# MERN STACK

## BEGINNER ⇨ ADVANCED GUIDE

# INDEX

M E N R

# MERN : Mongo DB, Express JS, Node JS, React JS

- **Frontend** : React JS
- **Backend** : Node JS, Express JS
- **Database** : Mongo DB
  SQL (optional) or as a secondary database

## # Introduction

The MERN Stack is a widely adopted full-stack development framework that simplifies the creation of modern web applications. It is a full-stack Javascript stack, meaning we can use Javascript both on client-side and server-side, leading to consistency and ease of development.

### Why learn MERN Stack?

- **Single Language (Javascript)** : No need to learn multiple languages for frontend & backend.
- **Fast & Scalable** : Node.js and Mongo DB handle large data and high traffic efficiently.
- **Popular & In-demand** : Many startups and companies use MERN because its open source, and backed by strong communities.

# Four Components of MERN

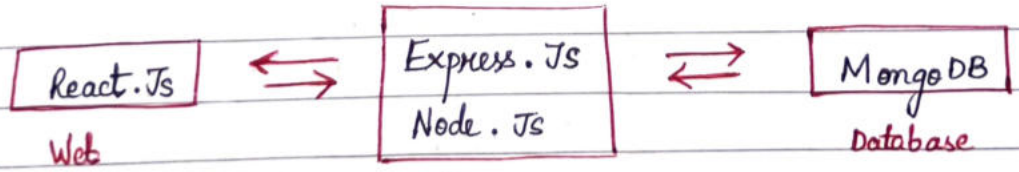
a) Mongo DB (Database)
- A NoSQL database that stores data in JSON-like documents instead of tables like traditional databases.

b) Express.js (Backend Framework)
- A lightweight web framework build on top of Node.js. It simplifies writing backend logic : like handling API requests, managing routes (URLs), connecting frontend to database.

c) React.js (Frontend Library)
- A Javascript library used to build user interfaces, especially single-page applications. It's component-based, fast, and helps developers create reusable and dynamic UI elements.

d) Node.js (Backend Runtime Environment)
- This is also a JavaScript runtime environment that lets you run Javascript on the server side. It's fast, scalable, and widely used for building backend services and APIs.

# How MERN Stack works ?




WORKING OF MERN STACK

↳ Roadmap to become a MERN Stack Developer :

- Learning basics of HTML, CSS and JavaScript
- Learning React which is a frontend library for building UI.
- Learning Node JS
- Learning Express JS, a framework build upon Node JS
- Learning MongoDB, a NoSQL database to store or retrieve data from database

↳ Typical Folder Structure

```
mern-project/
├── client/          → React front-end
├── server/          → Node + Express backend
│   ├── routes/
│   ├── models/
│   └── controllers/
├── package.json     → Project dependencies
└── .env             → Environment variables (eg, DB, URL)
```

* Here's an Easy Analogy to understand MERN :

Think of MERN as a pizza shop :

- React → The waiter serving a pizza (user interface)
- Express + Node → The kitchen staff preparing pizza. (server logic)
- MongoDB → The fridge where all ingredients are stored (data).

# Advantages of MERN

1. **Reusable Components**
   On the front-end, React components facilitate reuse and modularity

2. **High Performance**
   - React's virtual DOM optimizes UI updates
   - Node's non-blocking I/O allows high performance on server side.

3. **Active Community & Ecosystem**
   All 4 components are open-source and have large community support, many packages, tutorials, tools.

4. Flexibility & Modularity
   We can add or replace components, adjust architecture as business needs evolve.

# INTRO to JavaScript

JavaScript is a high-level programming language, mainly used for web development. It does interactive web applications, supporting both client-side and server-side development and integrating seamlessly with HTML, CSS and a rich standard library.

JavaScript can interact with DOM → Document Object Model to modify the structure, styling, content of the webpages. It is often used with Node.js, a ~~runtime~~ runtime environment which allows JavaScript codes to be executed at the outside of the browser.

→ single-threaded language that executes one task at a time
→ an interpreted language that executes code line by line.
→ data-type of the variable is decided at run-time, that's why it is called dynamically-typed.

↳ Key features of JS:

(i) Client-side scripting - JS runs on the user's browser, so has a faster response time without communicating the server.

(ii) Event-driven - It can respond to user actions {clicks, keystocks} in real time.

(iii) Asynchronous - JS can handle tasks like fetching data from servers.

↳ Client-side nature of JS:

- Involves controlling the browser and its DOM.
- Handles user events like clicks and form inputs
- Common library includes Angular JS, React JS and Vue JS.

↳ Server-side nature of JS:

- Involves interacting with databases, manipulating files and generating responses.
- Node JS and other frameworks like Express JS are widly used for server-side JS enabling full-stack developer.

↳ Applications of JS:

JavaScript is used in a wide range of applications, from enhancing websites to building complex applications. like -

(i) Web development: JavaScript adds interactivity and dynamic behaviour to static website, with popular frameworks like Angular JS, React JS etc enhancing development.

(ii) Server Application: Node.JS brings JavaScript to the server side, enabling powerful server application and full stack development

(iii) Game Development: JavaScript, combined with HTML5 and libraries like Ease JS enables creation of interactive games for the web.

@ SYNTAX ERROR
- Abhishek Rathor

# External JavaScript → This code can be placed at a different file, means external (.js) file linked with an HTML document.

Example

- html file

```
<HTML>
  <head>
    <script src = "first.js"></script>
  </head>
    <body>
        // HTML CODES
    </body>
</HTML>
```

- js file

```
Function myFunction () {
        alert ("Hello!");
      }
```

# Variables

In the JavaScript, variables are used for storing data values. It is like container in JavaScript which hold different types of information.

To declare a variable in JavaScript we use 'var', 'let' or 'const' keyword, followed by variable name.

(i) "Var" → The "var" keyboard is the main way for declaring variables in javascript. It is accessible within the entire

function where it is declared.

Example

```
// declaring and assigning : variables using var
var age = 20
console.log(age);
```

⇒ A variable can be assigned a data value using the assignment operator equal sign (=).

```
// Resigning the value of variable
age = 22
console.log(age);
```

⇒ A variable can be reassigned in JavaScript by assigning a new value to the variable without using the "var" keyword.

(ii) "let" → The "let" keyword was introduced to address the issues within "var" keyword & is accessible within the nearest block where it is declared.

Example

```
// declaring and assigning a variable using let
let name = 'Abhishek';   {// Output: Abhishek}
console.log(name)

// Block scope with let
if (true) {
    let x = 20;          {Output: 20}
    console.log(x);
}
```

(iii) "const" → The "const" stands for "constant". It has also block scope like "let", but in "const" declaration of variable cannot be reassigned after it has been assigned a value.

Example

```
// declaring a "const" variables
const PI = 3.14;
console.log (PI);
```

{Output = 3.14}

# Data Types

Data Types are used for holding and manipulating the values in the Javascript. The common data types are numbers, strings, boolean, undefined, null, array, object, function and symbols. Each data type has specific behaviour and properties.

(i) Number → It is used for representing the numeric value in the JavaScript.

Example

```
let age = 10;
let temperature = 99-1;
```

→ we can perform arithmetic operation also on number values.

(ii) String → It is used for representing the sequence of character values in JavaScript. It is enclosed within single quotes or double quotes

Example

```
let name = "Abhishek";
```

(iii) Boolean → It is used for representing the logical value that can either true or false.

Example

```
let hasPermission = False;
```

⇒ The variable "hasPermission" is assigned the Boolean value 'false', show that the user does not have permission.

(iv) Undefined → It is used for representing the variable that has been declared but has not been assigned a value.

Example

```
let x;
```

(v) Null → It is used for representing the intentional absence of any object value.

Example

```
let data = null;
```

(vi) Object → It is complex data type which represents the collection of properties and their value. It is used for organizing and stored related information for functioning together.

Example

```
let person = {
    name : 'Abhishek'
    age : 20
    isStudent : true
};
```

(vii) Array → used for representing an ordered list of values. It can store multiple values of any data type, such as strings, objects, numbers.

(viii) Function → treated as first-class objects. We can store functions in variables, pass arguments to other functions & can return functions.

So technically, the data type of a function is an object.

Data Types : Primitive → Number, String, Boolean, Null, Undefined
Non-primitive → Objects, Arrays, Function

# Operators

It is a keyword which is used for performing an operation on values, such as calculation, comparisons and more.

i) Arithmetic Operators

Addition (+) → used for adding two values
Subtraction (−) → used for subtracting one value from another
Multiplication (*) → used for multiplying two values
Division (/) → used for division operation
Modulus (%) → It returns the remainder
Exponentation → raises a number to the power of another.

ii) Assignment Operators

Assignment (=) → assigns a value to a variable
Addition assignment (+=) → adds value to variable and assign result
Subtract assignment (-=) → subtracts and assigns the result
Multiply assignment (*=) → multiplies and assigns the result
Division assignment (/=) → divide the variable by a value and assigns the result.

iii) Comparison Operators

equal to (==) → checks if two values are equal
not equal to (!=) → checks if two values are not equal
greater than (>) → checks if one value is greater than another
less than (<) → checks if one value is less than another
>= → checks if one value is greater than or equal to another
<= → checks if one value is less than or equal to another.

iv) Logical Operators

Logical AND (&&) → returns true if both operands are true.
Logical OR (||) → returns true if either operand is true.
Logical NOT (!) → denies the truth value of an expression

v) Conditional Operator → provides a short syntax for conditionally assigning a value based on condition.

Syntax:

condition ? ExpressionIfTrue : ExpressionIfFalse

# Control Statements

Control Statements in JavaScript are used to control the flow of execution of the program – i.e. to decide which part of the code should run, how many times and under what condition.

They can be broadly divided into :

1. Conditional Statements
2. Looping (Iterative) Statements
3. Jump (Branching) Statements

1. Conditional Statements → These statements are used to make decisions based on conditions (true / false)

→ If Statement → executes a block of code only if the condition is true.

Syntax

```
if (condition) {
        // code runs if condition is true
}
```

→ If - else Statement → executes one block if the condition is true, otherwise moves to the next block.

Syntax

```
if (condition) {
        // code if true
} else {
        // code if false
}
```

→ Else-if Statement → used when there are multiple conditions to test.

Syntax

```
if (condition1) {
        // code block 1
} else if (condition2) {
        // code block 2
} else {
        // default code block
}
```

→ Switch Statement → used when there are many possible values for a variable. It's an alternative to multiple if-else statements.

Syntax

```
switch (expression) {
    case value1:
        // code
        break;
    case value2:
        // code
        break;
    default:
        // default code
}
```

2. Looping (Iterative) Statements → used to execute a block of code repeatedly

→ for Loop → runs a block of code a specific number of times.

Syntax

```
for (initialization ; condition ; increment / decrement) {
    // code block
}
```

→ while Loop → executes a block while the condition is true.

Syntax

```
while (condition) {
    // code block
}
```

→ do-while Loop → similar to while, but runs at least once, even if the condition is false.

Syntax

```
do {
    // code block
} while (condition);
```

3. Jump Statements → used to change the normal sequence of execution.

→ break Statement → used to exit from a loop or switch immediately.

Example

```
for (let i=1; i<=10; i++) {
    if (i==5) break;
    console.log(i);
}                          // Output : 1 2 3 4.
```

→ continue Statement → skips the current iteration and continues with the next one.

Example

```
for (let i=1; i<=5; i++) {
    if (i==3) continue;
    console.log(i);
}                          // Output: 1 2 4 5
```

→ return Statement → used to exit from a function and optionally return a value.

Example

```
function add (a,b) {
    return a+b;
}
console.log(add(2,3));     //5
```

# Functions

JavaScript functions are effective tools for structuring and organising the code. They are basically a blocks of reusable codes designed to perform a specific task.

Functions helps make code modular, clean, and easy to maintain.

- Function Declaration (Named Function)

A block of reusable code that is simply called by its name. They support flexibility, readability and code organisation.

Functions can be defined by using the 'function' keyboard and it is followed by the name of function arguments and the body of functions.

Syntax

```
function functionName (parameters) {
    // code to be executed
}
```

Example

```
function greet (name) {
    console.log ("Hello" + name + "!");
}
greet ("Abhishek");   // Output : Hello Abhishek!
```

Key Points

- declared using the function keyword
- can be called before it is defined (because of hoisting)
- stored in memory at the start of the program execution.

- Function Expression

A function expression in JavaScript is a way to define a function and assign it to a variable.
It can be anonymous or named, and is not hoisted — meaning it cannot be called before it is defined.

Syntax

```
const functionName =
    function (parameters) {
        // code
    };
```

Example

```
const add = function (a, b) {
    return a+b;
};
console.log (add (5, 3));    // Output : 8
```

Key Points

- Function is stored in a variable.
- Not hoisted (unlike function declarations)
- Commonly used in callbacks or when passing functions as arguments.

- Anonymous Function

An anonymous function is a function without a name. It is often used when a function is needed temporarily, for example as a callback or inside another function.

Syntax

```
function () {
    // code block
}
```

Example

```
setTimeout ( function () {
    console.log ("Hello after 2 seconds");
}, 2000);
```

Key Points

- cannot be called directly (must be assigned or passed)
- commonly used in function expressions and callbacks.
- introduced mainly for short, one-time use functions.

- **Arrow Functions (ES6 Feature)**

Arrow Functions provide a shorter syntax for writing functions

Syntax

```
const functionName = (parameters) => {
    // code
};
```

Example

```
// Single line
const square = x => x * x;
// Multiple lines
console greet = (name) => {
    console.log("Hello, " + name);
};
```

Key Points

- don't need the function keyword
- automatically return the value (if single-line)
- do not have their own this, arguments, or super

- **Callback Functions**

A callback is a function passed as an argument to another function and executed later.

