# MERN ALL EXAMPLES

### UNIT - 1 WEB - D WITH MERN NOTES

#### 1.1 Fundamentals of Good Website Design

- **Types of Core Purpose (Page 1):**
  1. Describing Expertise
  2. Building Your Reputation
  3. Generating Leads
  4. Sales and After Care
- **Types of Key Factors (Page 1):**
  - Consistency
  - Colours
  - Typography
  - Imagery
  - Simplicity
  - Functionality
  - User Experience (UX)
- **Guidelines for Colours (Page 2):**
  - Communicate messages and evoke emotions.
  - Palette should fit the brand.
  - Limit to less than 5 colours. Complementary colours work well.
  - Pleasing combinations increase engagement.
- **Guidelines for Typography (Page 2):**
  - Visual interpretation of the brand voice.
  - Must be legible.
  - Use a maximum of 3 different fonts.
- **Guidelines for Imagery (Page 2):**
  - Includes photos, illustrations, videos, graphics.
  - Should be expressive, capture company spirit, and embody brand personality.
  - High-quality images convey professionalism and credibility.
- **Additional Design Principles (Types):**
  - Navigation

- F-Shaped Pattern Reading
- Visual Hierarchy
- Content
- Grid-Based Layout
- Load Time
- Mobile Friendly

**1.2 Web Page and Website**

- **Types of Embedded Resources in a Web Page (Page 5):**
  - Style information (CSS for look-and-feel).
  - Scripts (JavaScript for interactivity).
  - Media (images, sounds, videos).

**1.3 Web Application**

- **Example Components for Web App Functionality (Page 7):**
  - Web server (manages client requests)
  - Application server (completes tasks)
  - Database (stores information)
- **Types of App Development Approaches (Page 8):**
  - **Native App:** Built for a specific platform/device, installed, can use device-specific hardware (GPS, camera). Can operate offline.
  - **Web App:** Accessed via browser, typically needs internet.
  - **Hybrid App:** Combines approaches. Installs like native, built with web tech, can use device APIs. Typically doesn't work offline. Shares navigation elements with web apps.

**1.4 Client-Server Architecture**

- **Two Key Factors (Types - Page 9):**
  1. Server: Provides requested services.
  2. Client: Requests services.
- **Examples of Server Types (Page 10):**
  - Mail Servers
  - File Servers
  - Web Servers
- **Components of Client-Server Setup (Types - Page 10):**
  1. Workstations (Clients)
  2. Servers

3. Networking Devices (e.g., hubs, repeaters, bridges)

- **How Client-Server Works (Example Steps - Page 11):**
  1. User enters URL. Browser sends request to DNS server.
  2. DNS server finds and returns the IP address of the web server.
  3. Browser sends HTTP/HTTPS request to the web server's IP.
  4. Server transmits the essential website files.
  5. Browser processes files and displays the website.
- **Tiers of Architecture (Types - Page 12-14):**
  - **1-Tier:** All layers (presentation, business, data) on a single device.
  - **2-Tier:** Client-side stores UI; server houses database. Example: Online ticket reservation.
  - **3-Tier:** Middleware between client and server. Layers: Presentation (client), Application (middleware/server logic), Database (server).
    - **Diagram (Conceptual from Page 19 of MERN notes / Page 5 of Unit 1 notes):**

      ```
      [USER INTERFACE (CLIENT)] <---request---> [MIDDLEWARE (BL & DL)] <---response---> [DATABASE (SERVER)]
      ```

  - **N-Tier (Multi-tier):** Scaled form. Each function (presentation, application processing, data management) can be an isolated layer.
- **Difference: Client-Server vs. Peer-to-Peer (Table - Page 14):**

| Feature | Client-Server Architecture | Peer-to-Peer Architecture |
|---|---|---|
| Roles | Specific clients and servers. | No differentiation; peers are equal. |
| Data Management | Centralized. | Each peer has its own data/applications. |
| Objective | Disseminate knowledge/exchange relevant data. | Encourage peer connectivity, relationships. |
| Data Provision | Only in response to a request. | Peers can request and provide services. |
| Scalability | Suitable for small and large networks. | Better for limited users (e.g., <10 devices). |

### 1.5 Introduction to MERN Stack

- **MERN Stack Development Diagram (Conceptual from Page 6):**

```
[React JS (HTML/CSS, JavaScript, Bootstrap)] --- (Front-end Development)
    |
    v
[Node JS web server] --- (Back-end Development) --- [Express Web Framework]
```

```
                                                    |
                                                    v
                                                [Mongoose]
                                                    |
                                                    v
                                            [MongoDB (Database Management)]
```

- **Other Stacks Mentioned (Types - Page 17):**
  - LAMP: Linux, Apache, MySQL, PHP
  - MEAN: MongoDB, ExpressJS, AngularJS, NodeJS
- **MERN Stack Architecture (3-Tier) Diagram (Conceptual from Page 7):**

```
Client Machine (Browser with React Application)
    <-- HTTP Requests/Responses -->
Back-End Server (Node.js)
    - Express (API Logic)
    - MongoDB Driver
        <-- Database Queries -->
MongoDB (Database)
```

- **Detailed Components of MERN Stack (Types with characteristics):**
  - **MongoDB (Page 20):**
    - Data stored in BSON (Binary JavaScript Object Notation) format.
  - **Express.js (Page 21)**
  - **React.js (Page 21)**
  - **Node.js (Page 22)**

### 2.1 Introduction to HTML

- **Basic HTML Structure (Syntax/Code - "Basic HTML Page on VS Code Jan 19"):**

```
<!DOCTYPE html> <!-- Tells the browser you are using HTML5 -->
<html> <!-- Root element of an HTML page -->
<head> <!-- Contains meta-information about the HTML document (not displayed) --
>
<title>My 1st Page</title> <!-- Title shown in browser tab or window title bar -
->
</head>
<body> <!-- Contains the visible page content -->
<p>Hello world!</p> <!-- A paragraph element -->
</body>
</html>
```

- **Example HTML Element ("HTML tag Jan 17"):** `<p>This is a paragraph.</p>`
- **Examples of Empty elements:** `<br>`, `<img>`, `<input>`
- **Attribute Syntax:** `name="value"`
- **Example Attributes:**
  - `<html lang="en">`
  - `<meta charset="UTF-8">`
  - `<meta name="viewport" content="width=device-width, initial-scale=1.0">`

### 2.3 Basics of XHTML

- **Key Differences/Rules from HTML (Types of rules):**
  1. `<!DOCTYPE>` is mandatory.
  2. `xmlns` attribute in `<html>` is mandatory.
  3. Elements must be properly nested. (e.g., `<b><i>text</i></b>` is correct, `<b><i>text</b></i>` is not).
  4. Elements must always be closed. (e.g., `<p>text</p>`, `<br />`).
  5. Attribute names must be in lowercase.
  6. Attribute values must be quoted.
  7. Attribute minimization is forbidden. (e.g., use `checked="checked"`).
  8. Documents must have one root element.
- **XHTML `<!DOCTYPE>` Example (Code):**

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

- **XHTML `<html>` with `xmlns` Example (Code):** `<html xmlns="http://www.w3.org/1999/xhtml">`
- **XHTML Document Structure Example (Code - Jan 27):**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Title of a Web Page</title>
</head>
<body>
Contents to be displayed on the web page...
</body>
</html>
```

### 2.4 HTML Lists

- **Types of Lists (Page 23):**
  1. Unordered List (`<ul>`)
  2. Ordered List (`<ol>`)
  3. Description List (`<dl>`)
- **Unordered List Example (Code - Page 25):**

```
<h2>Shopping List</h2>
<ul>
<li>bread</li>
<li>coffee beans</li>
<li>milk</li>
<li>butter</li>
</ul>
```

- **Ordered List `type` Attribute Examples:** `a`, `A`, `i`, `I`
- **Ordered List `start` Attribute Example:** `start="4"`
- **Ordered List Example (Code - Page 27):**

```
<h2>Cooking Instructions</h2>
<ol>
<li>Gather ingredients</li>
<li>Mix ingredients together</li>
<li>Place ingredients in a baking dish</li>
</ol>
```

- **Description List Components (Syntax):** `<dl>`, `<dt>`, `<dd>`
- **Description List Example (Code - Page 30):**

```
<h2>List of Single Names with Single Values</h2>
<dl>
<dt>Bread</dt>
<dd>A baked food made of flour.</dd>
<dt>Coffee</dt>
<dd>A drink made from roasted coffee beans.</dd>
</dl>
```

- **Nesting Lists Example (Code - Page 33):**

```
<h4>Nesting List Example</h4>
<ol>
<li>Chapter One
<ol>
```

```
<li>Section One</li>
<li>Section Two</li>
</ol>
</li>
<li>Chapter Two</li>
</ol>
```

- **Summary of List Tags (Table - Page 32):**

| Tag | Description |
|---|---|
| `<ul>` | Defines an unordered list |
| `<ol>` | Defines an ordered list |
| `<li>` | Defines a list item |
| `<dl>` | Defines a description list |
| `<dt>` | Defines a name in a description list |
| `<dd>` | Describes the value in a description list |

### 2.5 HTML Tables

- **Basic Table Structure Tags (Syntax):** `<table>`, `<tr>`, `<td>`, `<th>`
- **Example Table Attributes:** `border="1"`, `cellpadding`, `cellspacing`, `colspan`, `rowspan`, `bgcolor`, `background`, `bordercolor`, `width`, `height`.
- **Table Grouping Elements (Syntax):** `<caption>`, `<thead>`, `<tbody>`, `<tfoot>`
- **Example Table with `<th>`, `colspan`, `rowspan` (Code - adapted from Page 43 & 45):**

```
<table border="1">
<thead>
<tr>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
</tr>
</thead>
<tbody>
<tr>
<td rowspan="2">Row 1 & 2, Cell 1</td>
<td>Row 1, Cell 2</td>
<td>Row 1, Cell 3</td>
</tr>
<tr>
<td>Row 2, Cell 2</td>
<td>Row 2, Cell 3</td>
```

```
</tr>
</tbody>
<tfoot>
<tr>
<td colspan="3">Row 3, Spanning all 3 Columns</td>
</tr>
</tfoot>
</table>
```

### 2.6 HTML Forms

- **`<form>` Tag Syntax (Page 35):**
  `<form action="Script URL" method="GET|POST"> ...form elements... </form>`
- **Form Attribute Examples (Page 35):**
  - `action`: URL of the script
  - `method`: `GET` or `POST`
  - `target`: `_blank`, `_self`, `_parent`
  - `enctype`: `application/x-www-form-urlencoded`, `multipart/form-data`
- **Types of Text Input Controls (Syntax - Page 36-38):**
  - `<input type="text">` (Attributes: `name`, `value`, `size`, `maxlength`, `placeholder`)
  - `<input type="password">`
  - `<textarea>` (Attributes: `name`, `rows`, `cols`)
- **Checkbox Control (Syntax - Page 38):**
  `<input type="checkbox">` (Attributes: `name`, `value`, `checked`)
- **Radio Button Control (Syntax - Page 39):**
  `<input type="radio">` (Attributes: `name`, `value`, `checked`)
- **Select Box Control (Syntax - Page 39-40):**
  `<select>` (Attributes: `name`, `size`, `multiple`)
  `<option>` (Attributes: `value`, `selected`, `label`)
- **File Select Box (Syntax - Page 40):**
  `<input type="file">` (Attributes: `name`, `accept`)
- **Types of Button Controls (Syntax - Page 41):**
  - `<input type="submit">`
  - `<input type="reset">`
  - `<input type="button">`
  - `<input type="image">`
  - `<button>` element

- **Hidden Controls (Syntax - Page 42):**
  `<input type="hidden">`
- **Basic Form Example (Code - Page 15-16, adapted):**

```
<body>
<form action="submit_form.php" method="post">
Name:
<input type="text" name="Name" size="15" maxlength="30" placeholder="Name">
<br><br>
Email:
<input type="email" name="email" size="15" maxlength="30" placeholder="Email">
<br><br>
Gender:
<input type="radio" name="gender" value="male"> Male
<input type="radio" name="gender" value="female"> Female<br><br>
Date of Birth:
<input type="date" name="date_of_birth"><br><br>
<input type="submit" value="Submit">
</form>
</body>
```

### 2.8 XML (Extensible Markup Language)

- **XML Example (Code):**

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

### 3.2 CSS Core Syntax

- **CSS Rule Syntax (Handwritten notes, "Internal CSS"):**

