChange={(e)=>setForm({...form, content:e.target.value})}
          placeholder='Content' required />
        <button>{editId ? 'Update' : 'Create'}</button>
      </form>

      <ul>
        {posts.map(p => (
          <li key={p._id}>
            <h3>{p.title}</h3>
            <p>{p.content}</p>
            <button onClick={()=>startEdit(p)}>Edit</button>
            <button onClick={()=>remove(p._id)}>Delete</button>
          </li>
        ))}
      </ul>
    </div>
  );
}
```

JavaScript logic explained

useEffect runs once when the page loads -> fetches the posts.
useState holds the form values, the post list, and which post (if any) is being edited.
When the user submits, we check **editId**: if it has a value, we PUT to update; otherwise we POST to create. This is a classic single-form create/edit pattern.

CHAPTER 09

Admin Panel

Admins can do everything a normal user can do, plus see all users and delete any user. We already built the **adminOnly** middleware in Chapter 6 - now we will use it.

Step 1 - User controller (admin endpoints)

```
// backend/controllers/userController.js
const User = require('../models/User');

// Get all users (admin only)
exports.getAllUsers = async (req, res) => {
  const users = await User.find().select('-password');
  res.json(users);
};

// Delete a user (admin only)
exports.deleteUser = async (req, res) => {
  const user = await User.findById(req.params.id);
  if (!user) return res.status(404).json({ message: 'Not found' });
  await user.deleteOne();
  res.json({ message: 'User deleted' });
};

// Promote user to admin
exports.makeAdmin = async (req, res) => {
  const user = await User.findById(req.params.id);
  if (!user) return res.status(404).json({ message: 'Not found' });
  user.role = 'admin';
  await user.save();
  res.json({ message: 'User promoted to admin' });
};
```

Step 2 - User routes

```
// backend/routes/userRoutes.js
const express = require('express');
const router = express.Router();
const { protect, adminOnly } = require('../middleware/authMiddleware');
const c = require('../controllers/userController');

router.get('/', protect, adminOnly, c.getAllUsers);
router.delete('/:id', protect, adminOnly, c.deleteUser);
router.put('/:id/promote', protect, adminOnly, c.makeAdmin);

module.exports = router;
```

Step 3 - ProtectedRoute on the frontend

```
// frontend/src/components/ProtectedRoute.jsx
import { Navigate } from 'react-router-dom';
import { useAuth } from '../context/AuthContext';

export default function ProtectedRoute({ children, admin = false }) {
  const { user } = useAuth();

  if (!user) return <Navigate to='/login' />;
  if (admin && user.role !== 'admin') return <Navigate to='/dashboard' />;

  return children;
}
```

Step 4 - AdminPanel page (logic)

```
// frontend/src/pages/AdminPanel.jsx
import { useEffect, useState } from 'react';
import api from '../api/axios';

export default function AdminPanel() {
  const [users, setUsers] = useState([]);

  const load = async () => {
    const { data } = await api.get('/users');
    setUsers(data);
  };
  useEffect(() => { load(); }, []);

  const remove = async (id) => {
    if (!confirm('Delete user?')) return;
    await api.delete(`/users/${id}`);
    load();
  };

  const promote = async (id) => {
    await api.put(`/users/${id}/promote`);
    load();
  };
```

AdminPanel JSX (return part)

```
  return (
    <div className='container'>
      <h2>Admin Panel - All Users</h2>
      <table>
        <thead>
          <tr>
            <th>Name</th><th>Email</th>
            <th>Role</th><th>Actions</th>
          </tr>
        </thead>
        <tbody>
          {users.map(u => (
            <tr key={u._id}>
              <td>{u.name}</td>
              <td>{u.email}</td>
              <td>{u.role}</td>
              <td>
                {u.role !== 'admin' &&
                  <button onClick={()=>promote(u._id)}>
                    Promote
                  </button>}
                <button onClick={()=>remove(u._id)}>
                  Delete
                </button>
              </td>
            </tr>
          ))}
        </tbody>
      </table>
    </div>
  );
}
```

Step 5 - How to make the FIRST admin

Since promoting users is admin-only, you need at least one admin to start. The simplest way: register a user normally, then promote them manually using **mongosh**:

```
mongosh
use mern_crud_db
db.users.updateOne(
  { email: "you@email.com" },
  { $set: { role: "admin" } }
)
```

Now log in as that user. You will see the Admin link in the navbar and can manage everyone else.

CHAPTER 10 Running Locally

Time for the moment of truth - running your full MERN app on your computer. You will need **two terminals open at the same time**.

Terminal 1 - Backend

