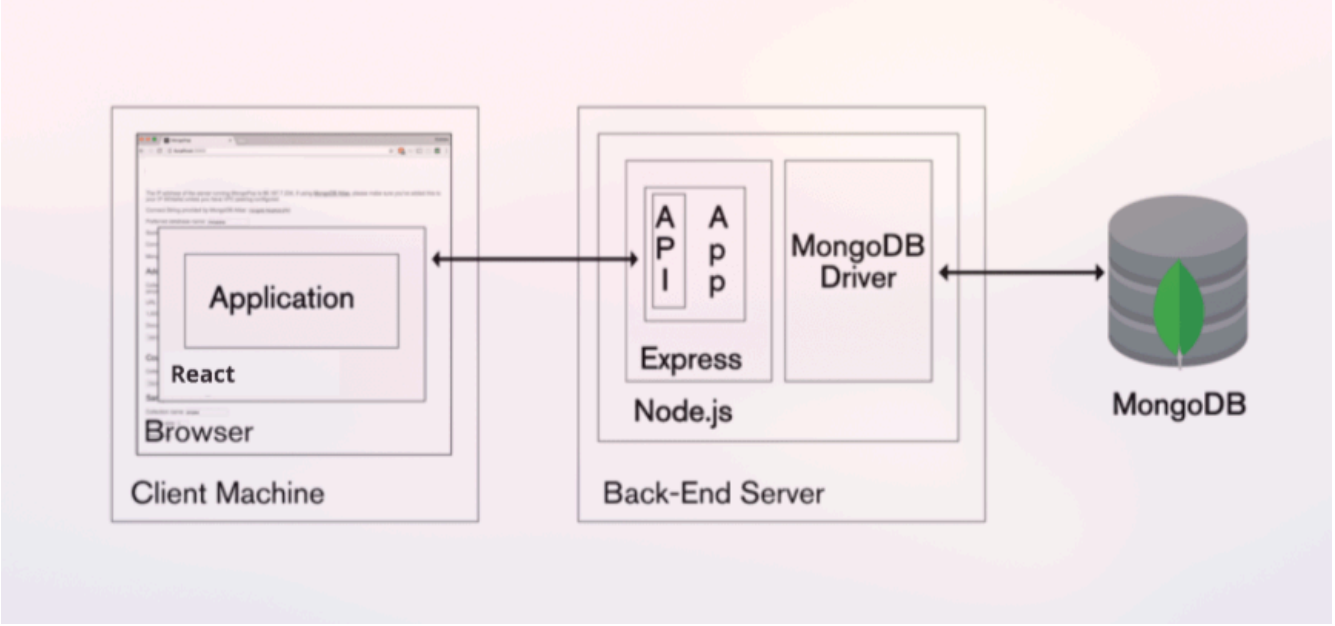
MERN ONESHOT

UNIT 1: WEB DEVELOPMENT FUNDAMENTALS

1. INTRODUCTION

- **Fundamentals of Good Website Design:**
  - **Definition:** Fulfills intended function, conveys message, engages visitor (aesthetics + functionality).
  - **Core Purpose:** Describing Expertise, Building Reputation, Generating Leads, Sales & After Care.
  - **Key Factors:** Consistency, Colours (brand-fit, <5, emotional impact), Typography (legible, max 3 fonts, brand voice), Imagery (expressive, high-quality), Simplicity, Functionality (ease of use), User Experience (UX) (usability, trust).
  - **Additional Principles:** Navigation (simple, intuitive, consistent) [PYQ], F-Shaped Pattern Reading, Visual Hierarchy (guides to important info), Content (compelling), Grid-Based Layout (organized, clean), Load Time (fast, optimize images), Mobile Friendly (responsive).
- **Web Page:** Document in browser (HTML), unique URL, embeds styles (CSS), scripts (JS), media.
- **Website:** Collection of linked web pages under a domain name; accessed via domain, displays homepage.
- **Web Application:** Program on remote server, accessed via browser; requires web server, app server, database. Benefits: no install, multi-user, cross-platform.
  - **Native App:** Platform-specific, installed, offline use, device hardware access.
  - **Hybrid App:** Installs like native, built with web tech, device API access, usually online.
- **Client-Server Architecture:** [PYQ] Server provides services, client requests.
  - **Components:** Workstations (clients), Servers (central repositories), Networking Devices.
  - **How it Works:** User URL -> DNS -> IP -> Browser HTTP request -> Server sends files -> Browser displays.
  - **Tiers:**
    - **1-Tier:** All layers on one device.
    - **2-Tier:** Client (UI) & Server (DB).
    - **3-Tier:** Client (Presentation) -> Middleware (Application Logic) -> Server (Database). More secure. [PYQ for MERN]
    - **N-Tier:** Scaled, isolated function layers.
  - **vs. Peer-to-Peer:** Client-Server (specific roles, centralized data) vs. P2P (equal peers, distributed data).