```
selector {
property1: value1;
property2: value2;
}
```

- **CSS Rule Example (Code):**

```
h1 {
color: red; /* Declaration 1: property 'color', value 'red' */
```

```
background-color: yellow; /* Declaration 2 */
}
```

### 3.3 Types of CSS (How to apply CSS)

- **Inline CSS Example (Code):**
  `<h1 style="color: red; font-size: 24px;">Welcome to our class</h1>`
- **Internal CSS Example (Code):**

```
<head>
<title>CSS</title>
<style>
h1 {
color: red;
background-color: yellow;
}
</style>
</head>
<body>
<h1>Welcome to our class</h1>
<h1>Computer class</h1>
</body>
```

- **External CSS Example (Code - "External CSS May 21"):**
  - `abc.css` file:

```
h1 {
color: white;
background-color: blue;
text-align: center;
height: 2cm;
line-height: 2cm; /* for vertical centering */
}
```

  - `abc.html` file:

```
<head>
<title>CSS</title>
<link rel="stylesheet" type="text/css" href="abc.css">
</head>
<body>
<h1>Welcome to our class</h1>
</body>
```

### 3.4 CSS Text Properties

- **Example Text Properties (with example values):**
  - `color: red;`
  - `background-color: yellow;`
  - `text-align: left | right | center | justify;`
  - `text-decoration: none | underline | overline | line-through;`
  - `text-transform: none | capitalize | uppercase | lowercase;`
  - `text-indent: 5px;`
  - `letter-spacing: 2px;`
  - `word-spacing: 10px;`
  - `line-height: 60px | 1.5;`
  - `text-shadow: 5px 4px red;` or `10px 10px 30px blue;`
- **CSS Text Properties Example (Code):**

```
h1 {
color: red;
background-color: yellow;
text-align: left;
text-decoration: overline;
text-transform: uppercase;
text-indent: 10px;
letter-spacing: 15px;
word-spacing: 10px;
line-height: 60px;
text-shadow: 10px 10px 30px blue;
}
```

### 3.5 CSS Box Model

- **Diagram (Conceptual based on handwritten notes "CSS Box Model May 26"):**

```
+---------------------------------------+
| Margin                                |
|   +-----------------------------------+
|   | Border                            |
|   |   +---------------------------+
|   |   | Padding                   |
|   |   |   +-------------------+   |
|   |   |   | Content           |   |
|   |   |   +-------------------+   |
|   |   +---------------------------+
```

```
|   +-----------------------------------+
+---------------------------------------+
```

- **Components of Box Model (Innermost to Outermost - Types):**
  1. Content
  2. Padding
  3. Border
  4. Margin
- **Box Model CSS Example (Code - "div span May 28, 29"):**

```
<style>
div {
height: 100px;
width: 100px;
background: red;
margin-bottom: 20px; /* Space between div and span */
padding: 10px;
border: 10px solid black;
outline: 10px solid blue; /* Outline is drawn outside the border */
}
span {
height: 100px; /* May not apply if display is inline */
width: 100px; /* May not apply if display is inline */
background: green;
display: block; /* To make height/width apply for span */
}
</style>
<body>
<div>Hey, I am box</div>
<span></span>
</body>
```

- **Output Depiction from example:**
  - `div`: `outline` (blue), `border` (black), `padding` (red color shows here), `content` (text).
  - `span`: `background` (green).
  - `margin-bottom` shows distance between the two elements.

### 3.6 Normal Flow Box Layout

- **Types of Elements in Normal Flow:**
  1. Block-level Elements (Examples: `<div>`, `<h1>`-`<h6>`, `<p>`, `<form>`, `<ul>`, `<ol>`, `<li>`)
  2. Inline Elements (Examples: `<span>`, `<a>`, `<img>`, `<strong>`, `<em>`, `<i>`)
- **CSS `display` Property Examples:** `block`, `inline`, `inline-block`, `none`

- **Normal Flow Example (Code - "Block Vs Inline Elements June"):**
  - `index.html`:

```
<head>
<title>Basic layout in CSS</title>
<link rel="stylesheet" href="style.css">
</head>
<body>
<h1>This is a block-level element.</h1>
<p>This is also a block-level element.</p>
<span>This an inline-level element.</span>
<span>This is another inline-level element.</span>
</body>
```

  - `style.css`:

```
h1, p { /* Block elements */
display: block;
background-color: red; /* For visibility */
}
span { /* Inline elements */
display: inline;
background-color: lightblue; /* For visibility */
padding: 5px;
margin: 5px;
}
```

### 3.7 Other Properties (like list, tables)

- **Styling Lists with CSS (Property examples):**
  - `list-style-type`: `none`, `disc`, `circle`, `square` (for `<ul>`); `decimal`, `lower-roman`, `upper-alpha` (for `<ol>`)
  - `list-style-image`: `url('image.gif')`
  - `list-style-position`: `inside`, `outside`
- **Styling Lists CSS Example (Code):**

```
ul.custom {
list-style-type: square;
padding-left: 20px;
}
ol.custom {
list-style-type: lower-roman;
margin-left: 30px;
}
```

- **Styling Tables with CSS (Property examples):**
  - `border`: `1px solid black` (applied to `table`, `th`, `td`)
  - `border-collapse`: `collapse`, `separate`
  - `padding`: (applied to `th`, `td`)
  - `text-align`: (applied to cells)
  - `background-color`: (applied to `table`, `tr`, `th`, `td`)
  - `:hover` pseudo-class
- **Styling Tables CSS Example (Code):**

```
table.styled {
width: 100%;
border-collapse: collapse;
}
table.styled th, table.styled td {
border: 1px solid black;
padding: 8px;
text-align: left;
}
table.styled th {
background-color: #f2f2f2;
}
table.styled tr:hover {
background-color: #ddd;
}
```

### 3.8 XSLT (Extensible Stylesheet Language Transformations)

- **XSLT Core Components (Types):** XSLT, XPath, XSL-FO (Optional)
- **Basic XSLT Example (Conceptual Code):**
  - XML data:

```
<book>
<title>Learning XSLT</title>
<author>John Doe</author>
</book>
```

  - Transformed HTML output:

```
<div>
<h1>Learning XSLT</h1>
<p>By: John Doe</p>
</div>
```

- XSLT stylesheet rule (snippet):

```
<xsl:template match="book">
<div>
<h1><xsl:value-of select="title"/></h1>
<p>By: <xsl:value-of select="author"/></p>
</div>
</xsl:template>
```

### 4.2 Basic Syntax & Embedding JS in HTML

- **Embedding JS in HTML (Type I - Common Syntax/Code):**

```
<script type="text/javascript"> // or just <script>
document.write("Hello World!");
alert('Welcome to JS');
</script>
```

- **Embedding JS in HTML (Type II - Older Syntax/Code):**

```
<script language="JavaScript">
document.write("Education for you");
</script>
```

- **`<script>` tag attribute examples:** `language="JavaScript"`, `type="text/javascript"`
- **External JS File Syntax:** `<script src="myscript.js"></script>`
- **JS Comments Syntax:** `// Single-line comment`, `/* Multi-line comment */`

### 4.3 Variables & Data Types

- **Variable Declaration Syntax:**
  - `var variableName;`
  - `let variableName;`
  - `const variableName = value;`
- **Primitive Data Types (with code examples):**
  - **String:** `let name = "Alice"; console.log(typeof name); // "string"`
  - **Number:** `let age = 30; let price = 19.99; console.log(typeof age); // "number"` (Special values: `Infinity`, `-Infinity`, `NaN`)
  - **BigInt:** `let bigNumber = 123...n; console.log(typeof bigNumber); // "bigint"`
  - **Boolean:** `let isActive = true; console.log(typeof isActive); // "boolean"`
  - **Undefined:** `let x; console.log(x); // undefined console.log(typeof x); // "undefined"`
  - **Null:** `let car = null; console.log(typeof car); // "object"`

- **Symbol (ES6+):** `let sym = Symbol("id"); console.log(typeof sym); // "symbol"`

- **Object Type (Reference Type - with code example):**
  - **Object:** `let person = {firstName: "John", lastName: "Doe"}; console.log(typeof person); // "object"`
- **Other Reference Types:** Array, Function

### 4.4 Literals

- **Examples of Literals:**
  - `100` (Number literal)
  - `3.14` (Number literal)
  - `"Hello"` (String literal)
  - `'World'` (String literal)
  - `true` (Boolean literal)
  - `false` (Boolean literal)
  - `null` (Null literal)
  - `{name: "John"}` (Object literal)
  - `[1, 2, 3]` (Array literal)
  - `234` and `456` in `Console.log(234+456)` (evaluates to `690`)
  - `"A"` in `Console.log("A" + "A")` (evaluates to `"AA"`)

### 4.5 Operators

- **Arithmetic Operators (Table):**

| Operator | Description | Example | Result |
|---|---|---|---|
| `+` | Addition | `x=2, y=2; x+y` | `4` |
| `-` | Subtraction | `x=5, y=2; x-y` | `3` |
| `*` | Multiplication | `x=5, y=4; x*y` | `20` |
| `/` | Division | `15/5 -> 3`, `5/2 -> 2.5` | |
| `%` | Modulus (remainder) | `5%2 -> 1`, `10%8 -> 2` | |
| `++` | Increment | `x=5; x++` (now x is 6) | |
| `--` | Decrement | `x=5; x--` (now x is 4) | |

- **Assignment Operators (Table):**

| Operator | Example | Same As |
|---|---|---|
| `=` | `x = y` | `x = y` |

| Operator | Example | Same As |
|---|---|---|
| `+=` | `x += y` | `x = x + y` |
| `-=` | `x -= y` | `x = x - y` |
| `*=` | `x *= y` | `x = x * y` |
| `/=` | `x /= y` | `x = x / y` |
| `%=` | `x %= y` | `x = x % y` |

- **Comparison Operators (Table with examples):**

| Operator | Description | Example |
|---|---|---|
| `==` | Equal to (value, type conversion) | `5 == "5" -> true` |
| `===` | Strictly equal to (value and type) | `5 === "5" -> false` |
| `!=` | Not equal to (value, type conversion) | `5 != "5" -> false` |
| `!==` | Strictly not equal to (value and type) | `5 !== "5" -> true` |
| `>` | Greater than | `5 > 8 -> false` |
| `<` | Less than | `5 < 8 -> true` |
| `>=` | Greater than or equal to | `5 >= 8 -> false` |
| `<=` | Less than or equal to | `5 <= 8 -> true` |

- **Logical Operators (Table with examples):**

| Operator | Description | Example |
|---|---|---|
| `&&` | Logical AND | `(x < 10 && y > 1)` |
| `!` | Logical NOT | `!(x == y)` |
| ` | | ` |

- **String Concatenation Example (Code):**
  `let greeting = "Hello" + " " + "World!"; // "Hello World!"`
- **Conditional (Ternary) Operator Example (Code):**
  `let voteable = (age < 18) ? "Too young" : "Old enough"; // "Old enough" if age is 20`

### 4.6 Functions

- **Function Declaration Syntax:**

```
function functionName(parameter1, parameter2) {
// code to be executed
return result; // optional
}
```

- **Function Invocation Syntax:** `functionName(argument1, argument2);`
- **Basic Function Example (Code - "Function Syntax in Java Script Feb 18"):**

```
<script>
function myIntro() {
document.write('I am Shyam.<br>');
document.write('I am an Animation Specialist.<br>');
}
myIntro(); // Calling the function
myIntro(); // Calling again
</script>
<!-- O/P: I am Shyam.
I am an Animation Specialist.
I am Shyam.
I am an Animation Specialist. -->
```

- **Function with Parameters Example (Code - "Function with Parameter Feb 19"):**

```
function sum(a, b) {
document.write(a + b + "<br>");
}
sum(10, 20); // O/P: 30
sum(20, 40); // O/P: 60
```

- **Function Expressions (Anonymous Functions) Example (Code):**

```
const greet = function(name) {
return "Hello " + name;
};
console.log(greet("Alice")); // "Hello Alice"
```

- **Arrow Functions (ES6+) Examples (Code):**

```
const add = (a, b) => a + b;
console.log(add(5, 3)); // 8

const greetByName = name => "Hello " + name; // Single parameter
console.log(greetByName("Bob")); // "Hello Bob"
```

- **Closures Example (Code - PYQ Q1(e)):**