Syntax

```
function mainFunction (callback) {
    // some code
    callback (); // execute the callback
}
```

@ SYNTAX ERROR
— Abhishek Rathor

Example

```
function sayHello () {
    console.log ("Hello!");
}
function greetUser (callback) {
    console.log ("Welcome user!");
    callback (); // calling the callback
}
greetUser (sayHello);
```

Key Points

- common in asynchronous programming (e.g. setTimeout, API calls)
- helps execute code after something else has finished.

## 13 Example of Callback with setTimeout

```
setTimeout (() => {
    console.log ("This runs after 2 seconds."); }, 2000);
```

Here, the arrow function acts as a callback for setTimeout.

B. Arrays → An array is a special type of object used to store multiple values in a single variable, accessed by index.

```
const fruits = ["Apple", "Banana", "Mango"];
```
↳ Creating an Array

```
console.log(fruits[0]);
```
→ Accessing array elements

```
fruits[1] = "Orange";
```
→ updating array elements

```
fruits.push("Grapes");
fruits.unshift("Kiwi");
```
→ adding elements

```
fruits.pop();
fruits.shift();
```
→ removing elements

LOOPING THROUGH AN ARRAY

```
for (let i = 0; i < fruits.length; i++) {
    console.log(fruits[i]);
}
```
→ for Loop

```
for (let fruits of fruits) {
    console.log(fruit);
}
```
→ for ... of Loop

```
fruits.forEach((fruit) =>
console.log(fruit
```
→ forEach() Method

c. Array Methods → These methods made working with arrays easier and cleaner.

1. forEach() → executes a function for each element in an array. (doesn't return anything)

Example

```
const numbers = [1, 2, 3];
numbers.forEach(num => console.log(num * 2));
// Output : 2, 4, 6.
```

2. map() → creates a new array by applying a function to each element.

Example

```
const numbers = [1, 2, 3];
const squared = numbers.map(num => num * num);
console.log(squared); // [1, 4, 9]
```

→ used when we want to transform array data

3. filter() → creates a new array with elements that pass a condition.

Example

```
const numbers = [1, 2, 3, 4, 5];
const even = numbers.filter(num => num % 2 == 0);
console.log(even); // [2, 4]
```

→ used to select certain elements from an array.

4. reduce() → reduces the array to a single value by applying a function repeatedly.

Example

```
const numbers = [1, 2, 3, 4];
const sum = numbers.reduce((acc, num) => acc + num, 0);
console.log(sum);  // 10
```

→ used for cumulative calculations (sum, product, etc.)

# ES6+ Features (Modern JavaScript)

Template literals : Def:

Template literals are a new way to work with strings. They use backticks (`) instead of quotes and allow embedding variables or expressions easily.

Syntax

```
string text ${expression}
```

Example

```
let name = "Abhishek";
let age = 20;
console.log("My name is ${name} and I am
                ${age} years old.");
```

Advantages:

- easier to combine text + variables
- can span multiple lines.

## Destructing : Def:

Destructing allows to unpack values from arrays or objects and assign them to variables easily.

### * Array Destructing

```
let fruits = ["apple", "banana", "mango"];
let [first, second, third] = fruits;
console.log(first);
console.log(second);
```

### * Object Destructing

```
let user = { name : "Abhishek", age : 20, city : "Kolkata"};
let { name, age } = user;
console.log(name);    // Abhishek
console.log(age);     // 20
```

Remaining variables →

```
let { city : location } = user;
console.log(location);    // Kolkata
```

```
let { country = "India" } = user;
```
→ Default values

## Spread & Rest Operators : Both use the same syntax (...) but behave differently depending on where they're used.

* Spread Operator → used to expand elements of an array or object

Example 1 – Arrays

```
let arr1 = [1, 2, 3];
let arr2 = [... arr1, 4, 5];
console.log(arr2);      // [1, 2, 3, 4, 5]
```

Example 2 – Objects

```
let person = {name: "Abhishek", age: 20};
let details = {... person, city: "Kolkata"}
console.log(details);
```

* Rest Operator → used to collect remaining values into a single variable.

Example – Functions:

```
function sum(... numbers) {
    return numbers.reduce((total, n) =>
        total + n);
}
console.log(sum(1, 2, 3, 4));      // 10
```

Example – Objects / Arrays:

```
let {name, ... rest} = {name: "Abhishek", age: 20,
                        city: "Kolkata"};
console.log(rest);
```

## Modules (import/export): Def:

Modules allow to split your code into multiple files and reuse functions, variables, or classes by exporting and importing them.

* Exporting

From file math.js:

```
export const add = (a,b) => a+b;
export const sub = (a,b) => a-b;
```

or export all at once:

```
~~export~~ const mul = (a, b) => a * b;
export default mul;
```

* Importing

In another file app.js:

```
import { add, sub } from './math.js';
import mul from './math.js';
console.log(add(5,3));    //8
console.log(mul(5,3));    //15
```

→ cleaner, modular code
→ easy to maintain & reuse functions.

# Asynchronous JavaScript

Def: JavaScript normally runs synchronously, meaning one line executes after another.
But some tasks (like fetching data from a server) take time.
To prevent the page from "freezing", JavaScript uses asynchronous programming — it lets the rest of the code run while waiting for that slow task to finish.

Examples of sync tasks:

- Fetching data from an API
- Reading files (in Node.js)
- Timer functions (setTimeout)

Promises : Def: A Promise is an object that represents a task that may complete in the future — either successfully or with an error.

* States of a Promise
  1. Pending → Waiting for the task to complete
  2. Resolved (Fulfilled) → Task succeeded
  3. Rejected → Task failed

* Syntax

```
let promise = new Promise (( resolve, reject ) => {
    let success = true;
    if (success) {
```

```
        resolve ("Task completed!");
    } else {
        reject ("Something went wrong.");
    }
});
```

* Using .then() and .catch()

```
promise.then(result => console.log(result))   // if resolved
promise.catch(error => console.log(error))    // if rejected
```

Example (Real Use Case):

```
function getData() {
    return new Promise(resolve => {
        setTimeout(() => resolve("Data received!"), 2000);
    });
}
getData().then(msg => console.log(msg)); // waits 2 sec →
                                         "Data received!"
```

Async / Await : Def: async and await are keywords introduced in ES8. They make asynchronous code look and behave like synchronous code, making it easier to read and write.

@ SYNTAX ERROR
- Abhishek Rathor

* Syntax

```
async function fetchData () {
    let promise = new Promise (resolve => {
        setTimeout (() => resolve ("Data loaded!"), 2000);
    });
    let result = await promise ; // waits until promise resolves
    console.log (result);
}
    fetchData ();
```

→ Output after 2 seconds → Data loaded!

Key Points

- always use await inside an async function.
- await pauses the function until the promise resolves
- errors can be handled with try ... catch.

Example

```
async function getUser () {
    try {
        let res = await
    fetch ("https://api.example.com/user");
        let data = await res.json ();
        console.log (data);
    } catch (err) {
        console.log ("Error:", err);
    }
}
```

## Fetch API (To call Backend) : Def: fetch() is a built-in JavaScript function to make HTTP requests to servers or APIs. It returns a Promise, which resolves to a Response object.

* GET Request Example

```
fetch("https://jsonplaceholder.typicode.com/users")
  .then(res => res.join())
// convert response to JSON
  .then(data => console.log(data))
  .catch(err => console.log("Error:", err));
```

* POST Request Example (Sending data to backend)

```
fetch("https://jsonplaceholder.typicode.com/posts", {
  method: "POST",
  headers: {
    "Content type": "application/json" },
  body: JSON.stringify({
      title: "Hello MERN",
      body: "Learning async JS!",
      userId: 1
  })
})
  .then(res => res.json())
  .then(data => console.log(data))
  .catch(err => console.log(err));
```

## * Using Fetch with Async / Await

```
async function createPost () {
    let res = await
fetch ("https://jsonplaceholder.typicode.com/posts", {
        method : "POST",
        headers : { "Content-Type" : "application/json" }
        body : JSON.stringify ({ title : "Async Await", body : "HERN
                                 Rocks!" })
    });
    let data = await res.json ();
    console.log (data);
}
createPost ();
```

# DOM & Events (Basic Understanding)

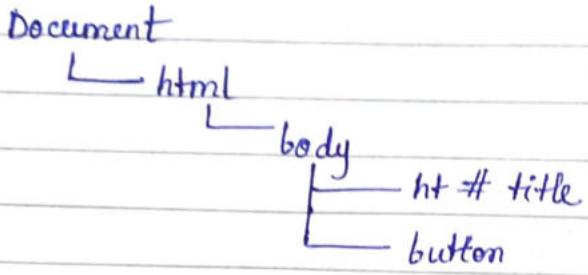
Def: DOM stands for Document Object Model.
It's a tree-like structure that represents your entire HTML webpage.
JavaScript can use the DOM to view, change, add, or delete elements dynamically.

Structure Example

```
<html>
  <body>
    <h1 id = "title"> Hello user </h1>
```

```
    <button> Click Me </button>
  </body>
</html>
```

The DOM Tree looks like :




Every HTML element (like <h1>, <button>) becomes a DOM node that JS can access.

## Accessing / Selecting Elements :

To work with the DOM, we must select the element we want to modify.

* document.getElementById()
  Selects an element by its id

```
let title =
  document.getElementById("title");
  console.log(title.innerText);
  // Output : Hello User
```

* document.querySelector()
Selects the first element that matches a CSS selector

```
let heading =
document.querySelector("#title."); // using ID
let button =
    document.querySelector("button"); // using tag name
```

* document.querySelectorAll()
Selects all matching elements (returns a NodeList)

```
let buttons =
    document.querySelectorAll("button");
    console.log(buttons.length); // number of buttons on page.
```

## Changing Context and Style

Changing text :

```
    title.innerText = "Welcome to MERN!";
```

Changing HTML :

```
    title.inner = "<i> Hello from JavaScript! </i>";
```

Changing styles :

```
    title.style.color = "blue";
    title.style.fontSize = "30px";
```

## Adding or removing classes :

```
title.classList.add("highlight");
title.classList.remove("old-style");
```

## Event Listeners : Def: An event is an action the user performs - like clicking a button, typing or scrolling. The event listener allows JS to "listen" for these actions and respond.

* **Basic Syntax**

```
element.addEventListener("event", function);
```

* **Example - Click Event**

```
let button =
document.querySelector("button");
button.addEventListener("click", function () {
    alert("Button clicked!");
});
```

* **Example - Mouse Over Event**

```
title.addEventListener("mouseover", function () {
    title.style.color = "purple";
});
```

## Commonly Used Events

| Event | Description |
|---|---|
| click | triggered when an element is clicked |
| mouseout | when mouse leaves an element |
| mouseover | when mouse pointer moves over an element |
| keydown | when a keyboad key is pressed |
| keyup | when a key is released |
| input | when text input changes |
| submit | when a form is submitted |
| load | when the page or resource is fully loaded. |

## In Short

- DOM lets JavaScript connect to HTML.
- Events let the user interact with your webpage.
- Together, they make websites dynamic and responsive — a must-have skill for MERN developers.

# Working of JS

JS is dynamically typed, cross-platform threaded scripting and programming language, it is an asynchronous and concurrent language that offers a lot of flexibity.

↳ Execution Context

| Variable Env. | Thread of Execution |
|---|---|
| Memory | Code |
| key : value i.e;<br>a : 10<br>fn : {π}<br>b : 1.2<br>x : true<br>y : "Prince" | o<br>o<br>o<br>o<br>o<br>o |

In thread of Execution that is executed one line at a time.

In Memory component which is an environment in the CPU memory where variables are present in the form of key, value pairs, and function is present i.e. whole function is present.

- Single-Threaded means JavaScript executes one command at a time.
- Synchronous : It only executes one command at a

time in a specific order i.e., we can go through the next line of code once the current line has been completely executed.

* Note: we can make JS to work or behave asynchronously by using various block of code like: async/await, promise (covered earlier) etc.

→ What happens when we run JS code:

"Everything in Javascript happens inside the execution context"

Let's say a program i.e;

```
var a = 2;
function square (num) {        ← Parameter
    var x = num * num;
    return ans;
}
var square 2 = square (d);     ← Argument
var square 4 = square (4);
```

↳ When we execute the code or any code, an execution context is created i.e., also known as memory creation and code component creation phase;

# MONGO DB

MongoDB is a cross-platform, document oriented database that provides high performance, high-availability and easy scalability. MongoDB works on concept of collection and document.

**Database:** Database is physical container for collections. Each database gets its own set of files on the file system. A single MongoDB server typically has multiple interface, database.

**Collection:** Collection is a group of MongoDB document. It is an equivalent to RDMS table. A collection exists within a single database collection do not enforce a schema document within a collection can have different fields.

**Document:** A document is a set of key-value pair. Document have dynamic schema. It means that document in the same collection do not need to have the same set of fields and structure and common fields in a collection document.

**Difference b/w SQL and MongoDB:**

| SQL | Mongo DB |
|---|---|
| • SQL database are relational databases i.e., in the form of table | • No SQL database are non-relational DB that is they are in the form of JSON document. |

@ SYNTAX ERROR
— Abhishek Rathor

| | |
|---|---|
| • They use structured tables to store data in rows & columns. | • They provide flexibility in data storage, allowing varied data types & structures. |
| • Suitable for applications with well-defined schemas and fixed data-structured | • Ideal for applications with dynamic or evolving data models. |
| • E-commerce platform, HR Management etc. | • CMS, Social Media Platforms. |
| • Examples: MySQL, PostgreSQL, Oracle | • Example: MongoDB, Cassandra |

Where to use MongoDB :

① MongoDB → Big Data
Content Management & Delivery
Mobile and Social Infrastructure
User Data Management
Data hub

Start MongoDB
Sudo service mongodb start

Stop MongoDB
Sudo service mongodb step

Restart MongoDB
Sudo service mongodb restart

# JSON vs BSON

→ JSON : JSON (JavaScript Object Notation) is a lightweight data interchange format that is :

- Human-readable and text-based
- Commonly used for data transfer between client and server (like in REST APIs).
- Based on key-value pairs and arrays.