```
cd mern-crud-app/backend
npm run dev

# Expected output:
# Server on port 5000
# MongoDB connected: localhost
```

Terminal 2 - Frontend

```
cd mern-crud-app/frontend
npm run dev

# Expected output:
# VITE v5.x.x  ready in 432 ms
# Local:   http://localhost:5173/
```

Test the full flow

Open **http://localhost:5173** in your browser:

#	Action	Expected result
1	Click 'Register'	Form appears
2	Fill name, email, password and submit	Redirects to dashboard
3	Create a post	Post appears in the list
4	Click Edit, change title, submit	Title updates
5	Click Delete	Post is gone
6	Promote yourself in mongosh	role becomes 'admin'
7	Log out and log back in	Admin link is now visible
8	Visit /admin	All users list appears

Hooray! You did it.

If everything above works, congratulations - you have built a complete MERN CRUD app with authentication and admin features. This is a real, production-grade pattern used by thousands of apps.

CHAPTER 11

Free Hosting in 2026

Localhost is great for learning, but no one else can see it. To put your app on the internet, we will use three free services. None of them require a credit card to start.

Part	Free service	What it does
Database	**MongoDB Atlas (Free M0)**	512MB cloud database forever
Backend	**Render.com Free Web Service**	Hosts your Node API (sleeps after 15 min idle)
Frontend	**Vercel or Netlify**	Hosts your React build globally on a CDN

Step 1 - MongoDB Atlas (cloud database)

1. Go to **cloud.mongodb.com** and sign up.
2. Create a free cluster (M0 - select the closest region).
3. Under **Database Access**, create a user + password.
4. Under **Network Access**, click 'Allow access from anywhere' (0.0.0.0/0).
5. Click **Connect** -> Drivers -> copy the connection string. Replace *password* and add the database name:

```
mongodb+srv://USER:PASSWORD@cluster0.xyz.mongodb.net/mern_crud_db
```

Step 2 - Push code to GitHub

```
# In your project root
git init
git add .
git commit -m "MERN CRUD complete"

# Create a new repo on github.com, then:
git remote add origin https://github.com/YOU/mern-crud-app.git
git branch -M main
git push -u origin main
```

Add a .gitignore

Before pushing, create **.gitignore** in both backend/ and frontend/ with at least: *node_modules* and *.env*. Otherwise your secret keys will end up public on GitHub.

Step 3 - Deploy backend to Render

1. Go to **render.com** and sign up with GitHub.
2. Click **New +** -> **Web Service**.
3. Select your GitHub repo.
4. Set:
   Root Directory: backend
   Build Command: npm install
   Start Command: npm start
5. Under **Environment**, add three variables:

```
MONGO_URI    = mongodb+srv://...your atlas string...
JWT_SECRET   = same_long_random_string_as_local
NODE_ENV     = production
```

Click **Create Web Service**. Render will give you a URL like *https://mern-crud-app.onrender.com*. Test it - you should see 'API is running...'.

Step 4 - Deploy frontend to Vercel

1. Go to **vercel.com** and sign in with GitHub.
2. **Add New** -> **Project** -> select your repo.
3. Set **Root Directory: frontend**.
4. Add an environment variable:

```
VITE_API_URL = https://mern-crud-app.onrender.com/api
```

5. Click **Deploy**. In about a minute, your app is live at a *vercel.app* URL. Share it with the world!

Step 5 - Update CORS for production

Once you know your Vercel URL, lock down CORS to only that origin. Edit **backend/server.js**:

```
app.use(cors({
  origin: [
    'http://localhost:5173',
    'https://your-app.vercel.app',
  ],
  credentials: true,
}));
```

Push to GitHub - Render auto-redeploys. Done.

About Render's free sleep

The free Render plan puts your backend to sleep after 15 minutes of no traffic. The first request after waking takes ~30 seconds. For a learning project this is fine. To keep it always-on, upgrade to Render's $7/month plan or self-host on a VPS.

CHAPTER 12

Common Errors & Fixes

Every developer gets errors - it is part of the job. Here are the most common ones beginners hit with MERN, and exactly how to fix them.

Error message	Cause & Fix
`MongooseError: Operation buffering timed out`	MongoDB is not running. Start the service or check MONGO_URI.
`CORS policy: No 'Access-Control-Allow-Origin'`	You forgot app.use(cors()) in server.js, or the order is wrong - cors must come BEF
`Cannot find module 'express'`	You forgot to run npm install, or you are in the wrong folder.
`JsonWebTokenError: invalid signature`	JWT_SECRET in .env was changed. Old tokens are now invalid. Log out and log ba
`EADDRINUSE :::5000`	Another process is using port 5000. Either stop it (kill the old Node) or change POR
`TypeError: Cannot read properties of undefined`	You used a value that is null or undefined. Use optional chaining (user?.name) or ch
`Network Error in axios`	Backend is not running, or VITE_API_URL points to the wrong URL. Check both ter
`E11000 duplicate key error`	You tried to register an email that already exists. The 'unique' rule on the schema re
`404 Not Found on a route`	Check the URL spelling in your axios call AND in your routes file. Method (GET vs P
`Render: Build fails - 'cannot find package.json'`	Set Root Directory to 'backend' (not the repo root) in Render settings.

Final words

You have just walked through a complete real-world MERN application: registration, login, JWT auth, full CRUD, role-based admin, and deployment. The skills you learned here transfer directly to almost every Node.js + React job posting in 2026. Keep building. Build a todo app, a blog, a chat app - repetition is what locks knowledge in.

Where to go next

1. Add image uploads with Multer + Cloudinary.
2. Replace localStorage with httpOnly cookies for better security.
3. Add pagination on the post list.
4. Move from REST to GraphQL or tRPC when you feel ready.
5. Visit egotechworld.com for more tutorials, source code, and free courses.

Thank you for reading. Built with care by EgoTechWorld in 2026.