# ONESHOT WEB-D WITH MERN NOTES - V3

### UNIT 1: WEB DEVELOPMENT FUNDAMENTALS

### 1. INTRODUCTION

- **Fundamentals of Good Website Design:**
  - **Definition:** Fulfills intended function, conveys message, and engages the visitor (a balance of aesthetics and functionality).
  - **Core Purpose:** Describing Expertise, Building Reputation, Generating Leads, Sales & After Care.
  - **Key Factors:** Consistency, Colours (brand-fit, typically <5, considering emotional impact), Typography (legible, max 2-3 fonts, reflecting brand voice), Imagery (expressive, high-quality, relevant), Simplicity (clean, uncluttered design), Functionality (ease of use, all elements work as expected), User Experience (UX) (overall satisfaction, usability, accessibility, building trust).
  - **Additional Principles:** Navigation (simple, intuitive, consistent, clearly labeled) [PYQ], F-Shaped Pattern Reading (users often scan in an F-shape), Visual Hierarchy (guides the eye to important information first), Content (compelling, relevant, well-organized), Grid-Based Layout (for organization and a clean look), Load Time (fast; optimize images and code), Mobile Friendly (responsive design for all devices).
- **Web Page:** A document, typically written in HTML, that is viewed in a web browser. It has a unique URL and can embed styles (CSS), scripts (JS), and multimedia content.
- **Website:** A collection of linked web pages and related content, identified by a common domain name and published on at least one web server. Accessed via its domain, it usually displays a homepage first.
- **Web Application:** A program that runs on a remote server and is accessed by users through a web browser over a network (like the internet). Requires a web server, an application server, and often a database.
  - **Benefits:** No installation required on the user's device, accessible to multiple users simultaneously, cross-platform compatibility.
  - **Native App:** Platform-specific (e.g., iOS, Android), installed directly onto the device, can often work offline, and has direct access to device hardware.
  - **Hybrid App:** Installs like a native app but is built using web technologies (HTML, CSS, JS) and runs in a webview. Can access device APIs through plugins, usually requires an internet connection for full functionality.
- **Client-Server Architecture:** [PYQ] A distributed application structure that partitions tasks or workloads between the providers of a resource or service, called servers, and service requesters, called clients.

- **Components:** Workstations (clients), Servers (central repositories of data and programs), Networking Devices (routers, switches for communication).
- **How it Works (Simplified):** User enters URL -> DNS resolves URL to IP address -> Browser sends HTTP request to server IP -> Server processes request and sends files (HTML, CSS, JS, media) back -> Browser renders and displays the page.
- **Tiers:**
  - **1-Tier:** All layers (presentation, application logic, data) are on a single device. Not common for web applications.
  - **2-Tier:** Client (User Interface) directly communicates with the Server (Database and sometimes application logic).
  - **3-Tier:** [PYQ for MERN & Q1(a)] Separates application into:
    1. **Presentation Tier (Client - e.g., React.js in MERN):** Handles UI/UX, user interaction. Runs on the client's browser.
    2. **Application Tier/Middleware (Server Logic - e.g., Node.js/Express.js in MERN):** Contains the business logic, processes client requests, interacts with the data tier. Runs on the web/application server.
    3. **Data Tier (Database - e.g., MongoDB in MERN):** Manages data storage and retrieval. Runs on a database server.
    - *Theory (3-Tier):* This separation promotes modularity, scalability, maintainability, and security. Each tier can be developed, managed, and scaled independently.
    - *Structure (3-Tier for MERN):* Client (Browser with React) <-> API Calls (HTTP) <-> Server Logic (Node.js/Express.js) <-> Database Operations (MongoDB).
  - **N-Tier:** Extends the 3-tier model with additional, more specialized layers for improved scalability and separation of concerns.
- **vs. Peer-to-Peer (P2P):** Client-Server has specific roles for clients and servers with centralized data/control, while P2P networks have equally privileged peers that share resources and data directly in a distributed manner.

- **MERN Stack Introduction:** [PYQ Q1(a)] A JavaScript-based technology stack for building full-stack web applications.
  - **Components:**
    - **M**ongoDB: A NoSQL, document-oriented database.
    - **E**xpress.js: A minimal and flexible Node.js web application framework (for the backend/API).
    - **R**eact.js: A JavaScript library for building user interfaces (for the frontend).
    - **N**ode.js: A JavaScript runtime environment that executes JavaScript code outside a web browser (for the server-side).
  - **3-Tier Architecture in MERN:** [PYQ Q1(a)] (As described above in Client-Server Architecture)
  - **Benefits:** Cost-effective (largely open-source), enables full-stack development using only JavaScript, generally good performance, SEO friendly (with SSR), robust security features can

be implemented, supports agile development and fast delivery cycles.

## 2. MARKUP LANGUAGES

- **HTML (HyperText Markup Language):** The standard markup language for creating the structure and content of web pages.
  - Composed of **Elements** (e.g., `<p>This is a paragraph.</p>`), which consist of a start tag, content, and an end tag.
  - **Tags** are keywords surrounded by angle brackets (e.g., `<body>`).
  - **Attributes** provide additional information about HTML elements and are defined within the start tag (e.g., `name="value"` as in `<img src="image.jpg" alt="description">`).
  - **Basic Structure:**

```
<!DOCTYPE html>
<html>
<head>
    <title>Page Title</title>
</head>
<body>
    <!-- Page content goes here -->
</body>
</html>
```

  - HTML is generally not case-sensitive for tags and attributes (though lowercase is convention). The `<html>` element is the root parent of `<head>` (metadata) and `<body>` (visible content).
- **XHTML (Extensible HTML):** A stricter, XML-based version of HTML. It follows XML's syntax rules.
  - **Key Rules:** Case-sensitive (tags and attributes usually lowercase), `<!DOCTYPE>` declaration is mandatory, `xmlns` attribute in `<html>` is required, all elements must be properly nested and closed (including empty elements like `<br />`), attribute values must be quoted.
- **HTML Lists:** Used to group related items of information.
  - Types:
    - `<ul>`: Unordered list (typically displayed with bullets).
    - `<ol>`: Ordered list (typically displayed with numbers or letters).
    - `<li>`: List item (used within `<ul>` and `<ol>`).
    - `<dl>`: Description list.
      - `<dt>`: Description term.
      - `<dd>`: Description definition/details.
  - Lists can be nested within other lists.
- **HTML Tables:** Used to arrange data in rows and columns.

- Core Elements: `<table>` (defines the table), `<tr>` (table row), `<td>` (table data cell), `<th>` (table header cell - bold and centered by default).
- Attributes: `border` (specifies border width), `colspan` (makes a cell span multiple columns), `rowspan` (makes a cell span multiple rows).
- Structural Elements: `<caption>` (table title), `<thead>` (groups header content), `<tbody>` (groups body content), `<tfoot>` (groups footer content).

- **HTML Forms:** Used to collect user input.
  - Main Element: `<form>`. Key attributes: `action` (URL where form data is sent), `method` (HTTP method, e.g., GET or POST).
  - **Form Controls:**
    - `<input type="text/password/checkbox/radio/file/submit/reset/button/image/hidden/email/date/etc.">`
    - `<textarea>`: For multi-line text input.
    - `<select>`: Creates a drop-down list.
    - `<option>`: Defines an option within a `<select>` list.
    - `<button>`: Defines a clickable button.
- **XML (Extensible Markup Language):** A markup language designed to carry data, not to display data. It uses user-defined tags, is hierarchical, and has a strict syntax. Commonly used for data exchange between different systems.

### 3. CSS STYLE SHEETS

- **CSS (Cascading Style Sheets):** A stylesheet language used to describe the presentation (look, formatting, layout) of a document written in HTML or XML.
- **Core Syntax:** `selector { property: value; }` (e.g., `p { color: blue; }`).
- **Types of CSS:**
  1. **Inline CSS:** Uses the `style` attribute directly within an HTML element (e.g., `<p style="color: red;">`). Highest precedence.
  2. **Internal/Embedded CSS:** Defined within a `<style>` tag in the `<head>` section of an HTML document.
  3. **External CSS:** Written in separate `.css` files and linked to the HTML document using a `<link>` tag in the `<head>` section (e.g., `<link rel="stylesheet" href="styles.css">`). Most common and recommended method for maintainability.
- **Text Properties (Examples):** `color`, `background-color`, `font-family`, `font-size`, `font-weight`, `text-align`, `text-decoration`, `text-transform`, `text-indent`, `letter-spacing`, `word-spacing`, `line-height`, `text-shadow`.
- **CSS Box Model:** [PYQ Q3(b)(ii)]

- *Theory:* Defines how HTML elements are rendered as rectangular boxes on a web page. Each box has distinct layers that contribute to its overall size and spacing relative to other elements.
- *Layers (from innermost to outermost):*
  1. **Content:** The actual content of the element (text, image, etc.). Its size is defined by `width` and `height` properties.
  2. **Padding:** Transparent space around the content, inside the border. Clears an area around the content.
  3. **Border:** A line that goes around the padding and content.
  4. **Margin:** Transparent space outside the border. Clears an area around the element, separating it from other elements.

- **Normal Flow (Document Flow):** The default way block and inline elements are displayed on a page.
  - **Block-level elements:** Start on a new line and take up the full width available (e.g., `<div>`, `<p>`, `<h1>`).
  - **Inline elements:** Flow along with the surrounding content, do not start on a new line, and only take up as much width as necessary (e.g., `<span>`, `<a>`, `<img>`).
  - The `display` property can be used to change an element's default flow behavior (e.g., `display: inline-block;`, `display: flex;`, `display: grid;`).
- **Styling Lists:** Properties like `list-style-type` (e.g., `disc`, `circle`, `square`, `decimal`), `list-style-image` (use an image as list marker), `list-style-position` (`inside` or `outside`).
- **Styling Tables:** Properties like `border`, `border-collapse` (`collapse` or `separate`), `padding` (for cells), `text-align`, `background-color`.
- **XSLT (Extensible Stylesheet Language Transformations):** A language for transforming XML documents into other XML documents, or to other formats like HTML or plain text. It uses XPath to navigate and select parts of an XML document.

**4. CLIENT-SIDE PROGRAMMING: JAVASCRIPT**

- **Introduction to JS:** A high-level, interpreted (or just-in-time compiled) scripting language primarily used to create dynamic and interactive content on web pages. It is not related to Java.
- **Basic Syntax & Embedding:**
  - Can be embedded in HTML using `<script>` tags, typically placed in the `<head>` or at the end of the `<body>` (recommended for faster perceived page load).
    - **Internal JS:** `<script> // JavaScript code here </script>`
    - **External JS:** `<script src="path/to/script.js"></script>` (links to an external `.js` file).
  - Statements are often separated by semicolons (`;`), though they are mostly optional if statements are on new lines (due to Automatic Semicolon Insertion - ASI).