- **MERN Stack Introduction:** [PYQ] JavaScript stack for full-stack apps.



  - **M**ongoDB (NoSQL DB), **E**xpress.js (Node.js framework), **R**eact.js (Frontend UI library), **N**ode.js (JS runtime environment).
  - **3-Tier Architecture:** [PYQ]
    1. **Front-end (React.js):** UI/UX, client-side rendering.
    2. **Middle-Tier/Server (Node.js, Express.js):** Application logic, API, request handling.
    3. **Backend/Database (MongoDB):** Data storage and management.
  - **Benefits:** Cost-effective (open-source), SEO friendly, good performance, security, fast delivery, agile, JS for full stack.

2. MARKUP LANGUAGES

- **HTML (HyperText Markup Language):** Standard language to structure web pages.
  - Elements (start tag, content, end tag), Tags (`<tag>`), Attributes (`name="value"`).
  - Basic Structure: `<!DOCTYPE html> <html> <head> <title>...</title> </head> <body> ... </body> </html>`.
  - Not case-sensitive. `<html>` is parent of `<head>`, `<body>`.
- **XHTML (Extensible HTML):** Stricter, XML-based HTML. Case-sensitive.
  - Rules: `<!DOCTYPE>` mandatory, `xmlns` in `<html>`, proper nesting, all elements closed, lowercase attributes, quoted values.
- **HTML Lists:** Group related info. Types: `<ul>` (unordered, bullets), `<ol>` (ordered, numbers), `<dl>` (description list: `<dt>` term, `<dd>` definition). `<li>` for list items. Nesting possible.
- **HTML Tables:** Arrange data in rows (`<tr>`) and columns (`<td>` data cell, `<th>` header cell) within `<table>`. Attributes: `border`, `colspan`, `rowspan`. Elements: `<caption>`, `<thead>`, `<tbody>`, `<tfoot>`.
- **HTML Forms:** Collect user data via `<form>`. Attributes: `action`, `method` (GET/POST).

- **Controls:** `<input type="text/password/checkbox/radio/file/submit/reset/button/image/hidden">`, `<textarea>`, `<select>`, `<option>`, `<button>`.

- **XML (Extensible Markup Language):** Carries data, user-defined tags, hierarchical, strict syntax. Used for data exchange.