Example :

```
{
    "name" : "Abhishek",
    "age" : 21,
    "skills" : ["JavaScript", "React"]
}
```

→ BSON : BSON (Binary JSON) is a binary-encoded serialization format used by MongoDB to store documents.

- It stands for Binary JSON
- It extends JSON with additional data types (like Date, ObjectId, Binary, etc.)
- It is optimized for speed and space when reading/writing data in databases.

Example (conceptually) :

```
{
    name : "Abhishek",
    age : 21,
    skills : ["JavaScript", "React"],
```

_id :
ObjectId ("507f191e810e19729de860ea")
}

## Key Differences Between JSON and BSON

| Feature | JSON | BSON |
|---|---|---|
| Full Form | JavaScript Object Notation | Binary JSON |
| Format Type | Text-based | Binary encoded |
| Readability | Human-readable | Machine-readable |
| Usage | Data transfer over network | Data storage & internal representa |
| Data Types Supported | Limited (string, number, boolean, null) | Rich (adds Date, ObjectId, etc.) |
| Performance | Slower for parsing large data | Faster due to binary encoding |
| Storage Size | Larger (plain text) | More compact and efficient |
| Indexing Support | Not applicable | Supports indexing in MongoDB |
| Traversal / Parsing | Requires string parsing | Direct memory access (faster for databases) |

## Why MongoDB uses BSON?

MongoDB uses BSON because :

1. Efficient storage → Binary data is smaller and faster to read/write.
2. Rich data types → can store Date, ObjectId, and more complex structures.
3. Fast encoding/decoding → BSON allows quick traversal for queries.
4. Extensibility → MongoDB can define new BSON types without breaking compatibility.

## Advantages of BSON over JSON

✓ Compact representation → Saves disk space
✓ Faster read/write → Especially for large documents
✓ Supports more data types → Especially needed in databases
✓ Easy traversal → useful for indexing and query optimization.

In short

JSON = Great for data exchange between systems
BSON = Great for internal storage in MongoDB due to speed and efficiency.

## # MongoDB – Data Types

- String → commonly used to store the data. String MongoDB must be UTF-8 valid.
- Integer → stores a numerical value.
- Boolean → stores boolean value
- Double → stores floating point values
- Array → store arrays or list or multiple values in one key
- Object → datatype used for embedded document.
- Null → used to store NULL value.
- Date → used to store the current data or time in UNIX time format. You can specify your own time by creating object of Date and passing data, month, year onto it.

- Object ID → This datatype is used to store document ID.
- Code → used to store javascript code into the document.

# MongoDB – Data Modelling : Data in MongoDB has flexible schema documents in the same collection. They do not need to have the same set of fields or structure common fields in the collections document may hold different types of data

Data Model Design :
MongoDB provides 2 types of data models.

① Embedded Data Model
In this model, we can have all the related data in a single document. It is also known as de-normalised data model.

Example –

```
{
    _id : ;
    [ mp_ID : "10025AE336"
      Personal_details : {
                First_Name : "Abhishek",
                Last_Name : "Rathor",
                Date_of_Birth : "1995-09-26"
      },
       Contact : {
                email : "syntaxerror@gmail.com",
                phone :  93037 04945  }
```

## ② Normalized Data Model

In this model, you can refer the sub-documents in the original document, using references.

Example —

```
Employee :
        {
            _id : < Object Id 01 >
            Emp_ID : "10025AE336"
        }
Personal_details :
        {
            _id : < Object Id 102 >
            empDOCID : " Object Id 101 "
            First_Name : " Abhishek "
            Last_Name : " Rathor ",
            Date_of_Birth : " 1995-09-26 "
        }
Contact :
        {
            _id : < Object Id 103 >
            empDOC ID : " Object Id 101 ".
            email : " syntaxerror @ gmail.com
            phone : 93037 04945 }
```

## ③ Hybrid Data Model (Embedded + Normalised)

In real-world apps (like in MERN projects), we can combine both approaches.

Example

```
{
    _id : 1,
    name : "Abhishek",
    city : "Delhi",
    orders : [ { order_id : 101, product : "Laptop" },
               { order_id : 102, product : "Phone" } ],
    wallet_id : ObjectId ("6753dca3...")
}
```

→ Embedded for frequent data (orders),
Referenced for large/complex data (wallet)

## Schema Design Rules

1. Model data around application needs, not tables.
2. Embed data that is read together often
3. Reference data that changes independently or grows large.
4. Avoid deep nesting (try not to exceed 2-3 levels)
5. Use indexes on frequently queried fields.
6. Limit document size to under 16 MB.

# Documents & Collections in MongoDB

A document in MongoDB is the basic unit of data — similar to a row in SQL.

- stored in BSON
- document made of field-value pairs
- each document has a unique _id field.

Example of a MongoDB document :

```
{
  "_id" :
  ObjectId("507f191e810c19729de860ea"),
  "name" : "Abhishek",
  "age" : 21,
  "email" : "syntaxerror@gmail.com",
  "skills" : ["JavaScript", "React", "Node.js"],
  "address" : {
        "city" : "Delhi",
        "country" : "India"
  }
}
```

Key Points :

- Flexible scheme : Documents in the same collection can have different fields.
- Stored internally as BSON, but represented as JSON when viewed
- _id field is automatically created if not specified.

TYPES:

MongoDB - Insert Document : An insert document in MongoDB refers to the actual data you want to add to a collection. It represents one or more records that you want to store in the database.

MongoDB stores data in the form of documents (BSON format) inside collections.

Syntax

```
db.collection.insertOne({ <field1> : <value1>, <field2> : <value2>, ... })
```

OR

```
db.collection.insertMany([{ <doc1> }, { <doc2> }, ...])
```

Example 1: Insert One Document

```
db.students.insertOne({
    _id : 1,
    name : "Abhishek",
    age : 21,
    city : "Delhi"
    course : "MERN Stack"
})
```

Explanation :

- db.students → collection name.
- insertOne() → method to insert a single document
- {...} → the insert document (actual data being stored)

Example 2 : Insert Many Documents

```
db.students.insertMany([
  { name : "Abhishek", age : 21, city : "Delhi" },
  { name : "Syntax", age : 22, city : "Mumbai" }
])
```

→ Insert multiple documents into the students collection at once.

Key Points

- MongoDB automatically adds an _id field if you don't provide one.
- Each document can have different fields (schema flexibility)
- Supports nested (embedded) and array data

Insert Methods Overview

| Method | Description | Example |
|---|---|---|
| insertOne() | insert one document | db.users.insertOne({name : "Abhishek"}) |
| insertMany() | insert multiple documents | db.users.insertMany([{...}, {...}]) |

# MongoDB : Query Document

A query document is a filter or condition used to search, update, or delete data in a collection.
Instead of storing data, it helps MongoDB find matching documents based on criteria you specify.

## Syntax

db.collection.find({ <field> : <value> })

We can also use comparison operators :

db.collection.find({ <field> :
{ $operator : <value> } })

## Example 1 : Simple Query

db.students.find({ city : "Delhi" })
→ Finds all students whose city is Delhi

## Example 2 : Query with Comparison Operator

db.students.find({ age : { $gt : 18 } })
→ Finds all students whose age is greater than 18.

## Example 3 : Using Logical Operator

```
db.student.find({
    $and: [{ city: "Delhi" }, { age:
{   $gte: 18 }]
})
```

→ Finds students who are from Delhi and 18 or older.

## Example 4 : Projection (selecting specific fields)

```
db.students.find({
    { city: "Delhi" },
    { name: 1, age: 1, _id: 0 }
})
```

## Query Operator Overview

| Operator | Meaning | Example |
|---|---|---|
| $eq | Equal to | {age:{ $eq: 20 } } |
| $ne | Not equal | { city: { $ne: "Delhi" } } |
| $gt | greater than | { marks: { $gt: 80 } } |
| $lt | less than | { age: { $lt: 25 } } |
| $in | value in array | { city: { $in: ["Delhi", "Mumbai"] } } |

## Common Query Methods

| Method | Description |
|---|---|
| find () | Retrieve documents |
| findOne () | Retrieve one document |
| UpdateOne () | Update first matching document |
| deleteOne. () | Delete first matching document |

Collection is a group of documents — similar to a table in SQL.

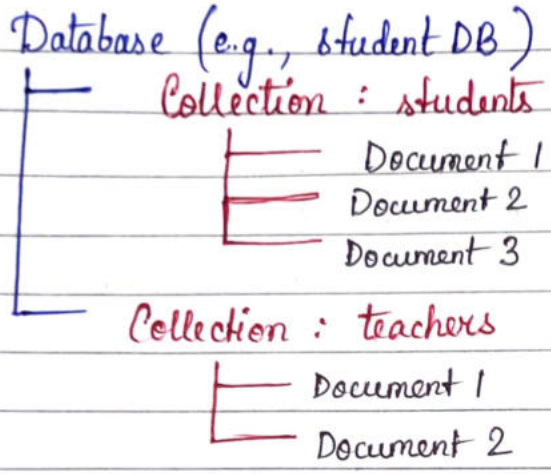
- A collection stores related documents.
- Collections do not enforce a fixed schema.
- They are created automatically when you insert the first document.

Example : If we have a collection named users, it can contain :

```
{
 "name" :  "Abhishek",
 "age" : 21
}
{
 "name" : "Syntax",
 "age" : 22,
 "skills" : ["Python", "Django"]
```

Both belong to the same collection (users) even though their structures differ.

## Relationship Between Documents, Collections, and Database




## Comparison with SQL

| Concept | SQL | MongoDB |
|---|---|---|
| Database | Database | Database |
| Table | Table | Collection |
| Row | Row | Document |
| Column | Column | Field |
| Primary Key | PRIMARY KEY | _id (auto-generated) |

## Example in MongoShell

Create / Insert a document :

```
db.users.insertOne({
    name : "Abhishek",
    age : 21,
    city : "Delhi"
});
```

@ SYNTAX ERROR
—Abhishek Rathor

Find documents :

```
db.users.find();
```

Output :

```
[
  {
    "_id":
ObjectId("6716b4efc9a3d7a2a7b19d003"),
    "name" : "Abhishek",
    "age" : 21,
    "city" : "Delhi"
  }]
```

## Key Characteristics

| Feature | Document | Collection |
|---|---|---|
| Definition | Single record | Group of records |
| Equivalent in SQL | Row | Table |
| Format | BSON (Binary JSON) | Set of BSON documents |

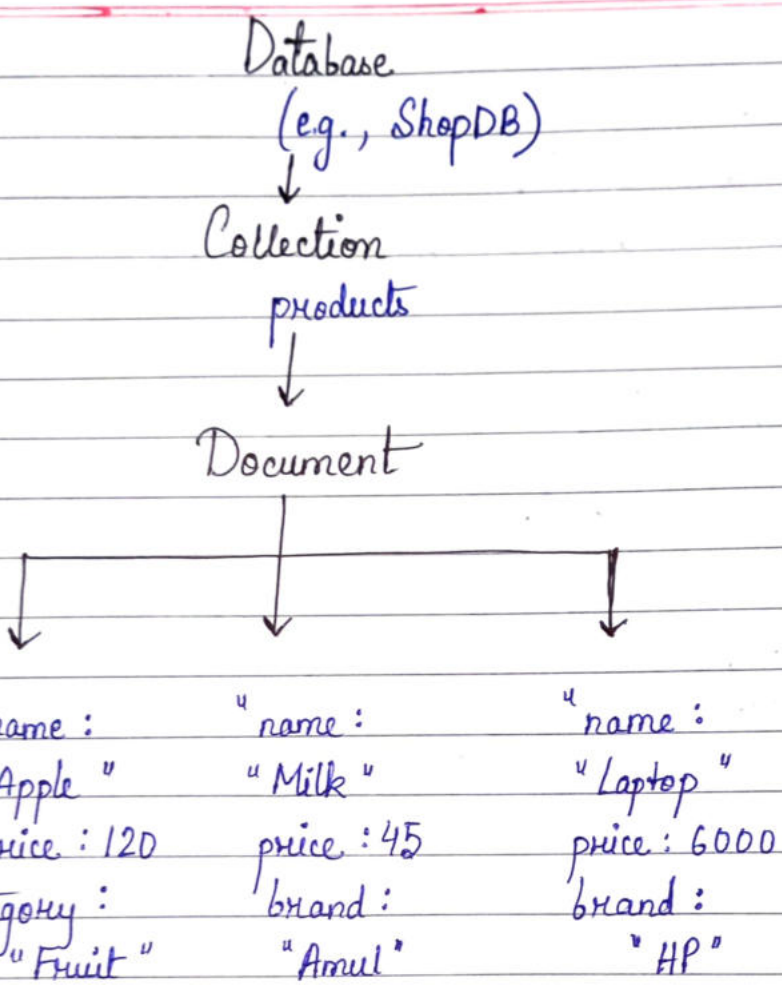
| | | |
|---|---|---|
| Contains | Fields | Documents |
| Structure | Key-value pairs | Schema-less group |
| Identified by | _id | Name |

## Example visualization

Database : ShopDB
└ Collection : products
  ├ Document 1 : { name : "Apple", price : 120, category : "Fruit" }
  ├ Document 2 : { name : "Milk", price : 45, brand : "Amul" }
  ├ Document 3 : { name : "Laptop", price : 60000, brand : "HP" }

## Summary

| Term | Description |
|---|---|
| Document | The smallest unit of data in MongoDB (like a row) |
| Collection | A group of related documents (like a table) |
| Schema-less | Different documents in the same collection can have different fields. |
| _id field | Automatically generated unique identifier for each document. |




## A Short Summary — CRUD Operations

* Create

```
db. users. insertOne ({ name : " Abhishek ", age : 21 });
db. users. insertMany ([
        { name : " Syntax ", age : 22 },
        { name : " Exnon ", age : 20 }
]);
```

* Read

```
db.users.find();
db.users.find({age: 21});
db.users.findOne({name: "Abhishek"});
```

* Update

```
db.users.updateOne({
    name: "Abhishek"},
  {$set: {age: 22}});

db.users.updateMany(
    {age: {$lt: 21}},
  {$set: {verified: true}}
);
```

* Delete

```
db.users.deleteOne({name: "Abhishek"});
db.users.deleteMany({age: {$gt: 25}});
```

# Projection (Selecting Specific Fields)

Projection in MongoDB means choosing which fields you want to include (or exclude) when retrieving documents using find() or findOne().