- Comments: Single-line `// comment` or multi-line `/* comment */`.
- JavaScript is case-sensitive.

- **Variables & Data Types:**
  - Variables are declared using `var` (function-scoped or global, pre-ES6), `let` (block-scoped, ES6+), `const` (block-scoped constant, value cannot be reassigned, ES6+).
  - **Primitive Data Types:**
    - `String`: Sequence of characters (e.g., `"Hello"`).
    - `Number`: Numeric values (integer or floating-point, e.g., `42`, `3.14`).
    - `BigInt`: For integers of arbitrary length.
    - `Boolean`: `true` or `false`.
    - `Undefined`: A variable that has been declared but not assigned a value.
    - `Null`: Represents the intentional absence of any object value.
    - `Symbol`: Unique and immutable primitive value (ES6+).
  - **Object Type (Reference Type):** Represents a collection of key-value pairs or more complex data structures.
    - `Object`: Includes plain objects `{}`, `Array`, `Function`, `Date`, etc.
  - JavaScript is dynamically typed (variable types are determined at runtime).
- **Literals:** Fixed values directly written in the code (e.g., `100`, `"Hello"`, `true`, `null`, `{}`, `[]`, `/regex/`).
- **Operators:**
  - Arithmetic: `+`, `-`, `*`, `/`, `%` (modulo), `++` (increment), `--` (decrement), `**` (exponentiation ES7+).
  - Assignment: `=`, `+=`, `-=`, `*=`, `/=`, `%=`.
  - Comparison: `==` (loose equality, type coercion), `===` (strict equality, no type coercion), `!=`, `!==`, `>`, `<`, `>=`, `<=`.
  - Logical: `&&` (AND), `||` (OR), `!` (NOT).
  - String: `+` (concatenation).
  - Ternary (Conditional): `condition ? valueIfTrue : valueIfFalse`.
- **Functions:** Reusable blocks of code designed to perform a particular task.
  - Declaration: `function functionName(parameter1, parameter2) { /* code to execute */ return value; }`
  - Invocation/Call: `functionName(argument1, argument2);`
  - Variables declared inside a function are typically local to that function (scope).

- **Arrow Functions (ES6+):** Provide a more concise syntax and lexical binding of `this` (e.g., `const add = (a, b) => a + b;`).

- **Closures:** [PYQ Q1(e) How to explain closures in JavaScript and when to use it?]
  - *Theory:* A closure is formed when an inner function has access to its outer (enclosing) function's variables and scope, even after the outer function has finished executing and its scope is normally destroyed. The inner function "remembers" the environment in which it was created.
  - *Use Cases:*
    - **Data Privacy/Encapsulation:** Creating private variables and methods (emulating private members of a class).
    - **Maintaining State in Asynchronous Operations:** In callbacks or event handlers, to preserve state specific to an event or iteration.
    - **Function Factories:** Functions that create and return other functions, customized by the outer function's parameters.
    - **Currying and Partial Application.**
  - *Structure/Example:*

```
function outerFunction(outerVariable) {
    return function innerFunction(innerVariable) {
        console.log('Outer Variable: ' + outerVariable);
        console.log('Inner Variable: ' + innerVariable);
    }
}
const newFunction = outerFunction('outside'); // outerFunction has executed
newFunction('inside'); // innerFunction still has access to 'outerVariable'
// Output:
// Outer Variable: outside
// Inner Variable: inside
```

- **Objects:** Collections of key-value pairs, where keys are strings (or Symbols) and values can be any data type, including other objects or functions (methods).
  - Creation: Object literals `{}`, `new Object()`, constructor functions, ES6 classes.
  - Accessing Properties: Dot notation `object.propertyName` or bracket notation `object["propertyName"]`.
  - **`this` keyword:** [PYQ Q2(a)] Refers to the object it belongs to. Its value depends on how a function is called:
    - In the global scope: `this` refers to the global object (`window` in browsers, `global` in Node.js).

    - In a method (function as an object property): `this` refers to the object the method is called on.
    - In a regular function (not a method, not arrow): `this` refers to the global object (in non-strict mode) or `undefined` (in strict mode).
    - In a constructor function (called with `new`): `this` refers to the newly created instance.
    - In an event handler: `this` often refers to the element that triggered the event.
    - In an arrow function: `this` is lexically bound; it inherits `this` from the surrounding (enclosing) non-arrow function's scope.
- **Arrays:** Ordered, 0-indexed collections of values of any type.
    - Creation: Array literals `[]` or `new Array()`.
    - Accessing Elements: `array[index]`.
    - `length` property: Gets or sets the number of elements.
    - Common Methods: `push()`, `pop()`, `shift()`, `unshift()`, `slice()`, `splice()`, `map()`, `filter()`, `reduce()`, `forEach()`, etc. Nested arrays (arrays of arrays) are possible.
- **Built-in Objects (Examples):** `Math` (for mathematical constants and functions), `Date` (for working with dates and times), `String` (provides methods for string manipulation), `JSON` (for parsing and stringifying JSON data).
- **JS Form Programming:** Interacting with HTML forms to retrieve user input, validate data, and dynamically update form elements or page content based on form interactions. Access form elements via DOM methods and manipulate their `value`, `checked`, `disabled`, etc., properties.
- **JavaScript Events & Event Handling:** [PYQ Q3(a) How to handle JavaScript Events in HTML? Explain with example.]
    - *Theory:* Events are actions or occurrences that happen in the browser, such as a user clicking a button, a page finishing loading, or a key being pressed. JavaScript can "listen" for these events and execute code in response.
    - **Ways to Handle Events:**
        1. **Intrinsic Event Attributes (HTML Attributes):** Directly assign JavaScript code to HTML attributes like `onclick`, `onmouseover`, `onload`.

```
<button onclick="alert('Button clicked!')">Click Me (Intrinsic)
</button>
```

        2. **DOM Property Assignment:** Assign a function to an element's `on<event>` property in JavaScript.

```
<button id="myButtonDomProp">Click Me (DOM Property)</button>
<script>
    document.getElementById('myButtonDomProp').onclick = function()
{
        alert('Button clicked via DOM property!');
```

```
    };
</script>
```

3. `addEventListener()` **(Modern & Recommended):** Attach one or more event listener functions to an element for a specific event type.

```
<button id="myButtonListener">Click Me (addEventListener)</button>
<script>

document.getElementById('myButtonListener').addEventListener('click'
, function() {
        alert('Button clicked via addEventListener!');
    });
    // Can add multiple listeners for the same event

document.getElementById('myButtonListener').addEventListener('click'
, function() {
        console.log('Another action on click!');
    });
</script>
```

- *Example (PYQ Q3(a)):* The `addEventListener()` method is generally preferred as it's more flexible (allows multiple handlers, more control over event phases).

- **Modifying Element Style:** JavaScript can dynamically change the CSS styles of HTML elements.
  - Example: `document.getElementById("myElement").style.color = "blue";`
  - Example: `document.getElementById("myElement").style.fontSize = "16px";` (CSS properties with hyphens like `font-size` become camelCase like `fontSize` in JS).

- **Document Object Model (DOM) & Document Trees:** [PYQ Q3(b)(iii)]
  - *Theory:* The DOM is a programming interface (API) for HTML and XML documents. It represents the page structure as a logical tree of nodes, where each node is an object representing a part of the document (e.g., an element, text, comment). JavaScript can use the DOM to access, create, and modify the content, structure, and style of web pages dynamically.
  - **Node Relationships:** Parent, Child, Sibling.
  - **Node Types:** Element node, Text node, Attribute node, Comment node, Document node, etc.
  - **Levels of DOM (Simplified History):**
    - **Level 0:** Early, informal specification, basic event handling (`onclick`) and document access.
    - **Level 1:** Standardized Core (generic XML/HTML manipulation) & HTML-specific models (access to HTML elements and attributes).
    - **Level 2:** Introduced advanced event handling (`addEventListener`), CSS manipulation (style objects), range and traversal interfaces.

    - **Level 3 & Beyond (Living Standard - WHATWG):** Added features like XPath, content models, saving/loading, and continues to evolve as the "DOM Living Standard".
- **Accessing HTML Elements in JS:** [PYQ Q2(a) In How many ways an HTML element can be accessed in JavaScript code?]
    - `document.getElementById('elementId')`: Selects a single element by its unique ID.
    - `document.getElementsByTagName('tagName')`: Selects a collection (HTMLCollection) of elements by their tag name (e.g., 'p', 'div').
    - `document.getElementsByClassName('className')`: Selects a collection (HTMLCollection) of elements by their class name.
    - `document.querySelector('cssSelector')`: Selects the first element that matches a specified CSS selector.
    - `document.querySelectorAll('cssSelector')`: Selects all elements (NodeList) that match a specified CSS selector.
- **ECMAScript (ES):** The specification that JavaScript is an implementation of. New versions bring new features.
    - **ES5 (2009):** Introduced "strict mode" (`'use strict'`), JSON support (`JSON.parse()`, `JSON.stringify()`), and new Array methods (`forEach`, `map`, `filter`, `reduce`, `isArray`).
    - **ES6 (2015) / ES2015:** A major update with many new features:
        - `let` and `const` for variable declarations (block scope).
        - Arrow functions (`=>`).
        - Template Literals (backticks `` ` `` for strings with embedded expressions `${expr}`).
        - Default function parameters (`function greet(name = 'Guest') {}`).
        - Rest parameters (`...args`) and Spread operator (`...iterable`).
        - Destructuring assignment (for objects and arrays).
        - Classes (`class MyClass extends ParentClass {}`).
        - Modules (`import`/`export`).
        - `Promise` objects for asynchronous operations.
        - `Map`, `Set`, `WeakMap`, `WeakSet` data structures.
        - `Symbol` primitive type.
        - `for...of` loop for iterating over iterable objects.
    - **ES5 vs ES6 Comparison Table:**

| Feature | ES5 | ES6 / ES2015+ |
| --- | --- | --- |
| Variable Declaration | `var` (function-scoped or global) | `let`, `const` (block-scoped) |

| Feature | ES5 | ES6 / ES2015+ |
|---|---|---|
| Function Syntax | `function name() {}` | Arrow functions (`() => {}`), lexical `this` |
| String Handling | Concatenation with `+` | Template Literals (`` `My name is ${name}` ``) |
| Default Parameters | Manual check inside function | `function(param = 'default') {}` |
| `arguments` object | Array-like, not a true array | Rest parameters (`...args`) -> true array |
| Iterables Expansion | Manual iteration or methods like `concat` | Spread operator (`...iterable`) |
| Destructuring | Manual assignment from obj/array props | `const {a,b}=obj; const [x,y]=arr;` |
| OOP | Constructor functions, prototype-based inheritance | `class`, `constructor`, `extends`, `super` (syntactic sugar over prototypes) |
| Modules | No native support (used CommonJS/AMD libraries) | Native `import`/`export` syntax |
| Async Operations | Callbacks (can lead to "callback hell") | `Promise` objects, `async`/`await` (ES2017) |
| Looping Iterables | `for` loop, `Array.prototype.forEach` | `for...of` loop (for iterable objects like Array, String, Map, Set) |
| New Data Structures | - (Objects and Arrays primarily) | `Map`, `Set`, `WeakMap`, `WeakSet` |
| Unique Identifiers | - | `Symbol` primitive type |