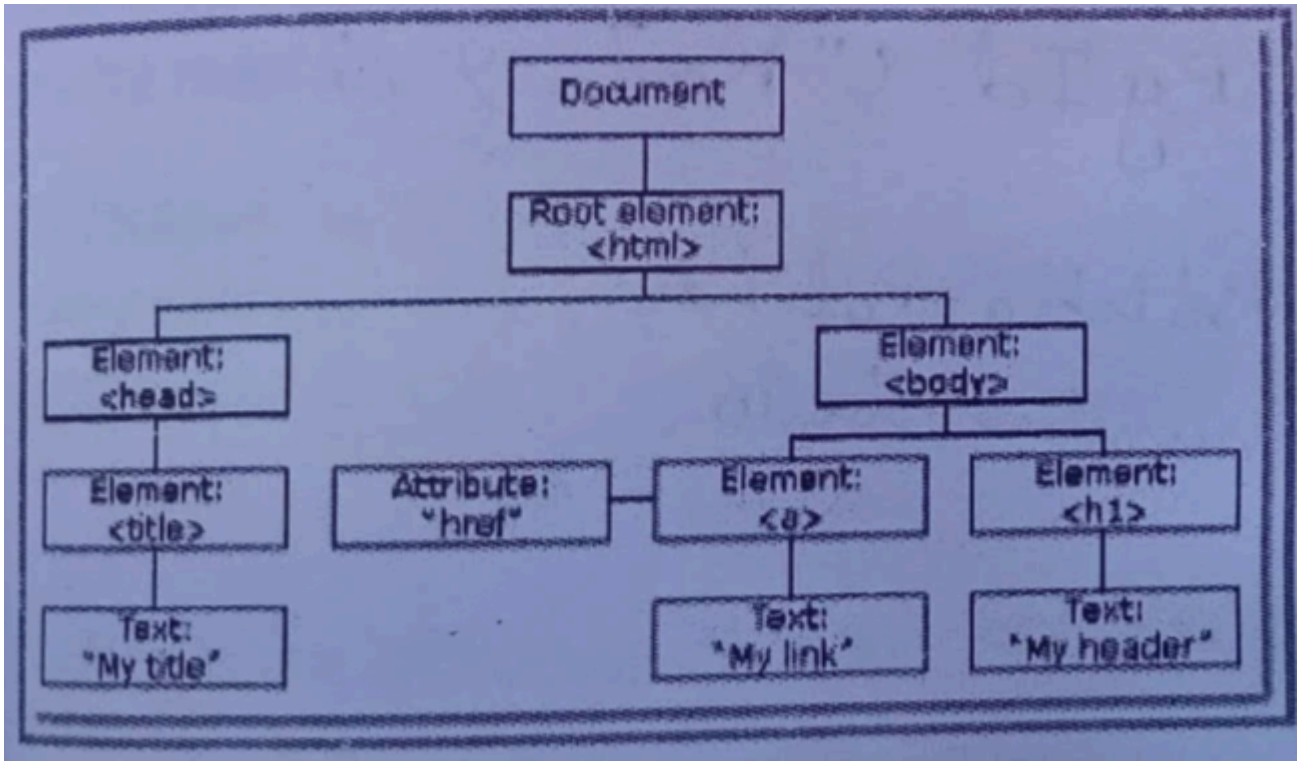
3. CSS STYLE SHEETS

- **CSS (Cascading Style Sheets):** Describes presentation (look, formatting) of HTML.
- **Core Syntax:** `selector { property: value; }`.
- **Types:**
  1. **Inline:** `style` attribute in HTML element.
  2. **Internal/Embedded:** `<style>` tag in HTML `<head>`.
  3. **External:** Linked `.css` file via `<link>` tag in `<head>`.
- **Text Properties:** `color`, `background-color`, `text-align`, `text-decoration`, `text-transform`, `text-indent`, `letter-spacing`, `word-spacing`, `line-height`, `text-shadow`.
- **CSS Box Model:** [PYQ] Element as a box: Content -> Padding -> Border -> Margin.
- **Normal Flow:** Default layout. Block-level (new line, full width) vs. Inline elements (flow with content, width as needed). `display` property modifies.
- **Styling Lists:** `list-style-type`, `list-style-image`, `list-style-position`.
- **Styling Tables:** `border`, `border-collapse`, `padding`, `text-align`, `background-color`.
- **XSLT (Extensible Stylesheet Language Transformations):** Transforms XML docs (e.g., to HTML) using XPath for selection.

4. CLIENT-SIDE PROGRAMMING: JAVASCRIPT

- **Introduction to JS:** High-level, interpreted scripting language for dynamic web pages. Not Java.
- **Basic Syntax & Embedding:** `<script>` tags in HTML (`<head>` or `<body>`), or external `.js` file. Semicolons separate statements (mostly optional if on new lines). Comments: `//` or `/* ... */`. Case-sensitive.
- **Variables & Data Types:**
  - `var` (function-scoped), `let` (block-scoped), `const` (block-scoped constant).
  - Primitives: String, Number, BigInt, Boolean, Undefined, Null, Symbol.
  - Object type (Reference): Object, Array, Function. Dynamically typed.
- **Literals:** Fixed values (e.g., `100`, `"Hello"`, `true`, `null`, `{}`, `[]`).
- **Operators:** Arithmetic (`+,-,*,/,%,++,--`), Assignment (`=,+=,-=`), Comparison (`==,===,!=,!==,>,<,>=,<=`), Logical (`&&,||,!`), String (`+`), Ternary (`condition ? trueVal : falseVal`).