By default, MongoDB returns all fields of a document.
Projection lets you control that - just like selecting certain columns in SQL (SELECT name, age FROM users)

## Syntax

```
db.collection.find(<query>, <projection>)
```

- `<query>` → filter / condition
- `<projection>` → fields to include (1) or exclude (0)

Example : Include Specific Fields

```
db.users.find({}, { name : 1, age : 1, _id : 0 });
```

Explanation :

- name : 1 → include name
- age : 1 → include age
- _id : 0 → exclude _id (because it's returned by default)

Output :

```
[ { "name" : "Abhishek", "age" : 21 },
  { "name" : "Syntax", "age" : 22 }
]
```

Example : Exclude Certain Fields

```
db.users.find({}, { password : 0, email : 0 });
```

This returns all fields except password and email.

## Why It's Useful in MERN

- Reduces data transfer (only send what React needs)
- Improves performance
- Hides sensitive info (like passwords or tokens).

# Sorting & Limiting Results

Sorting → used to arrange documents in ascending or descending order based on one or more fields.

Syntax

```
db.collection.find().sort({ fieldName : 1 or -1 });
```

- 1 → Ascending (A→Z/0→9)
- -1 → Descending (Z→A/9→0)

Example : Sort by Age

```
db.users.find().sort({age : 1});
```

→ Ascending order (younger first)

```
db.users.find().sort({age : -1});
```

→ Descending order (oldest first)

Sort by Multiple Fields

```
db.users.find().sort({age : 1, name : -1});
```

→ Sorts first by age ascending, then by name descending (if age match).

Limiting Results → Limits the number of documents ~~returned~~ returned by a query.

```
db.users.find().limit(5);
```

→ Returns only the first 5 documents.

Combined Example : Sort + Limit

```
db.users.find({}, { name: 1, age: 1, _id: 0 })
    .sort({ age: -1 })
    .limit(3);
```

→ Returns the top 3 oldest users, showing only their name and age.

## Why It's Useful in MERN?

- Pagination : Show limited results per page (like 10 products at a time).
- Trending / Top lists : Sort posts, users, or products.
- Performance : Avoid fetching large unnecessary data sets.

## Real-world Example (MERN Use Case)

⇒

```
app.get("/api/users", async (req, res) => {
  const users = await User.find({}, { name: 1, email: 1, _id: 0 })
                          .sort({ name: 1 })
                          .limit(5);
  res.json(users);
});
```

This sends only 5 users, sorted alphabetically, to the frontend.

# Querying and Filtering in MongoDB

Querying and filtering let you search, match, and retrieve specific data from your collections — the heart of any backend operation in MERN.

* **Basic Syntax of find()**

db.collection.find(<query>, <projection>)

- <query> → conditions (filters to find matching documents)
- <projection> → optional; select which fields to display

Example:

```
db.users.find({ age: 21 }, { name: 1, _id: 0 });
```

→ Returns only the name of users whose age is 21.

## 1. Comparison Operators

used to compare field values in documents.

| Operator | Meaning | Example | Description |
|---|---|---|---|
| $eq | equal to | {age: {$eq: 21}} | Finds docs where age = 21 |
| $ne | not equal to | {age: {$ne: 21}} | Excludes age = 21 |
| $gt | greater than | {age: {$gt: 18}} | age > 18 |
| $gte | greater than or equal | {age: {$gte: 18}} | age ≥ 18 |
| $lt | less than | {age: {$lt: 25}} | age < 25 |
| $lte | less than or equal | {age: {$lte: 25}} | age ≤ 25 |

Example:

```
db.users.find({ age: { $gte: 18, $lte: 25 } });
```

→ Finds users aged b/w 18 and 25 (inclusive).

## 2. Logical Operators

→ used to combine multiple conditions

| Operator | Meaning | Example | Description |
|---|---|---|---|
| $and | Both condition true | `{ $and: [ { age: { $gte: 18 } }, { city: "Delhi" } ] }` | Users age ≥ 18 AND city = Delhi |
| $or | Either condition true | `{ $or: [ { age: { $lt: 18 } }, { age: { $gt: 25 } } ] }` | users age < 18 OR > 25 |
| $not | negates a condition | `{ age: { $not: { $gt: 30 } } }` | users age ≤ 30 |
| $nor | none of the conditions | `{ $nor: [ { age: { $lt: 18 } }, { city: "Delhi" } ] }` | Not age < 18 and not city Delhi |

Example:

```
db.user.find({
    $and: [ { age: { $gte: 18 } },
            { verified: true }
    ] });
```

→ Returns all verified users age ≥ 18.

## 3. Array Operators

used when fields are arrays (like tags, hobbies, skills, etc.)

| Operator | Meaning | Example | Description |
|---|---|---|---|
| $in | Matches any value in the array | {city: {$in: ["Kolkata", "Delhi"]}} | city = Kolkata or Delhi |
| $nin | not in | {city: {$nin: ["Delhi", "Mumbai"]}} | Excludes Delhi, Mumbai |
| $all | matches all specified values | {hobbies: {$all: ["reading", "music"]}} | must contain both reading & music |

Example :

```
db.users.find({ hobbies: {$in: ["coding", "painting"]}});
```

→ Find users who like coding or painting.

## 4. Pattern Matching

Pattern matching in MongoDB means searching for text that matches a specific pattern rather than an exact value.

Mongodb uses the $regex (regular expression) operator for this.

It works like searching with wildcards - find all documents where a field contains, starts with, or ends with a specific substring.

Syntax

```
db.collection.find ( { fieldName : { $ regex : /pattern/,
                                     $ options : " options " } } )
```

- pattern : the text or regex pattern to match
- options : modify how the regex behaves

Example 1 : Basic Pattern Match

Find users whose name contains the word " Apple ".

```
db.users.find ( { name : { $ regex : / Apple/ } } );
```

Matches :

- " Apple "
- " Custard Apple "
- " Pine Apple "

Example 2 : Case-Insensitive Search

To make the search not case-sensitive, use $ options : " i ".

```
db.users.find ( { name : { $ regex : /Apple/i } } );
```

Matches :

- " Apple "
- " apple "
- " APPLE "
- " custard apple "

## Options You Can Use with $regex

| Option | Meaning | Example |
|---|---|---|
| i | Case-sensitive | /apple/i |
| m | multiline match | /^start/m |
| x | ignore whitespace/comments in pattern | /no or /x |
| s | allows . to match newline characters | /a.b/s |

## Common Use Cases in MERN

| Feature | Use Case | Example |
|---|---|---|
| Search bar | Find users/products by name | {name: { $regex: query, $options: "i" }} |
| Filter posts | Titles containing a keyword | { title: { $regex: keyword }} |
| Email lookup | Validate or search emails by domain | {email: { $regex: /@gmail.com$/ }} |

Tip

When your dataset grows large, using $regex can be slow, because it scans every document.

To improve performance:

- use indexes on the field
- Or use MongoDB's full-text search

@ SYNTAX ERROR
— Abhishek Rathor

# 5. Pagination & Sorting

Pagination means dividing your large result set into small chunks.

Example:

- Pg 1 → documents 1 - 10
- Pg 2 → documents 11 - 20
- Pg 3 → documents 21 - 30

In MongoDB, we achieve this using .skip() and .limit().

## Syntax

```
db.collection.find().skip(<number_to_skip>).limit(<number_to_limit>);
```

## Example 1 : Basic Pagination

Imagine you have 50 users in your database
To show 10 users per page :

- Page 1 : db.users.find().skip(0).limit(10);

→ Skips 0 users, shows first 10

- Page 2 : db.users.find().skip(10).limit(10);

→ Skips first 10 users, shows next 10

- Page 3 : db.users.find().skip(20).limit(10);

→ Skips first 20 users, shows next 10.

## Formula for Pagination

If :

- pageNumber = current page
- pagesize = number of documents per page

Then : skip = (pageNumber - 1) * page Size.

Example : Page 3 with 10 docs per page :

skip = (3-1) * 10 = 20

So :

```
db.users.find().skip(20).limit(10);
```

- Sorting → Sorting documents in ascending (1) or descending (-1) order

```
db.users.find().sort({age:1}); // ascending
db.users.find().sort({age:-1});
```

Multiple fields :

```
db.users.find().limit(5);
```

- Limiting → limits the number of results shown.

```
db.users.find().limit(5);
```

→ Returns first 5 documents

- Skipping → skips a certain number of results (used for pagination)

```
db.users.find().skip(5).limit(5);
```

→ Skips first 5 docs, then shows next 5.

## Example : Combined (Query [Pagination + Sort + Projection])

```
db.users.find({ verified : true },
  { name : 1, email : 1, _id : 0 })
        .sort ({ name : 1 })
        .skip (10)
        .limit (5) ;
```

## MERN Use Case Example

```
app.get("/api/users", async (req, res) => {
    const { page = 1, limit = 5 } =
                        req.query;

    const users = await
User.find ({ verified : true }, { name : 1, email : 1, _id : 0 })
                        .sort ({ name : 1 })
                        .skip ((page - 1) * limit)
                        .limit (limit);

    res.json (users);
});
```

→ This creates a paginated API endpoint React can call – showing 5 verified users per page.

## Summary Table

| Category | Operators | Use Case |
|---|---|---|
| Comparison | $eq, $gt, $lt, $gte, $lte, $ne. | compare numeric/data values |
| Logical | $and, $or, $not, $nor | combine multiple filters |
| Array | $in, $nin, $all | query array fields |
| Pattern Matching | $regex | Text search |
| Project | { field : 1/10 } | select specific fields |
| Pagination | .limit(), .skip(), .sort() | control displayed results |

# Mongoose (MongoDB with Node.js)

Mongoose is an ODM (Object Data Modeling) library for MongoDB and Node.js.

It acts as a bridge between your Express server and the MongoDB database.

1. Installing and Connecting Mongoose

Installation → run the command in project folder.

```
npm install mongoose
```

Connecting to MongoDB → In your server.js or db.js file :

```
const mongoose = require("mongoose");
...
```

For Cloud Database (MongoDB Atlas)

```
mongoose.connect("mongodb+srv://<username>:<password>@...
...")
```

Connection Flow :

1. Node.js app imports Mongoose.
2. Connects to MongoDB (local or cloud)
3. Once connected → ready to perform CRUD operations

## 2. Defining Schemas

A schema defines the structure of your document — like a blueprint or data model.

Example : User Schema

```
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 : 18 },
    isAdmin : { type : Boolean , default : false }, },
{ timestamps: true });
```

Explanation :

- type : data type of the field
- required : must have a value
- unique : no two documents can have same value
- default : default value if not provided
- timestamps : automatically adds createdAt & updatedAt

## 3. Creating Models

A Model is a compiled version of the schema — used to create, read, update, and delete documents.

```
const User = mongoose.model("User", userSchema);
```

Now we can use :

```
user.find()
user.create()
user.findById()
user.findByIdAndUpdate()
user.findByIdAndDelete()
```

## 4. CRUD Operations using Mongoose

### Create (Insert)

```
const newUser = new User({
    name: "Abhishek",
    email: "syntaxerror@gmail.com",
    age: 21
});
await newUser.save(); // Save to DB
```

OR directly

```
await User.create({ name: "Abhishek",
    email: "syntaxerror@gmail.com", age: 21});
```

### Read (Find)

Fetch all users : `const users = await User.find();`

Find one user : `const user = await User.findOne({ email: "syntaxerror@gmail.com" });`

Find by ID :

```
const user = await
    User.findById("652fdc44a10b3f9e13a77ac4");
```

Update :

```
await User.findByIdAndUpdate(
        "652fdc44a10b3f9e13a77ac4",
    { age: 23 },
    { new: true } // returns updated documents
);
```

Delete :

```
await User.findByIdAndDelete("652fdc4410b3f9e13a77ac4");
```

## All CRUD Summary

| Operation | Mongoose Method | Description |
|---|---|---|
| Create | • save() / • create() | Insert new doc |
| Read | • find() / • findOne() / • findById() | Retrieve docs |
| Update | • findByIdAndUpdate() / • updateOne() | Modify doc |
| Delete | • findByIdAndDelete() / • deleteOne() | Remove doc |

## 5. Schema Validation and Default Values

Schema-level validation ensures data is consistent before saving.

Example:

```
const productSchema = new mongoose.Schema({
    name: { type: String, required: [true,
"Name is required"] },
    price: { type: Number, min: 0 },
    category: { type: String, enum:
["Electronics", "Books", "Clothing"] },
    inStock: { type: Boolean, default: true } });
```

Features:

- required : must be filled
- min/max : Range validation
- enum : Restrict values
- default : auto-filled if not provided

## 6. Using Async / Await for Database Operations

Since MongoDB queries are asynchronous, you should use async/await for cleaner code.

```
app.get("/users", async (req, res) => { try {
    const users = await User.find();
    res.json(users);
} catch (err) {
    res.status(500).json({ message: err.message });
  }
});
```

## 7. Error Handling with try/catch

Always handle errors gracefully when performing DB operations.