---

**HTTP METHODS (WEB STANDARD, CRUD OPERATIONS)**

- **Definition:** Standardized request methods (verbs) used in HTTP to indicate the desired action to be performed for a given resource identified by a URL.
- **CRUD Operations Mapping:**
  - **C**reate -> **POST** (typically when the server assigns the ID) / **PUT** (if the client defines the ID for a new resource, and the operation is idempotent)
  - **R**ead -> **GET**
  - **U**pdate -> **PUT** (to replace an entire resource) / **PATCH** (to partially update a resource)
  - **D**elete -> **DELETE**
- **Common Methods & Examples:**

- **GET:** Retrieves data from a specified resource. Safe (should not change server state) and idempotent (multiple identical requests have the same effect as a single one). *Ex:* `GET /users` *(list all users),* `GET /users/123` *(get user with ID 123).*
- **POST:** Submits data to be processed to a specified resource, often causing a change in state or side effects on the server (e.g., creating a new resource). Not idempotent. *Ex:* `POST /users` *(with user data in the request body to create a new user).*
- **PUT:** Replaces all current representations of the target resource with the request payload. Idempotent. *Ex:* `PUT /users/123` *(with complete updated user data in the request body to replace user 123).*
- **DELETE:** Removes the specified resource. Idempotent (deleting a non-existent resource is still successful from HTTP perspective, or returns 404). *Ex:* `DELETE /users/123` *(delete user with ID 123).*
- **PATCH:** Applies partial modifications to a resource. Not necessarily idempotent (depends on the nature of the patch). *Ex:* `PATCH /users/123` *(with* `{"email": "new@example.com"}` *in the body to update only the email).*
- **HEAD:** Similar to GET, but it only transfers the status line and header section (no response body). Useful for checking resource metadata without transferring the whole content.
- **OPTIONS:** Describes the communication options (e.g., allowed HTTP methods) for the target resource. Often used for CORS preflight requests.

---

**UNIT 2: REACTJS**

**1. INTRODUCTION TO REACTJS**

- **Definition:** A JavaScript library created by Facebook for building user interfaces (UIs) or UI components. It is known for being fast, scalable, and simple. Often considered the "View" in an MVC (Model-View-Controller) architecture.
- **Why React?**
  - Efficiently creates dynamic web applications with less code compared to vanilla JS.
  - Improved performance due to the **Virtual DOM**.
  - Promotes **reusable UI components**, making code modular and easier to maintain.
  - **Unidirectional data flow** makes debugging easier and application state more predictable.
  - Has dedicated browser **debugging tools** (React DevTools).
- **Key Features (Building Blocks of React):** [PYQ Q1(c)]
  - **JSX (JavaScript XML):** A syntax extension for JavaScript that allows writing HTML-like structures directly within JS code to describe the UI. It is transpiled by tools like Babel into regular JavaScript function calls.
  - **Components:** Independent, reusable pieces of UI. They can be thought of as custom HTML elements. (More details below).

- **Virtual DOM:** [PYQ Q1(b) Differentiate between: Shadow DOM and Virtual DOM.]
  - *Theory (Virtual DOM):* The Virtual DOM (VDOM) is a lightweight, in-memory representation (a JavaScript object tree) of the actual Real DOM. When a component's state changes, React creates a new VDOM tree. It then "diffs" this new VDOM tree with the previous VDOM tree to identify the minimal changes required. Finally, React efficiently updates only those changed parts in the Real DOM, rather than re-rendering the entire page. This process is key to React's performance.
  - *Shadow DOM:* The Shadow DOM is a browser technology designed for encapsulation in web components. It allows a component to have its own isolated DOM tree ("shadow tree") and scoped CSS, preventing styles from leaking in or out. It's about creating encapsulated components, whereas the Virtual DOM is a React-specific concept for performance optimization.

| Feature | Virtual DOM | Shadow DOM |
|---|---|---|
| **Purpose** | Performance optimization (efficient DOM updates) | Encapsulation (scoped CSS, isolated DOM subtree) |
| **Nature** | JavaScript library concept (e.g., in React) | Browser standard/feature (Web Components API) |
| **Scope** | Manages representation of the entire component tree | Scopes a part of the DOM for a specific element |
| **Interaction** | Updates Real DOM based on diffs | Creates an isolated DOM fragment within an element |

- **One-way Data Binding (Unidirectional Data Flow):** Data flows in a single direction, typically from parent components down to child components via props. This makes applications easier to reason about and debug.
- **High Performance:** Achieved through the Virtual DOM and efficient update mechanisms.

**2. GETTING STARTED WITH REACT APP**

- **Prerequisites:** Node.js and npm (Node Package Manager) must be installed.
- **Create React App (CRA):** `npx create-react-app my-app` (A popular toolchain for creating React projects with no build configuration).
- **Vite (Alternative):** `npm create vite@latest my-app -- --template react` (A newer, faster build tool).
- **Running the App:** `cd my-app`, then `npm start` (for CRA) or `npm run dev` (for Vite).
- **Project Structure (Typical for CRA):**
  - `node_modules/`: Contains all project dependencies.
  - `public/`: Contains static assets like `index.html`, favicons, etc.
  - `src/`: Contains the React application code (e.g., `App.js`, `index.js`, components).
  - `package.json`: Lists project dependencies and scripts.

**3. TEMPLATING USING JSX**

- **Definition:** A syntax extension for JavaScript that looks very similar to HTML. It allows you to write UI structures declaratively within your JavaScript code. JSX is transpiled (converted) by tools like Babel into `React.createElement()` calls (plain JavaScript).
- **Embedding JavaScript Expressions:** Use curly braces `{}` to embed any valid JavaScript expression within JSX (e.g., variables, function calls, arithmetic operations).
  - Example: `<h1>Hello, {userName}</h1>` or `<div>{2 + 2}</div>`
- **Attributes:** JSX attributes generally follow HTML attribute naming conventions but use camelCase for multi-word attributes (e.g., `className` instead of `class`, `htmlFor` instead of `for`, `onClick` instead of `onclick`).
  - Attribute values can be strings (in quotes) or JavaScript expressions (in curly braces): `className="my-class"` or `style={{ color: 'blue' }}`.
- **Single Root Element:** A JSX expression returned by a component must have a single root element. If you need to return multiple elements, wrap them in a single parent `<div>`, a `<React.Fragment>`, or the shorthand `<>...</>`.

**4. CLASS COMPONENTS (Legacy, but important to understand)** [PYQ Q4(a)]

- Defined using ES6 classes that extend `React.Component`.
- Must implement a `render()` method that returns JSX to describe the UI.
- Can have local state (managed via `this.state` and `this.setState()`).
- Can utilize lifecycle methods (e.g., `componentDidMount`, `componentDidUpdate`, `componentWillUnmount`).
- Before Hooks, class components were necessary for state and lifecycle features.
- ES5 used `React.createClass()` (now deprecated).

**5. COMPONENTS** [PYQ Q4(a) Explain the components in React JS. Write the differences between class and functional components with example?]

- *Theory:* Components are the fundamental building blocks of a React UI. They are independent, reusable pieces of code that encapsulate a part of the user interface. Components accept inputs called "props" and return React elements (describing what should appear on the screen).
- Component names must start with a capital letter (e.g., `Welcome`, `Button`).
- **Types of Components:**
  1. **Functional Components:**
     - Defined as simple JavaScript functions that accept `props` as an argument and return JSX.
     - Simpler syntax, often preferred for presentational components.
     - With React Hooks (like `useState` and `useEffect`), functional components can now manage state and side effects, making them as powerful as class components.

```jsx
// Functional Component Example (PYQ Q4(a))
function Welcome(props) {
  return <h1>Hello, {props.name}!</h1>;
}
```

2. **Class Components:** (Described in section 4 above)

```jsx
// Class Component Example (PYQ Q4(a))
import React from 'react'; // or import { Component } from 'react';

class WelcomeMessage extends React.Component { // or extends Component
  render() {
    return <h1>Hello from class, {this.props.message}!</h1>;
  }
}
```

- **Differences: Class vs. Functional Components:** [PYQ Q4(a)]

| Feature | Class Components | Functional Components (with Hooks) |
|---|---|---|
| **Syntax** | ES6 Class (`class MyComponent extends React.Component`) | JavaScript Function (`function MyComponent(props)`) |
| **State** | `this.state` (object), updated with `this.setState()` | `useState()` Hook (manages individual state variables) |
| **Lifecycle** | Lifecycle methods (e.g., `componentDidMount`) | `useEffect()` Hook (for side effects, replaces most lifecycle methods) |
| **`this` Keyword** | Used extensively to access props, state, methods | Not used (props are direct arguments, state via Hook variables) |
| **Props Access** | `this.props.propName` | `props.propName` (or destructure: `{ propName }`) |
| **Readability** | Can be more verbose | Generally more concise and easier to read/test |
| **Performance** | Potentially slightly more overhead | Can be more optimized by React, easier to optimize |
| **Modern React** | Less common for new code; Hooks are preferred | The standard for new React code |

- **Stateful vs. Stateless Components (Conceptual distinction, often related to PYQ Q4(a)):**
  - **Stateless Components (Often Functional pre-Hooks, or purely presentational):**
    - Do not manage any internal state that changes over time.
    - Their output is solely determined by the `props` they receive.
    - Primarily responsible for rendering UI based on props.

        - Often simpler, easier to test, and more reusable.
    - **Stateful Components (Class Components, or Functional Components using `useState`):**
        - Manage internal data (`state`) that can change due to user interactions, network responses, etc.
        - When state changes, the component re-renders to reflect the new state.
        - Often responsible for application logic and data manipulation.
- **Comments in JSX:** Use curly braces with JavaScript block comments: `{/* This is a comment in JSX */}` or single-line comments: `{ // This is also a comment }` within JSX elements.
- **Embedding Components:** Components can be used (rendered) within other components by referencing them like HTML tags: `<MyComponent propName="value" />`.

**6. STATE AND PROPS** [PYQ Q5(a) Explain State and Props in React JS. Give an example to update the state of component.]

