Java Script

Node JS

- Asynchronous event-driven JavaScript runtime.

Node JS installation

- Visit: https://nodejs.org/en/download
- Download: Windows Installer (.msi)
- Install as per instructions (by installer)
- Verify:

```
node -v npm -v
```

Hello World

- Create new JS file e.g. demo01.js
- Write JS code. (Built-in objects available: "console").

```
console.log("Hello, World!!");
```

- To execute the program, open new terminal.

```
node demo01.js
```

Java Script

- Invented by Brendan Eich in 1995 at Netscape Corporation for Netscape2
- It is a Scripting language for web development

- It is an interpreted langugage
- Used to develop server side programs (Node) as well
- It is an Object Oriented Programming Language
- Use script tag to write JS code in html page
- Loosely typed language
- It is used to dynamically modify the html pages.It has full integration with HTML/CSS
- All major browsers support and by default enabled for javascript
- ECMAScript is the official name of the language.
- ECMAScript versions have been abbreviated to ES1, ES2, ES3, ES5, and ES6.
- Since 2016, versions are named by year (ECMAScript 2016, 2017, 2018, 2019, 2020).

Built-in Objects

1. console
    - represents the web console (terminal)
    - use log method to write output on console

Variables

- It is a container to store the data
- To declare a variable data type MUST NOT be used in its declaration
- To create variables in JS we can use

1. var
    - It is a normal variable whose value can be changed multiple times
    - var is not used as it is depricated, it is recommend to use let instead
2. let
    - It is a normal variable whose value can be changed multiple times
3. const
    - It is a variable whose value cannot be changed once initialized.

- modern JS prefer "let" & "const" over "var" keyword.
- Syntax: let name = initial value; const name = initial value; E.g.

```
let num = 100; // number
const salary = 4.5; // number
let = "test"; // string
const firstName = 'steve'; // string
let canVote = true; // Boolean
```

Variable Scope

1. global
    - a variable declared outside any function
    - can be declared with or without var keyword
    - can be accessed outside or inside any function
    - E.g.

```
num = 100;
var salary = 4.5;
```

    - can be declared inside a function without using a var keyword
    - E.g.

```
function function1() {
  // global
  firstName = "test";
}
```

2. Local
    - Must be declared inside a function with keyword var

- Can NOT be accessed outside the function in which it is declared
- E.g.

```
function function1() {
  // local
  var firstName = "test";
}
```

3. Block

- Declared with "const" or "let" keyword.
- Can NOT be accessed outside the block in which it is declared
- E.g.

```
function function1() {
  if(1 == 1) {
    // block
    const firstName = "test";
  }
  console.log(firstName); // error - not accessible outside the block
}
```

Data Types

- In JS, all Data Types are inferred (automatically decided by JS .....)
- typeof operator can be used to get the type of the variable.

1. number:
   - It supports both whole and decimal numbers
   - E.g.
     - num = 100;

        - salary = 4.5;
2. string: collection of characters E.g. - firstName = "steve"; - lastName = 'Jobs';
3. boolean:
    - may have only true or false value
    - E.g.
        - canVote = true;
        - canVote = false;
4. undefined:
5. object:
    - Represents JS object

Built-in Values

1. NaN
    - Not a Number
    - Is of type "number"
    - E.g. console.log(parseInt("test"));
2. Infinity:
    - When a number is divided by 0
    - E.g. answer = 10 / 0; // Infinity
3. undefined:
    - No value/type present
4. null:
    - No object.
    - is of type "object"

Strict Mode

- "use strict"; directive defines that JavaScript code should be executed in "strict mode".
- The "use strict" directive was new in ECMAScript version 5.
- With strict mode good practices are enforced e.g. use of undeclared/implicit variables raise error.

- You can use strict mode in all your programs. It helps you to write cleaner code, like preventing you from using undeclared variables. Supported in all modern browsers. Not supported in IE-9 and earlier.

Templated Strings

- strings can be enclosed in '..', "..", or `...`
- templated strings to print var values: `var=${varname}`

```
console.log(`addition: ${n1} + ${n2} = ${res}`);
```

JS Operators

- Basic operators are same as C/Java.
- == operator compares the contents (but not the data type).
- === operator compares the contents as well as data types.
- Example:

```
console.log(`123 == "123" : `, 123 == "123"); // true
console.log(`123 === "123" : `, 123 === "123"); // false
```

- similarly != compare contents, while !== compraes contents + data types. = Logical && returns first falsy value (this may be false, 0, empty string, or null)
- Logical || returns first truthy value (this may be true, non-zero number, non-empty string, or non-null object)
- falsy values: 0 (number), false (boolean), null (object), '' (string), undefined
- Example:

```
console.log(`0 && 12 : `, 0 && 12); // first false value
console.log(`0 || 12 : `, 0 || 12); // first true value
```

JS Functions

- function is a reusable block of statement - defined once & can be called multiple times.

Function types

- function may or may not have arguments (input to fn).

```
// function with no arguments
function printline() {
  console.log("----------------------------------------");
}
```

```
// function with arguments
function multiply(a, b) {
  const c = a * b;
  console.log("multiply result = " + c);
}
```

- function may or may not return result (output from fn).

```
// function returning result
function subtract(a, b) {
  const c = a - b;
  return c;
}

// function call
let res = subtract(22, 7);
```

Function argument types

- function argument types are inferred. fn can be called multiple times with different types of args.

```
// function arguments are not type-safe
function add(x, y) {
  const z = x + y;
  console.log(`${x} + ${y} = ${z} of type ${typeof z}`);
}
// function can be called with different arg types
add(22, 7);
add("Hello", "World");
add(10, true);
add("Bond", 7);
add("Flag", false);
// passing more arguments will ignore remaining args.
add(10, 20, 30);
// passing less arguments will consider remaining args undefined
add(10);
```

- if multiple fns defined with same name, last fn definition override/replace the earlier ones.

```
function f1() {
  console.log("f1() called.");
}
function f1(x, y) {
  console.log("f1(x, y) called: " + x + ", " + y);
}
function f1(x) {
  console.log("f1(x) called: " + x);
}
f1(10, 20); // calls last f1(x).
```

Built-in Functions

- JS have built-in functions as well.
- e.g. parseInt(), parseFloat(), eval()

```
// string to number (int)
let n1 = parseInt("123");
console.log("n1 = " + n1); // 123
let n2 = parseInt("123.45");
console.log("n2 = " + n2); // 123
let n3 = parseInt("Bond007");
console.log("n3 = " + n3); // NaN
let n4 = parseInt("007Bond");
console.log("n4 = " + n4); // 7
// string to number (float)
let v1 = parseFloat("3.14");
console.log("v1 = " + v1); // 3.14
let v2 = parseFloat("123");
console.log("v2 = " + v2); // 123
// eval() to evaluate string expr
let v3 = eval("2 + 3 * 4");
console.log("v3 = " + v3); // v3
```

Function Alias

- In JS, function can be treated as object. It can be assigned to another reference a.k.a. function alias. Using alias the actual function can be called.
- Function alias concept can be used to pass fn as argument to fn, Return fn from another fn, Create anonymous fns and Lambda expressions.
- Example:

```
// function definition
function add(x, y) {
  let z = x + y;
  console.log(`add(): ${x} + ${y} = ${z}`);
}
console.log("add() -- type = " + typeof add + ", name = " + add.name);
add(22, 7);

// function alias
const calc = add;
console.log("calc -- type = " + typeof calc + ", name = " + calc.name);
calc(23, 8);
```

- Anonymous functions are created using function keyword, but not given any name. Anonymous fns assigned to alias and called via aliases only.
- Example:

```
// anonymous function
const subtract = function (x, y) {
  let z = x - y;
  console.log(`subtract(): ${x} - ${y} = ${z}`);
};
subtract(22, 7);
```

- Lambda fns are Shorthand anonymous fns. They can be one-liner or multiline. However, one-liners are preferred. Lambda functions introduced in ES6.
- Example:

```
const multiply = (x, y) => {
  let z = x * y;
  console.log(`multiply(): ${x} * ${y} = ${z}`);
```

```
};
multiply(22, 7);

const divide = (x, y) => x / y;
const res = divide(22, 7);
console.log(`divide(): 22 / 7 = ${res}`);
```

- A closure in JS is the combination of a function and the lexical environment within which that function was declared. This environment consists of any variables that were in scope at the time the function was created. Closures allow a function to access variables from its outer scope even after that scope has finished executing.

```
// function closure
function outer() {
  let z = 1;
  // inner/nested fn -- accessing local var of outer fn
  function inner(x, y) {
    return x + y + z;
  }
  return inner;
}
```

```
// call outer fn & collect its return value
const infn = outer();
// call inner fn
const result = infn(22, 7);
console.log("final closure result = " + result);
```

OOP fundamentals

- In JS, an object is a complex data type that allows you to store collections of key-value pairs.

- Objects are dynamic, meaning you can add, modify, or delete properties after they are created.
- Everything in JS is an Object, even functions are also objects.
- Method1: Create object using new Object(). Properties can be added dynamically or accessed using dot operator.
- Example:

```
// create object & props in JS -- (using .)
let p = new Object();
p.name = "James Bond";
p.age = 65;
p.addr = "London";
console.log("type of p = " + typeof p);
console.log("p name = ", p.name);
console.log("p age = ", p.age);
console.log("p addr = ", p.addr);
```

- Method2: Create object using new Object(). Properties can be added dynamically or accessed using [] subscript.

```
// create object & props in JS -- (using [])
let m = new Object();
m["model"] = "iPhone 14 Pro";
m["company"] = "Apple";
m.price = 89000.0;
console.log("type of m = " + typeof m);
console.log("m model = ", m["model"]);
console.log("m company = ", m.company);
console.log("m price = ", m["price"]);
```

- Method3: Create object using JSON syntax. Properties can be accessed using dot operator or [] subscript.

```
// create object & props in JS -- (using {})
let i = {
  name: "Pencil",
  company: "Natraj",
  price: 5.0,
};
console.log("type of i = " + typeof i);
console.log("i name = ", i.name);
console.log("i company = ", m.company);
console.log("i price = ", i["price"]);
```

- Method4: Create object using constructor function i.e. new FnName(args). Properties are added into fn using "this" pointer (which refers to newly created object). To add common properties & methods for all objects use prototype (next demo).

```
// create object & props in JS -- (using constructor function)
function Car(model, company, price) {
  this.model = model;
  this.company = company;
  this.price = price;
}

let c = new Car("i10", "Hundai", 800000.0);
console.log("type of c = " + typeof c);
console.log("c model = ", c.model);
console.log("c company = ", c.company);
console.log("c price = ", c.price);
```

- To add common properties & methods for all objects use prototype.
- Example:

```

```

```
// create object & props in JS (using constructor function)
function Car(model, company, price) {
  this.model = model;
  this.company = company;
  this.price = price;
}
```

```
// add properties and methods for all objects using "prototype".
Car.prototype.color = "red";
Car.prototype.printInfo = function () {
  console.log("car model = ", this.model);
  console.log("car company = ", this.company);
  console.log("car price = ", this.price);
  console.log("car color = ", this.color);
};
```

```
// Create Car object
let c1 = new Car("i10", "Hundai", 800000.0);
c1.printInfo();
// Create another Car object
let c2 = new Car("Punch", "Tata", 900000.0);
c2.printInfo();
```

- JS ES6 (2015) introduced classes. Classes are templates for JS objects.
- The "constructor" method is a special method. It is executed automatically when a new object is created, to initialize object properties.
- The class methods are declared without "function" keyword and it receives implicit "this" argument.
- Class must be declared before its use.

- Example:

```
// ES6 - JS class
class Person {
  constructor(name, age, addr) {
    this.name = name;
    this.age = age;
    this.addr = addr;
  }
  printInfo() {
    console.log("person name = ", this.name);
    console.log("person age = ", this.age);
    console.log("person addr = ", this.addr);
  }
  toString() {
    return `Person: name=${this.name}, age=${this.age}, addr=${this.addr}`;
  }
}
```

```
// create objects
let p1 = new Person("James Bond", 65, "London");
console.log("p1 type = ", typeof p1); // object type
p1.printInfo();
```

```
let p2 = new Person("Superman", 60, "Crypton");
console.log("p2 type = ", typeof p2); // object type
p2.printInfo();
```

```
// call to string
console.log("p1 => " + p1.toString());
console.log("p2 => " + p2); // toString() method is implicitly called whenever object tried to convert into string.
```

- JS class can be inherited from another class.

```
// JS class inheritance
class Employee extends Person {
  constructor(name, age, addr, sal, dept) {
    super(name, age, addr);
    this.sal = sal;
    this.dept = dept;
  }
  showInfo() {
    super.printInfo()
    console.log("emp sal = ", this.sal);
    console.log("emp dept = ", this.dept);
  }
  toString() {
    return `Emp: [${super.toString()}] sal=${this.sal}, dept=${this.dept}`;
  }
}
```

JS Array

- In JS, arrays are objects. Array object have properties (e.g. length) and methods (e.g. push(), pop(), etc).
- Array is collection of elements. Array indices are in range 0 to n-1. Each element accessed using subscript.
- All array elements can be of same type (common use case) or of mixed types.
- Example 1:

```
function numberArray() {
  let nums = [11, 22, 33, 44, 55];
  console.log("nums type = " + typeof nums); // object
  console.log("nums = " + nums);
  console.log("nums length = " + nums.length);
  // traverse array
  for (let i = 0; i < nums.length; i++) {
    console.log("nums[" + i + "] = " + nums[i]);
  }
}

numberArray()
```

- Example 2:

```
function stringArray() {
  let names = ["Nilesh", "Rohan", "Rajiv", "Nitin"];
  console.log("names type = " + typeof names); // object
  console.log("names = " + names);
  console.log("names length = " + names.length);
  // traverse array
  for (let i = 0; i < names.length; i++) {
    console.log("names[" + i + "] = " + names[i]);
  }
}

stringArray();
```

- Example 3:

```
function mixedArray() {
  let mixed = [1, 3.142, "Nilesh", true, null];
  console.log("mixed type = " + typeof mixed); // object
  console.log("mixed = " + mixed);
  console.log("mixed length = " + mixed.length);
  // traverse array
  for (let i = 0; i < mixed.length; i++) {
    console.log("mixed[" + i + "] = " + mixed[i] + " of type " + typeof mixed[i]);
  }
}

mixedArray();
```

- for-of loop is introduced in ES6 (2015) to iterate through any iterable.
- Example 4:

```
function traverseArrayUsingForOfLoop() {
  let arr = [1.1, 2.2, 3.3, 4.4, 5.5];
  for (let ele of arr) {
    console.log("arr element: ", ele);
  }
}
traverseArrayUsingForOfLoop();
```

- Each array element can be object. Each object can be created with different method and/or of different type. Each element (object) accessed using arr[i].

```
// create array - Person objects
let people = [
  new Person("James Bond", 65, "London"),
  new Person("Superman", 70, "Crypton"),
```

```
  new Person("Batman", 50, "Gothom")
];
for (let i = 0; i < people.length; i++) {
  console.log(people[i].toString());
  //console.log(people[i].name + " | " + people[i].age + " | " + people[i].addr);
}
```