```
function outerFunction(outerVariable) {
return function innerFunction(innerVariable) {
console.log("Outer Variable: " + outerVariable);
console.log("Inner Variable: " + innerVariable);
console.log("Sum: " + (outerVariable + innerVariable));
```

```
}
}
const newFunction = outerFunction(10);
newFunction(5);
// Output:
// Outer Variable: 10
// Inner Variable: 5
// Sum: 15
```

### 4.7 Objects

- **Types of Creating Objects:**
  1. Object Literal
  2. `new Object()`
  3. Constructor Functions
  4. ES6 Classes
- **Object Literal Example (Code - "Object Mar 5"):**

```
let person = {
firstName: "Rohit", // property
lastName: "Sharma",
age: 60, // property
greet: function() { // method
console.log("Hello, my name is " + this.firstName);
}
};
console.log(person.firstName); // "Rohit"
person.greet(); // "Hello, my name is Rohit"
document.write(person.age); // 60 on page
```

- **`new Object()` Example (Code):**

```
let car = new Object();
car.make = "Toyota";
car.model = "Camry";
```

- **Constructor Functions Example (Code):**

```
function Person(first, last) {
this.firstName = first;
this.lastName = last;
}
let myFather = new Person("John", "Doe");
```

- **ES6 Classes Example (Code):**

```
class Rectangle {
constructor(height, width) {
this.height = height;
this.width = width;
}
get area() {
return this.calcArea();
}
calcArea() {
return this.height * this.width;
}
}
const square = new Rectangle(10, 10);
console.log(square.area); // 100
```

- **Accessing Properties Syntax:** `objectName.propertyName`, `objectName["propertyName"]`
- **Types of Objects (Handwritten "Object Feb 23"):**
  1. Built-in Objects (Native Objects): e.g., `String`, `Number`, `Boolean`, `Date`, `Math`, `Array`, `RegExp`.
  2. Host Objects: e.g., `window`, `document`, `location`, `history`, `NodeList`, `HTMLElement`.
  3. User-defined Objects.

### 4.8 Arrays

- **Types of Creating Arrays:**
  1. Array Literal
  2. `new Array()`
- **Array Literal Examples (Code):**

```
// Handwritten "let subject = ['Math', 'Physics'];"
let fruits = ["Apple", "Banana", "Orange"];
let numbers = [10, 20, 30, 40, 50];
let mixed = [10, "Harry", "Sarah", true, null];
```

- **`new Array()` Example (Code):**

```
// Handwritten "let subject = new Array('Math', 'Physics');"
let colors = new Array("Red", "Green", "Blue");
```

- **Accessing Array Elements Example (Code):** `console.log(fruits[0]); // "Apple"`
- **Modifying Array Elements Example (Code):** `fruits[1] = "Mango";`
- **Array `length` Property Example:** `console.log(fruits.length); // 3`
- **Common Array Methods (Types):** `push()`, `pop()`, `shift()`, `unshift()`, `splice()`, `slice()`, `concat()`, `join()`, `indexOf()`, `forEach()`, `map()`, `filter()`, `reduce()`.

- **Iterating through an Array (for loop) Example (Code):**

```
let nums = [10, 20, 30, 40, 50];
for (let i = 0; i < nums.length; i++) {
console.log(nums[i]);
}
```

- **Iterating through an Array (forEach) Example (Code):**

```
nums.forEach(function(num) {
console.log(num);
});
```

- **Nested Arrays Example (Code - "Nested Array in Java Script Mar 3"):**

```
// Handwritten "animals = ["cat", "dog", ["hawk", "eagle"]];"
let matrix = [
[1, 2, 3],
[4, 5, 6],
[7, 8, 9]
];
console.log(matrix[1][1]); // 5
// Handwritten "console.log(cities[3][1]) -> 'b'"
// var cities = ["Delhi", "Mumbai", "Chennai", ["a", "b", "c"]];
```

### 4.9 Built-in Objects

- **Math Object Method Examples (Code):**

```
console.log(Math.PI); // 3.141592653589793
console.log(Math.sqrt(16)); // 4
console.log(Math.random());
console.log(Math.floor(4.7)); // 4
console.log(Math.ceil(4.2)); // 5
console.log(Math.round(4.5)); // 5
```

- **Date Object Method Examples (Code):**

```
let now = new Date();
console.log(now.toString());
console.log(now.getFullYear());
console.log(now.getMonth()); // 0-11
console.log(now.getDate());
console.log(now.getHours());
```

- **String Object Method Examples (Code):**

```
let text = "Hello World";
console.log(text.length); // 11
console.log(text.toUpperCase()); // "HELLO WORLD"
console.log(text.indexOf("World")); // 6
console.log(text.slice(0, 5)); // "Hello"
```

- **JSON Object Method Examples (Code):**

```
let myObj = { name: "John", age: 30 };
let myJSON = JSON.stringify(myObj); // '{"name":"John","age":30}'
let parsedObj = JSON.parse(myJSON); // { name: "John", age: 30 }
```

- **Other Built-in Object Types:** `Number`, `Boolean`, `RegExp`

### 4.10 JavaScript Form Programming

- **Types of Accessing Form Elements:**
  - `document.getElementById('elementId')`
  - `document.forms['formName'].elements['elementName']`
  - `document.querySelector()` or `document.querySelectorAll()`
- **Getting/Setting Values Syntax:** `element.value`, `element.checked` (boolean)
- **Form Validation Example (Code):**

```
<form name="myForm" onsubmit="return validateForm()">
Name: <input type="text" name="fname">
<input type="submit" value="Submit">
</form>
<script>
function validateForm() {
let x = document.forms["myForm"]["fname"].value;
if (x == "") {
alert("Name must be filled out");
return false; // Prevents form submission
}
return true; // Allows form submission
}
</script>
```

### 4.11 Intrinsic Event Handling

- **Common Event Attributes (Syntax/Types with examples):**
  - `onclick`: `<button onclick="alert('Button Clicked!')">Click Me</button>`
  - `onload`: `<body onload="displayTime()">`
  - `onunload`

- `onchange`
- `onmouseover`
- `onmouseout`
- `onmousedown`
- `onmouseup`
- `onfocus`
- `onblur`
- `onsubmit`: (used on `<form>`)
- `onkeydown`, `onkeypress`, `onkeyup`

- **Event Handling Example (Code - "button onclick="hello()" Mar 13"):**

```
<head>
<title>Event Example</title>
<script>
function hello() {
document.write("Hello Everyone");
}
</script>
</head>
<body>
<button onclick="hello()">Click Me</button> <!-- O/P on click: Page shows "Hello Everyone" -->
</body>
```

- **`addEventListener` Example (Code):**

```
document.getElementById("myBtn").addEventListener("click", function() {
alert("Button clicked via addEventListener!");
});
```

### 4.12 Modifying Element Style

- **Accessing Style Property Syntax:** `document.getElementById("elementId").style.propertyName = "newValue";`
  - (CSS properties accessed using camelCase, e.g., `backgroundColor` for `background-color`)
- **Modifying Style Example (Code - "Modifying Element Style using JS Mar 14"):**

```
<h1 id="hid">JS</h1>
<button onclick="changeStyle()">Change Color</button>
<script>
function changeStyle() {
document.getElementById("hid").style.color = "red";
```

```
document.getElementById("hid").style.fontSize = "50px";
document.getElementById("hid").style.backgroundColor = "yellow";
}
</script>
```

### 4.13 Document Trees (DOM)

- **DOM Tree Diagram (Conceptual sketch from handwritten "Document Tree Mar 17" on Page 41):**

```
     Root (html)
     /       \
Element(head) Element(body)
    |           |
Element(title) Element(h1)
    |           |
Text("Tree")  Text("Intro")
```

- **DOM Node Types (Table - "DOM Node Types Mar 19" on Page 42):**

| Constant | Node Value | Description |
|---|---|---|
| `ELEMENT_NODE` | 1 | An element such as `<h1>` or `<div>` |
| `TEXT_NODE` | 3 | The actual Text of an Element or Attribute. |
| `COMMENT_NODE` | 8 | A Comment node. |
| `DOCUMENT_NODE` | 9 | A Document node. |
| `DOCUMENT_TYPE_NODE` | 10 | A `<!DOCTYPE html>` node. |
| `DOCUMENT_FRAGMENT_NODE` | 11 | A lightweight container for DOM nodes. |
| (Note: `ATTRIBUTE_NODE` (value 2) also exists) | | |

- **Categories/Levels of DOM (Diagram description from image on Page 43, "Categories of DOM Feb 26"):**
  - **DOM Core:** Specifies a generic model for viewing & manipulating a marked up document as a tree structure.
  - **DOM HTML:** Specifies an extension to the core DOM for use with HTML & represents DOM Level 0 with capabilities for manipulating all of that HTML element objects.
  - **DOM CSS:** Specifies the interfaces necessary to manipulate CSS rules programmatically.
  - **DOM Events:** It adds event handling to the DOM.
  - **DOM XML:** It specifies an extension to the core DOM for use with XML.

- **DOM Levels (Types):** Level 0, Level 1 (Core, HTML), Level 2 (Events, Style, Traversal and Range), Level 3, Level 4+ (Living Standard).
- **Accessing HTML Elements in JS (Methods - PYQ Q2(a)):**
  - `document.getElementById(id)`
  - `document.getElementsByTagName(tagName)`
  - `document.getElementsByClassName(className)`
  - `document.querySelector(cssSelector)`
  - `document.querySelectorAll(cssSelector)`

### 4.14 ECMAScript (ES)

- **Versions (Types):** ES1-ES3, ES4 (Abandoned), ES5, ES6, ES7 onwards.
- **Key ES6 (ECMAScript 2015) Features (Types):**
  - `let` and `const`
  - Arrow Functions (`=>`)
  - Template Literals ( `` `${variable}` `` )
    - **Template Literals Example (Code - handwritten "Template Literals Mar 23"):**

```
const fname = "Vinod";
const lname = "Thapa";
let message = `My name is ${fname} ${lname}. My lucky number is ${5+5}.`;
console.log(message);
// ReactDOM.render( `<h1>My name is ${fname} ${lname}</h1> <p>my lucky
number is ${5+5}</p>`, document.getElementById("root"));
```

  - Default Parameters
  - Rest and Spread Operators (`...`)
  - Destructuring Assignment
  - Classes
  - Modules (`import`/`export`)
  - Promises
  - New Collections: `Map`, `Set`, `WeakMap`, `WeakSet`
  - `for...of` loop
- **ES5 vs. ES6 (ECMAScript 2015) - Comparison Table (Example of feature differences):** (Selected rows)

| Feature | ES5 (ECMAScript 2009) | ES6 (ECMAScript 2015) |
|---|---|---|
| Variable Declaration | `var` (function-scoped/global) | `let`, `const` (block-scoped) |

| Feature | ES5 (ECMAScript 2009) | ES6 (ECMAScript 2015) |
|---|---|---|
| Function Syntax | Standard `function` declarations/expressions. | Arrow Functions (`=>`), concise syntax, lexical `this`. |
| String Handling | Concatenation with `+`. Multi-line via `\n` or `+`. | Template Literals (`` ` ``) with `${expression}`. |
| Default Parameters | Manual check for `undefined`. | `function greet(name = 'Guest')` |
| Modules | No built-in (used CommonJS, AMD). | Native Modules: `import` and `export`. |
| Asynchronous Ops | Primarily callbacks ("callback hell"). | Promises. |

---

**UNIT - 2: REACTJS**

**1. INTRODUCTION TO REACTJS**

- **Types of Reasons to Use React (Printed Notes pg 6):**
  1. Easy creation of dynamic applications
  2. Improved performance (Virtual DOM)
  3. Reusable components
  4. Unidirectional data flow
  5. Dedicated tools for easy debugging
- **Types of Key Features of React (Printed Notes pg 2):**
  1. JSX (JavaScript XML)
  2. Components
  3. Virtual DOM
  4. One-way data-binding (Unidirectional Data Flow)
  5. High performance

**2. GETTING STARTED WITH REACT APP**

- **Command to Check Node.js version:** `node -v`
- **Commands to Install `create-react-app` and Create App (Printed Notes pg 8):**
  - `npm install -g create-react-app` (Older method)
  - `npx create-react-app my-react-app` (Recommended)
  - `npx create-react-app project-name`
- **Alternative (Vite) Setup Commands (Handwritten Notes pg 1):**
  - `npm create vite@4.1.0`

    - Project name: `react-app`
    - Select a framework: `React`
    - Select a variant: `Javascript`
- **Command to Navigate to project directory:** `cd project-name` (or `cd react-app`)
- **Command to Install dependencies (if needed, e.g., with Vite):** `npm install` (or `npm i`)
- **Command to Start development server:**
    - `npm start` (for `create-react-app`)
    - `npm run dev` (for Vite projects)
- **Example Project Structure (Simplified for `create-react-app`):**
    - `node_modules/`
    - `public/`
        - `index.html` (Main HTML page, e.g., `<div id="root"></div>`)
    - `src/`
        - `App.css` (CSS for App component)
        - `App.js` (or `App.jsx`) (Main application component)
        - `index.css` (Global CSS)
        - `index.js` (or `index.jsx`) (Entry point, renders App to DOM)
- **Example `index.js` (or `main.jsx` for Vite - Handwritten Notes pg 1):**