- **Functions:** Reusable code blocks. `function name(params){...; return val;}`. Call: `name(args);`.
  - Scope: Variables inside usually local. Arrow functions (`=>`) have concise syntax, lexical `this`.
  - **Closures:** [PYQ] Inner function has access to outer function's scope, even after outer has executed. Used for private vars, state in async.
- **Objects:** Collection of key-value pairs (properties, methods). Create via literals `{}`, `new Object()`, constructor functions, ES6 classes. Access: `obj.prop` or `obj["prop"]`. `this` refers to the object in methods.
- **Arrays:** Ordered collection of values, 0-indexed. Create: `[]` or `new Array()`. Access: `arr[index]`. `length` property. Methods: `push, pop, map, forEach`, etc. Nested arrays possible.
- **Built-in Objects:** `Math`, `Date`, `String`, `JSON`.
- **JS Form Programming:** Interact with HTML forms for validation, dynamic updates. Access elements via DOM methods (e.g., `document.getElementById`). Get/set values: `element.value`, `element.checked`.
- **JS Events:**
  - *Theory:* Actions (click, load, keypress) JS can respond to.
- **Intrinsic Event Handling:** [PYQ] HTML attributes (e.g., `onclick`, `onload`, `onmouseover`) that execute JS on events. Modern: `addEventListener`.
- **Modifying Element Style:** `document.getElementById("id").style.property = "value";`.
- **Document Trees (DOM):** [PYQ] API for HTML/XML, represents page as a tree of nodes (elements, text, etc.). Allows JS to change structure, style, content.



  - Relationships: Parent, Child, Sibling. Node Types: Element, Text, Comment, etc.

- DOM Levels: Define standard interfaces
  - Level 0: Early, basic event handling.
  - Level 1: Core (generic XML/HTML) & HTML-specific models.
  - Level 2: Events, CSS manipulation, Traversal.
  - Level 3+: XPath, Saving/Loading, (Living Standard - WHATWG).
- **Accessing HTML Elements:** [PYQ] * `document.getElementById('id')`
  - `document.getElementsByTagName('tag')`
  - `document.getElementsByClassName('class')`
  - `document.querySelector('cssSelector')` (first match)
  - `document.querySelectorAll('cssSelector')` (all matches)
  - **`this` keyword:** [PYQ Q2(a)]

- **ECMAScript (ES):** Specification JavaScript implements.
  - **ES5 (2009):** Strict mode, JSON, new Array/Object methods.
  - **ES6 (2015)/ES2015:** Major update: `let/const`, arrow functions, template literals, default params, rest/spread, destructuring, classes, modules, Promises.
  - **ES5 vs ES6 Comparison Table:**

Feature	ES5	ES6
Variable Declaration	`var` (function-scoped)	`let`, `const` (block-scoped)
Function Syntax	`function name() {}`	Arrow functions (`() => {}`), lexical `this`
Destructuring	Manual assignment	`const {a,b}=obj; const [x,y]=arr;`
OOP	Constructor fns, prototypes	`class`, `extends`, `super` (syntactic sugar)
Modules	CommonJS/AMD (libraries)	Native `import`/`export`
Async Operations	Callbacks (callback hell)	`Promise` objects

UNIT 2: REACTJS

1. INTRODUCTION TO REACTJS

- **Definition:** JavaScript library for building UIs (User Interfaces) or UI components, created by Facebook. Fast, scalable, simple. View in MVC.
- **Why React?** Dynamic apps (less code), improved performance (Virtual DOM), reusable components, unidirectional data flow (easier debugging), dedicated debugging tools.

- **Key Features:** [PYQ Q1(c) Describe the building blocks of React?]
  - **JSX (JavaScript XML):** HTML-like syntax in JS to describe UI. Transpiled by Babel.
  - **Components:** [PYQ] Building blocks of UI, independent, reusable.
  - **Virtual DOM:** [PYQ Q1(b) Differentiate Shadow DOM and Virtual DOM] Lightweight copy of real DOM in memory. Efficiently updates real DOM by changing only necessary parts.
  - **One-way data-binding (Unidirectional Data Flow):** Data flows parent to child (via props). Predictable.
  - **High performance:** Updates only changed components.

2. GETTING STARTED WITH REACT APP

- **Prerequisites:** NodeJS and npm (Node Package Manager).
- **Create React App:** `npx create-react-app my-app` (or Vite: `npm create vite@latest my-app -- --template react`).
- `cd my-app`, `npm start` (or `npm run dev` for Vite).
- **Project Structure:** `node_modules`, `public` (index.html), `src` (App.js, index.js, components), `package.json`.