- **Props (Properties):**
    - *Theory:* Props (short for properties) are a way to pass data from a parent component to its child components. They are read-only for the child component (immutable from the child's perspective). Props allow components to be configured and customized, making them reusable and dynamic. Data flows unidirectionally (parent to child).
    - Passed as attributes in JSX: `<ChildComponent name="Alice" age={30} />`.
    - Accessed in functional components via the `props` argument: `props.name`.
    - Accessed in class components via `this.props`: `this.props.name`.
- **State:**
    - *Theory:* State is data that is private and managed *within* a component. It can change over time due to user interactions, network responses, or other events. When a component's state changes, React automatically re-renders the component (and its children) to reflect the new state.
    - **In Class Components:**
        - Initialized in the `constructor`: `this.state = { key: value };`
        - Accessed via `this.state.key`.
        - **Updated using `this.setState({ key: newValue })`**. Never modify `this.state` directly, as it won't trigger a re-render. `setState()` is asynchronous and may batch updates.
    - **In Functional Components (using Hooks):**
        - The `useState` Hook is used to add state. (See React Hooks section below).
    - *Example to Update State (PYQ Q5(a) - Functional Component with useState):*

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

function Counter() {
  // Declare a state variable 'count' and a function 'setCount' to
update it.
  // Initialize 'count' to 0.
  const [count, setCount] = useState(0);

  const incrementCount = () => {
    setCount(count + 1); // Update the state
  };

  return (
    <div>
      <p>Current Count: {count}</p>
      <button onClick={incrementCount}>Increment</button>
    </div>
  );
}
export default Counter;
```

- **Props vs. State Comparison Table:** [PYQ Q5(a)]

| Feature | Props | State |
|---|---|---|
| **Data Flow** | Passed from parent to child (Unidirectional) | Managed internally within the component |
| **Mutability** | Immutable (Read-only by the child component) | Mutable (Can be changed by the component via `setState` or setter fn) |
| **Ownership** | Owned by the parent component, passed down | Owned and managed by the component itself |
| **Purpose** | Configure/customize child components, pass data | Manage dynamic data internal to a component that changes over time |
| **Initialization** | Passed by the parent during component rendering | Initialized within the component (constructor or `useState`) |
| **Access** | `props.propName` or `this.props.propName` | `stateVariable` (Hooks) or `this.state.key` (Classes) |
| **Source** | External to the component (from parent) | Internal to the component |

**7. REACT HOOKS** [PYQ Q4(b) Define React Hooks. Demonstrate the useState hook and useEffect hook in react?]

- *Theory:* Hooks are functions that let you "hook into" React state and lifecycle features from functional components. They allow you to use state and other React features without writing a class.

Hooks were introduced in React 16.8.

- **Rules of Hooks:**
  - Only call Hooks at the top level (not inside loops, conditions, or nested functions).
  - Only call Hooks from React function components (or custom Hooks).
- **`useState` Hook:** [PYQ Q4(b)] Adds state to functional components.
  - *Syntax:* `const [stateVariable, setStateFunction] = useState(initialState);`
    - `stateVariable`: The current value of the state.
    - `setStateFunction`: A function to update the `stateVariable` and trigger a re-render.
    - `initialState`: The initial value of the state variable.
  - *Demonstration (PYQ Q4(b)):*

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

function CounterWithHook() {
  // Initialize 'count' state to 0
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      {/* Call setCount to update the state when button is clicked */}
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
export default CounterWithHook;
```

- **`useEffect` Hook:** [PYQ Q4(b)] Lets you perform side effects in functional components. Side effects include data fetching, subscriptions, manually changing the DOM, timers, etc.
  - `useEffect` runs after every render by default, but you can control when it runs by providing a dependency array.
  - *Syntax:* `useEffect(() => { /* side effect code */; return () => { /* cleanup code (optional) */ }; }, [dependencies (optional)]);`
    - The first argument is a function containing the side effect logic.
    - The optional second argument is an array of dependencies. `useEffect` will only re-run if one of these dependencies has changed since the last render.
      - No dependency array: Runs after every render.

    - Empty array `[]`: Runs only once after the initial render (like `componentDidMount`).
    - Array with values `[var1, var2]`: Runs after initial render and whenever `var1` or `var2` changes.
  - The optional return function is for cleanup (e.g., unsubscribing, clearing timers), executed before the component unmounts or before the effect runs again.
- *Demonstration (PYQ Q4(b) - updates document title):*

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

function DocumentTitleUpdater() {
  const [count, setCount] = useState(0);
  const [name, setName] = useState('');

  // useEffect to update document title when 'count' changes
  useEffect(() => {
    console.log('useEffect ran due to count change');
    document.title = `You clicked ${count} times`;

    // Cleanup function (optional)
    return () => {
      console.log('Cleanup from title effect (count changed or
unmount)');
    };
  }, [count]); // Dependency array: only re-run if 'count' changes

  // Another useEffect for a different side effect (e.g., logging name)
  useEffect(() => {
    console.log(`Name changed to: ${name}`);
  }, [name]); // Only re-run if 'name' changes