```
import React from 'react'
import ReactDOM from 'react-dom/client' // client for React 18+
import App from './App' // or App.jsx
import './index.css'

ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
)
```

- **Example Functional Component (`Message.jsx` - Handwritten Notes pg 1):**

```
// Message.jsx
function Message() {
  return <h1>Helloworld this is Message</h1>;
}
export default Message;
```

- **Example Usage in `App.jsx` (Handwritten Notes pg 1):**

```jsx
// App.jsx
import Message from './Message';

function App() {
  return (
    <div>
      <Message />
    </div>
  );
}
export default App;
```

- **Other Project Files (Types):** `.gitignore`, `package.json`, `package-lock.json` (or `yarn.lock`)

### 3. TEMPLATING USING JSX

- **JSX Transpilation Diagram (Printed Notes pg 4):**
  `Modern JavaScript (JSX) --> BABEL Transpiler --> Browser-compatible JavaScript (Regular JS Object)`
- **Embedding Expressions in JSX (Syntax and Example - Printed Notes pg 2):**

```jsx
// JavaScript
// HTML {JS}
render() {
  const name = "React Developer";
  const sum = 10 + 20;
  return (
    <div>
      <h1>Hello, {name}!</h1> {/* Variable expression */}
      <p>10 + 20 = {sum}</p> {/* Arithmetic expression */}
      <p>My lucky number is {Math.random() * 10}</p> {/* Function call
expression */}
    </div>
  );
}
```

- **Embedding JS Expressions in JSX (Handwritten Notes pg 5 "Template Literals"):**

```jsx
// const fname = "Kritika";
// const lname = "Kalihar";
// ReactDOM.render(
//   <> {/* React Fragment */}
//     <h1>My name is {fname + " " + lname}</h1>
//     {/* or <h1>My name is {`${fname} ${lname}`}</h1> using template literals
*/}
//     <p>my lucky number is {5+5}</p>
```

```
//   </>,
//   document.getElementById("root")
// );
// O/P: My name is Kritika Kalihar. my lucky number is 10.
```

- **JSX Attributes Example (camelCase, expressions):**

```
const myClass = "container";
const element = <div className={myClass} id="main-div">Content</div>;
```

- **JSX Transpilation to `React.createElement()` (Conceptual):**
  - JSX: `<MyButton color="blue" shadowSize={2}>Click Me</MyButton>`
  - JS: `React.createElement(MyButton, {color: 'blue', shadowSize: 2}, 'Click Me')`
- **JSX Single Root Element (Syntax examples):** `<div>...</div>` or `<>...</>` or `<React.Fragment>...</React.Fragment>`

### 4. CLASSES USING JSX (CLASS COMPONENTS)

- **ES5 Component Syntax (Using `React.createClass` - deprecated) (Printed Notes pg 7):**

```
// ES5
var MyComponentES5 = React.createClass({
  render: function() {
    return (
      <h3>Hello Simplilearn (ES5)</h3>
    );
  }
});
```

- **ES6 Class Component Syntax (Printed Notes pg 7):**

```
// ES6
class MyComponentES6 extends React.Component {
  render() {
    return (
      <h3>Hello Simplilearn (ES6)</h3>
    );
  }
}
```

- **Example of a Class Component (Printed Notes pg 17, simplified):**

```
import React from 'react'; // or import React, { Component } from 'react';

class Greeting extends React.Component { // or class Greeting extends Component
  render() {
```

```
    return <h1>Welcome to {this.props.name}</h1>; // Accessing props
  }
}
export default Greeting;
```

(Handwritten Notes pg 18 "Class Component Profile.jsx" is a similar example)

**5. COMPONENTS**

- **Types of Components (Printed Notes pg 15, 22):**
  1. Functional Components (Stateless Functional Components)
  2. Class Components (Stateful Class Components)
- **Functional Component Example (Printed Notes pg 17, Handwritten pg 16):**

```
// Functional Component
function Greeting(props) {
  return <h1>Welcome to {props.name}</h1>;
}
// Or using arrow function
// const Greeting = (props) => <h1>Welcome to {props.name}</h1>;
export default Greeting;
```

(Handwritten Notes pg 16 `Profile.jsx` (functional) is a similar example)

- **Class Component Example (Printed Notes pg 17, Handwritten pg 18 "Class component Profile.jsx"):**

```
import React from 'react';

class UserProfile extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      location: "New York"
    };
  }
  render() {
    return (
      <div>
        <h1>Hello, {this.props.name} from {this.state.location}!</h1>
      </div>
    );
  }
}
export default UserProfile;
```

- **Differences between Class and Functional Components (Table - Printed Notes pg 22):**

| Feature | Class Components | Functional Components |
| --- | --- | --- |
| State | Can hold/manage state (`this.state`). | Traditionally no state; now with `useState` Hook. |
| Simplicity | More complex. | Simple, easy to understand. |
| Lifecycle methods | Has all lifecycle methods. | Traditionally no lifecycle; now with `useEffect` Hook. |
| `this` keyword | Uses `this` to access props/state. | No `this`. Props passed as arguments. |
| Syntax | ES6 class extending `React.Component`. | JavaScript function. |
| Reusability | Can be reused. | Can be reused. |

- **Rendering a Component Example:**

```
const element = <Welcome name="Sara" />; // Using a functional or class component
ReactDOM.render(element, document.getElementById('root'));
```

- **Composing Components Example:**

```
function App() {
  return (
    <div>
      <Welcome name="Sara" />
      <Welcome name="Cahal" />
      <Welcome name="Edite" />
    </div>
  );
}
```

- **Comments in React (JSX) (Syntax/Examples - Printed Notes pg 12):**
  - Single-line: `{/* This is a single-line comment in JSX */}`
  - Multi-line:

```
{/*
  This is a
  multi-line comment
  in JSX
*/}
```

- **Embedding Two or More Components into One (Example - Printed Notes pg 21):**

```
class Simple extends React.Component {
  render() {
```

```
    return (<h1>Simplilearn</h1>);
  }
}

class App extends React.Component {
  render() {
    return (
      <div>
        <h1>Hello</h1>
        <Simple /> {/* Embedding Simple component */}
      </div>
    );
  }
}
ReactDOM.render(<App />, document.getElementById('index'));
```

### 6. STATE AND PROPS

- **How to pass props (Example - Printed Notes pg 20, Handwritten pg 16):**
  - Parent Component (`App.js`):

```
import Profile from './Profile';

function App() {
  return (
    <div>
      {/* Passing string prop */}
      <Profile text="Hello Props" />
      {/* Passing object prop (Handwritten Notes pg 16) */}
      <Profile data={{ name: 'Peter', detail: 'Profile data' }} />
    </div>
  );
}
```

  - Child Component (`Profile.js` - Functional):

```
function Profile(props) {
  return (
    <div>
      <h1>{props.text}</h1>
      {props.data && ( // Check if data prop exists
        <h2>Name: {props.data.name}, Detail: {props.data.detail}</h2>
      )}
    </div>
```

```
    );
  }
```

- **Initializing State in Class Components (Example - Printed Notes pg 19):**

```
class Clock extends React.Component {
  constructor(props) {
    super(props); // Always call super(props) in constructor
    this.state = { date: new Date(), message: "Welcome to Simplilearn" };
  }
  // ...
}
```

- **Accessing state (Syntax):** `this.state.propertyName`
  - Example: `<h1>{this.state.message}</h1>;`
- **Updating State (Example - Printed Notes pg 19, 20, Handwritten pg 19):**

```
class Profile extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: 'Peter',
      email: 'peter@test.com'
    };
    this.updateState = this.updateState.bind(this); // Binding for older JS
syntax
  }

  updateState() {
    this.setState({
      name: 'Bruce',
      email: 'bruce@test.com'
    });
  }

  render() {
    return (
      <div>
        <h1>Hello {this.state.name}</h1>
        <p>Email: {this.state.email}</p>
        <button onClick={this.updateState}>Update State</button>
        {/* Alt: <button onClick={() => this.setState({ name: 'Bruce' })}>Update
Name</button> */}
      </div>
```

```
    );
  }
}
// Output before click: Hello Peter, Email: peter@test.com
// Output after click: Hello Bruce, Email: bruce@test.com
```

- **Differences between State and Props (Table - Printed Notes pg 21):**

| Feature | State | Props |
|---|---|---|
| Use | Holds internal component data that can change. | Allows passing data from parent to child. |
| Mutability | Mutable (changed using `setState()`). | Immutable (read-only within the receiving component). |
| Read-Only | Can be changed by the component itself. | Read-only by the child component. |
| Child Components Access | Child components cannot directly access/modify parent's state. | Child component can access props passed by its parent. |
| Stateless Components | Functional components traditionally no state (now `useState`). | Functional components receive and use props. |
| Ownership | Owned and managed by the component itself. | Passed from parent; owned by the parent. |

### 7. LIFECYCLE OF COMPONENTS

- **Lifecycle Diagram (Conceptual from image on Printed Notes pg 15):**
  - **Mounting:** Constructor -> `getDerivedStateFromProps` (optional) -> render -> React updates DOM and refs -> `componentDidMount`
  - **Updating:** New props/`setState()`/`forceUpdate()` -> `getDerivedStateFromProps` (optional) -> `shouldComponentUpdate` (optional) -> render -> `getSnapshotBeforeUpdate` (optional) -> React updates DOM and refs -> `componentDidUpdate`
  - **Unmounting:** `componentWillUnmount`
- **Key Lifecycle Methods (Types - by phase):**
  - **Mounting:**
    1. `constructor(props)`
    2. `static getDerivedStateFromProps(props, state)` (Less common)
    3. `render()`
    4. `componentDidMount()`
  - **Updating:**
    1. `static getDerivedStateFromProps(props, state)`
    2. `shouldComponentUpdate(nextProps, nextState)`

3. `render()`
4. `getSnapshotBeforeUpdate(prevProps, prevState)` (Less common)
5. `componentDidUpdate(prevProps, prevState, snapshot)`

- **Unmounting:**
  1. `componentWillUnmount()`
- **Error Handling:**
  - `static getDerivedStateFromError(error)`
  - `componentDidCatch(error, info)`

### 8. RENDERING LISTS

- **Using `map()` function to render lists (Example - Printed Notes pg 10, 11):**

```
function NumberList(props) {
  const numbers = props.numbers; // e.g., [1, 2, 3, 4, 5]
  const listItems = numbers.map((number) =>
    // Each list item needs a unique "key" prop
    <li key={number.toString()}>
      {number}
    </li>
  );
  return (
    <ul>{listItems}</ul>
  );
}

// const numbersArray = [1, 2, 3, 4, 5];
// ReactDOM.render(
//   <NumberList numbers={numbersArray} />,
//   document.getElementById('root')
// );
```

- **Example with keys (string array - Printed Notes pg 11):**

```
const names = ['Kohli', 'Saif', 'Arun', 'Aamir', 'Arif'];

const ListOfNames = () => {
  const listItems = names.map((name, index) => // Using name as key
    <li key={name}>
      {name}
    </li>
  );
  // Alt if names not unique: use index (with caution) or unique ID from data
```

```
  // const listItems = data.map((item) => <li key={item.id}>{item.text}</li>);
  return (
    <ul>{listItems}</ul>
  );
};
```

### 9. PORTALS

- **Portal Syntax:** `ReactDOM.createPortal(child, container)`
- **Portal Example:**
  - `public/index.html`:

```
<div id="root"></div>
<div id="modal-root"></div> {/* The container for the portal */}
```

  - React Component (`Modal` and `App`):

```
import React from 'react';
import ReactDOM from 'react-dom';

class Modal extends React.Component {
  render() {
    return ReactDOM.createPortal(
      this.props.children, // The content to render
      document.getElementById('modal-root')
    );
  }
}

