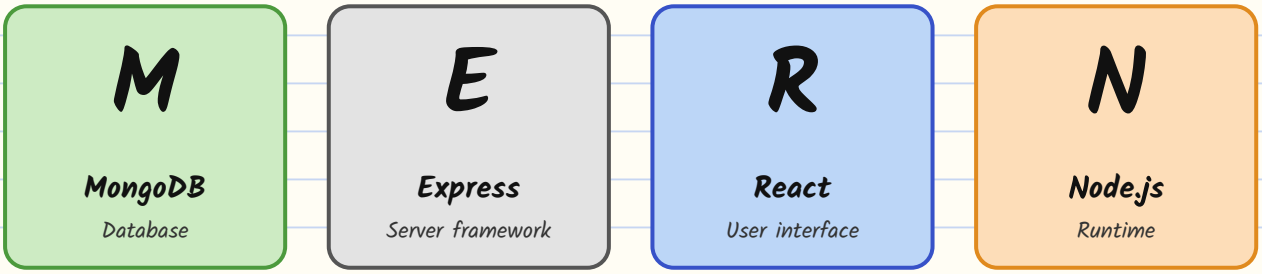
MERN Stack

MongoDB · Express · React · Node.js, explained simply

MERN Stack Notes · easy to understand · free for students

I. The Four Parts of the Stack

- **MERN** is a full stack made only with **JavaScript**. One language for the database queries, the server and the screen.



IDEA A food delivery app. React is the app screen you tap, Express on Node is the restaurant manager taking orders, MongoDB is the register where every order is written down.



Port 5173 (React) to port 5000 (API) to port 27017 (MongoDB)

Part	Job	Main tools
MongoDB	Store data as documents	Atlas, Compass, Mongoose
Express	Routes, middleware, API	cors, helmet, express.json
React	Build the user interface	Hooks, Router, axios
Node.js	Run JavaScript on the server	npm, dotenv, nodemon

- Why MERN? One language everywhere, JSON flows from database to screen with no changes, huge npm ecosystem, and plenty of jobs.

2. Project Setup and Folders

- Install the **Node LTS**, **VS Code**, **Git** and **Postman**. Use MongoDB **Atlas** (free cloud) or a local MongoDB with Compass.

EXAMPLE Recommended folder layout. Two apps in one repo.

```
mern-app/
  server/                    client/
    config/db.js               src/
    models/                      api/   components/   pages/
    controllers/                 context/   App.jsx   main.jsx
    routes/  middleware/       index.html
    server.js  .env            .env
```

Where	Command
server/	`npm init -y` then `npm i express mongoose cors dotenv bcrypt jsonwebtoken`
server/	`npm i -D nodemon` and set `"type": "module"` in package.json
client/	`npm create vite@latest client -- --template react`
client/	`npm i react-router-dom axios`

- Run both at once in two terminals: `npm run dev` in server (port 5000) and `npm run dev` in client (port 5173).

3. Backend with Express and Mongoose

- The backend follows a simple pattern. **Model** describes data, **controller** has the logic, **route** connects a URL to a controller.

EXAMPLE config/db.js and server.js.

```
// config/db.js
import mongoose from "mongoose";
export default async function connectDB() {
  await mongoose.connect(process.env.MONGO_URI);
  console.log("MongoDB connected");
}

// server.js
import "dotenv/config";
import express from "express";
import cors from "cors";
import connectDB from "./config/db.js";
import taskRoutes from "./routes/tasks.js";

await connectDB();
const app = express();
app.use(cors({ origin: "http://localhost:5173" }));
app.use(express.json());
app.use("/api/tasks", taskRoutes);
app.listen(process.env.PORT || 5000, () => console.log("API running"));
```

EXAMPLE server/.env (never push this to Git).

```
PORT=5000
MONGO_URI=mongodb://127.0.0.1:27017/mern-tasks
JWT_SECRET=change_me_to_a_long_random_text
```

EXAMPLE A model with a link to another collection.

```
const taskSchema = new mongoose.Schema(
  {
    title: { type: String, required: true, trim: true },
    done:  { type: Boolean, default: false },
    user:  { type: mongoose.Schema.Types.ObjectId, ref: "User" }
  },
  { timestamps: true }
);
export default mongoose.model("Task", taskSchema);
```

4. REST API Design

Method	Route	Action	Code
GET	/api/tasks	List tasks	200
POST	/api/tasks	Create a task	201
PATCH	/api/tasks/:id	Change part of a task	200
DELETE	/api/tasks/:id	Remove a task	204

- Use plural nouns, keep one error shape like `{ message: '...' }`, and return `404` when an id is not found, `400` for bad input, `401` when not logged in.