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment
Count</button>
      <input type="text" value={name} onChange={(e) =>
setName(e.target.value)} placeholder="Enter name" />
      <p>Name: {name}</p>
    </div>
  );
}
export default DocumentTitleUpdater;
```

**8. LIFECYCLE OF COMPONENTS (Primarily for Class Components; `useEffect` handles this for Functional Components)**

- Components go through several phases in their life:
  - **Mounting (Birth of the component):** When an instance of a component is being created and inserted into the DOM.
    - `constructor(props)`: Called before mounting. Used for initializing state and binding event handlers.
    - `static getDerivedStateFromProps(props, state)`: (Rarely used) Called right before `render()`. Returns an object to update state, or `null` if no update is needed.
    - `render()`: **Required method.** Examines `this.props` and `this.state` and returns React elements (JSX) describing the UI. Pure function.
    - `componentDidMount()`: Called immediately after the component is rendered into the DOM. Good place for network requests, subscriptions, or DOM manipulations.
  - **Updating (Growth of the component):** When a component's props or state change, triggering a re-render.
    - `static getDerivedStateFromProps(props, state)`: (Rarely used) Called on every re-render.
    - `shouldComponentUpdate(nextProps, nextState)`: (Rarely used for performance optimization) Determines if React should skip re-rendering. Returns `true` by default.
    - `render()`: Called to generate the updated UI.
    - `getSnapshotBeforeUpdate(prevProps, prevState)`: (Rarely used) Called right before the changes from the VDOM are reflected in the DOM. Allows capturing some information (e.g., scroll position) before it changes.
    - `componentDidUpdate(prevProps, prevState, snapshot)`: Called immediately after updating occurs (not for initial render). Good for network requests based on prop/state changes or DOM operations.
  - **Unmounting (Death of the component):** When a component is being removed from the DOM.
    - `componentWillUnmount()`: Called right before a component is unmounted and destroyed. Used for cleanup (e.g., invalidating timers, canceling network requests, removing event listeners).

**9. RENDERING LISTS**

- Use the JavaScript `map()` method on an array to transform each item into a React element (JSX).
- **Keys:** A special `key` prop is required for list items when rendering collections of elements.
  - Keys must be unique strings among siblings in the list.
  - React uses keys to identify which items have changed, are added, or are removed, enabling efficient updates to the VDOM and Real DOM.

- Best to use stable, unique IDs from your data as keys. Avoid using the array index as a key if the order of items can change, as this can lead to performance issues and bugs with component state.

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

**10. PORTALS**

- Portals provide a way to render children into a DOM node that exists outside the DOM hierarchy of the parent component.
- Common use cases: Modals, dialogs, tooltips, popovers that need to break out of their container's stacking context or overflow.
- Syntax: `ReactDOM.createPortal(child, domNodeContainer)`
  - `child`: Any renderable React child (element, string, fragment).
  - `domNodeContainer`: An existing DOM element where the child will be rendered.
- Event bubbling works through portals as if the portal content were a normal React child in the React tree, even though it's in a different DOM tree.

**11. ERROR HANDLING (ERROR BOUNDARIES)**

- Error boundaries are React class components that catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI instead of the component tree that crashed.
- To create an error boundary, a class component must define one or both of these lifecycle methods:
  - `static getDerivedStateFromError(error)`: Renders a fallback UI after an error has been thrown. Returns an object to update state.
  - `componentDidCatch(error, errorInfo)`: Logs error information.
- Error boundaries do *not* catch errors for:
  - Event handlers (use `try...catch` inside them).
  - Asynchronous code (e.g., `setTimeout` or `requestAnimationFrame` callbacks).
  - Server-side rendering.
  - Errors thrown in the error boundary itself (rather than its children).

**12. ROUTERS (REACT ROUTER)** [PYQ Q5(b)(i) React Router]

- React Router is a standard library for handling navigation and routing in React Single Page Applications (SPAs) without a full page refresh.
- It enables mapping different URLs to different components, allowing users to navigate between views.
- Installation: `npm install react-router-dom`
- **Key Components (React Router v6):**
  - `<BrowserRouter>`: Wraps your application, uses HTML5 history API to keep UI in sync with URL.
  - `<Routes>`: Container for multiple `<Route>` components. It looks through its children `<Route>`s to find the best match for the current URL.
  - `<Route path="/your-path" element={<YourComponent />} />`: Defines a mapping between a URL path and a React component that should be rendered when the path matches.
  - `<Link to="/your-path">Link Text</Link>`: Component for creating navigation links, similar to HTML `<a>` tags but aware of the router.
  - `useNavigate` hook: For programmatic navigation.
  - `useParams` hook: To access URL parameters.

**13. REDUX** [PYQ Q5(b)(ii) Redux]

- Redux is a predictable state container for JavaScript applications, often used with React (but can be used with any UI library/framework or vanilla JS). It helps manage global application state, especially in large applications where state needs to be shared across many components.
- **Core Concepts:**
  - **Store:** A single JavaScript object that holds the entire application state. It's the "single source of truth." Created using `createStore(reducer)` from the `redux` library (or `configureStore` from `@reduxjs/toolkit`).
  - **Action:** A plain JavaScript object that describes an intention to change the state. Actions must have a `type` property (usually a string constant) and can optionally have a `payload` with data. (e.g., `{ type: 'ADD_TODO', payload: { text: 'Learn Redux' } }`).
  - **Reducer:** A pure function that takes the previous `state` and an `action` as arguments, and returns the `nextState`. `(previousState, action) => newState`. Reducers specify how the application's state changes in response to actions. They must be pure (no side effects, same input gives same output).
  - **Dispatch:** The method `store.dispatch(action)` is used to send actions to the store. This is the only way to trigger a state change.
  - **Subscribe:** The method `store.subscribe(listenerFunction)` allows registering a callback that will be called whenever the state in the store changes. (Often handled by React-Redux

library bindings).

- **Redux vs. Flux:**
  - **Flux:** An architecture pattern (not a library) by Facebook. Can have multiple stores, a central dispatcher, and actions. State is mutable within stores.
  - **Redux:** A library inspired by Flux and Elm. Enforces a single store, uses reducers (pure functions) for state updates, and emphasizes immutable state.

**14. REDUX SAGA**

- Redux Saga is a middleware for Redux designed to handle side effects (asynchronous operations like API calls, accessing browser cache, etc.) in a more organized and testable way.
- It uses ES6 Generator functions (`function* mySaga() {}`) to make asynchronous flows look synchronous and easier to manage.
- Sagas listen for dispatched Redux actions and can trigger other actions or interact with the outside world.
- Common Effects: `call` (for calling functions), `put` (for dispatching actions), `takeEvery` / `takeLatest` (for listening to actions), `select` (for getting state from the store).
- Helps keep reducers pure by offloading side effect logic to Sagas.

**15. IMMUTABLE.JS**

- A library (by Facebook) that provides persistent immutable data structures like `List`, `Map`, `Set`, etc.
- When you "change" an immutable data structure, it returns a new instance with the changes, while the original structure remains unchanged.
- **Benefits:**
  - **Performance:** Makes change detection very efficient, especially useful for React's `shouldComponentUpdate` or Redux state comparisons (can compare by reference).
  - **Easier Change Tracking & Debugging:** Simplifies understanding how data has changed over time.
  - **Predictability:** Reduces unintended side effects as data structures cannot be altered in place.
- **Considerations:** Learning curve, potential performance overhead for converting to/from native JS objects (`toJS()`, `fromJS()`), adds to bundle size.

**16. SERVICE SIDE RENDERING (SSR) WITH REACT**

- Server-Side Rendering is the process of rendering React components on the server into an HTML string, which is then sent to the browser.
- The browser can display the HTML content immediately, and then client-side JavaScript ("hydration") takes over to make the page interactive.
- **Benefits:**

    - **Improved SEO:** Search engine crawlers can better index content that is present in the initial HTML.
    - **Faster Perceived Performance (Time To First Contentful Paint - FCP):** Users see content sooner, as they don't have to wait for all JS to download and execute.
- **Frameworks for SSR with React:** Next.js, Remix.
- Can also be implemented customly using Node.js and `ReactDOMServer.renderToString()` or `renderToPipeableStream()`.

17. **UNIT TESTING (IN REACT)** [PYQ Q5(b)(iii) Unit Testing]

- Unit testing involves testing individual components or functions of an application in isolation to ensure they work correctly.
- **Purpose:** Verify that each unit of the software performs as designed.
- **In React, you typically test:**
    - Component rendering (does it render without crashing?).
    - Output given specific props.
    - Behavior in response to user interactions (e.g., button clicks, form submissions).
    - Conditional rendering logic.
    - State changes.
- **Common Tools:**
    - **Jest:** A popular JavaScript testing framework (test runner, assertion library, mocking). Often comes pre-configured with Create React App.
    - **React Testing Library (RTL):** A library that encourages testing components in a way that resembles how users interact with them. Focuses on querying the DOM by accessible roles, text, etc., rather than internal implementation details.
    - **Enzyme:** (Older library, less common for new projects) Provides utilities for asserting, manipulating, and traversing React Components' output.

18. **WEBPACK**

- Webpack is a powerful static module bundler for modern JavaScript applications.
- It takes modules with dependencies (JavaScript files, CSS, images, etc.) and generates static assets (bundles) representing those modules, which can be served to a browser.
- **Core Concepts:**
    - **Entry:** The starting point(s) of the application's dependency graph.
    - **Output:** Specifies where to emit the bundled files and how to name them.
    - **Loaders:** Transform non-JavaScript files (e.g., CSS, SASS, TypeScript, images, fonts) into modules that can be included in the dependency graph. They also transpile newer JS syntax (e.g., ES6+ via Babel-loader).

- **Plugins:** Perform a wider range of tasks like bundle optimization, asset management, environment variable injection (e.g., `HtmlWebpackPlugin` to generate HTML files, `MiniCssExtractPlugin` to extract CSS into separate files).

- **Features:** Code splitting (breaking code into smaller chunks for on-demand loading), tree shaking (dead code elimination), development server (`webpack-dev-server`), Hot Module Replacement (HMR - updates modules in the browser without a full refresh).
- Tools like Create React App and Vite abstract away much of Webpack's (or similar bundler's) configuration for common use cases.

---

**UNIT 3: NODE.JS AND EXPRESS.JS**

**I. INTRODUCTION TO NODE.JS**

- **What is Node.js?** Node.js is an open-source, cross-platform JavaScript runtime environment. It allows developers to execute JavaScript code *outside* of a web browser, primarily for building server-side applications and networking tools. It uses Google's V8 JavaScript engine (the same engine that powers Chrome).
- **Key Features & Significance in Backend Development:** [PYQ Q6(a) Explain the significance of Node.js in backend development. How Node.js differs from traditional server-side technologies.]
  - **Asynchronous and Event-Driven:** Node.js uses a non-blocking I/O model. Most I/O operations (like reading files, network requests) are asynchronous, meaning Node.js doesn't wait for them to complete before moving on to other tasks. It relies on callbacks, Promises, and async/await to handle results. This makes it highly efficient for I/O-bound applications.
  - **Single-Threaded (with Event Loop):** [PYQ Q6(c) Explain the Node.js event loop mechanism. How does it help in handling asynchronous operations efficiently?]
    - *Theory (Event Loop):* The Event Loop is the core of Node.js's concurrency model. JavaScript itself is single-threaded (executes one command at a time). However, Node.js offloads long-running I/O operations to the system kernel (which often uses multiple threads). When an I/O operation completes, the kernel informs Node.js, and the corresponding callback function is added to an event queue. The event loop continuously monitors this queue and processes callbacks one by one in the main thread.
    - *Efficiency (Event Loop):* This mechanism allows a single Node.js process to handle thousands of concurrent connections efficiently without the overhead of creating and managing many threads (as in traditional multi-threaded servers). The main thread remains unblocked and available to handle new requests while I/O operations are pending.
  - **Non-Blocking I/O:** As mentioned, I/O operations do not block the execution of other code.
  - **npm (Node Package Manager):** Node.js has the world's largest ecosystem of open-source libraries and packages (managed via npm), greatly accelerating development.
  - **Cross-Platform:** Runs on Windows, macOS, and Linux.

- **Uses JavaScript:** Enables full-stack JavaScript development (using JS for both frontend and backend), reducing context switching for developers.
- **Performance:** Generally excellent for I/O-intensive applications, real-time applications (chats, live updates, streaming), APIs, and microservices.
- **Scalability:** Lightweight and can be scaled horizontally (by running multiple instances and distributing load) and vertically.

- **Node.js vs. Traditional Server-Side Technologies:** [PYQ Q6(a)]
  - **Language:** Node.js uses JavaScript, while traditional technologies often use Java, C#, PHP, Python, Ruby, etc.
  - **Concurrency Model:** Node.js is single-threaded with an event loop and non-blocking I/O. Many traditional servers use a multi-threaded model where each request might be handled by a separate thread (which can be resource-intensive).
  - **Performance:** Node.js often excels in scenarios with many concurrent connections and high I/O, due to its non-blocking nature. Traditional threaded servers might perform better for CPU-bound tasks if not properly optimized for I/O.
  - **Development Paradigm:** Node.js inherently encourages asynchronous programming patterns.

**II. SETTING UP NODE.JS**

- **Installation:** Download the LTS (Long Term Support) version from the official Node.js website (nodejs.org). npm (Node Package Manager) is included with the Node.js installation.
- **Verify Installation:** Open a terminal or command prompt and type `node -v` and `npm -v` to check the installed versions.
- **Basic Project Setup:**
  1. Create a project directory: `mkdir my-node-app && cd my-node-app`
  2. Initialize the project (creates `package.json`): `npm init -y` (the `-y` flag accepts default settings).
- **Managing Dependencies:**
  - Install a package: `npm install <package-name>` (e.g., `npm install express`). This adds it to `dependencies` in `package.json` and `node_modules/`.
  - Install a dev package: `npm install <package-name> --save-dev` or `npm install -D <package-name>`.
- **Running a Node.js Script:** `node app.js` (assuming `app.js` is your main file).

**III. NODE.JS CORE CONCEPTS**

- **Modules:** Reusable blocks of code. Node.js has a built-in module system.
  - **Types of Modules:**
    1. **Core Modules:** Built-in modules provided by Node.js (e.g., `http`, `fs`, `path`, `events`). Imported using `require('module-name')`.

2. **Local/Custom Modules:** Modules created by the developer within the project. Exported using `module.exports` or `exports` and imported using `require('./path/to/module')`.
3. **Third-Party Modules:** Modules installed from npm (e.g., `express`, `lodash`). Imported using `require('package-name')`.

- **ES6 Modules:** Node.js also supports ES6 module syntax (`import`/`export`). To use ES6 modules, you can either:
  - Set `"type": "module"` in your `package.json`.
  - Use the `.mjs` file extension for your module files.

- **Asynchronous Operations Handling:** [PYQ Q6(c)] Node.js relies heavily on asynchronous patterns to remain non-blocking.
  - **Callbacks:** [PYQ Q1(d) What is meant by “Callback” and “Callback Hell” in Node JS?]
    - *Theory (Callback):* A callback is a function passed as an argument to another function, which is then invoked (called back) inside the outer function to complete some kind of routine or action, often after an asynchronous operation has finished.
    - **Callback Hell (Pyramid of Doom):** [PYQ Q1(d)] Refers to deeply nested callback functions, which can make code difficult to read, debug, and maintain. This often arises when multiple sequential asynchronous operations depend on each other.
    - *Structure (Callback Hell):*