JS Default arguments

- By default, if any arg is not passed, it is undefined. Instead default value can be provided a.k.a. "default parameters". Arguments are assigned to params from left to right (irrespective of default values given to params).
- Example:

```
// function definition
function add(x = 0, y, z = 0) {
  let r = x + y + z;
  console.log(`add(): ${x} + ${y} + ${z} = ${r}`);
}
// function calls
add();
add(10);
add(22, 7);
add(11, 22, 33);
```

Node JS command line arguments

- Command line args is additional information passed to the program while running it from command line.
- C program - command line args

```
> hello.exe arg1 arg2 arg3
```

```
int main(int argc, char *argv[]) {
  // ... argv[i]
}
```

- Java program - command line args

```
> java Main arg1 arg2 arg3
```

```
class Main {
  public static void main(String[] args) {
    // args[i]
  }
}
```

- Node JS program - command line args

```
> node program.js arg1 arg2 arg3
```

```
//process.argv -- array of strings passed on command line.
//  [0] -- path of node.exe
//  [1] -- path of js program file
//  [2] -- first arg
//  [3] -- second arg -- so on
for(let i=0; i < process.argv.length; i++)
  console.log(process.argv[i]);
```

```
console.log("arg1 : ", process.argv[2]);
console.log("arg2 : ", process.argv[3]);
```

JS Hoisting

- In C/C++, variables or functions or classes must be declared before their use.
- Hoisting in JavaScript is a behavior where variable and function declarations are moved to the top of their scope before code execution. It allows using variables and functions before they are declared in the code. However, only declarations are hoisted, not initializations or assignments.
- **Variable Hoisting**
  - Variables declared with var are hoisted to the top of their scope and initialized with undefined.

```
console.log(x); // Output: undefined
var x = 5;

console.log(y); // Output: ReferenceError: Cannot access 'y' before initialization
let y = 10;
```

- **Function Hoisting**
  - Function declarations are fully hoisted, allowing them to be called before they are defined in the code.

```
greet("Sunbeam"); // Output: Hello, Sunbeam!

function greet(name) {
  console.log("Hello, " + name + "!");
}
```

- **Class Hoisting**
  - Classes are also hoisted, but unlike function declarations, they are not initialized. Therefore, you cannot use a class before it is declared, similar to let and const.

```
const myClass = new MyClass(); // Output: ReferenceError: Cannot access 'MyClass' before initialization

class MyClass {
  constructor() {
    console.log("MyClass constructor");
  }
}
```

JS Array (of objects)

- Array of objects can be created using JSON syntax.

```
// Array of Objects -- using JSON syntax
// create array
let mobiles = [
  {
    model: "iPhone 14 Pro",
    company: "Apple",
    price: 89000.0,
  },
  {
    model: "Google Pixel 9",
    company: "Google",
    price: 92000.0,
  },
  {
    model: "Galaxy S25",
    company: "Samsung",
```

```
    price: 120000.0,
  },
];
for (let i = 0; i < mobiles.length; i++) {
  console.log(mobiles[i].model + " | " + mobiles[i].company + " | " + mobiles[i].price);
}
```

JSON

- JSON (JavaScript Object Notation) is a lightweight data format used for storing and exchanging data between a server and a web application. It is easy to read and write for humans and machines.
- JSON is key-value format. Key is always string and value can be of any type (including array).

```
let person = {
  name: "Nilesh Ghule",
  gender: "M",
  age: 42,
  canVote: this.age >= 18,
  addr: {
    city: "Pune",
    pin: 411046,
    country: "India",
  },
  hobbies: ["Teaching", "Programming", "Cooking"],
};
// print object properties
console.log("person info: ");
console.log("name : " + person.name);
console.log("age : " + person.age);
console.log("addr : " + person.addr.city + ", " + person.addr.country);
console.log("canVote : " + person.canVote);
console.log("hobbies : " + person.hobbies);
console.log("person info: print using console.log(): ");
```

```
console.log(person);
// console.dir()
console.log("person info: pretty print using console.dir(): ");
console.dir(person);
// stringify JSON
console.log("person info: stringify JSON: ");
console.log(JSON.stringify(person));
// parse JSON
let json = '{"name": "James Bond", "age": 65, "addr": "London"}';
let obj = JSON.parse(json);
console.log("print obj: ", obj);
// Note: console.log() can take variable number of args.
```

JS destructuring

What is Destructuring?

- Destructuring is a **shortcut syntax** to **unpack values** from:
  1. **Arrays** → Extract elements by position
  2. **Objects** → Extract properties by name

Array Destructuring

```
const fruits = ["🍏", "🍌", "🍊"];

// Old way
const apple = fruits[0];
const banana = fruits[1];

// Destructuring
const [apple, banana, orange] = fruits;
console.log(apple); // "🍏"
```

Skipping Elements

```
const [first, , third] = ["A", "B", "C"];
console.log(third); // "C"
```

Default Values

```
const [a = 1, b = 2] = [10];
console.log(a, b); // 10, 2 (fallback for missing values)
```

Rest Pattern (...)

```
const [first, ...rest] = [1, 2, 3];
console.log(rest); // [2, 3]
```

Object Destructuring

```
const user = { name: "Nilesh", age: 25 };

// Old way
const name = user.name;
const age = user.age;

// Destructuring
const { name, age } = user;
console.log(name); // "Nilesh"
```

Renaming Variables

```
const { name: userName, age: userAge } = user;
console.log(userName); // "Nilesh"
```

Default Values

```
const { role = "Guest" } = user;
console.log(role); // "Guest" (if missing in object)
```

Practical Use Cases

1. **Function Parameters** (Cleaner API design)

```
function greet({ name, age }) {
  console.log(`Hello ${name}, you're ${age} years old!`);
}

greet(user); // "Hello Nilesh, you're 25 years old!"
```

2. **Swapping Variables**

```
let a = 1,
  b = 2;
[a, b] = [b, a]; // Swap without temp variable
```

3. **React Props Destructuring**

```
const UserCard = ({ name, email }) => (
  <div>
    {name} - {email}
  </div>
);
```

Function with Variable Args

- Fn args can be accessed within fn using "arguments" keyword. "arguments" object is like an array. Each argument can be accessed like arguments[i].
- Example:

```
function sum() {
  console.log("sum called with args: ", arguments);
  let args = [...arguments];
  console.log("all args : " + args);
  let total = 0;
  for (let i = 0; i < arguments.length; i++) {
    total = total + arguments[i];
  }
  return total;
}
```

```
// call fn with variable args
let res1 = sum(10, 20);
console.log("sum result : " + res1);
let res2 = sum(11, 22, 33, 44);
console.log("sum result : " + res2);
```

Spread operator

- ... is unpacking/spread operator that separates all elems of array or json object.
- Spread operator to merge two arrays (in a new array):

```
function combineArray() {
  let arr1 = ["Mango", "Melon", "Orange"];
  // add all arr1 elements at the end of arr2
  let arr2 = ["Lemon", "Grape", "Jackfruit", ...arr1];
  console.log("arr2 : " + arr2);
  // add all arr1 elements in between arr3
  let arr3 = ["Lemon", ...arr1, "Grape", "Jackfruit"];
  console.log("arr3 : " + arr3);
}

combineArray();
```

- Spread operator to append element to the array (and yield a new array).

```
function appendArray() {
  const oldArr = [11, 22, 33, 44];
  const newEle = 55;
  const newArr = [...oldArr, newEle];
  console.log("oldArr :" + oldArr);
  console.log("newArr :" + newArr);
}

appendArray()
```

- Spread operator to update object properties (in a new object):

```
function changeObject() {
  const p1 = {
    firstName: "James",
    age: 42,
  };
  console.log("p1 : ", p1);
  // add a lastName field into p1
  const p2 = {
    ...p1,
    lastName: "Bond",
  };
  console.log("p2 : ", p2);
  // change age field in p1
  const p3 = {
    ...p1,
    age: 45,
  };
  console.log("p3 : ", p3);
}
changeObject();
```

JS Array Operations

- push() - to push elements at the end.

```
function addItems() {
  let names = ["Nilesh", "Rajiv", "Rohan"];
  console.log("names : " + names);
  names.push("Nitin");
  names.push("Prashant");
  console.log("names (after pushing 2 items) : " + names);
}
addItems();
```

- splice(index, 0, newelems) - to insert elements at given index.

```
function addItemsAtPos() {
  let names = ["Nilesh", "Rajiv", "Rohan"];
  console.log("names : " + names);
  names.splice(1, 0, "Sarang", "Vishal");
  // from 1st index, delete 0 elements, and insert "Sarang", "Vishal"
  console.log("names (after splicing/insert 2 items at index 1) : " + names);
}
addItemsAtPos();
```

- pop() - to delete the last element.

```
function removeItems() {
  let names = ["Nilesh", "Rajiv", "Rohan", "Nitin", "Prashant", "Sarang", "Vishal"];
  console.log("names : " + names);
  let n1 = names.pop();
  console.log("popped item : " + n1);
  let n2 = names.pop();
  console.log("popped item : " + n2);
  console.log("names (after popping 2 items) : " + names);
}
removeItems();
```

- splice(index, delcount) - to delete num of elements from given index.

```
function removeItems() {
  let names = ["Nilesh", "Rajiv", "Rohan", "Nitin", "Prashant"];
  console.log("names : " + names);
```

```
  let n3 = names.splice(2, 1);
  // if delete count (arg2) missing all elements after that pos are deleted
  console.log("spliced/deleted item (at 2nd pos) : " + n3);
  console.log("names (after splice/delete 2nd pos) : " + names);
}

removeItems()
```

- at() - to access element at given index like []. at() can be used with -ve index to access elements from last e.g. at(-1) is last elem, at(-2) is second last elem, ...

```
function accessElements() {
  let names = ["Nilesh", "Rajiv", "Rohan", "Nitin", "Prashant"];
  console.log("names[3] = ", names[3]);
  console.log("names.at(3) = ", names.at(3));
  console.log("names[-1] = ", names[-1]); // undefined
  console.log("names.at(-1) = ", names.at(-1)); // access last element
  // forward traversal
  for (let i = 0; i < names.length; i++) {
    console.log("fwd : " + names[i]);
  }
  // reverse traversal
  for (let i = 0; i < names.length; i++) {
    console.log("rev : " + names.at(-i - 1));
  }
}
accessElements();
```

- delete arr[i] - to delete element at given index - that position left "undefined".

```
function removeItems() {
  let names = ["Nilesh", "Rajiv", "Nitin", "Prashant"];
  console.log("deleting 1st pos : " + names[1]);
```

```
  delete names[1]; // delete element at position 1
  // names[1] will be now "undefined" -- splice() is better than delete operator.
  console.log("names (after deleting 1st pos) : " + names);
}

removeItems();
```

- slice(start,end) - to get a part of an array. the start index is inclusive, end index is exclusive.

```
function sliceArray() {
  let names = ["Nilesh", "Rajiv", "Rohan", "Nitin", "Prashant"];
  let part = names.slice(2, 4);
  console.log("names: " + names);
  console.log("names(1,4): " + part); // "Rajiv", "Rohan", "Nitin"
  let part2 = names.slice(2);
  console.log("names(2): " + part2); // "Nilesh", "Rajiv"
}

sliceArray();
```

Functional Programming

- map(fn) - operates on each array element and collect results into a new array.

```
function sqrNumsMap() {
  let nums = [11, 22, 33, 44, 55];
  let sqrs = nums.map((n) => n * n);
  console.log("nums : " + nums);
  console.log("sqrs : " + sqrs);
}
sqrNumsMap();
```

- filter(fn) - test a condition on each array element and collect elems satisfying the condition into a new array.

```
function numsEvenFilter() {
  let nums = [11, 22, 33, 44, 55, 66, 77, 88];
  let evens = nums.filter((n) => n % 2 == 0);
  console.log("nums : " + nums);
  console.log("evens : " + evens);
}
numsEvenFilter();
```

- flatMap(fn) - operates on each array element so that each element produce a new array and then collect results into a new array (flattened).

```
function numsSplitDigitsFlatMap() {
  function separateDigits(n) {
    let digs = [];
    while (n) {
      digs.push(n % 10);
      n = Math.trunc(n / 10);
    }
    return digs.reverse();
  }
  let nums = [123, 456, 789];
  let digits = nums.flatMap((n) => separateDigits(n));
  console.log("nums : " + nums);
  console.log("digits : " + digits);
}
numsSplitDigitsFlatMap();
```

- reduce(fn, start) - calls given fn repeatedly with accumulated result and next element to produce a single final result.

```
function reduceSum() {
  let nums = [11, 22, 33, 44, 55];
  let total = nums.reduce((x, y) => x + y, 0);
  console.log("nums : " + nums);
  console.log("total : " + total);
}
reduceSum();
```

- sort(fn) - sorts given array with order defined by given fn and return result as new array.

```
function sortNums() {
  let nums = [44, 33, 66, 22, 11, 55];
  let sorted = nums.sort();
  console.log("nums : " + nums);
  console.log("sorted : " + sorted);
  let revsorted = nums.sort((x, y) => y - x);
  console.log("rev sorted : " + revsorted);
}
sortNums();
```

JS String

- Strings can be enclosed in double quotes, single quotes, or back quotes (template strings).
- ${expr} in template strings is evaluated dynamically and added in string.

```
const s1 = "Sunbeam";
const s2 = 'DMC';
const s3 = `${s1}, ${s2}!!`;
console.log(s1);
```

```
console.log(s2);
console.log(s3);
```

- length property returns number of chars in string.

```
const text = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
console.length(text.length); // 26
```

- Escape characters like \n, \r, \t, \b, \', \", \\ are supported (like Java).
- String objects can be created by a few ways:

```
const s1 = "Nilesh";
console.log("s1 = " + s1 + " of type " + typeof s1); // string
const s2 = String(123);
console.log("s2 = " + s2 + " of type " + typeof s2); // string
const s3 = new String("Sunbeam");
console.log("s3 = " + s3 + " of type " + typeof s3); // object
```

- Not recommended to create String objects explicitly (with new).
  - The new keyword slows down execution speed.
  - String objects can produce unexpected results (equality check, eval(), etc).
- Strings can be compared using == (content check) or === (content and type check).
- Strings can be compared using == (address check) and returns false for two different objects (irrespective of contents).

```
const s4 = "Nilesh";
console.log("s1 == s4 : ", s1 == s4); // true - two strings are equal
```

```
const s5 = new String("Sunbeam");
console.log("s3 == s5 : ", s3 == s5); // false - two objects are never equal (addr comparison)
```

- Strings are immutable. Methods modifying string, returns new String objects.
- toUpperCase() and toLowerCase()

```
let text1 = "SunBeam";
let text2 = text1.toUpperCase();
let text3 = text1.toLowerCase();
```

- subscript vs at(index) vs charAt(index)
  - charAt() works similar to subscript operator.
  - at() added in ES 2022 and supports negative indexes too (like array)

```
const str = "sunbeam";
console.log("str[2] = " + str[2]); // n
console.log("str[-2] = " + str[-2]); // undefined
console.log("str.charAt(1) = " + str.charAt(1)); // u
console.log("str.charAt(-1) = " + str.charAt(-1)); // (blank "")
console.log("str.at(1) = " + str.at(1)); // u
console.log("str.at(-1) = " + str.at(-1)); // undefined
```

- slice(startIndex, endIndex) vs substring(startIndex, endIndex)
  - slice() extracts a part of a string and returns the extracted part in a new string.
    - startIndex is included and endIndex is excluded.
    - if endIndex is not given, returns the whole string from startIndex.
    - startIndex and endIndex can be negative i.e. w.r.t. end of string.