- Search and pagination: `GET /api/tasks?q=milk&page=2&limit=10`. Use `.skip((page - 1) * limit).limit(limit)` in Mongoose.
- **Test every route in Postman first**, before you write any React code. If the API works there, any later bug is in the frontend.

5. React Frontend Essentials

Concept	Remember
Component	Function returning JSX, name starts with a capital letter
Props	Data passed down from parent. Read-only
useState	Memory of a component. Changing it redraws the screen
useEffect	Run code after render, like loading data. Add the [ ] list
key	Unique value for each list item, like the MongoDB `_id`
Controlled input	Value comes from state, onChange updates state

EXAMPLE A reusable hook for loading data.

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

  useEffect(() => {
    api.get(url)
      .then(res => setData(res.data))
      .catch(err => setError(err.message))
      .finally(() => setLoading(false));
  }, [url]);

  return { data, loading, error };
}
```

- Never change state directly. Make a new copy: `setTasks([...tasks, newTask])`, `tasks.filter(...)`, `tasks.map(...)`.
- Always show a **loading** text and an **error** message. Real users have slow networks.

6. Routing and Forms in React

- **React Router** shows different pages without reloading. `Link` changes the URL, `Routes` picks the page, `useParams` reads `:id`, `useNavigate` moves in code.

EXAMPLE *Pages and routes.*

```
import { BrowserRouter, Routes, Route, Link } from "react-router-dom";

<BrowserRouter>
  <nav><Link to="/">Home</Link> <Link to="/login">Login</Link></nav>
  <Routes>
    <Route path="/" element={<Home />} />
    <Route path="/tasks/:id" element={<TaskPage />} />
    <Route path="*" element={<NotFound />} />
  </Routes>
</BrowserRouter>
```

EXAMPLE *One state object for a whole form.*

```
const [form, setForm] = useState({ email: "", password: "" });
const change = (e) => setForm({ ...form, [e.target.name]: e.target.value });

<input name="email" value={form.email} onChange={change} />
<input name="password" type="password" value={form.password} onChange={change} />
```

- *On submit call* `e.preventDefault()`*, send the data with axios, then use* `navigate('/')` *after success.*

7. Connecting Frontend to Backend

- *The React app (port 5173) calls the Express API (port 5000). Browsers treat these as* ***different origins****, so the server must allow it with* `cors()`*.*

EXAMPLE *One axios instance for the whole app.*

```
// client/src/api/api.js
import axios from "axios";
const api = axios.create({ baseURL: import.meta.env.VITE_API_URL });
export default api;

// client/.env
// VITE_API_URL=http://localhost:5000/api
```

	fetch	*axios*
Install	*Built in*	`npm i axios`
JSON result	*Need* `res.json()`	*Ready in* `res.data`
Bad status like 404	*Does not throw*	*Throws an error*
Headers for all calls	*Write every time*	*Interceptors*

Problem	Usual cause
CORS error in console	Missing `cors()` or wrong origin in server
Network Error	Server is not running or wrong port
404 on API call	Wrong URL, missing /api in the path
req.body is undefined	Forgot `app.use(express.json())`
env value is undefined	Vite needs the `VITE_` prefix. Restart the dev server

8. Mini Project, Task Manager

- Build one full CRUD app first. Backend files below, then the React screen.

EXAMPLE server/controllers/tasks.js