Example :

```
try {
    Const user = await User.create ({ name : "Abhishek" });
    console.log ("User created :", user);
} catch (error) {
    console.error ("Error creating user :",
        error.message);
}
```

## 8. Schema Options and Useful Properties

Common Options in Mongoose Schema :

| Option | Description | Example |
|---|---|---|
| required | makes field mandatory | {type : String, required : true} |
| unique | ensures no duplicates | {type : String, unique : true} |
| default | sets default value | {type : Boolean, default : false} |
| min/max | numeric range validation | {type : Number, min : 18} |
| enum | restricts values | {type : String, enum : ['male', 'female']} |
| timestamps | auto createdAt, updatedAt | new mongoose.Schema({...}, {timestamps : true}) |

## Example : Complete Mongoose Setup

```
const mongoose = require ("mongoose");
mongoose.connect ("mongodb://localhost:27017/myAPP")
.then (() => console.log ("Connected to MongoDB"))
.catch (err => console.error ("Error :", err));

const userSchema = new mongoose.Schema ({ name : { type : String,
                                                   required : true },
email : { type : String, required : true, unique : true },
age : { type : Number, min : 0, default : 18 }, },
  { timestamps : true });

const User = mongoose.model ("User", userSchema);
(async () => {
    try {
        const newUser = await
User.create ({ name : "Abhishek", email : "syntaxerror@gmail
                                              .com" });
console.log ("User added :", newUser); }
catch (error) { console.log ("Error :", error.message); }
}) ();
```

**Flow : React → API call → Express route → Mongoose query → MongoDB → Response back to React**

# Relationships (References & Population)

Overview → Relationships are how you connect data across collections (e.g., users → posts). Two main strategies :

- Embedding (store related data inside the parent document)
- Referencing (store ObjectId references to other documents)

## Embedding vs Referencing — quick comparison

| Aspect | Embedding | Referencing |
|---|---|---|
| When to use | Small, bounded related data you read together (comments inside a post, profile inside user) | Large relationships, frequently-updated related items, or many-to-many |
| Read pattern | Fast – single document read | Requires extra query or $lookup / .populate() |
| Update pattern | more expensive if embedded data changes often | cheaper to update single document (separate collection) |
| Document size limits | Beware BSON limit (16MB) | Not limited by parent document size |
| Example | post.comments : [ {text, userId, ts} ] | post.author = ObjectId("...") referring to users. |

# Mongoose : ref + .populate()

## Schemas & reference example

```
// User model
const userSchema = new mongoose.Schema ({
    name : String,
    email : String });
const User = mongoose.model ("User", userSchema);

// Post model with reference to User
const PostSchema = new mongoose.Schema ({
    title : String,
    body : String,
    author : {type : mongoose.Schema.Types.ObjectId, ref: "User"}

    // <-- ref
});
const post = mongoose.model ('Post', postSchema);
```

## Populate usage

```
// Returns posts with author fields populated (join-like)
const posts = await
Post.find().populate ('author', 'name','email'); // include only name, email.
```

.populate() runs extra query (lies) behind the scenes ; good for convenience, but can be slower than aggregation $lookup for large datasets.

## Relationship patterns & example

### One-to-one

- Option A : embed the smaller doc inside larger if always read together (e.g., user.profile).
- Option B : reference each other (e.g., user.profileId) if profile large or shared.

### One-to-many

- If "many" is small & bounded → embed array of subdocuments
- If "many" can grow or needs independent queries → reference (array of ObjectIds or hold foreign key on child docs).

```
// child holds ref to parent (common)
const commentSchema = new Schema ({ text : String, post :
  ObjectId  /* ref : 'Post' */ });
```

### Many-to-many

- Option 1 : array of ObjectIds on both documents (if arrays small).
- Option 2 : linking collection (join table) – best for large many-to-many

```
// linking collection example
const enrollmentSchema = new Schema ({ student : ObjectId, course :
ObjectId, enrolledAt : Date });
```

Virtual populate (inverse populate without storing array)

```
userSchema.virtual('posts', { ref: 'Post',
    localField : '_id',
    foreignField : 'author'
});
await
User.findById(id).populate('posts'); // populates posts for user
```

When to choose what (rules of thumb)

- Read together, small, few updates → embed.
- Many related items, independent updates, or size grows → reference
- Need global uniqueness / lookup (users across posts) → reference
- Need fast analytics / aggregations across collections → use referencing + $lookup or keep denormalized counters.

# Data Modeling & Schema Design

Principles —

- Model for your query patterns (design reads first)
- Favor simplicity and predictability.
- Consider consistency, atomicity, and performance.
- Avoid huge arrays or embedding that may exceed BSON 16 MB.

## Common patterns

1. Embed small subdocuments :

```
product = { name, price, specs : { weight, color } }
```

2. Reference for large or many children :

```
post.author = ObjectId('....'); // author stored
                in users collection.
```

3. Hybrid : Stores frequently-read fields duplicated for fast reads (denormalize), but keep authoritative data in separate collection to update occasionally

4. Bucketing pattern : Groups items into buckets to avoid huge arrays - e.g., store events in a buckets collection where each bucket covers N events or a time-range.

5. Time-series pattern : Uses capped collections, TTL indexes, or pre-shard time-based ranges.

## Important constraints & practices

- Max BSON doc size = 16 MB → split or reference when approaching.
- Avoid large unbounded arrays inside a doc.
- Version your schema in-app (schemaVersion field) if you expect migrations.

- Plan deletion / archival : move old data to archive collections or storage (cold storage).
- Use unique constraints for fields requiring uniqueness (email).
- consider write vs read workload : more indexes improve reads but slow writes.

## Example designs

### Blog

- users (id, name, email)
- posts (id, title, body, author : ObjectId, tags [], createdAt)
- comments (id, postId : ObjectId, author : ObjectId, text, createdAt) – reference comments to posts (so posts don't grow unbounded).

### E-commerce

- products
- orders (embedded line items small? or reference product snapshot to capture price at order time)
- users
- use separate inventory collections or denormalized stock on product updated atomically.

# Aggregation Framework

A pipeline system to process and transform documents on the server — used for analytics, grouping, joining, reshaping data.

Pipeline = array of stages. Each stage transforms the documents passed to it.

@ SYNTAX ERROR
— Abhishek Rathor

## Common stages & meaning

- $match — filter documents (like WHERE). Put it early.
- $facet — run multiple pipelines in parallel and return combined output.
- $group — aggregate data (like GROUP BY). Use $sum, $avg, $push.
- $bucket / $bucketAuto — group values into buckets (histogram)
- $sort — order results
- $addFields / $set — add or modify fields
- $project — include / exclude / compute fields
- $replaceRoot — replace document root with a subdocument
- $lookup — join with another collection (left outer join)
- $replaceWith — replace document root
- $unwind — deconstruct an array field to multiple docs.
- $limit, $skip — pagination inside pipeline

## Example pipelines

### Posts per user (with username)

```
db.posts.aggregate([
  { $group: { _id: "$author", count:
```

```
{ $sum : 1 } } },
  { $lookup : { from : "users",
                localField : "_id",
                foreignField : "_id",
                as : "user" } },
  { $unwind : "$user" },
  { $project : { _id : 0, user : "$user.name", count : 1 } },
  { $sort : { count : -1 } }
])
```

Top 5 products by sales

```
db.orders.aggregate([
    { $unwind : "$items" },
  { $group : { _id : "$items.productId",
totalQty : { $sum : "$items.qty" } } },
    { $sort : { totalQty : -1 } },
    { $limit : 5 },
    { $lookup : { from : "products",
localField : "_id", foreignField : "_id", as : "product" } },
    { $unwind : "$product" },
    { $project : { _id : 0, product : "$product.name",
totalQty : 1 } }  ])
```

Using $facet for multiple aggregations in one pass

```
db.orders.aggregate([ { $facet : { recentOrders :
    [{ $sort : { createdAt : -1 } }, { $limit : 5 }],
```

```
salesByDay : [ { $ group : { _id : $
{ $ dateToString : { format : "%Y-%m-%d", date : "$ createdAt" }
  }, total : { $ sum : "$ total" } } },
        { $ sort : { _id : 1 } } ]
    } } ])
```

Mongoose usage

```
const pipeline = [ /* stages as above */ ];
const result = await
Post.aggregate (pipeline);
```

Performance tips

- Place $ match early to reduce documents processed.
- Use $ project early to remove fields you don't need.
- $ lookup can be expensive ; ensure the joined field is indexed
- For heavy aggregation, consider pre-aggregating (materialized views) or using MongoDB Atlas Online Archive / Aggregation Pipelines with indexes.
- $ expr allows expression evaluation in $ match.

# MongoDB Atlas & Deployment

Quick steps (high level)

1. Create Atlas account on MongoDB Atlas.
2. Create a Project → Build a Cluster (free tier available)
3. Create a Database User (username / password) — grant least privilege.

4. Network Access : whitelist your app IP(s) or use 0.0.0.0/0 (not recommended for prod).
5. Get connection string (URI) —
   mongodb+srv://<user>:<password>@cluster0.xxx.mongodb.net/<dbname>?retryWrites=true&w=majority
6. Use that URI in your Node app (via environment variable).

## Node/Mongoose connection example (with .env)

• env

```
MONGO_URI = mongodb+srv://myUser:myPass@cluster0.xxxx.mongodb.net/myDB
```

db.js

```
require('dotenv').config();
const mongoose = require('mongoose');
mongoose.connect(process.env.MONGO_URI,
{
    useNewUrlParser: true,
    useUnifiedTopology: true,
    // poolSize and other options if needed
})
.then(() => console.log('Connected to Atlas'))
.catch(err => console.error('Atlas connection error:', err));
```

## Product consideration & best practices

- Use environment variables for credentials (never commit .env)
- TLS in enabled by default for Atlas – keep it.
- Use priviledged DB user (different users for read-only or admin tasks)
- Enable IP allowlist / VPC peering for secure connectivity
- configure connection pool and timeouts per app load.
- use read preferences & write concern appropriately.
- Monitor with Atlas metrics (CPU, connections, IOPS).
- Use Atlas backups (continuous snapshots or scheduled backups). Enable PITR (point-in-time) if needed.

# Indexing & Performance

An index allows MongoDB to find documents efficiently without scanning the entire collection (similar to book index).

### Types of indexes

- Single-field index: { field : 1 }
- Compound index : { a : 1, b : -1 } – order matters
- Multikey index : on array fields (indexes each element)
- Text index : for text search ({ name : "text", description : "text" })

- TTL index : { createdAt : 1 } with { expireAfterSeconds : 3600 } - auto-delete docs.
- Hashed index : for sharding hashed shard key

Create index examples

Shell

```
db.users.createIndex ({ email : 1 }, { unique : true });
db.posts.createIndex ({ author : 1, createdAt : -1 });
db.products.createIndex ({ name : "text", description : "text" });
```

Mongoose

```
const userSchema = new Schema ({ email : String });
userSchema.index ({ email : 1 }, { unique : true });
```

Compound indexes Rule of thumb

- Build compound indexes to match common query patterns and sort order.
- For query find ({ a : 1, b : 1 }).sort({ c : -1 }) create index { a : 1, b : 1, c : -1 } to support both filter and sort

Explain plan (analyze performance)

```
db.users.find ({ email : "syntaxerror@gmail.com" })
.explain ("executionStats")
```

Look for:

- executionStats.totalDocsExamined — should be small if index used
- executionStats.executionTimeMillis
- queryPlanner.winningPlan.inputStage — shows index usage.

If a query scans many docs → create proper index.

## Indexing trade-offs

- Each index increases write cost (insert/update/delete)
- Indexes consume memory (wiredTiger cache / RAM)
- Avoid indexing low-selectivity fields (boolean flags) unless combined in a compound index.

## Quick tips

- Use covering indexes (index contains all fields requested) to avoid fetching documents.
- Add indexes only after measuring/observing slow queries.
- Monitor with .explain() and Atlas Performance Advisor suggestions.

# Advanced (Transactions, Change Streams, Sharding, Backup & Restore)

## Transactions (multi-document, ACID)

- Since MongoDB 4+, multi-document ACID transactions are supported in replica sets and sharded clusters.
- Use when you need atomic updates across multiple documents/collections (e.g., transfer money between accounts).

The full form of ACID in database is :

ACID = Atomicity, Consistency, Isolation, Durability

1. Atomicity → "All or nothing" rule.
   A transaction must complete entirely or not at all.
2. Consistency → The database must remain in a valid state before and after the transaction.
3. Isolation → Multiple transactions can occur at the same time, but each behaves as if it's only one running.
4. Durability → Once a transaction is committed, it's permanent — even if there's a system crash, the data stays saved.

## Mongoose example

```
const session = await mongoose.startSession();
try {
    session.startTransaction();
    await User.updateOne({ _id: from }, { $inc: { balance: -100 } },
{ session });
    await User.updateOne({ _id: to }, { $inc: { balance: 100 } },
{ session });
    await Session.commitTransaction();
} catch (err) {
    await session.abortTransaction();
} finally {
    session.endSession();
}
```

session.withTransaction(asyn() ⇒ {...}) is also supported (driver feature)

## Change Streams (real-time)

- Listen to DB changes in real-time (watch()).
- useful for live notifications, feeds, invalidating caches

### Node example

```
const changeStream = Post.watch();
changeStream.on('change', (change) ⇒ { // change.operationType:
    'insert' | 'update' | 'delete'
    console.log(change);
});
```

Notes:

- Requires replica set (Atlas clusters support this)
- Use filters on the change stream pipeline to reduce noise.

## Sharding (horizontal scaling)

- Sharding distributes data across multiple shards for scale.
- Key components : mongos (query router), config servers, shards (replica sets).
- Shard key selection is critical — choose a key with good cardi-nality and evenly distributed writes.
- Commands (very high-level):
  - sh.enableSharding("myDB")
  - sh.shardCollection("myDB.collection", {shardKey: 1})
- Sharding is complex — test & plan shard key before enabling

## Backup & Restore

mongodump / mongorestore (CLI)

- Dump

mongodump --uri = "mongodb+srv://user:pass@cluster0/.../myDB" --out = /backups/2025-10-08

- Restore

```
mongorestore --uri = "mongodb+srv://..." /backups/2025-10-08
```

Atlas

- Atlas provides automatically snapshots and point-in-time recovery (PITR) on certain tiers.
- Configure backup windows, retention, and test restores periodically.