3. TEMPLATING USING JSX

- **Definition:** Syntax extension for JS, looks like HTML. Describes UI. Transpiled by Babel.
- **Embedding Expressions:** Use curly braces `{}` for JS expressions (variables, math, function calls).
- **Attributes:** camelCase (e.g., `className`, `htmlFor`). Values: string quotes or `{JS expression}`.
- Single root element required per return (use `<div>` or `<React.Fragment>` / `<>...</>`).

4. CLASSES USING JSX (CLASS COMPONENTS) [PYQ Q4(a) Differences between class and functional components]

- ES6 classes extending `React.Component`.
- Must have `render()` method returning JSX.
- Can have state (`this.state`) and lifecycle methods.
- ES5 used `React.createClass` (deprecated).

5. COMPONENTS [PYQ Q4(a) Components in React JS]

- **Definition:** Reusable, independent UI pieces. Like JS functions, accept props, return React elements.
- Start with a capital letter.
- **Types:**
  1. **Functional Components:** JS functions (props) => JSX. Simpler, often stateless (but Hooks add state/lifecycle).

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

2. **Class Components:** `extends React.Component`. Have `render()`, can have state and lifecycle methods.

```
class Welcome extends React.Component { render() { return <h1>Hello, {this.props.name}</h1>; } }
```

- **Stateful vs. Stateless Components (More Points for PYQ Q4(a)):**
  - **Stateless (Often Functional pre-Hooks, or presentational):**
    - Do not manage internal state.
    - Receive data via props.
    - Primarily responsible for UI presentation.
    - Deterministic: same props yield same output.
    - Often simpler to understand, test, and reuse.
  - **Stateful (Often Class, or Functional with Hooks):**
    - Manage internal state (`this.state` or `useState`).
    - State can change over time, causing re-renders.
    - Often responsible for logic and data manipulation.
    - Can have lifecycle methods (or `useEffect` for side effects).
- **Differences: Class vs. Functional:** [PYQ Q4(a)]

Feature	Class Components	Functional Components (with Hooks)
State	`this.state`, `this.setState()`	`useState()` Hook
Lifecycle	Lifecycle methods	`useEffect()` Hook (for side effects)
`this`	Yes	No (props as args)
Syntax	ES6 Class	JS Function

- **Comments in JSX:** `{/* comment */}` or `{ // comment }` within JSX.
- **Embedding Components:** `<MyComponent />` within another component's JSX.

6. STATE AND PROPS [PYQ Q5(a) Explain State and Props. Example to update state.]

- **Props (Properties):**
  - Pass data parent to child. Read-only for child.
  - `<ChildComponent propName="value" data={object} />`. Child accesses via `props.propName`.
- **State:**
  - Data private to a component, can change over time, causing re-renders.

- Class Components: Initialize in `constructor(props){ super(props); this.state = { key: val }; }`. Access `this.state.key`.
- **Update State:** Use `this.setState({ key: newVal })`. Never modify `this.state` directly. `setState` is async.

```
// Example for PYQ Q5(a) - updating state
class Counter extends React.Component {
  constructor(props) { super(props); this.state = { count: 0 }; }
  increment = () => { this.setState({ count: this.state.count + 1 });
};
  render() { return (<div><p>{this.state.count}</p><button onClick=
{this.increment}>Inc</button></div>); }
}
```

- Functional Components use `useState` Hook for state.

- **State vs. Props Table:**

Feature	Props	State
Data Flow	Parent to Child (Unidirectional)	Internal to component
Mutability	Immutable (Read-only by child)	Mutable (by component via `setState`/setter)
Ownership	Owned by parent, passed down	Owned & managed by the component itself
Purpose	Configure/customize child components	Manage dynamic data internal to component
Initialization	Passed by parent during rendering	Initialized within component (constructor/`useState`)
Access	`props.propName` / `this.props.propName`	`state.key` / `this.state.key`

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

- *Theory:* Functions allowing functional components to use state and lifecycle features.
- **`useState`:** [PYQ Q4(b)] Adds state to functional components.
  - *Syntax:* `const [state, setState] = useState(initialValue);`
  - *Example:*

```
function Counter() {
  const [count, setCount] = useState(0);
  return (
    <> {/* Shorthand for React.Fragment */}
```

```
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Inc</button>
    </>
  );
}
```