    - Example:

```
let text = "0123456789";
let part = text.slice(2, 8);  // 234567
```

```
let text = "0123456789";
let part = text.slice(2);  // 23456789
```

```
let text = "0123456789";
let part = text.slice(-2);  // 89
```

```
let text = "0123456789";
let part = text.slice(-8, -2);  // 234567
```

  - substring() is similar to slice().
    - if start and end index values less than 0, they are considered 0 in substring().

```
let text = "0123456789";
let part = text.slice(-2);  // 0123456789
```

- concat() can be used for string concatenation like + operator.

```
let text = "Hello".concat("SunBeam", "DMC!");
console.log(text);
```

- trim(), trimStart(), trimEnd() removes white spaces.

```
let text = "    Hello    ";
console.log(`text = |${text}|, length = ${text.length}`);
let result1 = text.trim();
console.log(`trim() = |${result1}|, length = ${result1.length}`);
let result2 = text.trimStart();
console.log(`trimStart() = |${result2}|, length = ${result2.length}`);
let result3 = text.trim();
console.log(`trimEnd() = |${result3}|, length = ${result3.length}`);
```

- padStart(), padEnd() add given char to increase the length of string.

```
let text = "A";
let padded1 = text.padStart(4,"0"); // 000A
console.log(padded1);
let padded2 = text.padEnd(4,"x"); // Axxx
console.log(padded2);
```

- repeat() string multiple times.

```
let text = "Hello world!";
let result = text.repeat(4);
```

- replace() replaces a specified value with another value in a string.

```
let text = "Hello DMC! Hello!!!";
let result1 = text.replace("Hello", "Hi");  // Hi DMC! Hello!!!
  // only first occurrance is replaced.
let result2 = text.replaceAll("Hello", "Hi"); // Hi DMC! Hi!!!
```

- split() splits string into multiple parts.

```
let text = "DAC,DMC,DESD,DBDA,DITISS";
let parts = text.split(","); // ["DAC","DMC","DESD","DBDA","DITISS"]
```

Node.js Modules

What are User-Defined Modules?

- Modules are reusable blocks of code that can be **exported** from one file and **imported** into another.
- Helps in **organizing code**, **improving maintainability**, and **avoiding repetition**.
- Node.js uses **CommonJS** modules by default (`require` & `module.exports`).

Key Concepts

1. `module.exports` → Exports a module for use in other files.
2. `require()` → Imports an external module.
3. **Types of Exports:**
    - **Single Export** (Function, Object, Variable)
    - **Multiple Exports** (Using an Object)

Single Export (Function)

```
// greet.js
function greet(name) {
  return `Hello, ${name}!`;
}

module.exports = greet; // Exporting the function
```

```
// app.js
const greet = require("./greet"); // Importing
console.log(greet("Nilesh")); // Output: "Hello, Nilesh!"
```

Multiple Exports (Object)

```
// mathUtils.js

function add(a, b) {
  return a + b;
}
const subtract = (a, b) => a - b;

module.exports = {
  add,
  sub: subtract,
  multiply: function (a, b) {
    return a * b;
  },
}; // Exporting multiple functions
```

```
// app.js
const { add, sub, multiply } = require("./mathUtils");
console.log(add(5, 3)); // Output: 8
console.log(sub(5, 3)); // Output: 2
console.log(multiply(5, 3)); // Output: 15
```

Alternative Syntax (Direct `exports`)

```
// logger.js
exports.log = (message) => console.log(`[LOG]: ${message}`);
exports.error = (message) => console.error(`[ERROR]: ${message}`);
```

```
// app.js
const { log, error } = require("./logger");
log("This is a log message"); // Output: [LOG]: This is a log message
error("Something went wrong!"); // Output: [ERROR]: Something went wrong!
```

Best Practices

1. **Keep modules small & focused** (Single Responsibility Principle).
2. **Use descriptive names** (`userService.js`, `authUtils.js`).
3. **Prefer `module.exports` for clarity** (Avoid mixing with `exports`).
4. **Use ES Modules (`import/export`) if using modern Node.js (v12+)**.

ES Modules (Alternative in Modern Node.js)

- Supported in modern Node JS i.e. v12+.

- If using `.mjs` **extension** or `"type": "module"` in `package.json`:

```
// utils.mjs
export const multiply = (a, b) => a * b;
```

```
// app.mjs
import { multiply } from "./utils.mjs";
console.log(multiply(2, 4)); // Output: 8
```

Example Applications

1. **Reusable utility functions** (e.g., `validation.js`).
2. **Service layers** (e.g., `userService.js`, `authService.js`).
3. **Configuration files** (e.g., `config.js`).

Node.js "os" Module

What is the `os` Module?

- A **built-in Node.js module** that provides utilities for interacting with the **operating system**.
- Helps fetch system-related information like **CPU, memory, network, OS details**, etc.

Key Methods & Properties

Method/Property	Description
`os.platform()`	Returns OS platform (e.g., `'linux'`, `'win32'`)
`os.arch()`	Returns CPU architecture (e.g., `'x64'`)

Method/Property	Description
os.cpus()	Returns CPU core info (Model, Speed, Usage)
os.totalmem()	Total system memory (in bytes)
os.freemem()	Free system memory (in bytes)
os.hostname()	Returns the system hostname
os.networkInterfaces()	Returns network interfaces (IP, MAC, etc.)
os.userInfo()	Returns current user info (Username, Home Dir)
os.uptime()	System uptime (in seconds)

Basic OS Info

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

console.log("OS Platform:", os.platform()); // 'linux', 'win32', 'darwin' (macOS)
console.log("CPU Architecture:", os.arch()); // 'x64', 'arm'
console.log("Hostname:", os.hostname());
console.log("Current User:", os.userInfo().username);
```

Memory Usage

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

const totalGB = (os.totalmem() / 1024 ** 3).toFixed(2); // Convert bytes to GB
const freeGB = (os.freemem() / 1024 ** 3).toFixed(2);
```

```
console.log(`Total RAM: ${totalGB} GB`);
console.log(`Free RAM: ${freeGB} GB`);
```

CPU & Network Info

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

console.log("CPU Cores:", os.cpus().length); // Number of CPU cores
console.log("Network Interfaces:", os.networkInterfaces());
```

Node.js "fs" Module

What is the `fs` Module?

- A **built-in Node.js module** for interacting with the **file system**.
- Supports **synchronous** (blocking) and **asynchronous** (non-blocking) operations.
- Used for:
  - Reading/Writing files
  - Creating/Deleting files & directories
  - File permissions & stats

Reading Files

Asynchronous (Non-blocking)

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

fs.readFile("file.txt", "utf8", (err, data) => {
  if (err) throw err;
```

```
  console.log(data); // File content
});
```

Synchronous (Blocking)

```
const data = fs.readFileSync("file.txt", "utf8");
console.log(data);
```

Writing Files

Asynchronous

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

Synchronous

```
fs.writeFileSync("newFile.txt", "Hello, Node.js!");
```

Appending to Files

```
fs.appendFile("log.txt", "New log entry\n", (err) => {
  if (err) throw err;
```

```
  console.log("Log updated!");
});
```

File & Directory Operations

Method	Description	Example
`fs.unlink()`	Deletes a file	`fs.unlink('file.txt', (err) => {})`
`fs.mkdir()`	Creates a directory	`fs.mkdir('newFolder', (err) => {})`
`fs.rmdir()`	Removes an empty directory	`fs.rmdir('emptyFolder', (err) => {})`
`fs.rename()`	Renames a file/directory	`fs.rename('old.txt', 'new.txt', (err) => {})`
`fs.existsSync()`	Checks if a file exists	`if (fs.existsSync('file.txt')) {...}`

File Stats (Metadata)

```
fs.stat("file.txt", (err, stats) => {
  if (err) throw err;
  console.log("Is File:", stats.isFile()); // true
  console.log("Size:", stats.size); // bytes
  console.log("Created:", stats.birthtime); // Date object
});
```

Best Practices

1. **Prefer async methods** (Avoid blocking the event loop).
2. **Use `fs.promises` for Promise-based operations** (Modern approach).
3. **Handle errors properly** (Avoid crashes).
4. **Use `path.join()` for cross-platform paths** (Avoid `\` vs `/` issues).

Callback Hell

What is Callback Hell?

Callback Hell (a.k.a. "Pyramid of Doom") occurs when multiple nested callbacks make code:

- **Hard to read** (Deep indentation)
- **Difficult to maintain**
- **Prone to errors** (Error handling becomes messy)

Example 1: Callback Hell in FS Operations

- input a path from command-line.
- if path is of file, display its info (stat)
- if path is of directory, display its contents
- if path doesn't exist, give error

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

if (process.argv.length != 3) {
  console.log("pass file or directory path as command line arg.");
  process.exit(1);
}

const path = process.argv[2];

fs.stat(path, (err, status) => {
  if (err) {
    console.log("Error: ", err);
    return;
  }
  if (status.isFile()) {
    console.log("File Size: " + (status.size / 1024).toFixed(2) + " KB");
    console.log("Created on : " + status.birthtime);
```

```
      console.log("Modified on : " + status.mtime);
    } else if (status.isDirectory()) {
      fs.readdir(path, (err, files) => {
        // Nested Callback
        if (err) {
          console.log("Error reading directory: ", err);
          return;
        }
        console.log("Directory listing: ");
        files.forEach((f) => console.log(f));
      });
    } else {
      console.log("Neither file nor directory.");
    }
  });
```

Example 2: Callback Hell in FS Operations

- input a path from user.
- if path is of doesn't exists or if not of file, raise error.
- if path is of a file, copy it into "dest.txt".
- finally read that file and display its contents.

Problems in This Example

1. **Deep nesting** makes code harder to follow.
2. **Error handling is repetitive** (`if (err)` checks everywhere).
3. **Difficult to extend** (Adding more operations worsens nesting).

JS Promises

A **Promise** is an object representing the eventual **completion (or failure)** of an asynchronous operation.

Key States of a Promise

- `pending` → Initial state (not yet resolved/rejected)
- `fulfilled` → Operation completed successfully
- `rejected` → Operation failed

Creating a Promise

```
const myPromise = new Promise((resolve, reject) => {
  // Async operation (e.g., API call, file read)
  if (success) {
    resolve("Data fetched!"); // Success case
  } else {
    reject("Error occurred!"); // Failure case
  }
});
```

Consuming Promises

`.then()` → **Success Handler**

```
myPromise.then((result) => {
  console.log(result); // "Data fetched!"
});
```

`.catch()` → **Error Handler**

```
myPromise.catch((error) => {
  console.error(error); // "Error occurred!"
```

```
});
```

`.finally()` → **Runs Always**

```
myPromise.finally(() => {
  console.log("Operation complete (success or failure)");
});
```

Chaining Promises

Avoids **callback hell** by chaining `.then()` calls:

```
fetchData()
  .then((data) => processData(data))
  .then((result) => displayResult(result))
  .catch((err) => console.error(err));
```

Example 1: Simple Promise

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

const filePath = process.argv[2]; // file path from command-line arg

const readPromise = new Promise((resolve, reject) => {
  fs.readFile(filePath, (err, data) => {
    if (err) {
      reject(err);
    } else {
```

```
      resolve(data.toString());
    }
  });
});

readPromise
  .then((result) => {
    console.log("Success: " + result);
  })
  .catch((error) => {
    console.log("Error: " + error);
  });
```

Example 2: Solving Callback Hell

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

// 1. Manual Promise wrapper for fs.stat
function statPromise(path) {
  return new Promise((resolve, reject) => {
    fs.stat(path, (err, stats) => {
      if (err) reject(err);
      else resolve(stats);
    });
  });
}

// 2. Manual Promise wrapper for fs.readdir
function readdirPromise(path) {
  return new Promise((resolve, reject) => {
    fs.readdir(path, (err, files) => {
      if (err) reject(err);
      else resolve(files);
    });
```

```
  });
}

// 3. Main function using our promises
function inspectPath(path) {
  statPromise(path)
    .then((stats) => {
      if (stats.isFile()) {
        console.log(`File Size: ${(stats.size / 1024).toFixed(2)} KB`);
        console.log(`Created: ${stats.birthtime}`);
        console.log(`Modified: ${stats.mtime}`);
      } else if (stats.isDirectory()) {
        return readdirPromise(path);
      }
    })
    .then((files) => {
      if (files) {
        console.log("Directory contents:");
        files.forEach((file) => console.log(`- ${file}`));
      }
    })
    .catch((err) => {
      console.error("Error:", err.message);
    });
}

// 4. Usage
const path = process.argv[2];
inspectPath(path);
```

Key Improvements (over callback approach)

1. **Flat structure** (No nested callbacks).

```
statPromise(path)
  .then((stats) => {
    /* handle stats */
  })
  .then((files) => {
    /* handle files */
  })
  .catch((err) => {
    /* handle all errors */
  });
```

2. **Single** `.catch()` for all errors.

- One .catch() at the end handles all possible errors
- Better than checking if (err) at every level

3. **Reusable Promise-based functions**.

- The promise wrappers can be used anywhere in your code

Using fs/promises

- The fs/promises API provides asynchronous file system methods that return promises.
- The promise APIs use the underlying Node.js threadpool to perform file system operations off the event loop thread.

```
//const fs = require("fs").promises;  // old style Node v10.0
const fs = require("fs/promises"); // new style Node v14.0 (can still use old style)

const path = process.argv[2];

fs.stat(path)
  .then((status) => {
    if (status.isFile()) {
```

```
      console.log("File Size: " + (status.size / 1024).toFixed(2) + " KB");
      console.log("Created on : " + status.birthtime);
      console.log("Modified on : " + status.mtime);
    } else if (status.isDirectory()) {
      console.log("Directory listing: ");
      return fs.readdir(path);
    } else {
      console.log("Neither file nor directory.");
    }
  })
  .then((files) => {
    if (files) {
      files.forEach((f) => console.log(f));
    }
  })
  .catch((error) => {
    console.log("Error: " + error);
  });
```

Static Promise Methods

- Promises also have helper methods.

Method	Description	Example
`Promise.resolve()`	Creates a resolved Promise	`Promise.resolve("Done")`
`Promise.reject()`	Creates a rejected Promise	`Promise.reject("Error")`
`Promise.all()`	Waits for **all** Promises to resolve	`Promise.all([p1, p2])`
`Promise.race()`	Resolves when **any** Promise settles	`Promise.race([p1, p2])`
`Promise.any()`	Resolves when **any** Promise fulfills	`Promise.any([p1, p2])`

Promise.resolve()

Returns a Promise object that is resolved with the given value (data).

```
// resolved promise
const resPromise = Promise.resolve("Hello, World!");
resPromise.then((data) => {
  console.log("Resolved Callback : " + data);
});
```

Promise.reject()

Returns a new Promise object that is rejected with the given reason.

```
// rejected promise
const rejPromise = Promise.reject("Bye, World!");
rejPromise.catch((reason) => {
  console.log("Rejected Callback : " + reason);
});
```

Promise.all()

Takes an iterable of promises as input and returns a single Promise. This returned promise fulfills when all of the input's promises fulfill (including when an empty iterable is passed), with an array of the fulfillment values. It rejects when any of the input's promises reject, with this first rejection reason.

