MERN Stack – Complete Beginner Documentation

This document is designed for **freshers** who are starting their journey in **Full Stack Web Development** using the **MERN Stack**. Every concept is explained in simple language with examples.

What is MERN Stack?

MERN is a popular **JavaScript-based full stack development technology**. It allows developers to build complete web applications using a single programming language: **JavaScript**.

MERN stands for: - **M** → MongoDB (Database) - **E** → Express.js (Backend Framework) - **R** → React.js (Frontend Library) - **N** → Node.js (Runtime Environment)

---

1. MongoDB

What is MongoDB?

MongoDB is a **NoSQL database** used to store application data. Unlike traditional databases, MongoDB stores data in **JSON-like documents** instead of tables.

Why MongoDB?

- Flexible schema (no fixed structure)
- Fast and scalable
- Easy to store JavaScript objects

Example MongoDB Document

```
{
  name: "Sudheer",
  marks: 90,
  totalMarks: 595,
  occupation: "Senior Doctor",
  salary: 500000
}
```

---

2. Express.js

What is Express?

Express.js is a **Node.js framework** used to build **REST APIs** easily. It helps handle HTTP requests like GET, POST, PUT, PATCH, and DELETE.

Why Express?

- Lightweight and fast
- Easy routing
- Middleware support

Installing Express

```
npm install express
```

Simple Express Server Example

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

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

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

---

3. API (Application Programming Interface)

An API allows **communication between frontend and backend**. Frontend sends a request, backend processes it, and sends a response.

Example: - Frontend → requests user data - Backend → fetches from database - Backend → sends response to frontend

---

4. HTTP Methods

GET – Fetch Data

Used to retrieve data from the server.

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

POST – Insert Data

Used to create new data. Data is sent securely (not stored in browser cache).

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

PUT – Update Entire Object

Replaces the whole object.

```
app.put('/users/:id', (req, res) => {
  res.send('User updated completely');
});
```

PATCH – Partial Update

Updates only selected fields.

```
app.patch('/users/:id', (req, res) => {
  res.send('User partially updated');
});
```

DELETE – Remove Data

Deletes a record from the database.

```
app.delete('/users/:id', (req, res) => {
  res.send('User deleted');
});
```

---

5. HTTP Status Codes

Status codes tell the **result of an API request**.

Code	Meaning
200	Success
201	Created (New Record Inserted)
400	Bad Request
401	Unauthorized
404	Not Found
500	Internal Server Error

---

6. React.js

What is React?

React is a **JavaScript library** used to build **User Interfaces (UI)**. It is component-based and fast.

Why React?

- Reusable components
- Fast rendering (Virtual DOM)
- Easy to manage UI state

Simple React Component

```
function App() {
  return <h1>Hello React</h1>;
}

export default App;
```

---

7. Node.js

What is Node.js?

Node.js is a **runtime environment** that allows you to run JavaScript **outside the browser**.

Why Node.js?

- Fast execution
- Non-blocking (async)
- Uses JavaScript everywhere

V8 Engine

Node.js uses the **V8 JavaScript Engine**, written in **C++**, which converts JavaScript into machine code for fast execution.

---

8. NPM (Node Package Manager)

What is NPM?

NPM is used to **install, manage, and share JavaScript packages**.

Common Commands

```
npm init
npm install express
npm i
```

Official Website: https://www.npmjs.com/

8.1 NPM Commands – Detailed Explanation

NPM (Node Package Manager) is used to manage libraries and dependencies in a Node.js project. Below are the most commonly used NPM commands explained clearly for beginners.

1️⃣ `npm init`

Purpose: Initialize a new Node.js project.

When you run `npm init`, npm creates a file called `package.json`, which stores all project-related information and dependencies.

```
npm init
```

It asks questions like project name, version, entry file, etc.

Shortcut command:

```
npm init -y
```

This skips all questions and creates `package.json` with default values.

2️⃣ `npm install express`

Purpose: Install a specific package (Express in this case).

This command downloads Express from the npm registry and installs it inside the `node_modules` folder.

```
npm install express
```

What happens internally: - `node_modules` folder is created - Express is added to `dependencies` in `package.json`

You can now use Express in code:

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

3️⃣ `npm i`

Purpose: Install all dependencies listed in `package.json`.

`npm i` is a short form of `npm install`.

```
npm i
```

Important points: - Reads `package.json` - Installs all dependencies automatically - Mostly used after cloning a project from GitHub

---

⚠️Differences Between NPM Commands (Interview Important)

Command	Use Case	Creates package.json	Installs packages
`npm init`	Create new project	✅Yes	❌No
`npm install express`	Install Express	❌No	✅Express only
`npm i`	Install all dependencies	❌No	✅From package.json

Real-Time Usage Flow

For a new project:

```
npm init -y
npm install express
```

For an existing project:

```
npm i
```

8.2 package.json File – Complete Explanation

What is `package.json`?