- **`useEffect`**: [PYQ Q4(b)] Handles side effects (data fetching, DOM changes, subscriptions).
  - *Syntax:* `useEffect(() => { /* effect */; return () => {/* cleanup */}; }, [deps]);` (deps = dependencies array)
  - *Example (doc title):*

```
function TitleUpdater() {
  const [name, setName] = useState('Guest');
  useEffect(() => {
    document.title = `Hello ${name}`;
  }, [name]); // Runs when 'name' changes

  return <input value={name} onChange={e => setName(e.target.value)}
placeholder="Name"/>;
}
```

7. LIFECYCLE OF COMPONENTS

- Phases: Mounting, Updating, Unmounting. (Mainly for Class Components; `useEffect` for Functional).
- **Mounting (Birth):**
  - `constructor()`: Init state, bind methods.
  - `static getDerivedStateFromProps()`: (Rare) Sync state from props.
  - `render()`: Returns JSX. **Required.**
  - `componentDidMount()`: After component in DOM. API calls, subscriptions.
- **Updating (Growth):** (Triggered by props/state change)
  - `static getDerivedStateFromProps()`
  - `shouldComponentUpdate()`: (Rare) Optimize, return `false` to prevent re-render.
  - `render()`
  - `getSnapshotBeforeUpdate()`: (Rare) Capture DOM info before update.
  - `componentDidUpdate()`: After update. DOM ops, network requests based on prop/state changes.
- **Unmounting (Death):**
  - `componentWillUnmount()`: Before removal from DOM. Cleanup (timers, subscriptions).

8. RENDERING LISTS

- Use JS `map()` method on an array to return an array of JSX elements.
- **Keys:** Special `key` prop required for list items. Must be unique among siblings. Helps React identify changed/added/removed items for efficient updates. Use stable IDs from data if possible; avoid index if list order can change.

```
const numbers = [1, 2, 3];
const listItems = numbers.map((number) => <li key={number.toString()}>
{number}</li>);
// return <ul>{listItems}</ul>;
```

9. PORTALS

- Render children into a DOM node outside parent's DOM hierarchy (e.g., for modals, tooltips).
- `ReactDOM.createPortal(child, domNodeContainer)`.
- Event bubbling works through portals as if normal React children.

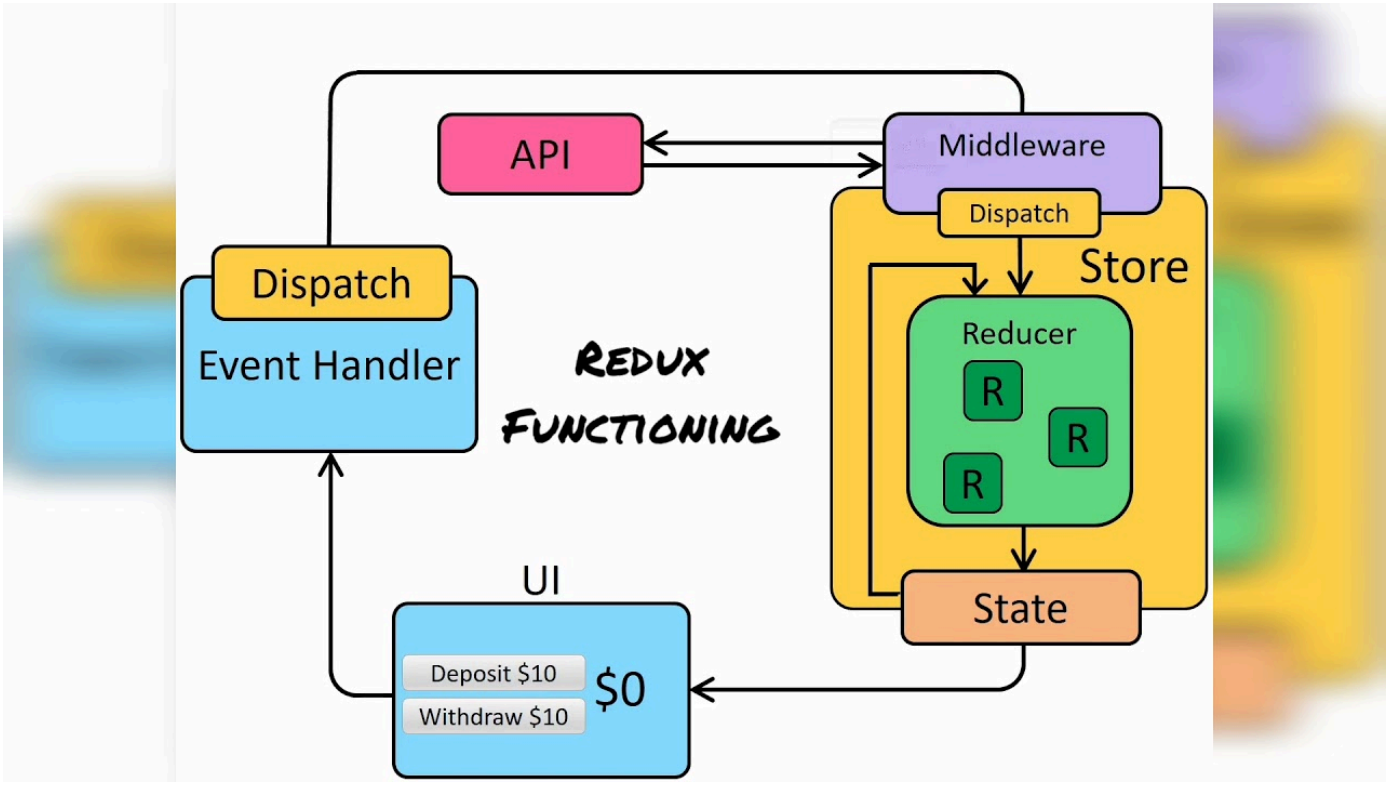
10. ERROR HANDLING (ERROR BOUNDARIES)