```
// wait for all promises
const promise1 = Promise.resolve("Promise1");
const promise2 = Promise.resolve("Promise2");
const promise3 = Promise.resolve("Promise3");
```

```
const allPromise = Promise.all([promise1, promise2, promise3]);
allPromise.then((data) => {
  for (let i = 0; i < data.length; i++) {
    console.log("Executing Promise (all) : " + data[i]);
  }
});
```

Promise.any()

Takes an iterable of promises as input and returns a single Promise. This returned promise fulfills when any of the input's promises fulfill, with this first fulfillment value. It rejects when all of the input's promises reject (including when an empty iterable is passed), with an AggregateError containing an array of rejection reasons.

```
// wait for any one promise
const promis1 = new Promise((resolve, reject) => {
  // resolved promise -- only success
  setTimeout(resolve, 1500, "Promise 1500");
});

const promis2 = new Promise((resolve, reject) => {
  // resolved promise -- only success
  setTimeout(resolve, 1200, "Promise 1200");
});

const promis3 = new Promise((resolve, reject) => {
  // resolved promise -- only success
  setTimeout(resolve, 2000, "Promise 2000");
});

// resolves with the value of the first fulfilled promise.
const anyPromis = Promise.any([promis1, promis2, promis3]);
anyPromis.then((data) => {
```

```
  console.log("Executing Promise (any) : " + data);
});
```

Promise.allSettled()

Takes an iterable of promises as input and returns a single Promise. This returned promise fulfills when all of the input's promises settle (including when an empty iterable is passed), with an array of objects that describe the outcome of each promise.

```
// here allSettled waits for all promises and return result
const allSettledPromise = Promise.allSettled([promis1, promis2, promis3]);
allSettledPromise.then((data) => {
  for (let i = 0; i < data.length; i++) {
    console.log("Executing Promise (allSettled) : " + JSON.stringify(data[i]));
  }
});
```

Promise use cases

1. **API calls** (`fetch`, Axios).
2. **File operations** (`fs.promises`).
3. **Database queries** (Mongoose, Sequelize).

Async/Await in JavaScript

What is Async/Await?

Async/await is syntactic sugar built on top of Promises that:

- Makes asynchronous code look synchronous
- Eliminates `.then()` chains
- Uses `try/catch` for error handling

- Requires the `async` keyword before functions

Basic Syntax

- Promise based version

```
function resolvedPromise1() {
  return Promise.resolve("Resolved Promise1");
}

const rp1 = resolvedPromise1();
console.log("rp1 type :", rp1);
rp1.then((data) => {
  console.log(data);
});
```

- Async/Await version

```
async function resolvedPromise2() {
  return "Resolved Promise2"; // returns data wrapped in a fulfilled promise
}

const rp2 = resolvedPromise2();
console.log("rp2 type :", rp2); // Promise
rp2.then((data) => {
  console.log(data);
});
```

- The await keyword can only be used inside an async function.
- The await keyword makes the function pause the execution and wait for a resolved promise before it continues.

```
async function waitForPromise2() {
  const resPromise2 = await resolvedPromise2();
  console.log("Result of Promise2 : " + resPromise2);
  return "Wait for Promise2 is Completed";
}

waitForPromise2().then((data) => {
  console.log(data);
});
```

Key Characteristics

- `async` functions always return Promises
- `await` pauses execution until Promise settles
- Works with any Promise-based API
- Errors are caught with `try/catch`

Async/Await Version

```
async function inspectPath(path) {
  try {
    const stats = await statPromise(path);

    if (stats.isFile()) {
      console.log(`File Size: ${(stats.size / 1024).toFixed(2)} KB`);
      console.log(`Created: ${stats.birthtime}`);
      console.log(`Modified: ${stats.mtime}`);
    } else if (stats.isDirectory()) {
      const files = await readdirPromise(path);
      console.log("Directory contents:");
      files.forEach((file) => console.log(`- ${file}`));
    }
```

```
  } catch (err) {
    console.error("Error:", err.message);
  }
}
```

Key Improvements with Async/Await

1. **Flatter Structure**
    - No more `.then()` chains
    - Sequential code flow
2. **Natural Error Handling**
    - Uses standard `try/catch`
3. **Variable Scope**
    - Variables available in same scope (no nested closures)
4. **Readability**
    - Looks like synchronous code

Important Rules

Rule 1: Await Only Works in Async Functions

```
// Wrong
function regularFunc() {
  await somePromise; // SyntaxError
}

// Correct
```

```
async function asyncFunc() {
  await somePromise; // Works
}
```

Rule 2: Async Functions Return Promises

```
async function hello() {
  return "Hello";
}

// Equivalent to:
function hello() {
  return Promise.resolve("Hello");
}
```

Rule 3: Top-Level Await (Modern JS v14.8.0+)

```
// Works in ES modules
const data = await fetchData();
console.log(data);
```

Error Handling

Method 1: Try/Catch

```
async function fetchData() {
  try {
    const data = await riskyOperation();
```

```
    return process(data);
  } catch (error) {
    console.error("Failed:", error);
    return fallbackData;
  }
}
```

Method 2: Promise.catch()

```
async function fetchData() {
  const data = await riskyOperation().catch((error) => {
    console.error("Failed:", error);
    return fallbackData;
  });
  return process(data);
}
```

Practical Examples

Example 1: Sequential Operations

```
async function getUserProfile(userId) {
  const user = await fetchUser(userId);
  const posts = await fetchPosts(user.id);
  const friends = await fetchFriends(user.id);

  return { user, posts, friends };
}
```

Example 2: Parallel Operations

```
async function getDashboardData() {
  const [user, posts, notifications] = await Promise.all([
    fetchUser(),
    fetchPosts(),
    fetchNotifications(),
  ]);

  return { user, posts, notifications };
}
```

Async/Await vs Promises

Feature	Promises	Async/Await
Syntax	`.then()` chains	Sequential code
Error Handling	`.catch()`	`try/catch`
Debugging	Harder (anonymous funcs)	Easier (proper stack)
Readability	OK for simple chains	Better for complex logic

Best practices

1. **Using `await` correctly**

```
async function getData() {
  // Returns Promise pending -- to be resolved/rejected later (outside getData())
  return fetch("/api");
}
```

```
async function getData() {
  // Returns resolved data -- when error handling is needed (in getData())
  try {
    return await fetch("/api");
  } catch (err) {
    // error handling
  }
}
```

2. **Avoid Unnecessary Serialization**

```
// Slow (sequential)
const a = await fetchA();
const b = await fetchB();

// Fast (parallel)
const [a, b] = await Promise.all([fetchA(), fetchB()]);
```

3. **Handle Errors**

```
// Proper handling
try {
  const data = await fetchData();
  // Process data
} catch (err) {
  // Handle error
}
```

When to Use Async/Await vs Promises

Async/Await

- Most asynchronous code
- Complex promise chains
- Error-heavy operations
- Anywhere you want synchronous-looking code

Raw Promises

- You need `Promise.all()`/`Promise.race()`
- Working with existing promise-based libraries
- Performance-critical parallel operations

HTTP Server

- Node.js has a built-in module called HTTP, which allows Node.js to transfer data over the Hyper Text Transfer Protocol (HTTP).
- The HTTP module can create an HTTP server that listens to server ports and gives a response back to the client.

HTTP Server - Minimal Example

```
const http = require("http"); // Core HTTP module

const server = http.createServer((req, res) => {
  res.writeHead(200, { "Content-Type": "text/plain" });
  res.end("Hello World!");
});

server.listen(3000, () => {
  console.log("Server running on http://localhost:3000");
});
```

- **Key Components**
  - `http.createServer()` - Creates server instance

- `req` (Request) - Incoming message object
- `res` (Response) - Server response object
- `server.listen()` - Starts server on specified port

HTTP Server - Handling Different Routes

```
const server = http.createServer((req, res) => {
  if (req.url === "/") {
    res.writeHead(200, { "Content-Type": "text/html" });
    res.end("<h1>Homepage</h1>");
  } else if (req.url === "/about") {
    res.writeHead(200, { "Content-Type": "text/html" });
    res.end("<h1>About Us</h1>");
  } else {
    res.writeHead(404, { "Content-Type": "text/html" });
    res.end("<h1>404 Not Found</h1>");
  }
});
```

HTTP Server - Handling Different HTTP Methods

```
const server = http.createServer((req, res) => {
  switch (req.method) {
    case "GET":
      // Handle GET requests
      break;
    case "POST":
      // Handle POST requests
      let body = "";
      req.on("data", (chunk) => {
        body += chunk.toString();
      });
```

```
      req.on("end", () => {
        console.log(body);
        res.end("OK");
      });
      break;
    default:
      res.writeHead(405, { "Content-Type": "text/plain" });
      res.end("Method Not Allowed");
  }
});
```

HTTP Server - Setting Response Headers

```
res.writeHead(200, {
  "Content-Type": "text/html",
  "Custom-Header": "SomeValue",
});
```

Creating a REST API

```
const server = http.createServer((req, res) => {
  const { method, url } = req;
  const baseURL = `http://${req.headers.host}/`;
  const parsedUrl = new URL(url, baseURL);
  const pathname = parsedUrl.pathname; // req.url
  const query = parsedUrl.searchParams;