```
asyncOperation1(data, function(err, result1) {
    if (err) { /* handle error */ return; }
    asyncOperation2(result1, function(err, result2) {
        if (err) { /* handle error */ return; }
        asyncOperation3(result2, function(err, result3) {
            if (err) { /* handle error */ return; }
            // ... and so on
        });
    });
});
```

    - *Avoidance:* Callback hell can be mitigated using Promises, Async/Await, named functions, or modularizing code.
  - **Promises:** Objects representing the eventual completion (or failure) of an asynchronous operation and its resulting value. They allow chaining asynchronous operations using `.then()` for success and `.catch()` for errors, improving readability over nested callbacks.
  - **Async/Await (ES2017):** Syntactic sugar built on top of Promises, making asynchronous code look and behave a bit more like synchronous code, further improving readability and maintainability. `async` functions return a Promise, and `await` pauses execution until a Promise settles.

- **Event Loop (Detailed Phases):** [PYQ Q6(c) Node.js event loop mechanism]

    - The event loop is the heart of Node.js's non-blocking, asynchronous architecture. It's a constantly running process that checks various event queues and executes their corresponding callbacks.
    - **Simplified Phases (in one iteration/tick):**
        1. **Timers:** Executes callbacks scheduled by `setTimeout()` and `setInterval()`.
        2. **Pending Callbacks (I/O Callbacks):** Executes almost all I/O callbacks deferred to the next loop iteration (e.g., network, file system errors).
        3. **Idle, Prepare:** Internal use only.
        4. **Poll:** Retrieves new I/O events; executes I/O-related callbacks (e.g., for incoming connections, data from files). If the poll queue is empty, it will block here if there are no timers or `setImmediate` calls.
        5. **Check:** Executes callbacks scheduled by `setImmediate()`.
        6. **Close Callbacks:** Executes close event callbacks (e.g., `socket.on('close', ...)`).
    - `process.nextTick()`: Callbacks registered with `process.nextTick()` are not technically part of the event loop's phases but are executed immediately after the current operation completes, before the event loop continues to the next phase. They have higher priority.
- **Events & `EventEmitter`:** Many objects in Node.js emit events (e.g., an HTTP server emits an event each time a request comes in, a stream emits an event when data is available).
    - The `events` module provides the `EventEmitter` class, which is used to create, fire, and listen for custom events.
    - Key methods: `on('eventName', listener)` (add listener), `emit('eventName', ...args)` (fire event), `once('eventName', listener)` (add one-time listener), `off('eventName', listener)` (remove listener).
- **File System (`fs` module):** Provides an API for interacting with the file system (reading, writing, updating, deleting files and directories).
    - Offers both synchronous (blocking) and asynchronous (non-blocking) methods. Asynchronous methods (callback-based, Promise-based) are generally preferred.
    - Promise-based API: `const fs = require('fs').promises;`
- **Streams:** [PYQ Q7(b)(i) Give brief description on following: Streams in Node JS]
    - *Theory:* Streams are objects that let you read data from a source or write data to a destination in a continuous fashion, piece by piece (chunks), without loading the entire data into memory at once. They are instances of `EventEmitter`.
    - *Significance:* Extremely useful for handling large amounts of data (e.g., large files, network responses) efficiently.
    - *Types:*
        1. **Readable Streams:** For reading data (e.g., `fs.createReadStream()`, HTTP request).
        2. **Writable Streams:** For writing data (e.g., `fs.createWriteStream()`, HTTP response).

3. **Duplex Streams:** Both readable and writable (e.g., TCP sockets).
4. **Transform Streams:** Duplex streams that can modify or transform data as it is written and read (e.g., `zlib.createGzip()` for compression).

- `pipe()`: A common method to connect a Readable stream to a Writable stream, e.g., `readable.pipe(writable)`.

- **Buffers:** Used to handle binary data directly. A Buffer is a fixed-size chunk of memory allocated outside the V8 JavaScript heap.
- **Command Line Interaction:**
  - Running scripts: `node script.js arg1 arg2`
  - REPL (Read-Eval-Print Loop): Accessed by typing `node` in the terminal.
  - `process.argv`: Array containing command line arguments.
  - `readline` module: For interactive command line input.
  - `child_process` module: To run external shell commands or spawn other Node.js processes.
- **`console` Module:** Provides debugging and logging capabilities (e.g., `console.log()`, `console.error()`, `console.warn()`, `console.table()`, `console.time()`/`console.timeEnd()`, `console.assert()`).
- **Concurrency Handling (via Single Thread):** Achieved primarily through the event loop and non-blocking I/O. For I/O operations, Node.js delegates tasks to the OS kernel or a thread pool (like libuv's thread pool), and callbacks are queued to be executed by the main thread once the tasks are complete.
- **Worker Threads (`worker_threads` module):** For CPU-intensive tasks that would otherwise block the main event loop. Allows running JavaScript code in parallel on separate threads, communicating with the main thread via message passing.
- **Clustering (`cluster` module):** Allows creating multiple Node.js processes (forks) that can share the same server port. This enables network applications to utilize multi-core CPUs more effectively. A master process manages worker processes.
- **Forking (`child_process.fork()`):** A special case of `child_process.spawn()` for creating new Node.js instances with an IPC (Inter-Process Communication) channel established.
- **Timing Features:** `setTimeout()`, `setInterval()` (schedule code execution after a delay/repeatedly), `setImmediate()` (execute code after current I/O poll phase), `process.nextTick()` (execute code immediately after current operation). `perf_hooks` module for performance measurement.

### IV. INTRODUCTION TO EXPRESS.JS

- **What is Express.js?** Express.js is a minimal, flexible, and unopinionated Node.js web application framework. It provides a robust set of features for building web and mobile applications, particularly APIs. It simplifies tasks like routing, middleware integration, and request/response handling.
- **Why Use Express.js?**

- Simplifies building web servers and APIs in Node.js.
- Provides helpful utilities for routing and HTTP request handling.
- Has a large and active ecosystem of middleware for various functionalities.
- Its unopinionated nature gives developers flexibility in how they structure their applications.

### V. EXPRESS.JS CORE CONCEPTS

- **Basic Setup:**
  1. Install: `npm install express`
  2. Create an `app.js` (or `server.js`) file:

```
const express = require('express');
const app = express(); // Create an Express application instance
const port = 3000;

// Define routes (see below)
app.get('/', (req, res) => {
  res.send('Hello World!');
});

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

- **Routing:** Defines how an application's endpoints (URIs) respond to client requests (based on HTTP method and path).
  - Syntax: `app.METHOD(PATH, HANDLER_FUNCTION)`
    - `METHOD`: An HTTP method in lowercase (e.g., `get`, `post`, `put`, `delete`).
    - `PATH`: A URL path on the server (e.g., `/`, `/users`, `/products/:id`).
    - `HANDLER_FUNCTION`: A function executed when the route is matched. It typically receives `(req, res, next)` arguments.
  - **Route Parameters (`req.params`):** Capture dynamic segments in the URL path (e.g., in `/users/:userId`, `req.params.userId` will contain the value).
  - **Query Parameters (`req.query`):** Key-value pairs appended to the URL after a `?` (e.g., in `/search?term=node`, `req.query.term` will be `"node"`).
  - **`express.Router`:** A mini Express application for creating modular, mountable route handlers. Useful for organizing routes into separate files.
  - **Route Chaining:** Group handlers for the same route path but different HTTP methods: `app.route('/items').get(handler1).post(handler2);`

- **Middleware:** [PYQ Q7(a) Discuss the concept of middleware in Express.js. Give two examples of middleware functions and write their purposes.]
  - *Theory:* Middleware functions are functions that have access to the request object (`req`), the response object (`res`), and the `next` middleware function in the application's request-response cycle. They can:
    1. Execute any code.
    2. Make changes to the request and the response objects.
    3. End the request-response cycle.
    4. Call the next middleware function in the stack (using `next()`).
  - *Types of Middleware:*
    - **Application-level middleware:** Bound to an instance of `app` using `app.use()` or `app.METHOD()`.
    - **Router-level middleware:** Works like application-level but bound to an instance of `express.Router()`.
    - **Route-specific middleware:** Passed directly as a handler to a route definition.
    - **Error-handling middleware:** Defined with four arguments `(err, req, res, next)`. Must be defined last, after other `app.use()` and routes calls.
    - **Built-in middleware:** Provided by Express (e.g., `express.json()`).
    - **Third-party middleware:** Installed via npm (e.g., `cors`, `morgan`).
  - *Example 1 (Logger Middleware - PYQ Q7(a)):*

```
const loggerMiddleware = (req, res, next) => {
  console.log(`[${new Date().toISOString()}] ${req.method}
${req.originalUrl}`);
  // Purpose: To log details of every incoming request (timestamp,
method, URL).
  next(); // Purpose: To pass control to the next middleware function
in the stack.
};
app.use(loggerMiddleware); // Apply to all requests
```

  - *Example 2 (Authentication Middleware - Conceptual for PYQ Q7(a)):*

```
const ensureAuthenticated = (req, res, next) => {
  if (req.session && req.session.user) { // Assuming session management
is set up
    // Purpose: To verify if the user is authenticated. If yes, allow
access.
    return next();
  } else {
    // Purpose: To block access for unauthenticated users and send an
```

```
    error response.
        res.status(401).send('Unauthorized: Please log in.');
    }
};
app.get('/protected-route', ensureAuthenticated, (req, res) => {
    res.send('This is a protected route.');
});
```

- **Handling HTTP Requests and Responses:** [PYQ Q6(b) Illustrate the process of handling HTTP requests and responses using Express.js.]
    - *Theory:* Express provides `req` (request) and `res` (response) objects as arguments to route handler functions and middleware.
        - The `req` object represents the incoming HTTP request and contains properties like `req.params`, `req.query`, `req.body`, `req.headers`, `req.method`, `req.url`.
        - The `res` object represents the HTTP response that an Express app sends when it gets an HTTP request. It has methods to send data back to the client (e.g., `res.send()`, `res.json()`, `res.status()`).
    - *Process:*
        1. A client (e.g., browser, mobile app) sends an HTTP request to a specific URL and method on the Express server.
        2. Express matches the request's path and HTTP method to a defined route.
        3. Any applicable middleware functions execute sequentially, potentially modifying `req` or `res`, or ending the cycle.
        4. If not ended by middleware, the matched route handler function executes.
        5. The handler uses the `req` object to access request details (e.g., `req.params.id` for route parameters, `req.body` for POST data if `express.json()` or `express.urlencoded()` middleware is used).
        6. The handler uses the `res` object to construct and send an HTTP response back to the client (e.g., `res.json(data)`, `res.send('Response text')`, `res.status(404).send('Not Found')`, `res.render('template')`).
    - *Example (PYQ Q6(b)):*

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

// Middleware to parse JSON request bodies
app.use(express.json());
// Middleware to parse URL-encoded request bodies
app.use(express.urlencoded({ extended: true }));
```

```
let items = [{id: 1, name: "Sample Item"}]; // In-memory data store

// GET request to fetch all items
app.get('/items', (req, res) => {
  // req.query could be used here for filtering/pagination
  res.status(200).json(items); // Sends JSON response with status 200
});

// GET request to fetch a single item by ID
app.get('/items/:id', (req, res) => {
  const itemId = parseInt(req.params.id);
  const item = items.find(i => i.id === itemId);
  if (item) {
    res.status(200).json(item);
  } else {
    res.status(404).send('Item not found');
  }
});