- Class components that catch JS errors in their child tree and display fallback UI.
- Define `static getDerivedStateFromError(error)` (to update state for fallback UI) and/or `componentDidCatch(error, errorInfo)` (for logging).
- Don't catch errors in event handlers, async code, SSR, or self.

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

- Library for navigation in SPAs without page refresh.
- Key Components (v6): `<BrowserRouter>`, `<Routes>`, `<Route path="/path" element={<Component />} />`, `<Link to="/path">Name</Link>`.
- Manages views based on URL. `npm install react-router-dom`.

12. REDUX [PYQ Q5(b)(ii) Redux]



- Predictable state container for JS apps, manages global application state.
- **Core Concepts:**
  - **Store:** Single object holding entire app state. `createStore(reducer)`.
  - **Action:** Plain JS object describing "what happened" (e.g., `{ type: 'TYPE', payload: data }`).
  - **Reducer:** Pure function `(prevState, action) => newState`. Specifies how state changes.
  - **Dispatch:** `store.dispatch(action)` to send actions.
  - **Subscribe:** `store.subscribe(listener)` to react to state changes.
- **Redux vs. Flux:** Redux (library, single store, immutable state, reducers) vs. Flux (architecture, multiple stores, mutable state, dispatcher).

13. REDUX SAGA

- Redux middleware for managing side effects (async ops like API calls).
- Uses ES6 Generator functions (`function* () {}`) and Effects (`call`, `put`, `takeEvery`, `takeLatest`, `select`).
- Keeps reducers pure by handling async logic in Sagas.

14. IMMUTABLE.JS

- Library providing persistent immutable data structures (`Map`, `List`).
- "Changing" an object returns a new object; original unchanged.

- Benefits: Performance (fast change detection for React/Redux), easier change tracking, predictability.
- Considerations: Learning curve, interop (`toJS()`, `fromJS()`), bundle size.

15. SERVICE SIDE RENDERING (SSR) WITH REACT

- Render React components on server to HTML, send to browser.
- Benefits: Improved SEO, faster perceived performance (Time To Content).
- Client-side JS then "hydrates" the HTML.
- Frameworks: Next.js, Remix. Or custom Node.js setup with `ReactDOMServer.renderToString()`.

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

- Testing individual components/functions in isolation.
- Tools: Jest (runner, assertions), React Testing Library (RTL) (user-centric testing).
- Test: Rendering, user interactions, conditional logic, props.