  // Set CORS headers
  res.setHeader("Access-Control-Allow-Origin", "*");
  res.setHeader("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE");
  res.setHeader("Access-Control-Allow-Headers", "Content-Type");
```

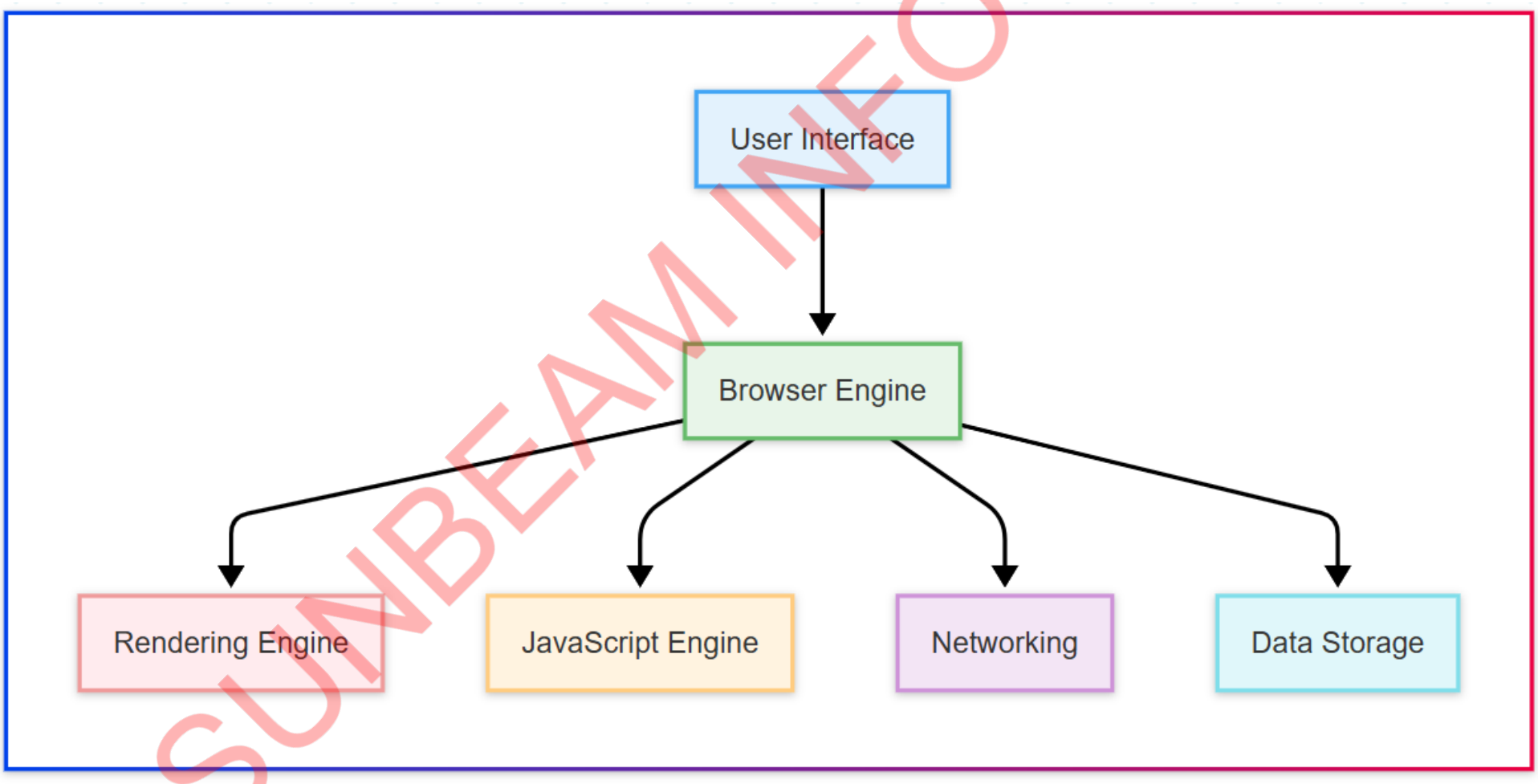
```
  // API routes
  if (pathname === "/api/users" && method === "GET") {
    res.writeHead(200, { "Content-Type": "application/json" });
    res.end(JSON.stringify([{ id: 1, name: "John" }]));
  } else if (pathname === "/api/users" && method === "POST") {
    let body = "";
    req.on("data", (chunk) => (body += chunk));
    req.on("end", () => {
      res.writeHead(201, { "Content-Type": "application/json" });
      res.end(body);
    });
  } else {
    res.writeHead(404, { "Content-Type": "application/json" });
    res.end(JSON.stringify({ message: "Route not found" }));
  }
});
```

Node JS Concepts & Internals

Browser Architecture

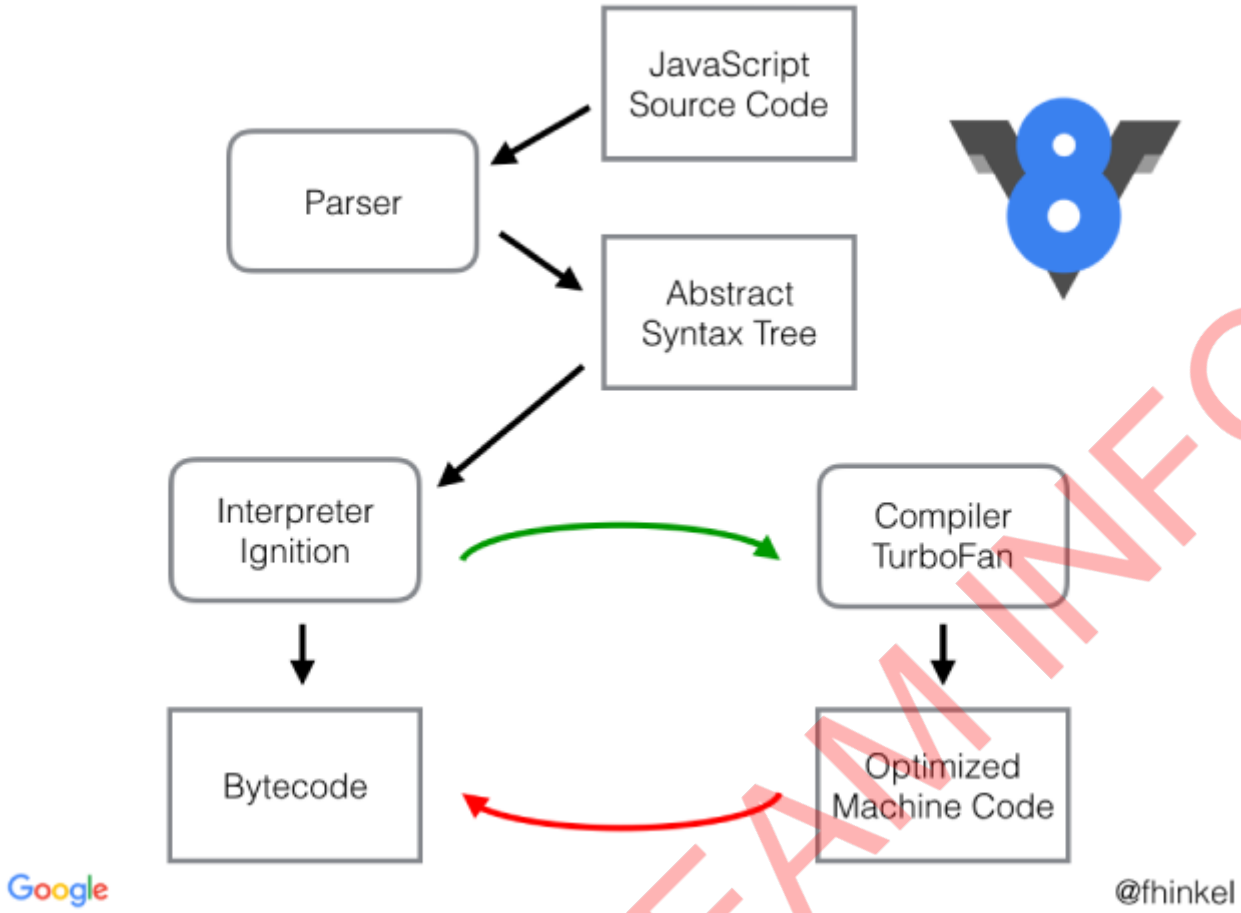
- **Browser**: Specialized application for website browsing with parsers (HTML, CSS, XML) to render UI.
    - **Key Components**:
        - **User Interface**: Browser user interface (menus, tabs, etc).
        - **Browser Engine**: Bridge between User interface and Browser core functionality.
        - **Network**: Handles HTTP requests/responses.
        - **Data Storage**: Manage persistent storage like local storage.
        - **Rendering Engine**: Renders HTML/CSS components.
        - **JavaScript Engine**: Executes JS code.
    - **Java Script Engine**:

- A JavaScript engine is a program or software component that executes JavaScript code. It translates human-readable JavaScript code into machine code that a computer can understand and execute. Modern JavaScript engines use just-in-time (JIT) compilation to improve performance.
- Popular JavaScript engines include:
  - V8: Developed by Google, used in Chrome and Node.js.
  - SpiderMonkey: Developed by Mozilla, used in Firefox. It was the first JavaScript engine.
  - JavaScriptCore: Used in Safari.
  - Chakra: Used in Internet Explorer and Microsoft Edge (legacy).



V8 Engine

- **Description**: Google's open-source, high-performance JS/Wasm engine written in C++.
- **Used In**: Chrome and Node.js.
- **Workflow**:
    1. **Parser**: Converts JS source code to an Abstract Syntax Tree (AST).
    2. **Interpreter (Ignition)**: Generates bytecode.
    3. **Compiler (TurboFan)**: Optimizes bytecode to machine code.
- **Key Role**: Powers Node.js's JavaScript runtime.
- **Details**: https://braineanear.medium.com/the-v8-engine-series-i-architecture-ba08a38c54fa



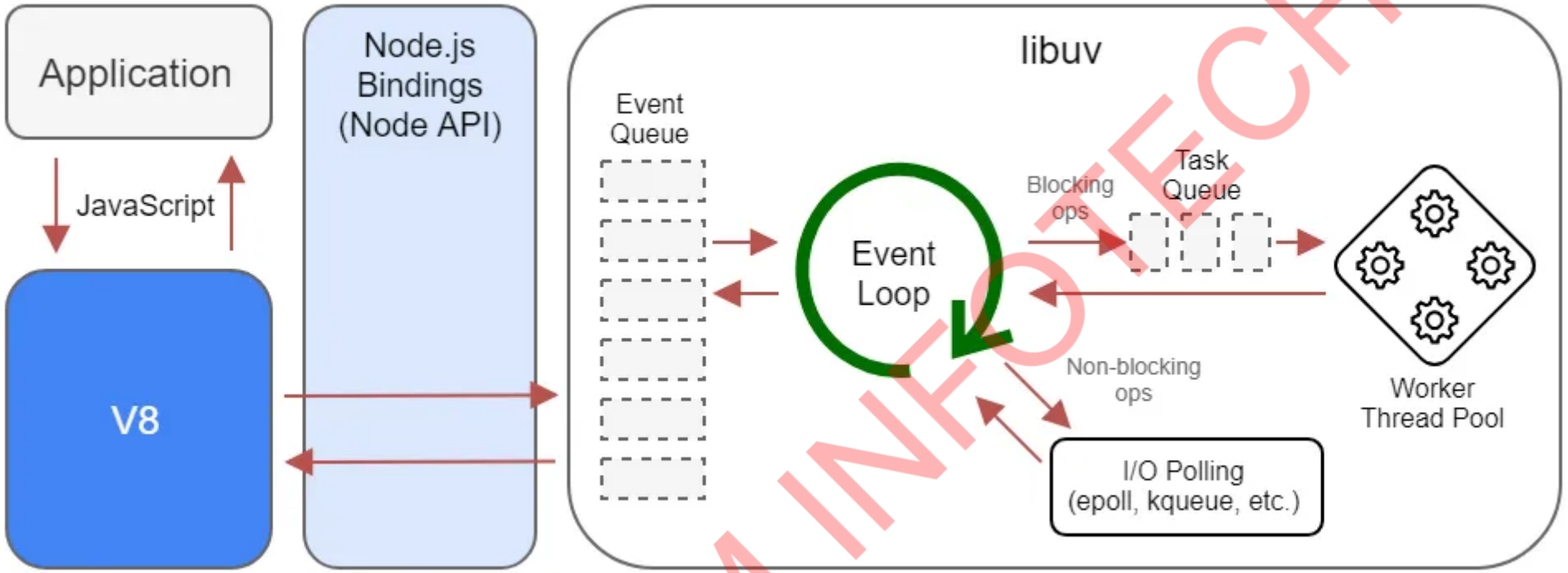
Node.js Overview

- **Definition**: JavaScript runtime built on V8 for server-side and networking apps.
- **Developed By**: Ryan Dahl (2009).
- **Cross-Platform**: Runs on macOS, Linux, Windows.
- **Core Features**:
  - Event-driven, non-blocking I/O model.
  - Single-threaded but scalable.

- Lightweight and efficient.

Node.js Architecture

- **Key Components**:
  - **V8 Engine**: Executes JS code.
  - **libuv**: Handles event loop and async I/O (uses OS async APIs like `epoll`, `kqueue`).
  - **Thread Pool**: Used for:
    - Crypto operations.
    - DNS lookups.
    - File I/O (when no OS async API exists).
  - **Standard Library**: Provides modules for:
    - Filesystem (`fs`), HTTP, TCP/UDP, Buffers, Crypto.
  - **Details**: https://youtu.be/P9csgxBgaZ8



Node JS Misconceptions

- **Single-Threaded?**:
  - **Yes**: Event loop and V8 run on a single thread (main call stack).
  - **But**: Thread pool (via libuv) handles blocking operations.
- **Thread Pool Usage**:
  - Not for all I/O! Only when no OS async API is available.
  - Used for CPU-bound tasks (e.g., `crypto.pbkdf2`).
- **Event Loop vs. V8 Threads**:
  - Event loop runs in one thread; V8 may use additional threads for optimization.

Event Loop & Non-Blocking I/O

- **Event-Driven Model**:
  - **Event Queue**: Holds pending callbacks.
  - **Event Loop**: Continuously checks and executes callbacks from phases:
    1. **Timers**: `setTimeout`, `setInterval`.
    2. **I/O Callbacks**: Network/File operations.
    3. **Idle/Prepare**: This phase is internal to Node.js and prepares for the Poll phase.
    4. **Poll Phase**: Waits for new I/O events.
    5. **Check Phase**: `setImmediate` callbacks.
    6. **Close Phase**: Cleanup (e.g., `socket.on('close')`).
- **Non-Blocking I/O**: Achieved via libuv's async OS APIs (e.g., `epoll` on Linux).
- **Example:**

```
// Shows event loop order
setImmediate(() => console.log("Immediate"));
setTimeout(() => console.log("Timeout 0"), 0);
Promise.resolve().then(() => console.log("Promise"));
process.nextTick(() => console.log("Next Tick"));

// Output order:
// Next Tick > Promise > Timeout 0 > Immediate
```

- **Details**: https://nodejs.org/en/learn/asynchronous-work/event-loop-timers-and-nexttick
- **Advanced Tutorial**: http://latentflip.com/loupe

Use Cases

- **Ideal For**:
  - I/O-bound apps (APIs, data streaming).
  - Real-time apps (chat, IoT).
  - JSON APIs and SPAs.
- **Avoid For**:
  - CPU-intensive tasks (use Worker Threads or other languages).

Advantages

- **Full-Stack JavaScript**: Unified language for frontend/backend.
- **Scalability**: Event loop handles thousands of concurrent connections.
- **Ecosystem**: NPM's vast package library.
- **Performance**: Fast execution (V8) and low-latency I/O.

Worker Threads for CPU Tasks

- **Purpose**: Offload CPU-heavy work from the main thread. Added in Node v10.5.0.
- **Example**:

```
const { Worker } = require("worker_threads");
const worker = new Worker("./cpu-task.js");
worker.on("message", (result) => console.log(result));
```

MERN vs. MEAN vs. XAMPP

Full-Stack Comparison

Feature	MERN Stack	MEAN Stack	XAMPP

Feature	MERN Stack	MEAN Stack	XAMPP
Components	MongoDB, Express, React, Node.js	MongoDB, Express, Angular, Node.js	Apache, MySQL, PHP, Perl (Linux: LAMP)
Frontend	React (JavaScript library)	Angular (TypeScript framework)	Typically plain PHP/HTML or CMS
Backend	Node.js + Express	Node.js + Express	PHP/Perl + Apache
Database	MongoDB (NoSQL)	MongoDB (NoSQL)	MySQL (SQL)
Language	JavaScript (full-stack)	JavaScript + TypeScript	PHP, SQL, HTML/CSS/JS
Performance	Virtual DOM (React) → Fast UI updates	Two-way data binding → Slightly slower	Server-side rendering → Moderate
Learning Curve	Moderate (React flexibility)	Steeper (Angular's complexity)	Easiest (traditional web dev)
Use Cases	SPAs, real-time apps, modern web apps	Enterprise apps, complex UIs	Traditional websites, WordPress, PHP apps
Deployment	Cloud-native (e.g., Vercel, Heroku)	Cloud-native	Shared hosting/cPanel

Highlights

1. **MERN/MEAN**:
    - **JavaScript everywhere** (shared language).
    - Best for **dynamic, modern web apps**.
    - Components
        - Platform: Node
        - Web Server: Node (HTTP)
        - Database: MongoDB (you may choose MySQL, ...)
        - Application: Backend=Express, Frontend=React/Angular
2. **XAMP**:
    - **Traditional LAMP stack** for PHP-based apps.
    - Ideal for **WordPress, legacy systems**.
    - Components

- Platform: X (Linux - LAMP or Windows - WAMP or Mac - MAMP)
- Web Server: Apache
- Database: MySQL (you may choose other)
- Application: PHP/Perl (Server side rendering)

3. **Angular vs. React**:
    - **Angular**: Full framework (more boilerplate).
    - **React**: Library (more flexibility).

npm package manager

What is npm?

- **Node Package Manager**: World's largest software registry for JavaScript packages.
- **Key Functions**:
    - Install/uninstall packages
    - Manage dependencies
    - Run project scripts
    - Publish packages

npm Commands Cheat Sheet

```
npm install                 # Install all dependencies
npm install package@1.2.3   # Install specific version
npm update                  # Update packages per package.json rules
npm list                    # View installed dependency tree
npm audit                   # Security vulnerability check
npm run script-name         # Run custom script
npm publish                 # Publish package to registry
```

npm init

- The npm init command is used to initialize a new npm project and create a package.json file.

```
npm init
```

Package Installation

- Third party packages are downloaded from central repository and installed to the project (node_modules directory) or system-wide (c:/Users//AppData/Roaming/npm).

```
npm install
```

- Package installation creates package-lock.json

Install Types

Command	Effect	Saves to
`npm install package`	Local install	`node_modules`
`npm install -g package`	Global install	System-wide

package.json

Example

```
{
  "name": "my-app", // Package name
  "version": "1.0.0", // Semantic versioning
  "description": "My Node.js app",
  "main": "index.js", // Entry point
  "scripts": {
    // Custom commands
    "start": "node index.js",
    "dev": "nodemon index.js"
```

```
  },
  "dependencies": {
    // Production packages
    "express": "^4.18.2"
  },
  "devDependencies": {
    // Development-only packages
    "nodemon": "^3.0.2"
  },
  "engines": {
    // Node/npm version requirements
    "node": ">=18.0.0",
    "npm": ">=9.0.0"
  }
}
```

Version Syntax

```
"dependencies": {
  "express": "^4.18.2",  // Minor/patch updates allowed (4.x.x)
  "lodash": "~4.17.21",  // Patch updates only (4.17.x)
  "react": "16.8.6"      // Exact version
}
```

package-lock.json

- Records **exact versions** of all installed packages
- Creates reproducible builds
- Generated automatically on `npm install`

Key Features

- Locks dependency tree structure
- Uses integrity hashes (SHA-512) for security
- Should be committed to version control

Example

```
{
  "name": "my-app",
  "version": "1.0.0",
  "lockfileVersion": 3,
  "requires": true,
  "packages": {
    "": {
      "dependencies": {
        "express": {
          "version": "4.18.2",
          "resolved": "https://registry.npmjs.org/express/-/express-4.18.2.tgz",
          "integrity": "sha512-5/PsL6iGPdfQ/lKM1UuielYgv3BUoJfz1aUwU9vHZT+...",
          "dependencies": {
            "accepts": "~1.3.8"
          }
        }
      }
    }
  }
}
```

npm alternatives

- **Yarn**: Faster, deterministic installs
- **pnpm**: Disk-efficient (hard links)
- **Bun**: All-in-one runtime with built-in package manager

What is Yarn?

- **Fast, reliable dependency manager** created by Facebook (now community-maintained)
- **Alternative to npm** with key improvements:
  - Deterministic installs via `yarn.lock` (instead of package-lock.json)
  - Parallel downloads for faster installation
  - Works with npm registry (same packages)
  - Caches packages for repeat installs without internet

Yarn Commands

```
yarn init          # Create package.json
yarn add package   # Add dependency
yarn remove package # Remove package
yarn install       # Install all deps (creates yarn.lock)
yarn upgrade       # Update packages
```

yarn.lock vs package-lock.json

Feature	yarn.lock	package-lock.json
Format	YAML-like	JSON
Deterministic	Yes	Yes
Human-readable	More readable	Less readable
Install Speed	Faster (parallel)	Slower (sequential)

Express.js

Fundamentals

What is Express?

- **Minimalist web framework** for Node.js
- **Simplifies server creation** with routing, middleware, and HTTP helpers
- **Used by** companies like Uber, Accenture, and Twitter

Installation