```
import Task from "../models/Task.js";

export const getTasks = async (req, res) => {
  const tasks = await Task.find().sort("-createdAt");
  res.json(tasks);
};
export const addTask = async (req, res) => {
  const task = await Task.create({ title: req.body.title });
  res.status(201).json(task);
};
export const toggleTask = async (req, res) => {
  const task = await Task.findById(req.params.id);
  if (!task) return res.status(404).json({ message: "Not found" });
  task.done = !task.done;
  res.json(await task.save());
};
export const deleteTask = async (req, res) => {
  await Task.findByIdAndDelete(req.params.id);
  res.sendStatus(204);
};
```

EXAMPLE server/routes/tasks.js

```
import { Router } from "express";
import * as c from "../controllers/tasks.js";

const router = Router();
router.route("/").get(c.getTasks).post(c.addTask);
router.route("/:id").patch(c.toggleTask).delete(c.deleteTask);
export default router;
```

EXAMPLE client/src/App.jsx, part 1 (state and API calls).

```
export default function App() {
  const [tasks, setTasks] = useState([]);
  const [title, setTitle] = useState("");

  useEffect(() => {
    api.get("/tasks").then(res => setTasks(res.data));
  }, []);

  const add = async (e) => {
    e.preventDefault();
    const { data } = await api.post("/tasks", { title });
    setTasks([data, ...tasks]);
    setTitle("");
  };
  const toggle = async (id) => {
    const { data } = await api.patch(`/tasks/${id}`);
    setTasks(tasks.map(t => (t._id === id ? data : t)));
  };
  const remove = async (id) => {
    await api.delete(`/tasks/${id}`);
    setTasks(tasks.filter(t => t._id !== id));
  };
```

EXAMPLE client/src/App.jsx, part 2 (the screen).

```
  return (
    <div>
      <form onSubmit={add}>
        <input value={title} onChange={e => setTitle(e.target.value)} />
        <button>Add</button>
      </form>
      {tasks.map(t => (
        <p key={t._id}>
          <span onClick={() => toggle(t._id)}>
            {t.done ? "[x] " : "[ ] "}{t.title}
          </span>
          <button onClick={() => remove(t._id)}>Delete</button>
        </p>
      ))}
    </div>
  );
}
```

9. Login and Protected Pages

- Flow: user registers, password is saved as a **bcrypt hash**, login returns a **JWT**, React saves it and sends `Authorization: Bearer token` with every request, and a **protect** middleware checks it.

EXAMPLE *server/controllers/auth.js*

```
const sign = (id) =>
  jwt.sign({ id }, process.env.JWT_SECRET, { expiresIn: "7d" });

export const register = async (req, res) => {
  const { name, email, password } = req.body;
  if (await User.findOne({ email }))
    return res.status(409).json({ message: "Email already used" });
  const hash = await bcrypt.hash(password, 10);
  const user = await User.create({ name, email, password: hash });
  res.status(201).json({ token: sign(user._id), name: user.name });
};

export const login = async (req, res) => {
  const { email, password } = req.body;
  const user = await User.findOne({ email });
  if (!user || !(await bcrypt.compare(password, user.password)))
    return res.status(401).json({ message: "Wrong email or password" });
  res.json({ token: sign(user._id), name: user.name });
};
```

EXAMPLE *server/middleware/protect.js*

```
export const protect = (req, res, next) => {
  const token = (req.headers.authorization || "").replace("Bearer ", "");
  try {
    req.userId = jwt.verify(token, process.env.JWT_SECRET).id;
    next();
  } catch {
    res.status(401).json({ message: "Not authorized" });
  }
};
// router.use(protect);   then use  Task.find({ user: req.userId })
```

EXAMPLE *Send the token automatically (axios interceptor).*

```
api.interceptors.request.use((config) => {
  const token = localStorage.getItem("token");
  if (token) config.headers.Authorization = `Bearer ${token}`;
  return config;
});
```

EXAMPLE *Block pages for logged out users.*