Other advanced bits

- GridFS — store files > 16MB inside Mongo (file chunks)
- TTL Indexes — automatic expiry for sessions or temporary tokens.
- Aggregation Pipelines optimizations — pipeline stages ordering, $merge to write results back to collection.

# Express.JS

Express is a fast, assertive, essential and moderate web framework of Node.js. You can assume express as a layer built on the top of the Nodejs that helps manage a server and routes. It provides a robust set of features to develop web and mobile applications.

Let's see some of the core features of Express framework:

- It can be used to design single-page, multi-page and hybrid web applications
- It allows to setup middlewares to respond to HTTP Requests.
- It defines a routing table which is used to perform different actions based on HTTP method and URL.
- It allows to dynamically render HTML pages based on passing arguments to templates

## Why use Express

- Ultra fast 1/10
- Asynchronous and single threaded
- MVC like structure
- Robust API makes routine easy

## How does Express look like

Let's see a basic Express.js app

File : basic_express.js

```
var express = require ('express');
var app = express ();
app.get ('/', function (req, res) {
res.send ('Welcome to JavaTpoint'); });
```

```
var server = app.listen(8000, function () {
    var post = server.address().address
    var post = server.address().port
    console.log("Example app listening at https://%s,%s',
                host, port);
});
```

## Setting Up Express

```
npm init -y
npm install express mongoose dotenv cors
```

Then create a file : server.js :

```
const express = require('express');
const app = express();
const PORT = 5000;
// Middleware
app.use(express.json()); // To parse JSON requests

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

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

Now run :

```
node server.js
```

# Basic Express Concepts

Each route in Express receives two main objects:

## Express.js Request Object

The express.js request object represents the HTTP request and has properties for the request query, string, parameters, body, HTTP headers, and soon.

```
app.get('/', function (req, res) { //-- })
```

### Properties

The following table specifies some of the properties associated with request object.

| Object properties | Description |
|---|---|
| req.app | → used to hold a reference to the instance of the express application i.e. using the middleware |
| req.body | → contains key-value pairs of data submitted in the request body |
| req.ips | → when trust proxy setting is true, this property contains an array of ip address specified in the ?x-forwarded-for? request header |
| req.params | → an object containing properties mapped to the named route? parameters? |
| req.query | → an object containing a property for each query string parameter in the route |
| req.route | → the currently matched route a string |
| req.secure | → a boolean that is true if a TLS connection is established |

@ SYNTAX ERROR
— Abhishek Rathor

# Express.js Response Object

The response object (res) specifies the HTTP response which is sent by an Express app when it gets an HTTP request.

## What it does

- It sends response back to the client browser.
- It facilitates you to put new cookies value and that will write to the client browser (under cross domain rule)
- Once you res.send() or res.direct() or res.render(), you cannot do it again otherwise, there will be uncaught error.

## Properties

Let's see some properties of response object

| Object Properties | Description |
|---|---|
| res.app | → It holds a reference to the instance of the express application that is using the middleware. |
| reshedders.sent | → It is a Boolean property that indicates if the app sent HTTP headers for the response. |
| res.locals | → It specifies an object that contains response local variables scoped to the request. |

Example :

```
app.post('/user', (req, res) => {
    const data = req.body; // get JSON data
    res.status(201).json({ message: 'User created', user : data });
});
```

# Express.js Routing

Routing is made from the word route. It is used to determine the specific behavior of an application. It specifies how an application responds a client request to a particular route URL or path and a specific HTTP request method (GET, POST etc.). It can handle different types of HTTP requests.
Let's take an example to see basic routing.

Example :

```
app.get('/users', (req, res) => {...}); // Read data
app.post('/users', (req, res) => {...}); // Create new
app.put('/users/:id', (req, res) => {...}). // update by Id
app.delete('/users/:id', (req, res) => {...}); // Delete
```

Each route can also handle route params :

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

# Express.js Middleware

Express.js Middleware are different types of functions that are involved in and invoked by the Express.js routing layer before the final request handler. As the name specified, middleware appears in the middle between an initial request and final intended route. In stack, middleware functions are always invoked in the order in which they are added.

Middleware is commonly used to perform tasks like body parsing for URL-encoded or JSON requests, cookie parsing for basic cookie handling or even building Javascript modules on the fly.

## What is a Middleware function?

Middleware function are the functions that access to the request and response object (req, res) in request-response cycle.

A middleware function can perform the following tasks:

- It can execute any code.
- It can make changes to the request and the response objects
- It can end the request-response only cycle.
- It can all the next middleware function in the stack.

## Express.js Middleware

Following is a list of possibly used middleware in Express.js app:

- Application-level middleware.
- Router-level middleware
- Error-handling middleware
- Built-in middleware
- Third-party middleware

Common uses :

- Parse JSON
- Enable CORS
- Handle authentication
- Log requests

Example :

```
app.use (express.json ()); // Built-in
app.use (cors ()); // Cross-Origin Resource Sharing
```

Custom middleware :

```
const logger = (req, res, next) => {
    console.log (`${req.method} ${req.url}`);
    next (); // pass to next middleware or route
});
app.use (logger);
```

Use of Express.js Middleware

If you want to record every time you get a request then you can use a middleware.

File : simple-middleware.js

```
var express = require ('express')
var app = express ();
app.use (function (req, res, next) {
    console.log ('%s %s', req.method, req.url);
    next ();
});
app.get ('/', function (req, res, next) {
    res.send ('How can I help you?'); });
```

```
var server = app.listen (8000, function () {
    var host = server.address().address
    var port = server.address().port
    console.log ("Example app listening at https://%s %s", host, port)
})
```

You see that server is listening
Now, you can see the result generated by server on the local host http://127.0.0.1:8000

Output

sample app listening at http://::::8000

you can see that output is same but command prompt is displaying a GET result.

Go to http://127.0.0.1:8000/help

How can I help you?

As many times as you reload the page, the command prompt will be updated

Note: In the above example next() middleware is used.

# Express.js Scaffolding

Scaffolding is a technique that is supported by some MVC frameworks. It is mainly supported by the following frameworks:
Ruby on Rails, OutSystem Platform, Express Framework, play Framework Django etc.

Scaffolding facilitates the programmers to specify how the application data may be used. This specification is used by the frameworks with predefined code templates to generate the final code that the application can use for CRUD operations.

Express.js Scaffold → An Express.js scaffold supports candy and more. web projects based on Node.js.

# Express + MongoDB (via Mongoose)

Connection setup :

```
const mongoose = require('mongoose');
require('dotenv').config();
mongoose.connect(process.env.MONGO_URI).then(()
  => console.log('MongoDB Connected'))
.catch(err => console.log(err));
```

Example Model:

```
const userSchema = new mongoose.Schema({
    name : String,
    email : String, });
const User = mongoose.model('User', userSchema);
```

Example Route :

```
app.post('/users', async (req, res) => {
try { const user = new User(req.body);
    const saved = await user.save();
res.status(201).json(saved); } catch (err) {
```

res.status(400).json({ error: err.message }); } });

# RESTful APIs

Express is perfect for building REST APIs that React frontend can call.

| Method | Meaning | Example URL |
|---|---|---|
| GET | Read data | /api/users |
| POST | Create data | /api/users |
| PUT | update data | /api/users/:id |
| DELETE | remove data | /api/users/:id |

Example

```
app.get('/api/users', async (req, res) =>
{
  const users = await User.find();
  res.json(users);
});
```

# Error Handling in Express.js

Error handling means catching and responding to any problems that occur while your backend processes a request — without crashing the server.

Types of Errors :

1. Synchronous errors

2. ~~Asy~~ Asynchronous errors
3. Operational errors
4. Programming errors

## Basic Error Handling

You can catch and respond with an error message instead of crashing.

```
app.get('/example', (req, res) => {
    try {
        // something goes wrong
        throw new Error('Oops! Something went wrong.');
    } catch (err) {
        res.status(500).json({ message: err.message }); } });
```

Output : {"message": "Oops! Something went wrong."}

## Centralized Error Handling Middleware

Create a custom error middleware :

```
// middleware/errorMiddleware.js
const errorHandler = (err, req, res, next) => {
    console.error(err.stack);
    res.status(err.statusCode || 500).json({
        success: false
        message: err.message || 'Internal Server Error' });
};
module.exports = errorHandler;
```

Use it in server.js

```
const errorHandler = require('./middleware/errorMiddleware');
app.use(errorHandler);
```

Now, anywhere in your routes, you can just throw an error, and Express will automatically pass it to this middleware.

## Error Handling in Auth Routes (Example)

```
app.post('/login', async (req, res, next) => {
  const { email, password } = req.body;
  if (!email || !password)
    return next(ErrorResponse('Please enter email and password', 400));
  const user = await
  User.findOne({ email });
  if (!user) return
  next(ErrorResponse('Invalid credentials', 401));
  res.json({ message: 'Login successful' });
});
```

## Final Error Flow Diagram

Client Request
↓
Express Route
↓
Route Handler (async/sync) → If error → next(err)

Response to client (JSON error)
↑
Error Middleware (centralized Handler)
↑

# Environment Variables (.env)

Don't hardcode credentials — store them in .env.

• env

```
PORT = 5000
MONGO_URI = mongodb+srv://user:
pass@cluster.mongodb.net/db
```

server.js

```
require('dotenv').config();
const PORT = process.env.PORT || 5000;
```

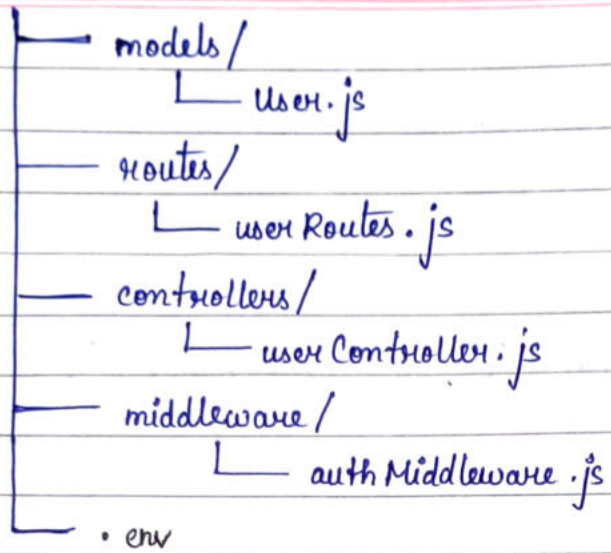
# CORS Setup (for React Frontend)

When React (frontend) calls Express (backend), CORS (Cross-Origin Resource Sharing) must be allowed.

```
const cors = required('cors');
app.use(cors({
        origin: 'http://localhost:3000', // frontend URL
        credentials: true
}));
```

# Express Folder Structure (Best Practice)

```
backend/
├── server.js
├── config/
│   └── db.js
```




Example

routes/userRoutes.js

```
const express = require('express');
const router = express.Router();
const { getUsers, addUser } =
require('../controllers/userController');
router.get('/', getUsers);
router.post('/', addUser);
module.exports = router;
```

controllers/userController.js

```
const User = require('../models/User');
exports.getUsers = async (req, res) => {
        const users = await User.find();
        res.json(users); };
exports.addUser = async (req, res) => {
        const newUser = new User(req.body);
```

```
    await newUser.save();
    res.status(201).json(newUser);
};
```

server.js

```
app.use('/api/users', require('./routes/userRoutes'));
```

# Express with React (Full MERN)

When deployed:

- React runs on frontend (Vite or CRA)
- Express services API routes
- MongoDB stores data

In production, Express can also serve React build:

```
app.use(express.static('client/build'));
app.get('*', (req, res) => {
    res.sendFile(path.resolve(__dirname,
'client', 'build', 'index.html'));});
```

# Express with Middleware Stack (Example)

```
app.use(express.json());
app.use(cors());
app.use('/api/users', userRoutes);
app.use('/api/auth', authRoutes);
app.use(errorHandler);
```

Each route → modular, maintainable, clean code

# Common Express Methods

| Method | Description |
|---|---|
| app.use() | mount middleware or routes |
| app.get() | Handle GET request |
| app.post() | Handle POST request |
| app.put() | Handle PUT request |
| app.delete() | Handle DELETE request |
| res.json() | Send JSON response |
| res.status() | Set HTTP status code |
| next() | pass to next middleware |
| req.params | route params |
| req.query | query string |
| req.body | POST body. |

# Authentication (Brief)

Used in MERN for user login/register

Common flow :

1. User sends credentials → /login
2. Server verifies credentials
3. Server returns JWT token
4. Client stores token (local Storage)
5. Future requests include token in headers.
6. Middleware verifies token → allows access

# NODE.JS

Node.js is a cross-platform runtime ~~application~~ environment and library for running JavaScript applications outside the browser. It is used for creating server-side and networking web applications. It is open source and free to use.

Many of the basic modules of Node.js are written in JavaScript. Node.js is mostly use to run real-time server applications.

Node.js also provides a rich library of various JavaScript modules to simplify the development of web applications.

Node.js = Runtime Environment + JavaScript library

## Features of Node.js

Following is a list of some important features of Node.js that makes it the first choice of software architects.

1) Extremely fast
2) I/O is Asynchronous and Event Driven.
3) Single threaded
4) Highly scalable
5) No buffering
6) Open source
7) License

# Role of Node.js in the MERN Stack

It serves as a backend engine (runtime environment)

It serves as —

1. Backend Server Creation → builds the server that handles requests and responses
2. Connects Frontend and Database → Node.js acts as a bridge between the react frontend (client) and the MongoDB database (data layer)

   It uses Express.js and Mongoose to receive data from React, store or fetch data from MongoDB and send results back to React.
3. Hosts the Express Framework → Express.js runs on top of Node.js.
4. Handles Asynchronous Operations
5. Uses npm → provides access to thousands of ready-made modules.

## Simple Flow Diagram

[ React Frontend ]
↓
(Sends API Request)
↓
[ Node.js + Express Backend ]
↓
[ MongoDB Database ]
↓
(Returns Response)

@ SYNTAX ERROR
— Abhishek Rathor

# Main Components of Node.js

## 1. V8 JavaScript Engine

→ developed by Google written in C++
→ It converts JavaScript code into machine code directly.
→ ensures high performance and fast execution
→ The same engine used in the Google Chrome browser.

Role in Node.js
Executes JavaScript code efficiently on the server side.

Example :