```
npm init -y          # Initialize project (with defaults -y)
npm install express  # Install Express
```

Basic Server Setup

```
const express = require("express");
const app = express();
const PORT = 3000;

// Basic route
app.get("/", (req, res) => {
  res.send("Hello World!");
});

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

Add Routes

```
// GET request
app.get("/api/users", (req, res) => {
  res.send("Get all users");
});

// POST request
app.post("/api/users", (req, res) => {
  res.send("Create new user");
});

// Route parameters
app.get("/api/users/:id", (req, res) => {
  res.send(`User ID: ${req.params.id}`);
});
```

Handling Requests

Method	Example	Description
`req.params`	`req.params.id`	Route parameters
`req.query`	`req.query.search`	URL query strings
`req.body`	`req.body.name`	Request payload (needs middleware)

Response Methods

Method	Example	Description
`res.send()`	`res.send('Hello')`	Send response
`res.json()`	`res.json({data})`	Send JSON
`res.status()`	`res.status(404).send()`	Set status code

Method	Example	Description
`res.redirect()`	`res.redirect('/new')`	Redirect

Starting the Server

```
node app.js      # Basic start
nodemon app.js   # Auto-restart on changes (install via npm i -g nodemon)
```

Getting Started

Introduction to Express

- **Developed by**: TJ Holowaychuk (initial release) and now maintained by Node.js Foundation and open-source contributors
- **What it is**:
  - A minimal, unopinionated web framework for Node.js
  - Provides robust routing and HTTP utility methods
  - Designed for building web applications and APIs
- **Key Characteristics**:
  - Fast server-side development
  - Middleware-centric architecture
  - Minimal core that can be extended with middleware

```
npm install express  # Using npm
# OR
yarn add express     # Using Yarn
```

Serving Static Files

- Arrange the project directory structure

```
project/
    ├── index.js
    ├── public/
            css/
            │   └── style.css
            ├── js/
            │   └── app.js
            └── images/
            │    └── logo.png
            ├── pages
                └── index.html
```

- Static resources are placed in "public" directory

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

- Static resources accessed with URL.
    - http://localhost:3000/pages/index.html
    - http://localhost:3000/css/style.css
    - http://localhost:3000/images/logo.png

Routing

Routing syntax

```
```javascript
app.METHOD(PATH, HANDLER)
```
```

```
```

- `app`: Express application instance
- `METHOD`: HTTP method (`get`, `post`, `put`, `delete`, etc.)
- `PATH`: URL path (string or regex pattern)
- `HANDLER`: Callback function executed when route matches

Route Examples

```
// GET request to homepage
app.get("/", (req, res) => {
  res.send("Home Page");
});

// POST request to create user
app.post("/users", (req, res) => {
  res.status(201).json({ message: "User created" });
});

// Route with parameter
app.get("/users/:userId", (req, res) => {
  res.send(`User ID: ${req.params.userId}`);
});

// Route with query parameters
app.get("/search", (req, res) => {
  res.send(`Search term: ${req.query.q}`);
});
```

app.route()

- Can contain multiple callback functions

- Can be chained for the same path

```
app
  .route("/books")
  .get((req, res) => res.send("Get a book"))
  .post((req, res) => res.send("Add a book"));
```

Request & Response Objects

Request (req) Methods

Method	Example	Description
`req.params`	`req.params.userId`	Route parameters (from `/users/:userId`)
`req.query`	`req.query.search`	URL query parameters (`?search=term`)
`req.body`	`req.body.username`	Request payload (requires `express.json()` middleware)

Response (res) Methods

Method	Example	Description
`res.send()`	`res.send('Hello')`	Sends HTTP response
`res.json()`	`res.json({data})`	Sends JSON response
`res.status()`	`res.status(404).send()`	Sets HTTP status code
`res.sendFile()`	`res.sendFile(path)`	Sends a file

REST Protocol

What is REST?

- **REpresentational State Transfer** (REST) is an architectural style for distributed systems
- **Key Principles**:
  - **Stateless**: Each request contains all necessary information
  - **Client-Server**: Separation of concerns
  - **Cacheable**: Responses can be cached
  - **Uniform Interface**: Consistent resource identification and manipulation
  - **Layered System**: Intermediary servers can be inserted without client awareness

REST Resource Representation

```
// JSON Example
{
  "id": 101,
  "title": "REST API Design",
  "author": "John Doe",
  "price": 29.99
}
```

HTTP Methods & CRUD Mapping

HTTP Method	CRUD Operation	Description	Status Codes
GET	Read	Retrieve resource(s)	200 (OK), 404 (Not Found)
POST	Create	Create new resource	201 (Created), 400 (Bad Request)
PUT	Update/Replace	Replace entire resource	200 (OK), 204 (No Content)
PATCH	Update/Modify	Partially update resource	200 (OK), 204 (No Content)
DELETE	Delete	Remove resource	200 (OK), 204 (No Content)

REST Request/Response Examples

1. GET All Books

Request

```
GET /books HTTP/1.1
Host: api.example.com
Accept: application/json
```

Response (200 OK)

```
[
  {
    "id": 1,
    "title": "Clean Code",
    "author": "Robert Martin"
  },
  {
    "id": 2,
    "title": "Designing Data-Intensive Applications",
    "author": "Martin Kleppmann"
  }
]
```

2. GET Single Book

Request

```
GET /books/1 HTTP/1.1
Host: api.example.com
```

Response (200 OK)

```
{
  "id": 1,
  "title": "Clean Code",
  "author": "Robert Martin",
  "price": 39.99
}
```

3. POST Create Book

Request

```
POST /books HTTP/1.1
Host: api.example.com
Content-Type: application/json

{
  "title": "RESTful Web APIs",
  "author": "Leonard Richardson",
  "price": 34.99
}
```

Response (201 Created)

```
HTTP/1.1 201 Created
Location: /books/3
```

4. PUT Update Book

Request

```
PUT /books/1 HTTP/1.1
Host: api.example.com
Content-Type: application/json

{
  "title": "Clean Code (Revised)",
  "author": "Robert C. Martin",
  "price": 45.99
}
```

Response (200 OK)

5. DELETE Book

Request

```
DELETE /books/1 HTTP/1.1
Host: api.example.com
```

Response (204 No Content)

JSON Format

```
{
  "id": 1,
  "name": "JSON Example",
  "isActive": true,
  "tags": ["rest", "json"],
  "metadata": {
    "createdAt": "2023-01-01"
  }
}
```

Advantages:

- Lightweight
- Native JavaScript support
- Easy to parse
- Human-readable

REST Best Practices

1. **Resource Naming**:
    - Use nouns (not verbs): `/books` not `/getBooks`
    - Plural form for collections: `/users` not `/user`
2. **HTTP Status Codes**:
    - 200 - Successful GET/PUT/PATCH
    - 201 - Resource created
    - 204 - No content (successful DELETE)
    - 400 - Bad request
    - 401 - Unauthorized
    - 404 - Not found

- 500 - Server error

3. **Versioning**:

```
GET /v1/books
Accept: application/vnd.example.v1+json
```

4. **Filtering/Sorting**:

```
GET /books?author=Martin&sort=-price
```

5. **Consistent response (like JSend format)**:

```
{
  "status": "success",
  "data": {
    "id": 1,
    "title": "Clean Code",
    "author": "Robert Martin",
    "price": 39.99
  }
}
```

```
{
  "status": "error",
  "message": "Book not found"
}
```

REST API in Express.js

Setup & Installation

```
npm init -y
npm install express mysql2
```

Database Setup (MySQL)

```
CREATE DATABASE bookdb;
USE bookdb;

CREATE TABLE books (
  id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
  author VARCHAR(100) NOT NULL,
  subject VARCHAR(100),
  price DECIMAL(10,2) NOT NULL
);

INSERT INTO books (name, author, subject, price) VALUES
('Clean Code', 'Robert Martin', 'Programming', 39.99),
('Designing Data-Intensive Applications', 'Martin Kleppmann', 'Database', 49.99);
```

Server Implementation (`app.js`)

```
const express = require("express");
const mysql = require("mysql2");
```

```
const app = express();
app.use(express.json());
// convert incoming req body contents into json format and make it available as req.body
```

```
// Create MySQL connection pool
const pool = mysql.createPool({
  host: "localhost",
  user: "root",
  password: "yourpassword",
  database: "bookdb",
});
```

```
// 1. GET all books
app.get("/books", (req, res) => {
  const sql = "SELECT * FROM books";
  pool.query(sql, (err, results) => {
    if (err) {
      console.error("Database error:", err);
      return res.status(500).json({ error: "Database error" });
    }
    return res.json(results);
  });
});
```

```
// 2. GET single book by ID
app.get("/books/:id", (req, res) => {
  const sql = "SELECT * FROM books WHERE id = ?";
  pool.query(sql, [req.params.id], (err, results) => {
    if (err) {
      console.error("Database error:", err);
```

```
      return res.status(500).json({ error: "Database error" });
    }
    if (results.length === 0) {
      return res.status(404).json({ error: "Book not found" });
    }
    return res.json(results[0]);
  });
});
```

```
// 3. POST create new book
app.post("/books", (req, res) => {
  const { name, author, subject, price } = req.body;

  if (!name || !author || !price) {
    return res.status(400).json({ error: "Missing required fields" });
  }

  const sql =
    "INSERT INTO books (name, author, subject, price) VALUES (?, ?, ?, ?)";
  pool.query(sql, [name, author, subject, price], (err, results) => {
    if (err) {
      console.error("Database error:", err);
      return res.status(500).json({ error: "Database error" });
    }

    // Fetch the newly created book
    const fetchSql = "SELECT * FROM books WHERE id = ?";
    pool.query(fetchSql, [results.insertId], (err, newBook) => {
      if (err) {
        console.error("Database error:", err);
        return res.status(500).json({ error: "Database error" });
      }

      return res.status(201).json(newBook[0]);
```

```
    });
  });
});
```

```
// 4. PUT update entire book
app.put("/books/:id", (req, res) => {
  const { name, author, subject, price } = req.body;

  const sql =
    "UPDATE books SET name = ?, author = ?, subject = ?, price = ? WHERE id = ?";
  pool.query(
    sql,
    [name, author, subject, price, req.params.id],
    (err, results) => {
      if (err) {
        console.error("Database error:", err);
        return res.status(500).json({ error: "Database error" });
      }

      if (results.affectedRows === 0) {
        return res.status(404).json({ error: "Book not found" });
      }

      return res.json({ id: req.params.id, name, author, subject, price });
    }
  );
});
```

```
// 5. DELETE book
app.delete("/books/:id", (req, res) => {
  const sql = "DELETE FROM books WHERE id = ?";
  pool.query(sql, [req.params.id], (err, results) => {
```

```
    if (err) {
      console.error("Database error:", err);
      return res.status(500).json({ error: "Database error" });
    }

    if (results.affectedRows === 0) {
      return res.status(404).json({ error: "Book not found" });
    }

    return res.status(204).end();
  });
});
```

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

Middleware in Express.js

Core Concepts

- **Definition**: Functions that execute during the request-response cycle

- **Key Characteristics**:
  - Access to `req`, `res` objects
  - Can modify requests/responses
  - Must call `next()` to pass control (or end the cycle)

Types of Middleware

1. **Application-level Middleware**

```
app.use((req, res, next) => {
  console.log("Time:", Date.now());
  next();
});
```

2. **Route-level Middleware**

```
app.get("/protected", authMiddleware, (req, res, next) => {
  // ...
});
```

3. **Built-in Middleware**

```
app.use(express.json()); // Parse JSON bodies
app.use(express.urlencoded({ extended: true })); // Parse form data
```

4. **Third-party Middleware**

```
const cors = require("cors");
app.use(cors()); // Enable CORS
```

Middleware Examples

1. **Logging Middleware**

```
app.use((req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next();
});
```

2. **Request Timing Middleware**

```
app.use((req, res, next) => {
  const start = Date.now();
  res.on("finish", () => {
    console.log(`Request took ${Date.now() - start}ms`);
  });
  next();
});
```

Express Router

Why Use Router?

- Organize routes into separate files
- Create modular, maintainable code
- Apply middleware to specific route groups
- Mount multiple routers with different paths

Router Implementation

1. **Create Router File (`routes/books.js`)**

```
const express = require("express");
const router = express.Router();

// GET /books
router.get("/", (req, res) => {
  res.send("All books");
});

// GET /books/:id
router.get("/:id", (req, res) => {
  res.send(`Book ${req.params.id}`);
});

module.exports = router;
```

2. **Mount in Main App (`app.js`)**

```
const booksRouter = require("./routes/books");
app.use("/books", booksRouter);
```

Project Structure

```
project/
    ├── app.js
    ├── middleware/
    │   ├── auth.js
    │   └── logger.js
    └── routes/
        ├── books.js
        └── users.js
```

Express Security

Authentication (AuthN)

Purpose: Verify user identity ("Who are you?")

Authentication Methods

Type	Examples
Knowledge-Based	Username/password, security questions, PINs
Possession-Based	SMS/email codes, authenticator apps, smart cards, security tokens
Biometric	Fingerprint, facial recognition, iris scan
Multi-Factor (MFA)	Combination of 2+ methods (e.g., password + SMS code)

Authorization (AuthZ)

Purpose: Control access rights ("What can you do?")