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = { showModal: false };
    this.handleShow = this.handleShow.bind(this);
    this.handleHide = this.handleHide.bind(this);
  }
  handleShow() { this.setState({ showModal: true }); }
  handleHide() { this.setState({ showModal: false }); }

  render() {
    const modal = this.state.showModal ? (
      <Modal>
        <div className="modal-content">
          This is a modal!
```

```
          <button onClick={this.handleHide}>Hide modal</button>
        </div>
      </Modal>
    ) : null;

    return (
      <div className="app">
        This is the main app content.
        <button onClick={this.handleShow}>Show modal</button>
        {modal}
      </div>
    );
  }
}
// ReactDOM.render(<App />, document.getElementById('root'));
```

### 10. ERROR HANDLING (ERROR BOUNDARIES)

- **Error Boundary Lifecycle Methods (Types):**
  1. `static getDerivedStateFromError(error)`
  2. `componentDidCatch(error, errorInfo)`
- **Error Boundary Example (Code):**

```
import React from 'react';

class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false, error: null, errorInfo: null };
  }

  static getDerivedStateFromError(error) {
    return { hasError: true, error: error };
  }

  componentDidCatch(error, errorInfo) {
    console.error("Uncaught error:", error, errorInfo);
    this.setState({ errorInfo: errorInfo });
  }

  render() {
    if (this.state.hasError) {
      return (
        <div>
```

```
          <h1>Something went wrong.</h1>
          <details style={{ whiteSpace: 'pre-wrap' }}>
            {this.state.error && this.state.error.toString()}
            <br />
            {this.state.errorInfo && this.state.errorInfo.componentStack}
          </details>
        </div>
      );
    }
    return this.props.children;
  }
}

// Example of a component that might throw an error
class BuggyCounter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { counter: 0 };
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() { this.setState(({counter}) => ({ counter: counter + 1 }));}
  render() {
    if (this.state.counter === 3) { throw new Error('I crashed!'); }
    return <h1 onClick={this.handleClick}>{this.state.counter}</h1>;
  }
}

// Usage example:
// <ErrorBoundary>
//   <BuggyCounter />
// </ErrorBoundary>
```

### 11. ROUTERS (REACT ROUTER)

- **React routing vs. Conventional routing (Table - Printed Notes pg 27):**

| Feature | React Routing (SPA) | Conventional Routing (Multi-Page App) |
|---|---|---|
| Page Structure | Single HTML page (typically `index.html`). | Each view often a new HTML file. |
| Navigation | User navigates multiple views within same file/DOM. | User navigates multiple files. |
| Page Refresh | Does not refresh (only relevant parts update). | Refreshes every time user navigates. |

| Feature | React Routing (SPA) | Conventional Routing (Multi-Page App) |
|---|---|---|
| Performance | Generally improved performance, faster transitions. | Slower due to full page reloads. |

- **Key Components of React Router (Types - v5/v6 syntax varies):**
  - `<BrowserRouter>` (or `<HashRouter>`)
  - `<Routes>` (v6) / `<Switch>` (v5)
  - `<Route>` (Attributes: `path`, `element` (v6) / `component` or `render` (v5))
  - `<Link>` (or `<NavLink>`)
- **Command to Install React Router:** `npm install react-router-dom`
- **React Router Example (Conceptual, adapted for v6 - Printed Notes pg 28 for v5 context):**

```
// App.js
import React from 'react';
import { BrowserRouter as Router, Routes, Route, Link } from 'react-router-dom';

const HomePage = () => <h1>Home Page</h1>;
const AboutPage = () => <h1>About Us</h1>;
const ContactPage = () => <h1>Contact Us</h1>;
const NotFoundPage = () => <h1>404 - Page Not Found</h1>;

function App() {
  return (
    <Router>
      <div>
        <nav>
          <ul>
            <li><Link to="/">Home</Link></li>
            <li><Link to="/about">About</Link></li>
            <li><Link to="/contact">Contact</Link></li>
          </ul>
        </nav>
        <hr />
        <h1>React Router Example</h1>
        <Routes>
          <Route path="/" element={<HomePage />} />
          <Route path="/about" element={<AboutPage />} />
          <Route path="/contact" element={<ContactPage />} />
          <Route path="*" element={<NotFoundPage />} /> {/* Catch-all for 404
*/}
        </Routes>
```

```
      </div>
    </Router>
  );
}
export default App;
```

### 12. REDUX

- **Core Concepts/Components of Redux (Types - Printed Notes pg 25):**
  1. Store (Created using `createStore(reducer)`)
  2. Action (Example: `{ type: 'ADD_TODO', text: 'Learn Redux' }`)
  3. Reducer (Syntax: `(previousState, action) => newState`)
  4. Dispatch (Method: `store.dispatch(action)`)
  5. Subscribe (Method: `store.subscribe(listener)`)
- **Data Flow in Redux (Unidirectional - Steps):**
  1. UI interaction.
  2. UI dispatches an Action (e.g., `{ type: 'INCREMENT' }`).
  3. Store calls Reducer(currentState, action).
  4. Reducer returns new state.
  5. Store saves new state.
  6. Store notifies subscribed UI components.
  7. UI re-renders.
- **Redux vs. Flux (Table - Printed Notes pg 27):**

| SN | Redux | Flux |
|---|---|---|
| 1. | Open-source JS library for state management. | Application architecture (pattern) by Facebook. |
| 2. | Store's state is immutable (reducers return new state). | Store's state can be mutable (immutability often encouraged). |
| 3. | Single-store for the entire application. | Can have multiple stores. |
| 4. | Uses the concept of a reducer. | Uses the concept of a dispatcher (central hub for actions). |

### 13. REDUX SAGA

- **Saga Generator Function Syntax:** `function* mySaga() {}`
- **Redux Saga Effects (Types/Syntax):**
  - `call(fn, ...args)`
  - `put(action)`

- `take(pattern)`
- `takeEvery(pattern, saga, ...args)`
- `takeLatest(pattern, saga, ...args)`
- `select(selector, ...args)`

- **Example Action for Saga:** `{ type: 'FETCH_USER_REQUEST', userId: '123' }`
- **Example API call in Saga:** `const user = yield call(Api.fetchUser, action.userId)`
- **Example Success/Failure Actions in Saga:**
  - `yield put({ type: 'FETCH_USER_SUCCESS', user: user })`
  - `yield put({ type: 'FETCH_USER_FAILURE', error: error.message })`
- **Setup Commands/Syntax:**
  1. Install: `npm install redux-saga`
  2. Create middleware: `const sagaMiddleware = createSagaMiddleware()`
  3. Apply middleware: `const store = createStore(rootReducer, applyMiddleware(sagaMiddleware))`
  4. Run root saga: `sagaMiddleware.run(rootSaga)`

### 14. IMMUTABLE.JS

- **Key Data Structures (Types):** `List`, `Stack`, `Map`, `OrderedMap`, `Set`, `OrderedSet`, `Record`.
- **`Map` Example (Code):**

```
import { Map } from 'immutable';
let map1 = Map({ a: 1, b: 2, c: 3 });
let map2 = map1.set('b', 50);
// map1 is still {a:1, b:2, c:3}
// map2 is Map { "a": 1, "b": 50, "c": 3 }
console.log(map1.get('b')); // 2
console.log(map2.get('b')); // 50
```

- **`List` Example (Code):**

```
import { List } from 'immutable';
let list1 = List([1, 2, 3]);
let list2 = list1.push(4);    // list1 is still List [ 1, 2, 3 ]
                              // list2 is List [ 1, 2, 3, 4 ]
let list3 = list2.set(0, 100); // list3 is List [ 100, 2, 3, 4 ]
console.log(list1.get(0));    // 1
```

- **Interoperability Methods (Types):** `toJS()`, `fromJS()`

### 16. UNIT TESTING (IN REACT)

- **Tools for Unit Testing React (Types):**
  1. Jest
  2. React Testing Library (RTL)
  3. Enzyme (Older, less recommended)
- **"What to Test" in a React Component (Types of tests):**
  - Rendering
  - User Interaction
  - Conditional Logic
  - Props
- **Unit Testing Example (Jest and RTL - Conceptual Code):**
  - Component (`Greeting.js`):

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

function Greeting({ initialName = "Guest" }) {
  const [name, setName] = useState(initialName);
  const [inputValue, setInputValue] = useState("");

  const handleChangeName = () => {
    if (inputValue.trim() !== "") {
      setName(inputValue);
      setInputValue("");
    }
  };

  return (
    <div>
      <h1>Hello, {name}!</h1>
      <input
        type="text"
        value={inputValue}
        onChange={(e) => setInputValue(e.target.value)}
        placeholder="Enter a name"
      />
      <button onClick={handleChangeName}>Change Name</button>
    </div>
  );
}
export default Greeting;
```

  - Test file (`Greeting.test.js`):

```
import React from 'react';
import { render, screen, fireEvent } from '@testing-library/react';
import '@testing-library/jest-dom';
import Greeting from './Greeting';

describe('Greeting Component', () => {
  test('renders with default initial name', () => {
    render(<Greeting />);
    expect(screen.getByText('Hello, Guest!')).toBeInTheDocument();
  });

  test('renders with provided initial name', () => {
    render(<Greeting initialName="Alice" />);
    expect(screen.getByText('Hello, Alice!')).toBeInTheDocument();
  });

  test('changes name when button is clicked with input value', () => {
    render(<Greeting />);
    const inputElement = screen.getByPlaceholderText('Enter a name');
    const buttonElement = screen.getByText('Change Name');

    fireEvent.change(inputElement, { target: { value: 'Bob' } });
    fireEvent.click(buttonElement);

    expect(screen.getByText('Hello, Bob!')).toBeInTheDocument();
    expect(inputElement.value).toBe('');
  });
});
```

### 17. WEBPACK

- **Key Features and Benefits (Types):**
  1. Dependency Graph
  2. Loaders
  3. Plugins
  4. Code Splitting
  5. Tree Shaking
  6. Development Server (`webpack-dev-server`)
  7. Hot Module Replacement (HMR)
- **Types of Loaders:** `babel-loader`, `css-loader`, `style-loader`, `file-loader`, `url-loader`
- **Types of Plugins:** `HtmlWebpackPlugin`, `MiniCssExtractPlugin`, `TerserWebpackPlugin`
- **Basic Webpack Configuration (`webpack.config.js` - Simplified Example):**

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  mode: 'development', // or 'production'
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist'),
    clean: true,
  },
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: { loader: 'babel-loader' },
      },
      {
        test: /\.css$/,
        use: ['style-loader', 'css-loader'],
      },
      {
        test: /\.(png|svg|jpg|jpeg|gif)$/i,
        type: 'asset/resource',
      },
    ],
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: './src/index.html',
    }),
  ],
  devServer: {
    static: './dist',
    hot: true,
  },
};
```

---

**UNIT - 3: Node.js and Express.js**

**1. What is Node.js?**

- Node.js utilizes Google's V8 JavaScript engine.

### 2. Key Features of Node.js (Types)

- Asynchronous & Event-Driven (uses callbacks or promises)
- Single-Threaded (with Event Loop)
- Non-Blocking I/O
- npm (Node Package Manager)
- Cross-Platform
- Uses JavaScript

## II. Setting Up Node.js

- **Commands to Verify Installation:**
  - `node -v`
  - `npm -v`
- **Commands to Set Up a Basic Node.js Project:**
  1. `mkdir my-node-app`
  2. `cd my-node-app`
  3. `npm init -y` (creates `package.json`)
- **Command to Install Project Dependencies:**
  - `npm install <package-name>`
  - Example: `npm install express`
- **Command to Run a Simple Node.js Script:**
  - `node app.js`
- **Simple Node.js Script Example (`app.js`):**

```
// app.js
console.log("Hello, Node.js!");
// Output: Hello, Node.js!
```

## III. Node.js Core Concepts

- **Types of Modules:**
  1. **Core Modules:** (e.g., `http`, `fs`, `path`, `os`) - `require` them.
  2. **Local/Custom Modules:** Export using `module.exports` or `exports`, import using `require('./path/to/module')`.
  3. **Third-Party Modules:** Install using `npm install`, import using `require('package-name')`.
- **Core Module Examples (`http`):**

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

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

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

- **Core Module Examples (`path`):**

```
const path = require('path');
const filePath = '/users/admin/file.txt';
console.log(path.basename(filePath)); // Output: file.txt
console.log(path.dirname(filePath)); // Output: /users/admin
console.log(path.extname(filePath)); // Output: .txt
```

- **Core Module Examples (`os`):**