```
console.log("Executed by V8 Engine");
```

## 2. Libuv (Library for Asynchronous I/O)

- A C library that provides Node.js with :
  - Event loop
  - Asynchronous I/O operations
  - Thread pool.

It helps Node.js handle :
- File operations
- Network requests
- Timers
  ↳ without blocking the main thread.

## 3. Node.js APIs (Bindings & Core Modules)

Node.js includes built-in APIs that developers can use directly — no need for external libraries.

### Common Core Modules:

| Module | | Purpose |
|---|---|---|
| http | → | create web servers |
| fs | → | file system operations (read/write files) |
| path | → | handle file and directory paths |
| os | → | system info (CPU, memory, OS details) |
| url | → | parse and format URLs |
| events | → | handle custom events |
| crypto | → | hashing, encryption, security. |

### In short

| Component | Description | Function |
|---|---|---|
| V8 Engine | Google's JS Engine | Executes JS code. |
| Libuv | C library | Handles async I/O operation |
| Core Modules | Built-in Node.js libraries | Enable system and server tasks. |

# npm (Node Package Manager)

What is npm?

npm stands for Node Package Manager. It is default package manager for Node.js and is used to :

- Install, manage, and update libraries or dependencies.
- Share and reuse code packages across projects

Every MERN project uses npm to manage backend and frontend dependencies.

Why npm is Important in the MERN Stack

The MERN stack depends on several Node.js packages such as :

- express → Backend framework
- mongoose → MongoDB integration
- cors, dotenv, bcrypt, jsonwebtoken → Security & configuration
- nodemon → Development tool

All of these are installed and managed using npm.

How npm works

When you install Node.js, npm gets installed automatically.

It provides :

1. Command Line Tool (CLI) → For installing and manging packages.
2. Online Registry → Stores thousands of open-source packages.

Example :

```
npm install express
```

## npm Packages

A package (or module) is a collection of JavaScript files bundled together to perform specific functionality.

Examples :

| Package | Purpose |
|---|---|
| express | → backend framework for APIs |
| mongoose | → MongoDB connection and schema modeling |
| cors | → allows cross-origin requests (React → Node) |
| dotenv | → loads environment variables |
| bcrypt | → password hashing |
| jsonwebtoken | → user authentication |
| nodemon | → automatically restarts server when files change |

## Common npm Commands

| Command | Description |
|---|---|
| npm init | → Initializes a new Node.js project (creates package.json) |
| npm install <package> | → Installs a specific package locally |
| npm install -g <package> | → Installs a package globally |
| npm install | → Installs all dependencies listed in package.json |
| npm uninstall <package> | → Removes a package |

npm update <package> → updates a package to latest version
npm list → lists all installed packages
npm start → runs the start script from package.json
npm run <script> → runs a custom script defined in package.json

## package.json File

Every Node.js or MERN project includes a package.json file.

Purpose:
- Stores project info (name, version, description)
- Lists dependencies (required packages)
- contains scripts for automation

Example:

```
{
  "name": "mern-backend",
  "version": "1:0.0",
  "description": "Backend for MERN project",
  "main": "server.js",
  "scripts": {
      "start": "node server.js",
      "dev": "nodemon server.js"
  },
  "dependencies": {
      "express": "^4.18.2",
      "mongoose": "^7.0.3",
      "dotenv": "^16.0.0"
  }
}
```

## package-lock.json

- Automatically created when you install packages.
- Records exact versions of dependencies and sub-dependencies.
- ensures the same versions are installed when sharing the project.

## node_modules Folder

- stores all installed npm packages
- created automatically after running npm install.
- should not be uploaded to Github (it's large.) → instead, share package.json.

## Local vs Global Installation

| Type | Command Example | Description |
|---|---|---|
| Local | npm install express | Installs inside current project (recommended) |
| Global | npm install -g nodemon | Available system-wide (for tools like nodemon) |

## Scripts in npm

You can define custom scripts in package.json and run them using npm run.

Example:

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

## npm in the MERN Development workflow

| Stack Layer | Example npm Packages |
|---|---|
| Backend (Node + Express) | express, cors, dotenv, bcrypt, jsonwebtoken |
| Database (MongoDB) | mongoose |
| Frontend (React) | react, react-dom, axios |
| Development Tools | nodemon, concurrently |

## Example: Installing All Dependencies in a MERN Project

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

Then add this in package.json:

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

Now run the backend in development mode:

```
npm run dev
```

# Asynchronous Programming in Node.js

Node.js is asynchronous and non-blocking, which means it can handle multiple operations at the same time without waiting for one to finish before starting another.

- In synchronous programming, tasks run one after another – each must finish before the next starts.
- In asynchronous programming, tasks can start, continue, and complete independently.
- Node.js uses an event loop to manage async data / tasks, allowing it to handle many requests efficiently.

Example :

```
console.log ("Start");
setTimeout (() => {
    console.log ("Async Task Done."); }
2000);
console.log ("End");
```

Output :

```
Start
End
Async Task Done
```

"Async Task Done" runs later because setTimeout() is non-blocking.

## Event-Driven Programming

Node.js uses an EventEmitter model - it listens for events and triggers callbacks (listeners) when those events occur.

Example:

```
const EventEmitter = require('events');
const emitter = new EventEmitter();
emitter.on('start', () => {
    console.log('Started!');  });
emitter.emit('start');
```

used in: HTTP servers, streams, file handling, etc.

## Error Handling in Async Code

Always handle errors in asynchronous features using try/catch (for async/await) or .catch() (for promises)

## Example with Async/Await

```
async function processData() {
  try {
    let result = await fetchData();
    console.log(result);  }
  catch (error) {
    console.error("Something went wrong:", error);  }}
```

Example with Promise :

```
fetchData()
  .then(result => console.log(result))
  .catch(err => console.error(err));
```

## Summary Table

| Concept | Description | Example |
|---|---|---|
| Callback | Function passed to another function | fs.readFile('file.txt', cb) |
| Promise | Object representing eventual completion/failure. | .then()..catch() |
| Async/Await | syntactic sugar over Promises | await fetch() |
| Event-driven | Responds to emitted events | emitter.on() |
| Error handling | Use try/catch or .catch() | try {...} catch(e) {} |

## Working with Databases (MongoDB Integration)

Node.js applications often use MongoDB as a NoSQL database. The connection between Node.js and MongoDB is usually handled through Mongoose, an object Data Modeling (ODM) library.

## Connecting Node.js to MongoDB using Mongoose

Mongoose simplifies interaction with MongoDB by providing a scheme-based structure to your documents.

**Installation** npm install mongoose

**Connection setup**

```
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost:27017/myDatabase')
  .then(() => console.log("Connected to MongoDB"))
  .catch(err => console.log("Connection failed:", err));
```

**Note:**

- mongodb://localhost:27017 → Default MongoDB address
- myDatabase → Your database name
- Then .then() and .catch() handle success/failure asynchronously.

## Defining Schemas and Models

A Schema defines the structure of a document while a Model is used to interact with the database.

**Schema Example**

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

Creating a Model — `const User = mongoose.model('User', userSchema);`

Schema → Model → Collection → Document

Schema (structure) → Model (logic) → Collection (users) → Document (one user)

## Handling Async Queries (Async/Await)

Instead of using .then() and .catch(), you can use async/await for cleaner syntax.

Example

```
async function getUsers() {
  try {
    const users = await User.find();
    console.log(users);
  } catch (error) {
    console.log("Error fetching users:", error); }
}
getUsers();
```

## Validation and Error Handling

Mongoose provides built-in validation for data consistency:

Example

```
const productSchema = new mongoose.Schema({
  name: { type: String, required: [true, "Product name required"] },
  price: { type: Number, min: [0, "Price must be positive"]
```

```
}, category : { typ: String, enum : ["Food", "Clothing", "Electronics"] }
});
```

Error handling

```
async function createProduct() {
  try {
    const Product = mongoose.model('Product', productSchema);
    const item = new Product({ price: -20 });
    await item.save();
  } catch (err) {
    console.error("Validation Error:", err.message);
} }
createProduct();
```

## Mongoose Validation Features

- required, min, max, enum, match
- Custom validators using validate:
  { validator: fn, message: "..." }

Node.js ↔ Mongoose ↔ MongoDB

Client (React)
↓
Express.js Routes
↓
Mongoose Models
↓
MongoDB Collections

## Environment Variables

Environment variables are key-value pairs used to store configuration data outside the code.
They help you secure sensitive information like API keys, database URIs, and passwords.

Example:

```
PORT = 5000
MONGO_URI = mongodb://localhost:27017/myDB
JWT_SECRET = mySuperSecretKey
```

## Securing API Keys & Credentials

- Store keys, tokens, DB credentials inside .env files.
- Access using process.env.KEY_NAME.

Example

```
const apiKey =
    process.env.MY_API_KEY;
```

## API Testing and Debugging

Testing with Postman or Thunder Client

- Test API endpoints (GET, POST, PUT, DELETE)
- Check headers, params, and JSON responses

```
}, category : { typ: String, enum : ["Food", "Clothing", "Electronics"] }
});
```

Error handling

```
async function createProduct() {
    try {
        const Product = mongoose.model('Product', productSchema);
        const item = new Product ({ price : -20 });
        await item.save();
    } catch (err) {
        console.error("Validation Error :", err.message);
} }
createProduct();
```

## Mongoose Validation Features

- required, min, max, enum, match
- Custom validators using validate :
  { validator : fn, message : "..." }

Node.js ↔ Mongoose ↔ MongoDB

Client (React)
↓
Express.js Routes
↓
Mongoose Models
↓
MongoDB Collections

## Using console.log() and morgan

npm install morgan

```
const morgan = require('morgan');
app.use(morgan('dev'));
```

Shows request logs in the console — method, URL, status, and time

## Logging & Error Tracking

- Use console.error() for errors.
- In production, use logging libraries like winston or pino.

## Deployment

### Preparing for Production

- Remove console logs.
- Handle errors gracefully
- Use process.env.PORT.
- Use helmet for security headers.

### Using Environment Variables for Production

```
NODE_ENV = production
PORT = 8080
MONGO_URI = "mongodb+srv://cluster.mongodb.net/myDB"
```

# Project Structure in MERN

Understand proper folder organization :

```
/backend
  ├── server.js
  ├── config/
  ├── models/
  ├── routes/
  ├── controllers/
  ├── middleware/
  └── .env
```

- server.js → main entry point
- config/ → DB connection, environment setup
- models/ → Mongoose schemas
- routes/ → Express routes
- controllers/ → logic for each route
- middleware/ → Custom middleware functions

# REACT JS

## # Introduction to React JS :

React JS is a component-based JavaScript Library used to build dynamic and interactive user Interfaces. It was developed for the view layer of the application by Facebook. React allows developers to create reusable UI components that efficiently update and render based on changes in data providing a responsive and interactive user experience.

↳ uses a virtual DOM for faster updates
↳ supports a declarative approach to designing UI components
↳ ensure better application control with one-way data binding

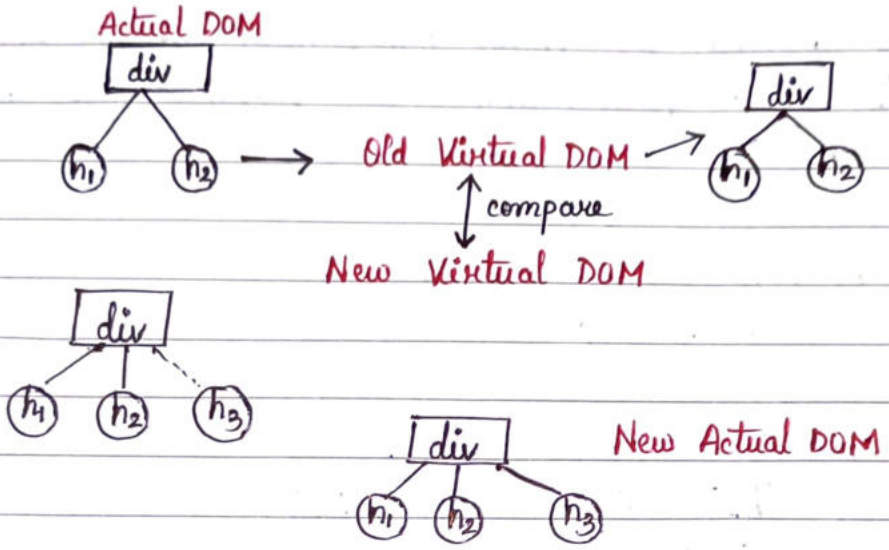
### Why React JS is developed :

React JS was developed by Facebook to simplify building dynamic user interfaces. Before React, updating the UI was complex and messy. React introduced the virtual DOM to efficiently update only the changed parts of a page, making web apps faster and easier to manage. It was created by Jordan walk and Released in 2013.

@ SYNTAX ERROR
— Abhishek Rathor

## ↳ How ReactJS works?

React operates by creating an in-memory virtual DOM rather than directly manipulating the browser's DOM. It performs necessary manipulations with the virtual representation before applying changes to the actual browser DOM i.e;




Browser DOM ↔ Virtual DOM

## ↳ Explanation:

(i) Actual DOM and Virtual DOM:

- Initially, there is an Actual DOM (Real DOM) containing a div with two children elements: $h_1$ and $h_2$
- React maintains a previous virtual DOM to track the UI state before any updates.

(ii) Detecting changes:

- When a change occurs (eg. adding a new $h_3$ elements), React generates a New Virtual DOM.

• React compares the previous virtual DOM with the New Virtual DOM using a process called reconciliation.

(iii) Efficient DOM update:

• React identifies the differences (in this case, the new h3 element)

• Instead of updating the entire DOM, React updates only the changed part in the New Actual DOM, making the update process more efficient.

→ "Hello World", Boiler Plate Code for React;