Key Concepts

Concept	Description	Banking System Example
Principal	Authenticated user/account	Currently logged-in bank employee
Authority	Fine-grained permission for specific actions	`VIEW_BALANCE`, `PROCESS_WITHDRAWAL`
Role	Group of authorities assigned to a position	`ROLE_CASHIER`, `ROLE_MANAGER`

Role Based Security - Bank Example

Role	Authorities
Cashier	View balances, process deposits/withdrawals, print receipts
Branch Manager	Approve loans, override transactions, manage staff
Loan Officer	Process loan applications, verify documents, approve credit lines

Express - Security Implementation

Database Setup

```
CREATE TABLE users (
  id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
  email VARCHAR(100) UNIQUE NOT NULL,
  passwd VARCHAR(255) NOT NULL,  -- Store hashed passwords only
  mobile VARCHAR(15),
  addr TEXT,
  enabled BOOLEAN DEFAULT TRUE,
  role VARCHAR(20) DEFAULT 'ROLE_CUSTOMER'
);

-- Sample data
INSERT INTO users (name, email, passwd, role) VALUES
('Admin User', 'admin@bank.com', '$2a$10$xJw...', 'ROLE_ADMIN'),
('Branch Manager', 'manager@bank.com', '$2a$10$yPw...', 'ROLE_MANAGER'),
('Cashier', 'cashier@bank.com', '$2a$10$zQv...', 'ROLE_CASHIER');
```

User Registration (Sign-Up)

```
router.post("/signup", (req, res) => {
  const { name, email, passwd, mobile, addr } = req.body;

  // Basic validation ...

  // Hash password
  const encPasswd = bcrypt.hashSync(passwd, 10);
  const role = req.body.role || "ROLE_CUSTOMER";

  db.query(
    `INSERT INTO users
     (name, email, passwd, mobile, addr, role)
     VALUES (?, ?, ?, ?, ?, ?)`,
    [name, email, encPasswd, mobile, addr, role],
    (err, result) => {
      if (err) {
        // Handle duplicate email
        if (err.code === "ER_DUP_ENTRY") {
          return res.status(400).json({
            success: false,
            message: "Email already registered",
          });
        }
        return serverError(res, "Registration failed");
      }

      res.status(201).json({
        success: true,
        message: "Registration successful",
        userId: result.insertId,
      });
    }
  );
});
```

Authentication (Sign-In)

```
router.post("/signin", (req, res) => {
  const { email, passwd } = req.body;

  // Validate input ...

  // Check user exists
  db.query(
    "SELECT * FROM users WHERE email = ? AND enabled = 1",
    [email],
    (err, results) => {
      if (err)
        return res.status(500).json({
          status: "error",
          error: err,
        });

      const user = results[0];

      // Verify password
      if (!bcrypt.compareSync(passwd, user.passwd))
        return authError(res, "Invalid credentials");

      // Successful login -- Ideal reponse: create token and return
      return res.json({
        status: "success",
        message: "Login successful",
        user,
      });
    }
  );
});
```

JWT (JSON Web Token)

JWT Overview

- **Definition**: Open standard (RFC 7519) for secure information exchange
- **Key Characteristics**:
  - Compact (URL-safe string)
  - Self-contained (includes all necessary information)
  - Digitally signed (verifiable integrity)
- **Common Uses**:
  - Authorization (primary use case)
  - Secure information exchange
  - Stateless authentication

JWT Structure/Components

- JWT format: header.payload.signature

Part	Content Type	Description	Example
Header	JSON	Token type + signing algorithm	`{"alg":"HS256","typ":"JWT"}`
Payload	JSON	Claims (user data) + standard fields (`iss`, `exp`, `sub`, etc.)	`{"userId":123,"role":"admin"}`
Signature	Binary	Cryptographic signature	HMAC-SHA256 output

Configuration

```
const jwt = require("jsonwebtoken");
const JWT_SECRET = process.env.JWT_SECRET || "your-256-bit-secret"; // Always use env vars in production
const TOKEN_EXPIRY = "7d"; // 7 days
```

Token creation

- To be created after successful login.

```
function createToken(user) {
  const payload = {
    userId: user.id,
    role: user.role,
    // Add other necessary claims (but avoid sensitive data)
  };
  return jwt.sign(payload, JWT_SECRET, { expiresIn: TOKEN_EXPIRY });
}
```

- Standard claims in JWT:
  - `iss` (issuer)
  - `sub` (subject)
  - `aud` (audience)
  - `exp` (expiration)

Token verification

- To be verified on each request

```
// 2. Token Verification
function verifyToken(token) {
  try {
    return jwt.verify(token, JWT_SECRET);
  } catch (err) {
    console.error("JWT verification failed:", err.message);
    return null;
  }
}
```

Authentication Middleware

- Ensure that token verification is done for each request

```
function jwtAuth(req, res, next) {
  // Skip auth for public routes
  const publicRoutes = ["/users/signin", "/users/signup"];
  if (publicRoutes.includes(req.path)) {
    req.user = { userId: 0, role: "ROLE_ANONYMOUS" };
    return next();
  }

  // Extract token
  const authHeader = req.headers.authorization;
  if (!authHeader || !authHeader.startsWith("Bearer ")) {
    return res.status(401).json({ error: "Authorization header missing" });
  }

  const token = authHeader.split(" ")[1];
  const decoded = verifyToken(token);
  if (!decoded) {
    return res.status(401).json({ error: "Invalid or expired token" });
  }

  // Attach user to request
  req.user = decoded;
  next();
}
```

Best practice: Promises & Explicit algorithm check

```
// Enhanced token verification
function verifyToken(token) {
  return new Promise((resolve) => {
    jwt.verify(
      token,
      JWT_SECRET,
      {
        algorithms: ["HS256"], // Explicit algorithm check
      },
      (err, decoded) => {
        if (err) {
          console.error("JWT error:", err.name);
          resolve(null);
        } else {
          resolve(decoded);
        }
      }
    );
  });
}

// Usage in middleware
const decoded = await verifyToken(token);
```

File Upload with Multer

Core Concepts

- **Multer**: Node.js middleware for handling `multipart/form-data`
- **Key Features**:
  - Disk storage engine (saves to filesystem)
  - Memory storage engine (buffers in RAM)
  - File filtering and size limits

- Multiple file upload support

Implementation

```
const multer = require("multer");
const upload = multer({ dest: "uploads/" });
```

```
// upload file: from frontend -- <input name='icon' type='file'/>
router.post("/", upload.single("icon"), (req, res) => {
  const newFileName = req.file.filename + ".jpg";
  fs.rename(
    req.file.path,
    `${req.file.destination}${newFileName}`,
    (err) => {}
  );
  const { title, ...rest } = req.body;
  // ...
});
```

Java Script

XMLHttpRequest and fetch()

XMLHttpRequest (XHR)

- The original API for making HTTP requests in JavaScript
- Supported by all browsers (including very old ones)

Basic Usage

```
const xhr = new XMLHttpRequest();

xhr.open("GET", "https://api.example.com/data", true); // async=true

xhr.onload = function () {
  if (xhr.status >= 200 && xhr.status < 300) {
    console.log(JSON.parse(xhr.responseText));
  } else {
    console.error("Request failed:", xhr.statusText);
  }
};

xhr.onerror = function () {
  console.error("Network error occurred");
};

xhr.send();
```

Advanced Features

- **Progress tracking**:

```
xhr.upload.onprogress = (e) => {
  if (e.lengthComputable) {
    const percent = (e.loaded / e.total) * 100;
    console.log(`${percent}% uploaded`);
  }
};
```

- **Request cancellation**:

```
xhr.abort(); // Cancel the request
```

- **Synchronous requests** (avoid in production):

```
xhr.open("GET", "url", false); // sync=false
```

fetch() API

- Modern replacement for XHR
- Promise-based (cleaner syntax)
- Not supported in IE

Basic Usage

```
fetch("https://api.example.com/data")
  .then((response) => {
    if (!response.ok) throw new Error(`HTTP error! status: ${response.status}`);
    return response.json();
  })
  .then((data) => console.log(data))
  .catch((error) => console.error("Error:", error));
```

Advanced Features

- **POST request**:

```

```

```
fetch("https://api.example.com/data", {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
  },
  body: JSON.stringify({ key: "value" }),
});
```

- **Request cancellation** (using AbortController):

```
const controller = new AbortController();
const signal = controller.signal;

fetch("url", { signal })
  .then((response) => response.json())
  .catch((err) => {
    if (err.name === "AbortError") {
      console.log("Request aborted");
    }
  });

// Abort the request
controller.abort();
```

- **Streaming responses**:

```
fetch("large-file.txt").then((response) => {
  const reader = response.body.getReader();
  // Process stream chunks
});
```

Prime Differences

Feature	XMLHttpRequest	fetch()
Syntax	Callback-based	Promise-based
Response Handling	Manual JSON parsing	Built-in `.json()` method
Error Handling	Manual status code checking	Requires explicit `response.ok` check
Cancellation	`.abort()` method	Requires `AbortController`
Progress Events	Built-in support	No native support (use streams)
Browser Support	All browsers (including IE)	Modern browsers (no IE)

Modern Best Practices

1. **Prefer** `fetch()` for new projects
2. **Always handle errors**:

```
// Good fetch() error handling
async function getData() {
  try {
    const response = await fetch("url");
    if (!response.ok) throw new Error("Network response was not ok");
    const data = await response.json();
    // data processing
  } catch (error) {
    console.error("Fetch error:", error);
    throw error; // Re-throw for caller
  }
}
```

3. **Use async/await** for cleaner code:

```
async function fetchData() {
  const response = await fetch("url");
  const data = await response.json();
  console.log(data);
}
```

4. **For file uploads with progress**:

```
// XHR is still better for upload progress
function uploadWithProgress(file) {
  const xhr = new XMLHttpRequest();
  xhr.upload.onprogress = (e) => {
    console.log(`Uploaded ${e.loaded} of ${e.total} bytes`);
  };
  xhr.open("POST", "upload-url", true);
  xhr.send(file);
}
```

5. **Parallel requests**:

```
Promise.all([
  fetch("url1").then((r) => r.json()),
  fetch("url2").then((r) => r.json()),
]).then(([data1, data2]) => {
  console.log(data1, data2);
});
```

Web Storage: localStorage and sessionStorage

Key Characteristics

Feature	localStorage	sessionStorage
Persistence	Survives browser restart	Cleared when tab closes
Scope	Shared across all tabs/windows	Limited to current tab
Storage Limit	~5-10MB per domain	~5-10MB per domain
Accessibility	Accessible from any window	Only in originating tab
Expiration	Manual removal required	Automatic on tab close

Basic Operations

- **Setting Data**

```
// localStorage - persists until explicitly cleared
localStorage.setItem("username", "john_doe");
localStorage.setItem("user", JSON.stringify({ id: 1, name: "John" }));

// sessionStorage - cleared when tab closes
sessionStorage.setItem("session_token", "abc123xyz");
sessionStorage.setItem("temp_data", JSON.stringify({ page: 1, filter: "new" }));
```

- **Getting Data**

```
// Get string values
const username = localStorage.getItem("username"); // "john_doe"
```

```
// Get parsed objects
const user = JSON.parse(localStorage.getItem("user")); // { id: 1, name: 'John' }

// sessionStorage example
const token = sessionStorage.getItem("session_token"); // "abc123xyz"
```

- **Removing Data**

```
// Remove single item
localStorage.removeItem("username");

// Clear all items (for current domain)
sessionStorage.clear();
```

- **Storage Event Listeners (Advanced Use)**

```
// Listen for changes across tabs/windows
window.addEventListener("storage", (event) => {
  console.log(`Key changed: ${event.key}`);
  console.log(`Old value: ${event.oldValue}`);
  console.log(`New value: ${event.newValue}`);
  console.log(`Storage area: ${event.storageArea}`);
});
```

- **Iterating Through all Items**

```
// Loop through all localStorage items
for (let i = 0; i < localStorage.length; i++) {
  const key = localStorage.key(i);
  const value = localStorage.getItem(key);
```

```
  console.log(`${key}: ${value}`);
}

// Object-style iteration (modern approach)
Object.entries(localStorage).forEach(([key, value]) => {
  console.log(`${key}: ${value}`);
});
```

Best Practices

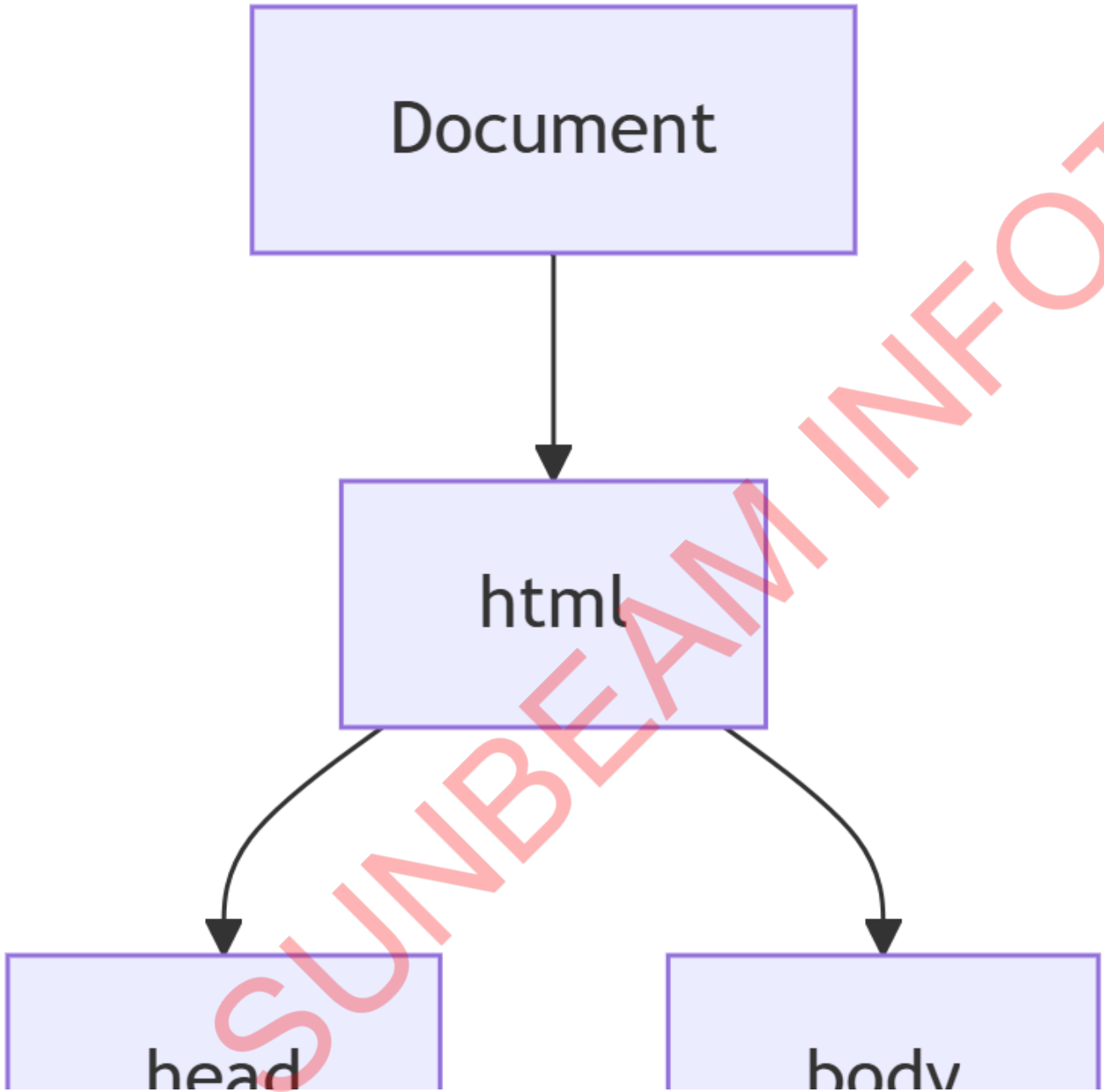
1. **Never store sensitive data**:
   - Avoid passwords, credit card numbers
   - JWTs should have short expiration
2. **Validate stored data**:

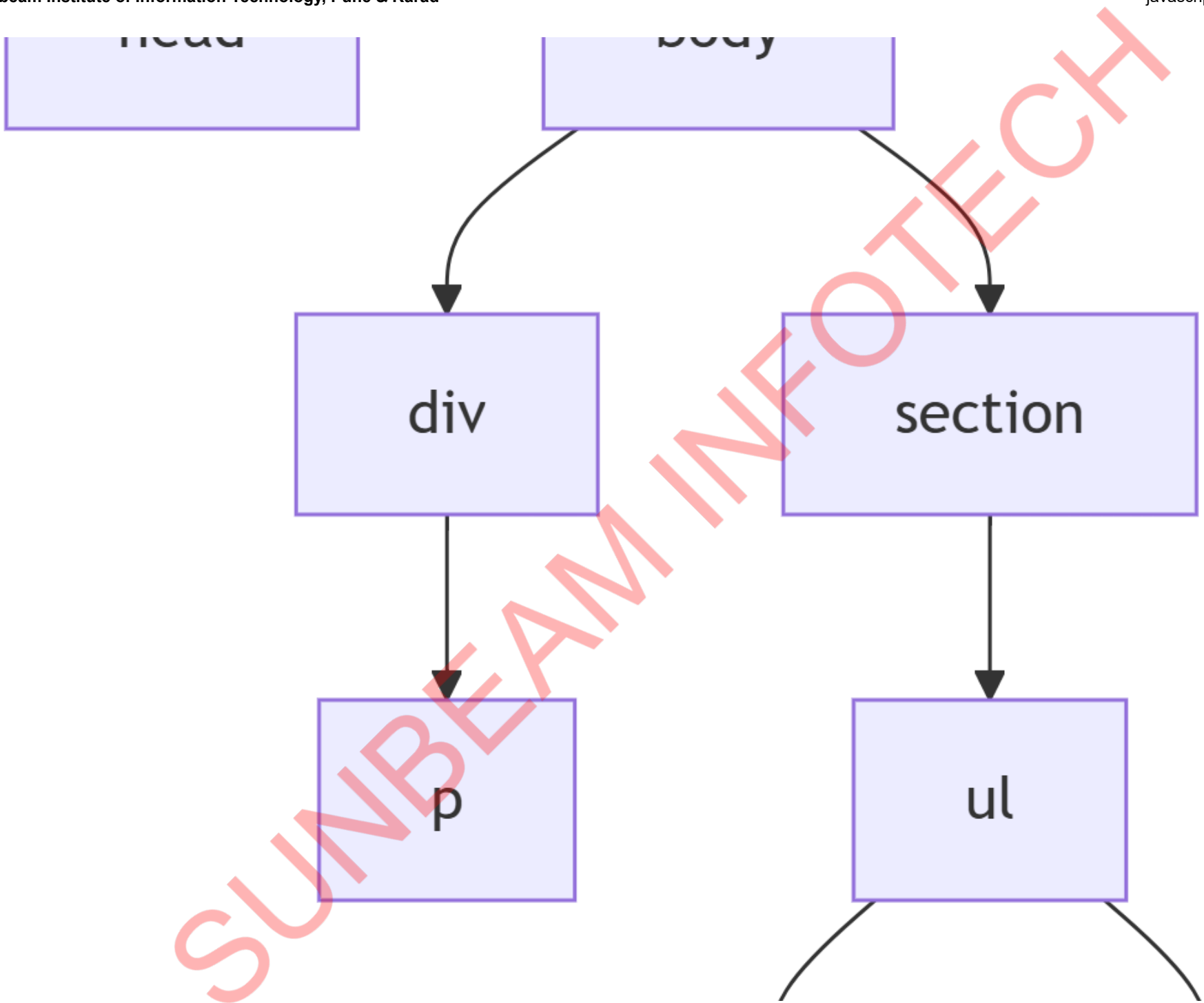
```
function getSafeUser() {
  try {
    const user = JSON.parse(localStorage.getItem("user"));
    return user && user.id ? user : null;
  } catch {
    return null;
  }
}
```

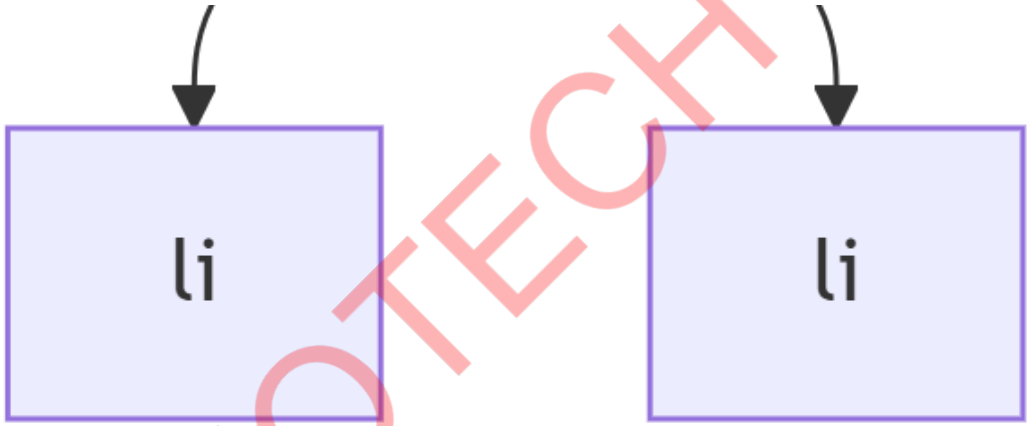
Common Vulnerabilities

- **XSS Attacks**: Malicious scripts can access storage
- **Data Tampering**: Users can modify storage directly
- **Overwriting**: Concurrent tabs may overwrite data

DOM (Document Object Model)







What is the DOM?

- **Tree-like representation** of HTML documents
- **Programming interface** for web documents
- **Enables dynamic interaction** with:
  - HTML elements
  - CSS styles
  - Event handling
  - Content updates

DOM Node Types

Node Type	Example	NodeType Value
Element Node	`<div>`, `<p>`, `<a>`	1
Attribute Node	`class="header"`	2
Text Node	`Hello World` (inside elements)	3
Comment Node	`<!-- comment -->`	8

DOM Traversal

Basic Navigation

```
// Parent/child relationships
document.body.parentNode; // <html>
document.body.firstChild; // First child node
document.body.lastChild; // Last child node
document.body.childNodes; // NodeList of all children

// Sibling navigation
element.nextSibling;
element.previousSibling;
```

Modern Methods

```
// Get children (element nodes only)
parentElement.children;
```

```
// Querying
document.getElementById("header");
document.getElementByName("header");
document.getElementByTagName("div");
document.getElementByClassName("container");
```

```
document.querySelector(".main-content"); // Returns first match
document.querySelectorAll("button"); // Returns NodeList
```

Element Selectors

- Element Selectors e.g. h1, div
- Multiple Element Selectors e.g. "h1, h2"
- Id Selectors e.g. #para1, p#para2
- Class Selectors e.g. .h1para
- Descendent Selectors e.g "div a"
- Child Selectors e.g. body > div
- Attribute Selectors e.g. input[type="submit"], a[href="products.html"]
- Universal Selectors e.g. *

DOM Manipulation

Creating Elements

```
// Create new element
const newDiv = document.createElement("div");
newDiv.textContent = "Hello World";

// Add to DOM
document.body.appendChild(newDiv);

// Insert at specific position
parent.insertBefore(newDiv, referenceElement);
```

Modifying Elements

- **Modifying contents**

```
// Content
element.innerHTML = "<strong>Warning!</strong>"; // Parses HTML
element.innerText = "Plain text only"; // Safer alternative
```

- **Modifying attributes**

```
// Attributes
element.attributeName = attrValue;
element.setAttribute("data-id", "123");
const id = element.getAttribute("data-id");
element.attributeName;
element.removeAttribute("data-id");
```

- **Modifying style**

```
// Styles
element.style.color = "red";
element.style.fontSize = "16px"; // Use camelCase
```

- **Modifying style class**

```
// Classes
element.className = "myclass";
element.classList.add("active");
element.classList.remove("hidden");
element.classList.toggle("visible");
```

Event Handling

Event Listener

```
button.addEventListener("click", (event) => {
  console.log("Button clicked!", event.target);
});
```

Common Events

Event Type	Description	Example Elements
`click`	Mouse click	Buttons, links
`submit`	Form submission	`<form>`
`keydown`	Keyboard key pressed	Input fields
`mouseover`	Mouse hovers over element	Any visible element
`change`	Element contents changed	input (text change), select (selection change)

Examples

1. **Create dynamic list**:

```
const data = ["Apple", "Banana", "Orange"];
const list = document.getElementById("fruits");
data.forEach((item) => {
  const li = document.createElement("li");
  li.innerText = item;
  list.appendChild(li);
});
```

2. **Form validation**:

```
document.querySelector("form").addEventListener("submit", (e) => {
  const input = document.getElementById("email");
  if (!input.value.includes("@")) {
    e.preventDefault();
    alert("Invalid email!");
  }
});
```

Real DOM vs Virtual DOM

Real DOM

What real DOM?

- The **actual representation** of your webpage in the browser
- A **tree structure** where each HTML element is a node
- **Directly linked** to what you see in the browser

Key Characteristics:

- **Heavyweight**: Every element is a full object with many properties
- **Slow updates**: Changing any part requires re-rendering affected elements
- **Immediate**: Changes appear instantly in the browser

How it Works:

```
// Real DOM manipulation example
const element = document.getElementById("myElement");
element.style.color = "red"; // Immediate visual change
```

Performance Problem:

```
// Inefficient for frequent updates
for (let i = 0; i < 1000; i++) {
  document.body.innerHTML += `<div>Item ${i}</div>`; // Triggers 1000 reflows!
}
```

Virtual DOM

What is Virtual DOM?

- A **lightweight copy** of the Real DOM
- Created and managed by libraries like React
- Pure **JavaScript object representation** of the UI

Example Virtual DOM

```
// Simplified Virtual DOM object example
{
  type: 'div',
  props: {
    className: 'container',
    children: [
      {
        type: 'h1',
        props: {
          children: 'Hello World'
        }
      }
```

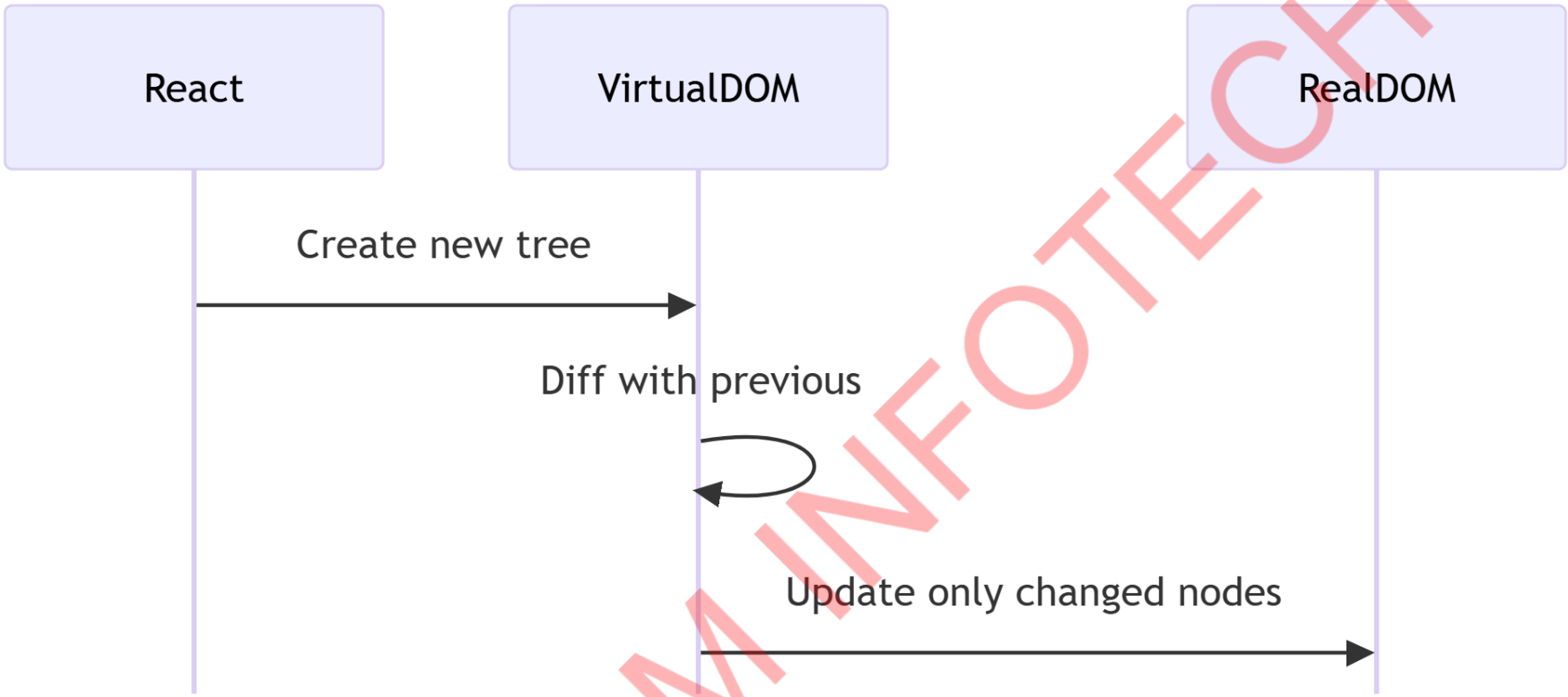
```
      ]
    }
  }
```

Key Characteristics:

- **Lightweight**: Just JavaScript objects (no browser overhead)
- **Fast comparisons**: Can diff changes efficiently
- **Batch updates**: Minimizes Real DOM operations

How it Works?

1. **Initial Render**: Creates Virtual DOM tree
2. **On Changes**:
    - Creates new Virtual DOM tree
    - Compares with previous version (diffing)
    - Updates only changed parts in Real DOM (reconciliation)



Performance Benefit:

```
// React handles this efficiently
function MyComponent() {
  return (
    <div>
      {Array(1000)
        .fill()
        .map((e, i) => (
          <div key={i}>Item {i}</div>
        ))}
    </div>
```

```
    );
}
// Only updates what changed between renders
```

Key Differences

Feature	Real DOM	Virtual DOM
Update Speed	Slow (direct manipulation)	Fast (batched updates)
Memory Usage	High (browser objects)	Low (JavaScript objects)
Update Process	Immediate	Diff + batch update
Typical Usage	Vanilla JS/jQuery	React/Vue/Other frameworks
Developer Control	Full control	Managed by framework

React Virtual DOM

1. **Performance Optimization**:
    - Minimizes expensive Real DOM operations
    - Only updates what actually changed
2. **Declarative Programming**:

```
// Tell React WHAT you want (not HOW to do it)
function Counter() {
  const [count, setCount] = useState(0);
  return <button onClick={() => setCount((c) => c + 1)}>{count}</button>;
}
```

3. **Cross-Platform**:
    - Same Virtual DOM concept works for:
        - Web (React DOM)
        - Mobile (React Native)
        - Even 3D scenes (React Three Fiber)

Common Misconceptions

❌ "Virtual DOM is faster than direct DOM manipulation"
☑ **Truth**: Virtual DOM is faster *for complex UIs with frequent updates*

❌ "Virtual DOM eliminates reflows/repaints"
☑ **Truth**: It minimizes them through batching

❌ "Virtual DOM is a React invention"
☑ **Truth**: The concept existed before (e.g., in Elm), but React popularized it

When to use?

Use Real DOM When:

- Building simple static websites
- Need direct element control
- Working without frameworks

Use Virtual DOM When:

- Building complex, dynamic UIs
- Frequent state changes
- Using component-based architecture

Practical Example

Problem: Update a list of 1000 items

Real DOM Approach (Slow):

```
const list = document.getElementById("list");
list.innerHTML = ""; // Wipes entire list
items.forEach((item) => {
  list.innerHTML += `<li>${item}</li>`; // 1000 DOM writes
});
```

Virtual DOM Approach (Fast):

```
function List({ items }) {
  return (
    <ul>
      {items.map((item) => (
        <li key={item.id}>{item.text}</li>
      ))}
    </ul>
  );
}
// React only updates changed <li> elements
```