// POST request to add a new item
app.post('/items', (req, res) => {
  const newItem = req.body; // Access request body (e.g., { name: "New Item" })
  if (!newItem.name) {
    return res.status(400).send('Item name is required');
  }
  newItem.id = items.length > 0 ? Math.max(...items.map(i => i.id)) + 1 : 1;
  items.push(newItem);
  res.status(201).json(newItem); // Sends 201 Created status and the new item
});

app.listen(port, () => {
  console.log(`Express server listening on port ${port}`);
});
```

- **Built-in Middleware:**
  - `express.json()`: Parses incoming requests with JSON payloads (populates `req.body`).
  - `express.urlencoded({ extended: true/false })`: Parses incoming requests with URL-encoded payloads (populates `req.body`).

    - `express.static('public')`: Serves static files (like images, CSS, client-side JS) from a specified directory (e.g., 'public').
- **Third-Party Middleware (Examples):**
    - `cors`: Enables Cross-Origin Resource Sharing.
    - `morgan`: HTTP request logger.
    - `cookie-parser`: Parses `Cookie` header and populates `req.cookies`.
    - `multer`: Handles `multipart/form-data` (primarily for file uploads).
    - `helmet`: Helps secure Express apps by setting various HTTP headers.
- **Request (`req`) Object Properties (Common):** `req.params`, `req.query`, `req.body`, `req.headers`, `req.method`, `req.url`, `req.ip`, `req.cookies`.
- **Response (`res`) Object Methods (Common):** `res.send()`, `res.json()`, `res.status()`, `res.sendStatus()`, `res.redirect()`, `res.render()` (for template engines), `res.sendFile()`, `res.set()` (sets response headers), `res.cookie()`.
- **Template Engines (View Engines):** Allow embedding dynamic data into HTML templates to generate HTML sent to the client.
    - Examples: EJS, Pug (formerly Jade), Handlebars.
    - Configuration: `app.set('view engine', 'ejs'); app.set('views', './views');` (sets EJS as engine and `views` as template directory).
    - Rendering: `res.render('templateName', { dataObject });`
- **File Uploads (`multer`):** Middleware for handling `multipart/form-data`, typically used for uploading files.
    - Configure storage (disk storage or memory storage) and file filters.
    - Access uploaded files via `req.file` (for a single file) or `req.files` (for multiple files).
- **Cookies (`cookie-parser`):** Middleware to parse the `Cookie` header from requests and populate `req.cookies` with an object keyed by cookie names. `res.cookie()` is used to set cookies.
- **Express Generator (Scaffolding):** [PYQ Q7(b)(ii) Give brief description on following: Express.js Scaffolding]
    - *Theory:* `express-generator` is a command-line tool (CLI) that quickly creates a basic skeleton structure for an Express.js application. This includes standard directories for routes, views (if a view engine is specified), public assets, and a main `app.js` file.
    - *Usage:*
        1. Install globally: `npm install -g express-generator`
        2. Generate app: `express my-express-app --view=ejs` (creates `my-express-app` with EJS as view engine).
        3. `cd my-express-app && npm install` (install dependencies).
        4. `npm start` (to run the app).

- **Common Project Structure (MVC-like pattern often adopted):**
  - `config/`: Configuration files (database, environment variables).
  - `controllers/` or `handlers/`: Logic for handling requests (often called by routes).
  - `models/`: Data schemas, database interaction logic (e.g., Mongoose models).
  - `routes/`: Route definitions, often organized by resource.
  - `views/`: Template files (if using a view engine).
  - `public/`: Static assets (CSS, client-side JS, images).
  - `middleware/`: Custom middleware functions.
  - `utils/`: Utility functions.
  - `app.js` or `index.js`: Main application setup, middleware configuration, starts the server.

**VI. NODE.JS DATABASE INTEGRATION**

- Node.js connects to various databases using specific driver libraries (npm packages).
- **MongoDB with Mongoose (ODM - Object Data Modeling):** Highly recommended for MongoDB.
  - Install: `npm install mongoose`
  - Connect: `mongoose.connect('mongodb_uri', { useNewUrlParser: true, useUnifiedTopology: true });`
  - Define Schema: Describes the structure of documents and data types. `const mySchema = new mongoose.Schema({ fieldName: String, age: Number });`
  - Create Model: A constructor compiled from a Schema definition. An instance of a model represents a MongoDB document. `const MyModel = mongoose.model('ModelNameInDB', mySchema);`
  - CRUD Operations:
    - Create: `MyModel.create({ fieldName: 'value', age: 25 })` or `new MyModel({ ... }).save()`
    - Read: `MyModel.find({ age: { $gt: 20 } })`, `MyModel.findOne({ _id: 'someId' })`, `MyModel.findById('someId')`
    - Update: `MyModel.updateOne({ _id: 'id' }, { $set: { age: 26 } })`, `MyModel.findByIdAndUpdate('id', { age: 27 }, { new: true })`
    - Delete: `MyModel.deleteOne({ _id: 'id' })`, `MyModel.findByIdAndDelete('id')`
- **PostgreSQL with `pg` driver:**
  - Install: `npm install pg`
  - Connect using a `Pool` (recommended for managing multiple client connections) or a single `Client`.

```
const { Pool } = require('pg');
const pool = new Pool({ /* connection config */ });
```

- Execute SQL queries: `pool.query('SELECT * FROM users WHERE id = $1', [userId])`.

### VII. ADVANCED NODE.JS TOPICS / FAQ

- **First-Class Functions:** In JavaScript (and thus Node.js), functions are treated like any other variable. They can be assigned to variables, passed as arguments to other functions, and returned from other functions.
- **Callback Hell & Avoidance:** [PYQ Q1(d)] (See detailed explanation under Node.js Core Concepts: Asynchronous Operations). Avoid with Promises, Async/Await, named functions, and modularization.
- **Promises vs. Callbacks:** Promises provide better readability for complex asynchronous sequences (chaining with `.then()`), more robust error handling (`.catch()`), and easier composition compared to deeply nested callbacks.
- **`process.nextTick()` vs. `setImmediate()`:**
  - `process.nextTick(callback)`: Schedules the `callback` to run immediately after the current operation completes, *before* the event loop proceeds to the next phase. It has very high priority. Use sparingly, as recursive `nextTick` calls can starve the I/O poll.
  - `setImmediate(callback)`: Schedules the `callback` to run in the "check" phase of the event loop, which occurs after the "poll" (I/O) phase. Generally safer for deferring execution after I/O callbacks have run.
- **Node.js Exit Codes:** Indicate the process termination status.
  - `0`: Successful termination.
  - `1` (or other non-zero): Uncaught fatal exception or error.
- **Stubs in Testing:** Test doubles that replace real dependencies with controlled, simulated objects or functions. They return pre-defined values, allowing tests to focus on the unit under test in isolation.
- **Reactor Pattern:** An architectural pattern for handling concurrent service requests delivered to a service handler from one or more inputs. The Node.js event loop is an implementation of this pattern.
- **`perf_hooks` Module:** Provides an API for measuring the performance of asynchronous operations and other code segments (`performance.mark()`, `performance.measure()`).
- **WASI (WebAssembly System Interface):** An API specification that allows WebAssembly modules to interact with system resources (like file system, network) outside of a web browser environment, potentially in Node.js.
- **Package Management (`package.json` & `package-lock.json`):**
  - `package.json`: Manifest file for a Node.js project. Records project metadata (name, version), `dependencies` (required for production), `devDependencies` (required for development/testing), `scripts` (e.g., `start`, `test`).

- `package-lock.json` (or `npm-shrinkwrap.json` or `yarn.lock`): Records the exact versions of all installed dependencies and their sub-dependencies, ensuring reproducible builds across different environments.
- Common npm commands: `npm install <pkg>`, `npm install -D <pkg>`, `npm uninstall <pkg>`, `npm update`, `npm list`, `npm run <script-name>`.

---

**UNIT 4: MONGODB**

**1. INTRODUCTION TO MONGODB**

- **What is MongoDB? Definition & Features:** [PYQ Q8(a) Write the features of MongoDB. How does MongoDB differ from traditional relational databases?]
  - *Theory:* MongoDB is an open-source, NoSQL (non-relational) database program. It is document-oriented, meaning it stores data in flexible, JSON-like documents called BSON (Binary JSON). This allows for dynamic schemas, where documents in the same collection do not need to have the same set of fields or structure. MongoDB is written in C++ and is designed for scalability and high performance.
  - **Core Concepts:**
    - **Document:** A record in MongoDB, equivalent to a row in RDBMS (but much more flexible).
    - **Collection:** A grouping of MongoDB documents, equivalent to a table in RDBMS (but schema-less).
  - **Key Features:** [PYQ Q8(a)]
    - **Dynamic Schema/Schemaless:** Fields can vary from document to document within a collection.
    - **Document-Oriented Storage (BSON):** Data stored in JSON-like documents.
    - **Indexing:** Supports various types of indexes (single field, compound, multikey, geospatial, text, hashed) to improve query performance. `_id` field is automatically indexed.
    - **Aggregation Framework:** A powerful pipeline for data processing, transformation, and analysis (e.g., grouping, filtering, calculating sums/averages).
    - **Replication (High Availability):** [PYQ Q9(a)] Uses **Replica Sets** (a group of MongoDB servers maintaining the same data set) to provide redundancy and automated failover. If a primary node fails, an eligible secondary can be elected as the new primary.
    - **Sharding (Horizontal Scaling):** [PYQ Q9(a)] Distributes data across multiple servers or clusters (shards) to handle large datasets and high throughput. (More details below).
    - **GridFS:** A specification for storing and retrieving large files (e.g., images, videos, audio files) that exceed BSON document size limits (16MB).
    - **Ad Hoc Queries:** Supports rich query language for filtering and sorting.
    - **TTL (Time-To-Live) Collections:** Automatically remove documents from a collection after a certain period or at a specific clock time.

- **SQL (Relational) VS NoSQL (Non-Relational, e.g., MongoDB): Key Differences** [PYQ Q8(a)]

| Feature | SQL (Relational Databases - RDBMS) | NoSQL (e.g., MongoDB) |
|---|---|---|
| **Data Model** | Tables with rows & columns | Documents (MongoDB), Key-Value, Graph, Column-family |
| **Schema** | Predefined, fixed schema (structured) | Dynamic, flexible schema (MongoDB is schema-less by default) |
| **Primary Key** | Typically user-defined primary key | `_id` field, automatically generated if not provided |
| **Relationships** | JOIN operations across tables | Embedding documents, linking via references, `$lookup` (aggregation) |
| **Query Language** | SQL (Structured Query Language) | Varies (MongoDB uses MQL - MongoDB Query Language, JSON-like queries) |
| **Scalability** | Primarily Vertical (scale-up: more CPU/RAM/Disk on one server) | Primarily Horizontal (scale-out: distribute data across multiple servers, e.g., Sharding) |
| **Transactions** | ACID-compliant (Atomicity, Consistency, Isolation, Durability) - strong consistency | Typically BASE (Basically Available, Soft state, Eventual consistency) - often tunable. MongoDB offers ACID for multi-document transactions since v4.0. |
| **Data Integrity** | Enforced by schema, constraints, FKs | Often managed at application level; schema validation can be added |
| **Best For** | Complex transactions, structured data, well-defined relationships | Big Data, unstructured/semi-structured data, rapid development, high scalability needs, real-time apps |

**2. CRUD OPERATIONS IN MONGODB (USING `mongosh` SHELL)** [PYQ Q8(b) Perform with code (any two): Add data in MongoDB? Delete a Document? Update a Document?]

- Connect to MongoDB using `mongosh` shell. Switch to your database: `use myDatabaseName`
- **Create (Add/Insert Data):** [PYQ Q8(b)(i) Add data]
  - `db.collectionName.insertOne({ field1: "value1", field2: 123 })`
    - Inserts a single document into `collectionName`.
  - `db.collectionName.insertMany([{ name: "Alice", age: 30 }, { name: "Bob", city: "New York" }])`
    - Inserts an array of documents.
- **Read (Query Data):**
  - `db.collectionName.find({ query_filter })`
    - Retrieves documents matching the `query_filter`.
    - Example: `db.users.find({ age: { $gt: 25 } })` (find users older than 25).

    - Example: `db.users.find()` (find all documents in `users` collection).
  - `db.collectionName.findOne({ query_filter })`
    - Retrieves the first document matching the `query_filter`.
- **Update Data:** [PYQ Q8(b)(iii) Update a Document]
  - `db.collectionName.updateOne({ filter_criteria }, { $set: { fieldToUpdate: newValue, anotherField: "updated" } })`
    - Updates the first document matching `filter_criteria` using update operators like `$set`, `$inc`, etc.
    - Example: `db.users.updateOne({ name: "Alice" }, { $set: { age: 31 } })`
  - `db.collectionName.updateMany({ filter_criteria }, { $set: { status: "active" } })`
    - Updates all documents matching `filter_criteria`.
  - `db.collectionName.replaceOne({ filter_criteria }, { new_document_structure_except_id })`
    - Replaces the entire document (except its `_id`) with the new document.
- **Delete Data:** [PYQ Q8(b)(ii) Delete a Document]
  - `db.collectionName.deleteOne({ filter_criteria })`
    - Deletes the first document matching `filter_criteria`.
    - Example: `db.users.deleteOne({ name: "Bob" })`
  - `db.collectionName.deleteMany({ filter_criteria })`
    - Deletes all documents matching `filter_criteria`.
    - Example: `db.products.deleteMany({ status: "inactive" })`

### 3. MONGODB CONCEPTS (RELEVANT TO PYQS)

- **Document:** [PYQ Q9(b)(i) Document in MongoDB]
  - A record in a MongoDB collection, stored in BSON (Binary JSON) format.
  - A set of key-value pairs, where keys are strings and values can be various data types (strings, numbers, booleans, arrays, embedded documents, dates, etc.).
  - Documents are schema-less, meaning documents in the same collection can have different fields.
  - Each document has a unique `_id` field which acts as its primary key (automatically generated if not provided).
  - *Structure Example:*