```
import React from 'react';
function App() {
    return ( <div>
                <h1> Hello, World </h1>
             </div>
           );
}
export default App;
```

* Note: we will use Vite + React app for creating React Apps i.e;

# Vite :

Vite is a build tool that provides a faster and more efficient development experience for modern web projects, and its works are seamlessly with React. It leaveroges native ES modules to enables near - instant server stand - up and fast hot module replacement (HMR)

→ To create React Project with Vite, run the following command in your terminal ;

(make sure that node is installed in the pc)

cmnds :

- npm create vite @ latest my. react - app - template react
- cd my - react - app
- npm install
- npm run dev.

→ Why use vite with React JS :

compared to older tools like Create React App (CRA), vite offer :

| Feature | Vite | CRA. |
|---|---|---|
| • Development start fine. | • Almost Instant | • Slower (due to Bundling) |
| • Hot Reload | • Fast & Effecient | • Slower & less reliable. |
| • Build Speed | • Optimized with Roll up | • webpack - based, slower. |

→ Folder structure overview for Vite + React App:

```
my-react-app/
├── public/          → static files
├── src/             → application source code → Its contains components
│   ├── assets/      → images and other static file
│   ├── app.css      → component-specific styles
│   ├── app.jsx      → main APP component
│   ├── index.css    → Global styles
│   └── main.jsx     → entry point of the app
├── .gitignore
├── index.html       → Root HTML files
├── package.json
├── vite.config.js   → Vite configuration
└── Readme.md
```

- For running the vite + React App we have to write the code;
  npm run dev.

# Quick Review on how to make Components:

In React, a component is a reusable and independent building block of the user interface components divide the UI into smaller, manageable pieces, making it easier to develop, maintain, and reuse code. They are similar to Java Script functions but work in isolation and return HTML via a render() function.

So basically, we have to create components of the code that is reusing in our website or repeating i.e;
Let's say the Navbar, Header, Footer, Image Slider etc. These parts are repeating in our website and are using repeatedly so we can using or writing code for multiple times we will create a compound of it or a separate component of it and then call it or use it in our main file by using import i.e.;

Let's say there is a component image slider that is in Image.JSX file, So we can import it as;

```
import image from "path-of-file";
—
<image>
—
```

# How React Render in the browser from index.html

Since, we know that React makes SPA (Single Page Application) i.e; it doesn't change or reload when changes occurs in the website that provides better UI experience for the user.

- So, how the whole React App Render with the single index.html file Root div?

This is all done by the JavaScript i.e;
The root that is present in the index.html

```
<div id = "root" > </div>
```

This renders all the data dynamically using JavaScript. Since the index.html file becomes an entry point for all the dynamic data and that data is entered into root element with the help of JavaScript.

- From where the root gets the dynamic data?

Every gets the dynamic data through main.jsx file tell React where and what to render inside the browser.

- Virtual DOM of React Renders to Real DOM

React builds a virtual DOM tree based on your JSX. and React efficiently updates the real DOM inside the root div using different algorithms when state or prop changes.

# JSX

What is JSX?

JSX is JavaScript Syntax Extention which is basically used to insert JavaScript with HTML in react components.

JSX looks like HTML, But its not.

- const element = <h1> Hello World </h1>

In JSX, we use:

- className instead of Class
- Returns only one parent element
- all tags must be closed

For writting JavaScript in JSX.

- we have use {} (curly Braces)
- Inside this, all the code is will JS or dynamic code.

## * Two Types of Components in React

- Functional components (Modern & Preffered)
- Class based components (Older & use void)

### Functional components

Functional components are JavaScript functions that return JSX (HTML-like code).

They are stateless (do not manage state directly) in older React versions but with React Hooks, they can handle state and lifecycle too.

Syntax:

```
function Welcome (props) {
    return <h1> Hello, {props.name}! </h1>;
}
```

Or using ES6 arrow function:

```
const Welcome = (props) => {
    return <h1> Hello, {props.name}! </h1>;
};
```

Example with Hooks:

```
import React, {~~useState~~ useState} from 'react';
```

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

Key Features :

- Simple and readable
- Uses React Hooks (useState, useEffect, etc.)
- Easier testing and debugging
- Recommended for modern React development

## Class-based Components

Class components are ES6 classes that ~~attend~~ extend React.Component They can use state and lifecycle methods directly.

Syntax :
```
import React, { Component } from "react";
class Welcome extends Component {
    render () {
        return <h1> Hello, {this.props.name}! </h1> ;
} }
```

Example with State:

```
import React, { Component } from "react";
class Counter extends Component {
    constructor() {
        super();
        this.state = { count: 0 }; // initialize state
    }
    increment = () => {
        this.setState({ count :
    this.state.count + 1 }); // update state
};
    render() {
      return ( <div>
                <h2> Count: { this.state.count} </h2>
                <button On Click = { this.increment } >
    Increase </button>
                </div>
            );
        }
    }
```

Key Features:

- uses this.state and this.setState() for state.
- Has lifecycle methods (componentDidMount, componentDidUpdate, etc.)
- More verbose than functional components.
- used before Hooks were introduced.

## * Including Tailwind CSS with React

Tailwind CSS is a utility-first CSS framework for rapidly building modern websites and applications. Instead of writing custom CSS, you use pre-defined utility classes directly in HTML or JSX.

Since, There are some predefined classes like;

bg-blue-500, Font-bold etc.

### How to use Tailwind

We can use Tailwind either with CDN link or installing it i.e;

using Tailwind with Vite + React

1: Go to Tailwind CSS website;
(tailwind css.com)

2: Then go to DOCS and select Vite
(or using Vite)

3: Copy the CMDs written there.
npm install tailwindcss @tailwindcss/vite.

4: Copy the vite.config.js file and CSS like copy the vite content and paste to vite.config.js file and CSS cmds for our global CSS file.
(make sure that it is properly installed)

## * Starting with React

Making our first React website

* Since for this:
  - We will make a basic website
  - We will use components like header, footer, main etc.
  - We will use react-router-dom

So, for making components i.e.;

In the Src, make a folder to separate components i.e.; src → components

Inside this we will make all the components like Header.jsx, Footer.jsx etc and using the react router-dom we will ~~input~~ import these components to main file that is App.jsx.

To import components in the App.jsx file i.e.;

First import them as:

- import Header from 'Path-of-Component';
- and under the return;

  <Header/>

It will automatically Render all the components accordingly.

# Note: The codes can be imported from App.jsx file.

## * Importing static data likes images to website ;

For importing static images to website make sure that this image is present inside public / images folder and you can make images Folder accordingly or use it to render images ;

It will use ; public / images i.e ;
Now for importing static images i.e. ;

- First import the image ;
  import logo from "path-of-images" ;

- Then use <img/> tag i.e.

* If your are using import make sure that the images or static files are in assets / folder
  or if using public directory then use <img> tag or path for it.

## Props :

Since Props are properties in React are a way to pass data from and component to another i.e; typically from a parent component to a child component.

↳ What is Props ?

Props are react only JavaScript objects that store the value of attributes passed to a React component. They help make components reusable and dynamic.

↳ How props works ?

Eg.

```
// Parent component
function App() {
    return <Greeting name = "Apple" />;
}
// Child component
function Greeting (props) {
    return <h1> Hello, { props.name } </h1>;
}
```

Output : Hello, Apple.

Since, we can use props for differents components and files for putting dynamic data to it.

# Let's take another Eg. of Props as :

```
<Main SearchBar = "True" />
```

(In App.Jsx file)

{ props.searchBar ? <h1> Hello Search </h1> :
" No Search Bar " }

In this, there is prop that is searchBar = " True "; if this searchBar = True, write Hello Search or if searchBar = False, write No Search Bar, so this is how we can use props for a condition.

* Default Props: are used to specify default values for a components props in case no value is provided by the parent component.

Eg:
```
function Greeting (props) {
    return <h1> Hello, { props.name } </h1>;
}
Greeting.defaultProps = {
    name = " Guest ",
};
```

Key Points:

| Feature | Description |
|---|---|
| Direction | Data flows one way (parent → child) |
| Read-only | Props cannot be changed by the child component |
| Purpose | used to customize components & pass information |
| Type | can be any data type — string, number, object, array, function etc. |

Example with Multiple Props

```
function UserCard (props) {
    return (
        <div>
            <h2> { props.username } </h2>
            <p> Age: { props.age } </p>
        </div>
    );
}
function App() {
    return < UserCard username = " Abhishek "
age = {20} />;
}
```

@ SYNTAX ERROR
— Abhishek Rathore

* Hooks

Hooks are special functions introduced in React 16.8 that let you use state, lifecycle methods, and other React features in functional components without writing a class.

↳ Commonly used Hooks;

- useState () : Add state to a functional component
- useEffect () : run side effects (eg: fetching data etc)
- useContext () : access value from ReactJS context API
- useRef () : creates a reference to a DOM
- useCallback(): memoizes a callback function
- useReducer (): manages complex state logic (like Redux but simpler)

## Before Hooks

Earlier, only class components could use features like:

- State (this.state)
- Lifecycle methods (componentDidMount, componentDidUpdate, etc.

Functional components were stateless and simple

## After Hooks

Now, with Hooks, functional components can:

Store data using state

→ React to changes using effects

→ Use context, refs, reducers, and much more.

Example: useState

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

    return (
        <div>
            <p> You clicked {count} times </p>
            <button onClick = {() => setCount (count + 1)}>
        Click Me </button>
            </div>
    );
}
```

Example : useEffect

```
import { useEffect, useState } from "react";
function Timer() {
    const [seconds, setSeconds] = useState(0);
    useEffect(() => {
        const interval = setInterval(() => {
            setSeconds(s => s+1);
        }, 1000);
        return () =>
    clearInterval(interval); // cleanup
}, []);
    return <p> Time: {seconds}s </p>;
}
```

## Why Hooks Are Useful

Simplify code – no need for classes
Reuse logic across components
Easier to read and test
Make functional components powerful

## * Connecting React with Node.js

1. Set up the Backend (Node.js + Express)

Folder: / backend

```
mkdir backend
cd backend
npm init -y
npm install express cors mongoose dotenv
```

server.js

```
const express = require("express");
const cors = require("cors");
const app = express();
app.use(cors());   // allow React to access backend

app.use(express.json());   // parse JSON data

~~app.use(express.~~

// Simple test route
app.get("/api/message", (req, res) => {
    res.json({ message: "Hello from Node.js backend!" });
});
app.listen(5000, () =>
  console.log("Server running on port 5000"));
```

Run the backend : node server.js

2. Set up the Frontend (React App)

Folder :/ frontend

```
npx - create-app - app frontend
cd frontend
npm start
```

3. Fetch Data from Backend in React

```
import React { useEffect, useState }
from "react";
import axios from "axios" ; // install with : npm install
                              axios
function App() {
      const [message, setMessage] = useState("");
      useEffect(() => {
axios.get("http://localhost:5000/api/message")
      .then(res => setMessage(res.data.message))
      .catch(err => console.error(err)); }, []);
return (
      <div>
        <h1> React + Node.js Connection </h1>
        <p> {message} </p>
      </div>
    );
}
export default App;
```

Output :

React + Node.js Connection
Hello from Node.js backend!

## 4. Connecting with MongoDB (Full MERN Flow)

Once React ↔ Node is connected, add MongoDB:

```
const mongoose = require("mongoose");
mongoose.connect("mongodb://127.0.0.1:27017/mernapp");
```

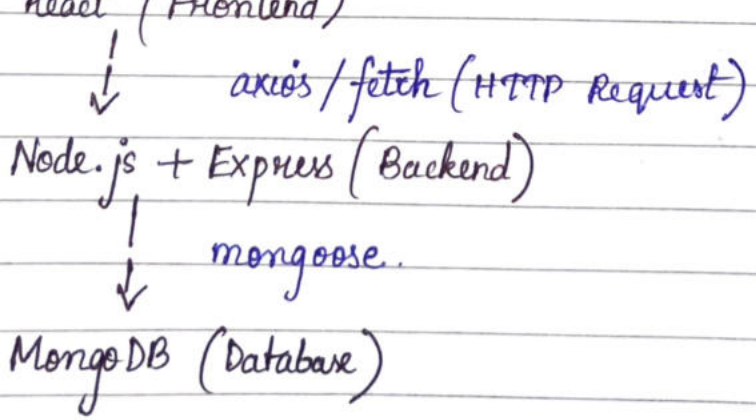
Then you can create routes like:

```
app.get("/api/users", async (req, res) => {
    const users = await User.find();
    res.json(users);
});
```

And in React:

```
axios.get("http://localhost:5000/api/users").then(res =>
    setUsers(res.data));
```

### Summary Diagram



# Authentication Flow in MERN

Here's the general flow :

1. User signs up/logs in via the React frontend.
2. React sends the credentials to the Express/Node.js backend via an API request.
3. Express + Node.js verify credentials with the MongoDB database.
4. If valid, the server generates a JWT token. (JSON web Token)
5. The token is sent back to React and stored in localStorage or cookies.
6. For future requests, React includes this token in the Authorization header.
7. The server verifies the token before allowing access to protected routes

# Deployment

Frontend → Deploy React on Vercel/Netlify

Backend → Deploy Node.js on Render/Railway/Heroku

Database → Use MongoDB Atlas (cloud)

# Tools & Libraries

| Tool | Use |
| --- | --- |
| Axios / Fetch | API calls |
| Mongoose | MongoDB modeling |
| React Router DOM | Navigation |
| JWT / bcrypt | Auth & password hashing |
| Dotenv | Environment variables |
| Cors | Cross-origin requests |

# MERN Project Flow

1. Design UI in React
2. Create Express API routes
3. Connect API to MongoDB
4. Integrate React with API using Axios
5. Add Authentication & Validation
6. Deploy Frontend + Backend