```
function PrivateRoute({ children }) {
  const { user } = useAuth();
  return user ? children : <Navigate to="/login" replace />;
}
// <Route path="/" element={<PrivateRoute><Dashboard /></PrivateRoute>} />
```

REMEMBER `localStorage` *is simple but scripts on the page can read it (XSS). httpOnly cookies are safer for real apps. Never trust the frontend check alone, always check on the server.*

10. Global State with Context

- **Context** shares data (like the logged in user) with any component, without passing props through every level.

EXAMPLE AuthContext.jsx

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

export function AuthProvider({ children }) {
  const [user, setUser] = useState(
    JSON.parse(localStorage.getItem("user") || "null")
  );
  const login = (data) => {
    localStorage.setItem("token", data.token);
    localStorage.setItem("user", JSON.stringify({ name: data.name }));
    setUser({ name: data.name });
  };
  const logout = () => { localStorage.clear(); setUser(null); };

  return (
    <AuthContext.Provider value={{ user, login, logout }}>
      {children}
    </AuthContext.Provider>
  );
}
// main.jsx:  <AuthProvider><App /></AuthProvider>
```

State type	Best tool
One component	useState
Complex logic in one component	useReducer
Few global values (user, theme)	Context
Large app state	Redux Toolkit or Zustand
Data from the server (cache, refetch)	TanStack Query

11. Uploads and Extra Features

EXAMPLE Image upload with multer.

```
// server
import multer from "multer";
const upload = multer({ dest: "uploads/" });
router.post("/avatar", protect, upload.single("avatar"), (req, res) => {
  res.json({ file: req.file.filename });
});

// client
const fd = new FormData();
fd.append("avatar", file);
await api.post("/users/avatar", fd);
```

- On most hosts the server disk is temporary. Store images on **Cloudinary** or **S3** and save only the image URL in MongoDB.

Feature	Tool
Real-time chat, live updates	Socket.io
Emails (welcome, reset)	Nodemailer
Payments	Razorpay or Stripe
Search and filters	Mongoose queries, `$regex`, text index
Charts	Recharts or Chart.js
Input checks	Zod, Joi or express-validator

12. Deployment and Best Practices

Part	Where	Key steps
Database	MongoDB Atlas	Create cluster, user, allow server IP, copy the URI
Backend	Render or Railway	Set env vars, start command `node server.js`, read `process.env.PORT`
Frontend	Vercel or Netlify	`npm run build`, set `VITE_API_URL` to the live API

- In the server, change the CORS origin to your **live frontend URL**. For React Router, add a rewrite so every path opens `index.html` (Netlify: `/* /index.html 200`).

Best practice	Why
Keep secrets in .env, add it to .gitignore	Leaked keys get abused fast
Hash passwords, keep tokens short lived	Protect users if data leaks
helmet, rate limiting, input validation	Stop common attacks
Central error handler and logging	Find bugs in production
Indexes on fields you search	Fast queries when data grows
Separate controllers from routes	Code stays clean and testable

13. Projects and Practice

Project	What you practice
1. Task manager	CRUD end to end (done above)
2. Notes app with login	JWT, protected routes, user owned data
3. Blog platform	Posts, comments, populate, pagination
4. E-commerce store	Cart, orders, payments, admin role
5. Chat app	Socket.io, real-time events, rooms
6. Social media mini	Likes, follows, image upload, feed

➤ Common mistakes

Mistake	Fix
Pushing .env to GitHub	Use .gitignore, rotate any leaked secret
Not awaiting database calls	Use async and await everywhere
Changing state directly	Make a new array or object
Returning password hash to React	Use `.select('-password')`
Hardcoding localhost URLs	Use env variables
Checking login only in React	Protect the API routes too

➤ Practice questions

1. What is the MERN stack? Role of each letter.
2. Explain the full flow of a request from React to MongoDB.
3. Why is CORS needed and how do you fix it?
4. What is middleware in Express? Write a protect middleware.
5. Why use Mongoose instead of the plain driver?
6. SQL vs MongoDB. When do you choose MongoDB?
7. How does JWT login work in a MERN app?
8. Context API vs Redux. When to use which?
9. How do you deploy a MERN app?
10. Build a Task Manager with login using MERN.

- Finish one complete app with login and deploy it. A live link on your resume is worth more than ten tutorials.