```
const os = require('os');
console.log(os.platform()); // e.g., 'win32', 'linux', 'darwin'
console.log(os.arch());     // e.g., 'x64'
console.log(os.freemem());  // Free system memory in bytes
```

- **Custom Modules Example:**
  - `math.js`:

```
const add = (a, b) => a + b;
const subtract = (a, b) => a - b;
module.exports = { add, subtract };
// Alternatively:
// exports.add = (a, b) => a + b;
// exports.sub = (a, b) => a - b;
```

  - `app.js`:

```
const math = require('./math'); // Use relative path
console.log(math.add(5, 3)); // Output: 8
console.log(math.subtract(10, 4)); // Output: 6
```

- **Third-Party Modules (npm) Example (`lodash`):**
  1. Install: `npm install lodash`
  2. Use:

```
const _ = require('lodash');
const numbers = [1, 2, 3, 4, 5];
console.log(_.shuffle(numbers)); // Outputs a shuffled version
```

- **ES6 Modules (`import`/`export`) Example:**
  1. `package.json` setup:

```
{
  "name": "my-es6-app",
  "version": "1.0.0",
  "type": "module", // Add this line
  "main": "app.mjs" // Optional: use .mjs extension or keep .js
}
```

  2. Module file (`math.mjs`):

```
// math.mjs
export function add(a, b) {
  return a + b;
}
export function subtract(a, b) {
  return a - b;
}
```

  3. Import and use (`app.mjs`):

```
// app.mjs
import { add, subtract } from './math.mjs';
console.log(add(5, 3)); // Output: 8
console.log(subtract(10, 4)); // Output: 6
```

- **Asynchronous Operations Patterns (Types & Examples):**
  - **Callbacks:**

```
const fs = require('fs');
fs.readFile('file.txt', 'utf8', (err, data) => { // Callback function
  if (err) { console.error("Error reading file:", err); return; }
  console.log("File content:", data);
});
console.log("Reading file..."); // This logs first
```

  - **Promises:**

```
const fs = require('fs').promises;
fs.readFile('file.txt', 'utf8')
  .then(data => { console.log("File content:", data); })
```

```
  .catch(err => { console.error("Error reading file:", err); });
console.log("Reading file..."); // This logs first
```

- **Async/Await:**

```
const fs = require('fs').promises;
async function readFileContent() {
  console.log("Reading file..."); // This logs first
  try {
    const data = await fs.readFile('file.txt', 'utf8'); // Pauses here
    console.log("File content:", data);
  } catch (err) {
    console.error("Error reading file:", err);
  }
}
readFileContent();
```

- **Event Loop - How it works (Simplified Steps):**
  1. Main JavaScript code executes.
  2. Async operation (e.g., `fs.readFile`) is handed off, callback registered.
  3. Main thread continues.
  4. Async operation completes, callback placed in event queue.
  5. Event Loop picks up callback and executes it.
- **Event Loop - Phases (Simplified Order - Type of phases):** Timers -> Pending Callbacks (I/O) -> Idle/Prepare -> Poll (New I/O events) -> Check (`setImmediate`) -> Close Callbacks. (`process.nextTick()` runs between phases).
- **EventEmitter - Key Methods (Types):** `on()`, `emit()`, `once()`, `off()`/`removeListener()`, `removeAllListeners()`, `listenerCount()`.
- **EventEmitter Example:**

```
const EventEmitter = require('events');
const myEmitter = new EventEmitter();
const greetHandler = (name) => { console.log(`Hello, ${name}!`); };
myEmitter.on('greet', greetHandler);
myEmitter.once('special', () => console.log('Special event happened once!'));
myEmitter.emit('greet', 'Alice'); // Output: Hello, Alice!
myEmitter.emit('special');       // Output: Special event happened once!
myEmitter.emit('special');       // No output
myEmitter.off('greet', greetHandler);
myEmitter.emit('greet', 'Charlie'); // No output
```

- **File System (`fs`) Module - Reading/Writing (Async with Promise Examples):**
  - Reading:

```
const fs = require('fs').promises;
async function readFile() {
  try {
    const data = await fs.readFile('example.txt', 'utf8');
    console.log("File content:", data);
  } catch (err) { console.error("Error reading file:", err); }
}
readFile();
```

- Writing:

```
const fs = require('fs').promises;
async function writeFile() {
  try {
    await fs.writeFile('output.txt', 'Hello from Node.js!', 'utf8');
    console.log("File written successfully!");
  } catch (err) { console.error("Error writing file:", err); }
}
writeFile();
```

- **Node.js Streams - Types:** Readable (`fs.createReadStream`), Writable (`fs.createWriteStream`), Duplex, Transform (`zlib`).
- **Streams - Piping Syntax:** `readableStream.pipe(writableStream)`
- **Interacting with Command Line - Types of Interactions & Examples:**
  - **Running Scripts (Command):** `node your_script.js`
  - **REPL (Command):** `node` (Exit with `.exit` or `Ctrl+D`)
  - **Command-Line Arguments (`process.argv`):**

```
// args.js
console.log(process.argv);
const args = process.argv.slice(2);
console.log("Arguments:", args);
// Run: node args.js arg1 arg2
```

  - **Reading User Input (`readline` module):**

```
const readline = require('readline').createInterface({
  input: process.stdin, output: process.stdout,
});
readline.question('What is your name? ', (name) => {
  console.log(`Hello, ${name}!`);
  readline.close();
});
```

  - **Executing Shell Commands (`child_process` module):**

```
const { exec } = require('child_process');
exec('ls -l', (error, stdout, stderr) => { // Use 'dir' on Windows
  if (error) { console.error(`exec error: ${error}`); return; }
  console.log(`stdout:\n${stdout}`);
  if (stderr) console.error(`stderr: ${stderr}`);
});
```

- **Building CLI Tools (`yargs` example):**
  1. Install: `npm install yargs`
  2. Code (`cli.js`):

```
const yargs = require('yargs/yargs');
const { hideBin } = require('yargs/helpers');
const argv = yargs(hideBin(process.argv))
  .command('greet <name>', 'Greet someone', (yargs) => {
    yargs.positional('name', { describe: 'Name to greet', type:
'string'});
  }, (argv) => {
    console.log(`Hello, ${argv.name}!`);
  })
  .demandCommand(1, 'You need at least one command')
  .help()
  .argv;
// Run: node cli.js greet Alice
```

- **`console` Module for Debugging - Common Methods (Types):** `console.log()`, `console.error()`, `console.warn()`, `console.table(data)`, `console.time(label)`/`console.timeEnd(label)`, `console.count(label)`, `console.group(label)`/`console.groupEnd()`, `console.assert(assertion, message)`.
- **`console` Module Example:**

```
console.log("Starting script...");
console.warn("This is a warning.");
const users = [{ name: 'Alice', age: 30 }, { name: 'Bob', age: 25 }];
console.table(users);
console.time("Loop");
for (let i = 0; i < 1000; i++) {}
console.timeEnd("Loop"); // Outputs: Loop: X.XXXms
console.assert(users.length > 5, "Assertion failed: Not enough users!"); // Will
log
```

- **Single-Threaded Node.js Concurrency Mechanism:** Event Loop and Non-blocking I/O, Thread Pool (`libuv`).

### IV. Introduction to Express.js

- **Basic Express Application Setup (Commands & Code):**
  1. Ensure Node.js project initialized: `npm init -y`
  2. Install Express: `npm install express`
  3. Create `server.js`:

```
// server.js
const express = require('express');
const app = express();
const port = 3000;

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

app.listen(port, () => {
  console.log(`Express server listening at http://localhost:${port}`);
});
```

  4. Run server: `node server.js`
  5. Open in browser: `http://localhost:3000`

### V. Express.js Core Concepts

- **Routing Syntax:** `app.METHOD(PATH, HANDLER)`
  - `METHOD`: e.g., `get`, `post`, `put`, `delete`
  - `PATH`: e.g., `/`, `/users`, `/products/:id`
  - `HANDLER`: Function `(req, res, next) => {}`
- **Basic Routes Examples:**

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

- **Route Parameters (`req.params`) Example:**

```
app.get('/users/:userId', (req, res) => {
  const userId = req.params.userId;
  res.send(`Fetching user profile for ID: ${userId}`);
});
// Request URL: /users/123 -> userId will be "123"

app.get('/products/:category/:productId', (req, res) => {
  const { category, productId } = req.params;
```

```
  res.send(`Product ID ${productId} in category ${category}`);
});
// Request URL: /products/electronics/456
```

- **Query Parameters (`req.query`) Example:**

```
app.get('/search', (req, res) => {
  const keyword = req.query.keyword;
  const limit = req.query.limit || 10;
  res.send(`Searching for "${keyword}" with limit ${limit}`);
});
// Request URL: /search?keyword=express&limit=5
```

- **`express.Router` for Modular Routes Example:**
  1. Router file (`routes/userRoutes.js`):

```
const express = require('express');
const router = express.Router();
router.use((req, res, next) => { console.log('Time:', Date.now()); next();
});
router.get('/', (req, res) => { res.send('List of users'); });
router.get('/:userId', (req, res) => { res.send(`Details for user
${req.params.userId}`); });
module.exports = router;
```

  2. Mount in main app file (`server.js`):

```
const express = require('express');
const app = express();
const userRoutes = require('./routes/userRoutes');
app.use('/users', userRoutes); // Mount user routes
app.get('/', (req, res) => res.send('Homepage'));
app.listen(3000, () => console.log('Server running...'));
```

- **Route Chaining (`app.route()`) Example:**

```
app.route('/items')
  .get((req, res) => { res.send('Get a list of items'); })
  .post((req, res) => { res.send('Add a new item'); });

app.route('/items/:itemId')
  .get((req, res) => { res.send(`Get item ${req.params.itemId}`); })
  .put((req, res) => { res.send(`Update item ${req.params.itemId}`); })
  .delete((req, res) => { res.send(`Delete item ${req.params.itemId}`); });
```

- **Types of Middleware & Examples:**

- **Application-Level Middleware:**

```
const logger = (req, res, next) => {
  console.log(`${req.method} ${req.originalUrl} - ${new
Date().toISOString()}`);
  next();
};
app.use(logger); // Applied to all requests
```

- **Router-Level Middleware:** (Used with `router.use()` or `router.METHOD()`)
- **Route-Specific Middleware:**

```
const checkAuth = (req, res, next) => {
  if (req.session && req.session.user) { next(); }
  else { res.status(401).send('Unauthorized'); }
};
app.get('/dashboard', checkAuth, (req, res) => { res.send('Welcome!'); });
```

- **Error-Handling Middleware (Syntax):**

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Something broke!');
});
```

- **Common Built-in Express Middleware (Types & Syntax):**
  - `express.json()` (Parses JSON, `req.body`)
  - `express.urlencoded({ extended: true })` (Parses URL-encoded, `req.body`)
  - `express.static('public')` (Serves static files)
- **Common Third-Party Middleware Examples (Types & Install Commands):**
  - `cors` (`npm install cors`)
  - `morgan` (`npm install morgan`)
  - `cookie-parser` (`npm install cookie-parser`)
  - `multer` (`npm install multer`)
  - `helmet` (`npm install helmet`)
- **`req` Object Properties (Types):** `req.params`, `req.query`, `req.body`, `req.headers`, `req.method`, `req.url`/`req.originalUrl`, `req.ip`, `req.cookies`, `req.file`/`req.files`.
- **`res` Object Methods (Types & Syntax):** `res.send()`, `res.json()`, `res.status(code)`, `res.sendStatus(code)`, `res.redirect()`, `res.render()`, `res.sendFile()`, `res.set()`/`res.header()`, `res.cookie()`, `res.clearCookie()`, `res.end()`.
- **Template Engines (View Engines) - Steps & Example Syntax:**

1. Install engine: `npm install ejs` (or `pug`, `hbs`)
2. Configure Express:

```
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'ejs'); // Use EJS
```

3. Create template file (e.g., `views/profile.ejs`).
4. Use `res.render()`:

```
app.get('/profile/:username', (req, res) => {
  const userData = { username: req.params.username, email:
'user@example.com', posts: ['Post 1', 'Post 2'] };
  res.render('profile', userData); // Renders views/profile.ejs
});
```