`package.json` is a **configuration file** for a Node.js project. It contains **metadata about the project** and manages all the **dependencies (packages)** required for the application.

Think of `package.json` as the **identity card + dependency list** of your project.

Why do we need `package.json`?

We need `package.json` because: - It defines **project details** (name, version, author) - It keeps track of **installed packages** - It allows others to **install dependencies easily** using `npm i` - It helps npm understand **how to run the project**

Without `package.json`, managing packages in real-world projects becomes very difficult.

How is `package.json` created?

It is created when you run:

```
npm init
```

Or

```
npm init -y
```

Sample `package.json` File

```
{
  "name": "mern-project",
  "version": "1.0.0",
```

```
  "description": "My first MERN application",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "dev": "nodemon index.js"
  },
  "author": "Sudheer",
  "license": "ISC",
  "dependencies": {
    "express": "^4.18.2"
  }
}
```

Important Properties of `package.json`

1️⃣ `name`

- Name of the project
- Must be lowercase

```
"name": "mern-project"
```

2️⃣ `version`

- Current version of the project
- Follows semantic versioning (major.minor.patch)

```
"version": "1.0.0"
```

3️⃣ `description`

- Short explanation of what the project does

```
"description": "My first MERN application"
```

4️⃣ `main`

- Entry point of the application
- Node.js starts execution from this file

```
"main": "index.js"
```

5️⃣ `scripts`

Scripts are **custom commands** to run the application.

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

Run scripts using:

```
npm start
npm run dev
```

6️⃣ `dependencies`

Dependencies are the **packages required to run the application**.

```
"dependencies": {
  "express": "^4.18.2"
}
```

- Installed using `npm install <package>`
- Required in production

7️⃣ `devDependencies`

Packages required **only during development**.

```
"devDependencies": {
  "nodemon": "^3.0.0"
}
```

- Installed using:

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

What happens when we run `npm i`?

- npm reads `package.json`

- Installs all `dependencies` and `devDependencies`
- Creates the `node_modules` folder

Interview One-Liners (Very Important)

- `package.json` manages project dependencies
- It is mandatory for real-world Node.js projects
- It helps in collaboration and deployment
- `npm i` works because of `package.json`

Simple Analogy

`package.json` = Shopping list
`npm install` = Buying items
`node_modules` = Store room

8.3 Nodemon – Why We Need It (Beginner Explanation)

What is Nodemon?

Nodemon is a **development tool** used in Node.js applications. It automatically **restarts the server whenever file changes are detected**.

In simple words:

> Nodemon saves us from stopping and restarting the server again and again.

Problem Without Nodemon

When we use normal Node.js, we start the server like this:

```
node index.js
```

If we make **any code change**, we must: 1. Stop the server (Ctrl + C) 2. Restart it again

This becomes **very frustrating** during development.

Solution With Nodemon

Nodemon automatically watches files and restarts the server.

Start server using:

```
nodemon index.js
```

Now: - Change code ✅- Server restarts automatically ✅- No manual restart ❌

Installing Nodemon

Install Nodemon as a **dev dependency**:

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

Why dev dependency? - Needed only during development - Not required in production

---

Using Nodemon with Scripts (Best Practice)

Add this in `package.json`:

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

Run using:

```
npm run dev
```

Normal Node vs Nodemon (Difference)

Feature	node	nodemon
Auto restart	❌No	✅Yes
Development friendly	❌No	✅Yes
Used in production	✅Yes	❌No

Interview One-Liners (Very Important)

- Nodemon improves developer productivity
- It automatically restarts the Node server
- Used only in development phase

- Installed under devDependencies

Simple Analogy

🆕 **Without Nodemon** → Switch OFF & ON fan every time
9 **With Nodemon** → Automatic sensor fan

---

9. Postman

Postman is an **API testing tool** used to test GET, POST, PUT, PATCH, DELETE requests without frontend.

Why Postman?

- Easy API testing
- Debug backend
- View responses clearly

---

10. CRUD Operations

CRUD represents basic database operations.

Operation	HTTP Method
Create	POST
Read	GET
Update	PUT / PATCH
Delete	DELETE

Example CRUD Flow

- Create user → POST
- Fetch user → GET
- Update user → PUT/PATCH
- Delete user → DELETE

Final Summary

- MERN stack uses **JavaScript everywhere**
- MongoDB stores data
- Express + Node handle backend logic
- React builds frontend UI
- APIs connect frontend and backend
- CRUD is the core of any application

🔝Next Steps for You

1. Practice Express APIs
2. Learn MongoDB queries
3. Build small React apps
4. Connect frontend & backend

Happy Learning 🚶