```
{
  "_id": ObjectId("60c72b2f9b1d8e1f8c8b4567"),
```

```
  "name": "Alice Wonderland",
  "age": 30,
  "email": "alice@example.com",
  "address": {
    "street": "123 Main St",
    "city": "New York"
  },
  "hobbies": ["reading", "coding"]
}
```

- **Collection:** [PYQ Q9(b)(ii) Collection in MongoDB]
  - A grouping of MongoDB documents.
  - Roughly equivalent to a table in a relational database but does not enforce a schema (documents within a collection can have different structures).
  - Collections are created implicitly when the first document is inserted into them, or explicitly using `db.createCollection("collectionName")`.
- **Database:** [PYQ Q9(b)(iii) Databases in MongoDB]
  - A physical container for collections.
  - Each database gets its own set of files on the file system.
  - A single MongoDB server can host multiple databases, each logically separate.
  - To use a database, you switch to it using the `use <databaseName>` command in `mongosh`. The database is created implicitly when you first store data in it.
- **Sharding & High Availability:** [PYQ Q9(a) Describe the process of sharding in MongoDB. How does MongoDB ensure high availability?]
  - **Sharding (Process):** [PYQ Q9(a)]
    - *Theory:* Sharding is MongoDB's method for horizontal scaling. It involves distributing data across multiple machines or clusters called "shards." Each shard contains a subset of the collection's data. This allows MongoDB to handle very large datasets and high throughput applications that would exceed the capacity of a single server.
    - *Components of a Sharded Cluster:*
      1. **Shards:** Store the actual data. Each shard can be a replica set for high availability.
      2. **Mongos (Query Routers):** Instances that route client requests to the appropriate shard(s). Clients connect to `mongos`, not directly to shards.
      3. **Config Servers:** Store metadata about the sharded cluster, including the mapping of data to shards. They must be deployed as a replica set for production.
    - *Process:*
      1. **Choose a Shard Key:** Select a field (or fields) from the documents in a collection to be sharded. The choice of shard key is crucial for even data distribution and query performance.

2. **Enable Sharding:** Enable sharding for a database and then for specific collections using the chosen shard key.
3. **Data Partitioning:** MongoDB partitions data into "chunks" based on ranges of the shard key values.
4. **Chunk Distribution:** These chunks are distributed across the available shards. A balancer process automatically migrates chunks between shards to ensure even distribution.

- **High Availability (via Replica Sets):** [PYQ Q9(a)]
  - MongoDB ensures high availability primarily through **Replica Sets**.
  - A replica set is a group of `mongod` instances (servers) that maintain the same data set.
  - One node in the replica set is designated as the **primary** node, which receives all write operations.
  - Other nodes are **secondary** nodes, which replicate the data from the primary. Secondaries can also serve read requests (if configured).
  - **Automatic Failover:** If the primary node becomes unavailable, the replica set automatically elects one of the eligible secondary nodes to become the new primary, minimizing downtime.
  - In a sharded cluster, *each shard should be deployed as a replica set* to provide high availability for the data on that shard. The config servers also must be deployed as a replica set.

**4. MANAGING MONGODB**

- **Installation:** Download the MongoDB Community Server from the official MongoDB website. Follow platform-specific installation instructions (MSI for Windows, Homebrew for macOS, package managers for Linux).
- **Starting the Server:** Typically `mongod --dbpath /path/to/your/data/directory`. (Configuration can also be done via a config file).
- **Connecting with `mongosh`:** Open a terminal and run `mongosh`. This connects to a MongoDB instance running on `localhost:27017` by default.
- **Create/Switch Database:** `use myNewDatabase` (The database is implicitly created when you first insert data into a collection within it).
- **Create Collection:** Implicitly on first insert, or explicitly: `db.createCollection("myCollection")`.
- **Drop Collection:** `db.myCollection.drop()`.
- **Drop Database:** First, switch to the database: `use databaseToDrop;` Then: `db.dropDatabase();`
- **Connecting MongoDB to Node.js:**
  - Use the official `mongodb` Node.js driver: `npm install mongodb`

```
const { MongoClient } = require("mongodb");
// Replace with your MongoDB deployment's connection string.
const uri = "mongodb://localhost:27017/myDatabaseName";
const client = new MongoClient(uri);

async function run() {
  try {
    await client.connect();
    console.log("Connected successfully to MongoDB server");
    const database = client.db("myDatabaseName");
    const collection = database.collection("myCollectionName");
    // Perform operations:
    const doc = await collection.findOne({ name: "Test" });
    console.log(doc);
  } finally {
    await client.close();
  }
}
run().catch(console.dir);
```

- Or use an ODM like Mongoose (see Unit 3, Node.js Database Integration).

**5. DATA MIGRATION INTO MONGODB**

- **From JSON/CSV:** Use the `mongoimport` command-line tool.
  - JSON: `mongoimport --uri="mongodb://localhost:27017/mydb" --collection=mycoll --file=data.json --jsonArray` (if file is a JSON array)
  - CSV: `mongoimport --uri="mongodb://localhost:27017/mydb" --collection=mycoll --type=csv --headerline --file=data.csv`
- **From SQL Databases (e.g., MySQL, PostgreSQL):**
  1. **Export Data:** Export data from the SQL database into a common format like CSV or JSON.
  2. **Transform Data (Optional but often needed):** Adjust the data structure to fit MongoDB's document model (e.g., denormalize by embedding related data, decide on referencing). This might involve writing custom scripts.
  3. **Import Data:** Use `mongoimport` or write a custom script (e.g., using Node.js with the MongoDB driver) to read the exported data and insert it into MongoDB.
- **From Firebase (Firestore):**
  1. Use the Firebase Admin SDK (e.g., in a Node.js script) to export data from Firestore collections.
  2. Write a script to transform and import this data into MongoDB using the MongoDB Node.js driver or Mongoose.
- **From Excel:**
  1. Convert the Excel file to CSV or JSON format first.

2. Alternatively, use a Node.js library like `xlsx` to read data directly from the Excel file and then import it into MongoDB using a custom script.

**6. MONGODB SERVICES (KEY ONES)**

- **MongoDB Atlas:** A fully managed, global cloud database service (DBaaS) available on AWS, Google Cloud, and Azure. Handles infrastructure, provisioning, scaling, backups, monitoring, and security.
- **MongoDB Enterprise Advanced:** A commercial, self-managed version of MongoDB offering advanced features, including enterprise-grade security, auditing, Kubernetes integration, management tools like Ops Manager/Cloud Manager, and official support.
- **MongoDB Community Edition:** The free, open-source, self-managed version of MongoDB.
- **MongoDB Realm (formerly Stitch):** A mobile database (Realm Database) with automatic synchronization to Atlas, and a serverless backend application development platform (functions, triggers, GraphQL).
- **MongoDB Charts:** A data visualization tool for creating charts and dashboards directly from data in MongoDB Atlas.
- **MongoDB Compass:** A GUI tool for interacting with MongoDB (Community, Enterprise, Atlas). Allows schema visualization, query building, index management, aggregation pipeline construction, etc.
- **MongoDB Atlas Search:** A managed full-text search engine built on Apache Lucene, integrated with MongoDB Atlas.
- **MongoDB Atlas Data Lake:** Allows querying data stored in cloud object storage (like Amazon S3, Azure Blob Storage) using the MongoDB Query Language (MQL) as if it were in MongoDB collections.

---