- **EJS Syntax Example:** `<h1>Welcome, <%= username %>!</h1>` , `<%- unescapedHtml %>`, `<% if (condition) { %> ... <% } %>`
- **Pug Syntax Example:** `h1 Welcome, #{username}!` , `each post in posts\n li= post`
- **Handlebars Syntax Example:** `<h1>Welcome, {{ username }}!</h1>` , `{{#each posts}} <li> {{this}}</li> {{/each}}`

- **File Uploads (using `multer`) Example:**
  1. Install: `npm install multer`
  2. Configure `multer`:

```
const multer = require('multer');
const path = require('path');
const storage = multer.diskStorage({
  destination: (req, file, cb) => { cb(null, 'uploads/'); },
  filename: (req, file, cb) => { cb(null, file.fieldname + '-' + Date.now() +
path.extname(file.originalname)); }
});
const fileFilter = (req, file, cb) => {
  if (file.mimetype === 'image/jpeg' || file.mimetype === 'image/png') {
cb(null, true); }
  else { cb(new Error('Invalid file type!'), false); }
};
const upload = multer({ storage: storage, limits: { fileSize: 1024 * 1024 * 5
}, fileFilter: fileFilter });
// Alt memory storage: const upload = multer({ storage:
multer.memoryStorage() });
```

  3. Use middleware in route (Types of upload methods):
     - `upload.single(fieldname)`: `req.file`

    - `upload.array(fieldname, maxCount)`: `req.files`
    - `upload.fields([...])`: `req.files`
  4. Example Route:

```
app.post('/upload-profile-pic', upload.single('profilePic'), (req, res) => {
  if (!req.file) { return res.status(400).send('No file uploaded.'); }
  res.send(`File uploaded: ${req.file.path}`);
});
```

  5. HTML form `enctype`: `enctype="multipart/form-data"`

- **Working with Cookies (using `cookie-parser`) Example:**
  1. Install: `npm install cookie-parser`
  2. Use middleware:

```
const cookieParser = require('cookie-parser');
app.use(cookieParser('your secret key here')); // Secret for signed cookies
```

  3. Setting Cookies (`res.cookie`):

```
app.get('/set-cookie', (req, res) => {
  res.cookie('username', 'john_doe');
  res.cookie('session_id', 'abc123xyz', { maxAge: 3600000, httpOnly: true /*, secure: true, signed: true */ });
  res.send('Cookies have been set!');
});
```

  4. Reading Cookies (`req.cookies`, `req.signedCookies`):

```
app.get('/read-cookies', (req, res) => {
  const username = req.cookies.username;
  const sessionId = req.signedCookies.session_id;
  res.send(`Username: ${username}, Session ID: ${sessionId}`);
});
```

  5. Deleting Cookies (`res.clearCookie`):

```
app.get('/clear-cookie', (req, res) => {
  res.clearCookie('username');
  res.clearCookie('session_id');
  res.send('Cookies cleared!');
});
```

- **`express-generator` for Scaffolding (Commands):**
  1. `npm install -g express-generator`
  2. `express my-app --view=ejs` (or `--view=pug`)

3. `cd my-app`
4. `npm install`
5. `npm start` (or `DEBUG=my-app:* npm start`)

- **Common Project Structure (MVC-like) (Directory Layout Example):**

```
/my-project
├── /config/         # Configuration files
├── /controllers/    # Request handling logic
├── /models/         # Data definitions, DB interactions
├── /routes/         # Route definitions
├── /views/          # Template files (EJS, Pug)
├── /public/         # Static assets (CSS, JS, images)
├── /middleware/     # Custom middleware
├── /utils/          # Utility functions
├── app.js           # Main Express app setup
├── server.js        # Server initialization (app.listen)
├── package.json
└── .env             # Environment variables
```

### VI. Node.js Database Integration

- **Common Database Choices & Driver Libraries (Types):**
  - MongoDB (NoSQL): `mongoose` (ODM), `mongodb` (native)
  - PostgreSQL (SQL): `pg`
  - MySQL (SQL): `mysql2`, `mysql`
  - Firebase (NoSQL Cloud): `firebase-admin`
  - Redis (In-Memory): `redis`
- **Connecting to MongoDB using Mongoose Example:**
  1. Install: `npm install mongoose`
  2. Connect and Define Schema/Model:

```
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost:27017/myDatabase', {
  useNewUrlParser: true, useUnifiedTopology: true,
})
  .then(() => console.log('MongoDB connected.'))
  .catch(err => console.error('MongoDB connection error:', err));

const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  age: Number,
```

```
  createdAt: { type: Date, default: Date.now }
});
const User = mongoose.model('User', userSchema); // Model 'User' -> 'users' collection
module.exports = User;
```

3. Perform CRUD Operations (Examples in Express routes):
   - CREATE:

```
// const User = require('./models/User');
app.post('/users', async (req, res) => {
  try {
    const newUser = new User(req.body);
    const savedUser = await newUser.save();
    res.status(201).json(savedUser);
  } catch (error) { res.status(400).json({ message: error.message }); }
});
```

   - READ (All):

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

   - READ (By ID):

```
app.get('/users/:id', async (req, res) => { /* ... await User.findById(req.params.id); ... */ });
```

   - UPDATE:

```
app.put('/users/:id', async (req, res) => {
  /* ... await User.findByIdAndUpdate(req.params.id, req.body, { new: true, runValidators: true }); ... */
});
```

   - DELETE:

```
app.delete('/users/:id', async (req, res) => { /* ... await User.findByIdAndDelete(req.params.id); ... */ });
```

- **Connecting to PostgreSQL using `pg` Example:**
  1. Install: `npm install pg`
  2. Connect and Query:

```
const { Pool } = require('pg');
const pool = new Pool({
  user: 'db_user', host: 'localhost', database: 'my_database', password:
'db_password', port: 5432,
});

async function getUsers() {
  try {
    const result = await pool.query('SELECT * FROM users');
    console.log(result.rows); return result.rows;
  } catch (err) { console.error('Error executing query', err.stack); throw
err; }
}
async function addUser(name, email) {
  try {
    const result = await pool.query('INSERT INTO users(name, email)
VALUES($1, $2) RETURNING *', [name, email]);
    console.log('Inserted user:', result.rows[0]); return result.rows[0];
  } catch (err) { console.error('Error executing insert', err.stack); throw
err; }
}
```

**VII. Frequently Asked Questions (FAQ) / Advanced Topics**

- **Callback Hell Example ("Pyramid of Doom"):**

```
asyncA(resultA => {
  asyncB(resultA, resultB => {
    asyncC(resultB, resultC => {
      // ...and so on...
    });
  });
});
```

- **Node.js Exit Codes (Common Types):** `0` (Success), `1` (Uncaught Fatal Exception), `5` (Fatal Error V8), `7` (Internal Exception Handler Run-Time Failure), `9` (Invalid Argument), `12` (Invalid Debug Argument).
- **Reactor Pattern Components (Types):** Reactor, Dispatcher/Demultiplexer, Handlers/Request Handlers.
- `perf_hooks` **Example for Measuring Async Performance:**

```
const { PerformanceObserver, performance } = require('perf_hooks');
const obs = new PerformanceObserver((items) => { /* ... */ });
obs.observe({ entryTypes: ['measure'], buffered: true });
async function someAsyncTask() {
```

```
  performance.mark('task-start');
  await new Promise(resolve => setTimeout(resolve, 100));
  performance.mark('task-end');
  performance.measure('Async Task Duration', 'task-start', 'task-end');
}
someAsyncTask();
```

- **`package.json` - Crucial Parts (Types of info):** Project metadata, `dependencies`, `devDependencies`, `scripts`.
- **Common `npm` Commands (Types of operations):** `npm install <pkg>`, `npm install <pkg> --save-dev`, `npm install`, `npm uninstall <pkg>`, `npm update`, `npm list`.
- **Node.js Timing Features (Types & Syntax):**
  - `setTimeout(callback, delay, [...args])` / `clearTimeout(timeoutId)`
  - `setInterval(callback, delay, [...args])` / `clearInterval(intervalId)`
  - `setImmediate(callback, [...args])` / `clearImmediate(immediateId)`
  - `process.nextTick(callback, [...args])`
- **Simple Node.js Server Returning "Hello World" (using `http` module):** (Same as earlier `http` example)
- **Types of API Functions in Node.js:** Asynchronous (Non-blocking), Synchronous (Blocking - e.g., `fs.readFileSync`).
- **`module.exports` Example:**
  - `utils.js`:

```
const PI = 3.14159;
function calculateCircumference(radius) { return 2 * PI * radius; }
module.exports = { calculateCircumference, PI };
```

  - `app.js`:

```
const utils = require('./utils');
console.log(utils.PI); // 3.14159
console.log(utils.calculateCircumference(10)); // ~62.83
```

- **Tools for Consistent Code Style (Types):** Linters (ESLint), Formatters (Prettier).
- **Async/Await Example in Node.js (using `fs.promises`):**

```
const fs = require('fs').promises;
async function readAndLogFile(filePath) {
  console.log(`Attempting to read: ${filePath}`);
  try {
    const data = await fs.readFile(filePath, 'utf8');
    console.log('File content:', data);
```

```
    return data;
  } catch (error) {
    console.error('Error reading file:', error);
    throw error;
  }
}
async function main() {
  await readAndLogFile('myFile.txt');
  console.log('Finished reading file.');
}
main();
```

- **Event Emitter Example (using custom class):**

```
const EventEmitter = require('events');
class MyEmitter extends EventEmitter {}
const myEmitter = new MyEmitter();
myEmitter.on('data', (chunk) => { console.log(`Received data chunk: ${chunk}`);
});
console.log("Emitting event...");
myEmitter.emit('data', 'Sample Data 1');
myEmitter.emit('data', 'Sample Data 2');
console.log("Finished emitting.");
// Output:
// Emitting event...
// Received data chunk: Sample Data 1
// Received data chunk: Sample Data 2
// Finished emitting.
```

- **Cluster Module Implementation Check (Syntax):** `cluster.isMaster` or `cluster.isPrimary` / `cluster.isWorker`
- **Measuring Duration of Async Operations (using `perf_hooks` - focused example):**

```
const { PerformanceObserver, performance } = require('perf_hooks');
// ... (Observer setup as above) ...
function simulateAsyncWork(callback) {
  performance.mark('work-start');
  setTimeout(() => {
    performance.mark('work-end');
    performance.measure('Simulated Work Duration', 'work-start', 'work-end');
    callback();
  }, 150);
}
simulateAsyncWork(() => { console.log('Async work finished.'); });
```

**UNIT - 4: MongoDB**

### 3. How to Perform Basic CRUD Operations in MongoDB?

(Using `mongosh` shell)

- **a. Add/Insert Data:**
  - **`insertOne` (Syntax/Command):** `db.collection.insertOne({document})`
    - Example: `db.books.insertOne({"title" : "Start With Why"})`
  - **`insertMany` (Syntax/Command):** `db.collection.insertMany([{doc1}, {doc2}])`
    - Example: `db.users.insertMany([{ "name": "Alice", "age": 30 }, { "name": "Bob", "age": 28, "city": "New York" }])`
- **b. Update Data:**
  - **`updateOne` (Syntax/Command):** `db.collection.updateOne({filter}, {update_operator})`
    - Example: `db.users.updateOne({ "name": "Alice" }, { $set: { "age": 31 } })`
  - **`updateMany` (Syntax/Command):** `db.collection.updateMany({filter}, {update_operator})`
    - Example: `db.users.updateMany({ "age": { $gt: 25 } }, { $set: { "status": "active" } })`
  - **`replaceOne` (Syntax/Command):** `db.collection.replaceOne({filter}, {new_document})`
    - Example `_id` remains: `ObjectId("4b2b9f67a1f631733d917a7a")`
    - Example: `db.users.replaceOne({ "name": "alice" }, { "username": "alice_new", "level": 5, "loggedIn": true })`
- **c. Delete Data:**
  - **`deleteOne` (Syntax/Command):** `db.collection.deleteOne({filter})`
    - Example: `db.books.deleteOne({"_id" : 3})`
    - Example: `db.users.deleteOne({ "name": "Bob" })`
  - **`deleteMany` (Syntax/Command):** `db.collection.deleteMany({filter})`
    - Example: `db.users.deleteMany({ "age": { $lt: 22 } })`
- **d. Perform Queries (Read Data):**
  - **`find()` (Syntax/Command):** `db.collection.find({filter})`
    - Find all: `db.users.find({})` or `db.users.find()`
    - Example: `db.users.find({"age" : 24})`
    - Example with operator: `db.users.find({ "age": { $gt: 25 } })` (Query operators: `$gt`, `$lt`)

### 4. SQL vs NoSQL: Key Differences

- **Example SQL Table (Users):**

| ID | Name | Age | Email |
|---|---|---|---|
| 1 | John | 28 | john@mail.com |
| 2 | Sarah | 24 | sarah@mail.com |

- **Example SQL Query:** `SELECT * FROM Users WHERE Age > 25;`
- **Types of NoSQL Databases (with examples):**

| Type | Description | Example |
|---|---|---|
| Document | Stores data as JSON-like documents | MongoDB, CouchDB |
| Key-Value | Simple key-value pairs | Redis, DynamoDB |
| Column-Family | Stores data in columns, not rows | Cassandra, HBase |
| Graph | Stores relationships in nodes/edges | Neo4j, ArangoDB |

- **Example NoSQL (MongoDB Document):**

```
{
  "_id": 1,
  "name": "John",
  "age": 28,
  "email": "john@mail.com",
  "interests": ["coding", "music"]
}
```

- **Example MongoDB Query:** `db.users.find({ age: { $gt: 25 } });`
- **Key Differences Table: SQL vs NoSQL:**

| Feature | SQL (Relational) | NoSQL (Non-Relational) |
|---|---|---|
| Data Model | Tables (Rows & Columns) | Documents, Key-Value, Graph, Columnar |
| Schema | Fixed Schema (Structured) | Dynamic Schema (Flexible) |
| Query Language | SQL (SELECT, JOIN, etc.) | NoSQL queries (MongoDB uses JSON-based) |
| Scalability | Vertical Scaling (More CPU/RAM) | Horizontal Scaling (Distributed systems) |
| Transactions | ACID-compliant (strong consistency) | BASE (Eventual Consistency) - often tunable |
| Best For | Complex relationships, structured data | Big Data, Real-time apps, unstructured data |

- **When to Use SQL vs NoSQL (Table):**

| Use Case | SQL | NoSQL |
|---|---|---|
| Banking & Financial Apps | ✅ Yes (Requires ACID) | ❌ No (Generally, due to ACID needs) |
| Real-Time Analytics | ❌ No (Can be slow at scale) | ✅ Yes |
| E-commerce & Inventory | ✅ Yes | ✅ Yes (Depends on structure needs) |
| Social Networks | ❌ No (Graph relations complex) | ✅ Yes (Especially Graph databases) |
| Content Management | ❌ No (Less flexible for content) | ✅ Yes (Flexible structure benefits) |

### 5. How to Create and Manage MongoDB?

- **Install MongoDB (Example Commands/Sources):**
  - Windows: MSI from MongoDB Official Site
  - Mac (Homebrew): `brew tap mongodb/brew`, `brew install mongodb-community@6.0`
  - Linux (Ubuntu/Debian): `sudo apt update`, `sudo apt install -y mongodb`
- **Start MongoDB Server (Command):** `mongod --dbpath /path/to/your/data/directory`
  - Default port: `27017`
- **Connect to MongoDB Shell (Command):** `mongosh`
- **Create/Switch Database (Command):** `use myDatabase`
- **Create Collection Explicitly (Command):** `db.createCollection("users")`
- **Implicit Collection Creation Example:** `db.products.insertOne({ name: "Laptop", price: 1200 })`
- **List All Databases (Command):** `show dbs`
- **List All Collections (Command):** `show collections`
- **Delete a Collection (Command):** `db.users.drop()`
- **Delete a Database (Commands):** `use myDatabase`, `db.dropDatabase()`
- **Connect MongoDB to Node.js (Basic Example):**
  - Install driver: `npm install mongodb`
  - `server.js` (example):

```
const { MongoClient } = require("mongodb");
const url = "mongodb://localhost:27017";
const client = new MongoClient(url);
const dbName = "myDatabase";
```

```
async function run() {
  try {
    await client.connect();
    console.log("Connected successfully to server");
    const db = client.db(dbName);
    const usersCollection = db.collection("users");
    await usersCollection.insertOne({ name: "David", age: 35 });
    console.log("Inserted a document...");
    const result = await usersCollection.find().toArray();
    console.log("Found documents =>", result);
  } catch (err) { console.error("An error occurred:", err);
  } finally { await client.close(); console.log("Connection closed."); }
}
run().catch(console.error);
```

- Run script: `node server.js`

- **Summary Table: Common Mongo Shell Commands:** (Partial list for brevity, full table in notes)

| Task | Command |
|---|---|
| Start MongoDB | `mongod --dbpath /path/to/data` |
| Connect Shell | `mongosh` |
| Switch/Create DB | `use myDatabase` |
| Insert One | `db.users.insertOne({ name: "John" })` |
| Find All | `db.users.find()` |
| List Databases | `show dbs` |
| Drop Collection | `db.users.drop()` |

### 6. How to Migrate Data into MongoDB?

- **Migration from JSON/CSV Files:**
  - **`mongoimport` for JSON Array (Command):**
    `mongoimport --db myDatabase --collection users --file users.json --jsonArray`
  - **Example `users.json` (for `--jsonArray`):**

```
[
  { "name": "Alice", "age": 25, "email": "alice@mail.com" },
  { "name": "Bob", "age": 28, "email": "bob@mail.com" }
]
```

- `mongoimport` for CSV (Command):
  `mongoimport --db myDatabase --collection users --type csv --file users.csv --headerline`
- Example `users.csv`:

```
name,age,email
Alice,25,alice@mail.com
Bob,28,bob@mail.com
```

- **Migration from SQL Databases (e.g., MySQL):**
  - **Steps (Types):** 1. Export SQL (CSV/JSON), 2. Transform data, 3. Import (mongoimport/script)
  - **Example MySQL Export to CSV (SQL Query):**

```
SELECT *
FROM users
INTO OUTFILE '/path/on/server/to/users.csv'
FIELDS TERMINATED BY ',' ENCLOSED BY '"' LINES TERMINATED BY '\n';
```

  - **Node.js Script for MySQL to MongoDB Migration (`migrate.js` - conceptual structure):**

```
const mysql = require("mysql");
const { MongoClient } = require("mongodb");
// ... (SQL & Mongo connection configs) ...
async function migrateData() {
  // ... (Connect to Mongo) ...
  sqlConnection.connect(err => {
    // ... (Connect to MySQL) ...
    sqlConnection.query("SELECT * FROM users", async (error, results) => {
      // ... (Map results to MongoDB format) ...
      // ... (Insert into MongoDB usersCollection.insertMany(usersToInsert))
...
      // ... (Close connections) ...
    });
  });
}
migrateData();
```

- **Migration from Firebase (Firestore) to MongoDB (Node.js script `firebase-migrate.js` - conceptual structure):**

```
const admin = require("firebase-admin");
const { MongoClient } = require("mongodb");
// ... (Firebase & Mongo setup with service account and connection string) ...
async function migrateFirebaseToMongo() {
  // ... (Connect to Mongo) ...
```

```
  const snapshot = await firestore.collection(firebaseCollectionName).get();
  const firestoreDocs = snapshot.docs.map(doc => ({ _id: doc.id, ...doc.data()
}));
  // ... (Insert into MongoDB mongoCollection.insertMany(firestoreDocs)) ...
  // ... (Close Mongo connection) ...
}
migrateFirebaseToMongo();
```

- **Migration from Excel to MongoDB (Node.js script `excel-to-mongo.js` - conceptual structure):**

```
const xlsx = require("xlsx");
const { MongoClient } = require("mongodb");
// ... (Excel file path & Mongo setup) ...
async function migrateExcelToMongo() {
  // ... (Read Excel: xlsx.readFile, xlsx.utils.sheet_to_json) ...
  // ... (Connect to Mongo) ...
  // ... (Insert jsonData into MongoDB collection.insertMany(jsonData)) ...
  // ... (Close Mongo connection) ...
}
migrateExcelToMongo();
```

- **Summary of Migration Methods (Table):**

| Source | Method(s) | Notes |
|---|---|---|
| JSON | `mongoimport --jsonArray` or `mongoimport` (per line) | Easiest for standard JSON. |
| CSV | `mongoimport --type csv --headerline` | Simple for flat, tabular data. |
| MySQL/PostgreSQL | Export to CSV/JSON -> `mongoimport` OR Custom Script (Node.js) | Script needed for transformation. |
| Firebase Firestore | Firebase Admin SDK + Custom Script (Node.js) | Requires service account key. |
| Excel (XLSX) | Convert to CSV/JSON -> `mongoimport` OR `xlsx` lib + Script | Script offers more control. |

### 7. How to Use MongoDB with Node.js?

- **Node.js Database Connection Module (`db.js` - example structure):**

```
// db.js
const { MongoClient } = require("mongodb");
const url = "mongodb://localhost:27017";
const dbName = "myDatabase";
const client = new MongoClient(url);
let dbConnection;
```

```
module.exports = {
  connectToServer: async function () { /* ... connect client, set dbConnection ... */ },
  getDb: function () { /* ... return dbConnection ... */ }
};
```

- **Node.js CRUD Operations (`app.js` using `db.js` - conceptual structure):**

```
// app.js
const db = require("./db");
async function main() {
  await db.connectToServer();
  const database = db.getDb();
  const usersCollection = database.collection("users");
  // CREATE
  // const insertResult = await usersCollection.insertOne({ name: "Alice", ... });
  // READ
  // const users = await usersCollection.find({ age: { $gte: 25 } }).toArray();
  // UPDATE
  // const updateResult = await usersCollection.updateOne({ name: "Alice" }, { $set: { age: 26 } });
  // DELETE
  // const deleteResult = await usersCollection.deleteOne({ name: "Alice" });
}
main();
```

- **MongoDB with Express.js API (`server.js` - conceptual structure for routes):**
  - Install: `npm install express mongodb cors body-parser`
  - Code:

```
const express = require("express");
const db = require("./db"); // From above
// ... (app setup, middleware: cors, bodyParser.json()) ...
let database;
db.connectToServer().then(() => {
  database = db.getDb();
  // GET All Users
  app.get("/users", async (req, res) => { /* ... database.collection("users").find().toArray() ... */ });
  // GET User by Name
  app.get("/users/:name", async (req, res) => { /* ... database.collection("users").findOne({ name: req.params.name }) ... */ });
```

```
  // POST (Create) User
  app.post("/users", async (req, res) => { /* ...
database.collection("users").insertOne(req.body) ... */ });
  // PUT (Update) User
  app.put("/users/:name", async (req, res) => { /* ...
database.collection("users").updateOne({name: req.params.name}, {$set:
req.body}) ... */});
  // DELETE User
  app.delete("/users/:name", async (req, res) => { /* ...
database.collection("users").deleteOne({name: req.params.name}) ... */});
  app.listen(port, () => { /* ... server running ... */ });
});
```

- Test API endpoints (Examples):
  - `GET http://localhost:3000/users`
  - `POST http://localhost:3000/users` with JSON body `{ "name": "Bob", "age": 30 }`
  - `PUT http://localhost:3000/users/Bob` with JSON body `{ "age": 32 }`
  - `DELETE http://localhost:3000/users/Bob`

**8. What are the Key MongoDB Services?**

- **Types of MongoDB Services (with brief description/use case):**
  1. **MongoDB Atlas (Cloud Database):** Fully managed DBaaS. (Use Case: Cloud apps, Serverless)
  2. **MongoDB Enterprise Advanced (Self-Managed):** Commercial, self-hosted with advanced features. (Use Case: Large orgs, compliance)
  3. **MongoDB Community Edition (Self-Managed):** Free, open-source core DB. (Use Case: Dev, small projects)
  4. **MongoDB Realm (Mobile & App Dev Platform):** Mobile DB (offline-first) + backend services. (Use Case: Mobile apps, cloud sync)
  5. **MongoDB Charts (Data Visualization):** Real-time dashboards from Atlas data. (Use Case: BI, embedded analytics)
  6. **MongoDB Compass (GUI Tool):** Visual interface for MongoDB. (Use Case: Developers, DBAs)
  7. **MongoDB Atlas Search (Integrated Full-Text Search):** Managed search on Atlas data. (Use Case: Rich text search apps)
  8. **MongoDB Atlas Data Lake (Query Data in Place):** Query data in cloud object storage (S3, Azure Blob). (Use Case: Big Data analytics on S3)
  9. **MongoDB Backup & Restore Solutions:** (Tools: `mongodump`, `mongorestore` for Community; Atlas/Enterprise have managed solutions)

---