17. WEBPACK

- Static module bundler for JS apps. Builds dependency graph, outputs bundles.
- **Loaders:** Process non-JS files (CSS, Babel for ES6+, images).
- **Plugins:** Optimization, asset management (e.g., `HtmlWebpackPlugin`).
- Features: Code splitting, tree shaking, dev server, Hot Module Replacement (HMR).
- `create-react-app` and Vite abstract much of its configuration.

UNIT 3: NODE.JS AND EXPRESS.JS

I. INTRODUCTION TO NODE.JS

- **What is Node.js?** Open-source, cross-platform JavaScript runtime environment. Executes JS code *outside* a web browser, mainly for server-side. Uses Google's V8 JS engine.
- **Key Features:**
  - **Asynchronous & Event-Driven:** Non-blocking I/O model. Efficient for concurrent connections using callbacks/Promises.
  - **Single-Threaded (with Event Loop):** [PYQ Q6(c) Node.js event loop mechanism] Manages concurrency efficiently on a single main thread without multi-threading overhead for user code.
  - **Non-Blocking I/O:** I/O operations don't block the main thread.
  - **npm (Node Package Manager):** World's largest ecosystem of open-source libraries.
  - **Cross-Platform:** Runs on Windows, macOS, Linux.
  - **Uses JavaScript:** Full-stack JS development.

- **Node.js for Backend Development:** [PYQ Q6(a) Significance of Node.js in backend] Efficient for I/O-bound tasks, real-time apps (chats, streaming), APIs. Scalable. Large community.
- **Node.js vs. Traditional Server-Side Technologies:** [PYQ Q6(a) How Node.js differs]
  - **Language:** JS vs. Java/C#/PHP.
  - **Concurrency:** Single-threaded, event-driven, non-blocking I/O vs. multi-threaded/blocking.
  - **Performance:** Often higher for I/O-intensive scenarios.
  - **Paradigm:** Encourages async patterns.

II. SETTING UP NODE.JS

- **Installation:** Download LTS from nodejs.org. npm is included. Verify: `node -v`, `npm -v`.
- **Basic Project:** `mkdir my-app`, `cd my-app`, `npm init -y` (creates `package.json`).
- **Dependencies:** `npm install <package-name>` (e.g., `npm install express`). Listed in `package.json`.
- **Run Script:** `node app.js`.

III. NODE.JS CORE CONCEPTS

- **Modules:** Reusable code blocks.
  - **Types:** Core (built-in: `http`, `fs`, `path`), Local/Custom (created by user, `require('./path')`, `module.exports`), Third-Party (from npm, `require('package-name')`).
  - **ES6 Modules:** Use `"type": "module"` in `package.json`. `import`/`export` syntax.
- **Asynchronous Operations:** [PYQ Q6(c) handling asynchronous operations]
  - **Callbacks:** Function passed as argument, called on completion. Can lead to "Callback Hell". [PYQ Q1(d) "Callback" and "Callback Hell"]
  - **Promises:** Represent eventual completion/failure. Chain `.then()`, `.catch()`.
  - **Async/Await:** Syntactic sugar over Promises. `async function() { await promiseCall(); }`. Makes async code look synchronous.
- **Event Loop:** [PYQ Q6(c) Node.js event loop mechanism] Core of Node.js non-blocking async behavior. Single-threaded. Constantly checks event queues (timers, I/O) and processes callbacks.
  - **Phases (Simplified):** Timers -> Pending I/O -> Idle/Prepare -> Poll (New I/O) -> Check (`setImmediate`) -> Close Callbacks. `process.nextTick()` runs between phases.
- **Events & `EventEmitter`:** Many Node.js objects emit events. `events` module provides `EventEmitter` class for event-driven architecture. Methods: `on()`, `emit()`, `once()`, `off()`.
- **File System (`fs` module):** Interact with file system (read, write, etc.). Sync and Async (Promise-based preferred: `require('fs').promises`).
- **Streams:** [PYQ Q7(b)(i) Streams in Node JS] Efficiently handle sequential data (files, network) without loading all into memory. Types: Readable, Writable, Duplex, Transform. Use `pipe()`.

- **Buffers:** Handle binary data directly. Fixed-size memory chunk outside V8 heap.
- **Command Line Interaction:** Run scripts (`node script.js`), REPL (`node`), `process.argv` (args), `readline` (input), `child_process` (shell commands).
- **`console` Module:** Debugging/logging (`log`, `error`, `warn`, `table`, `time`/`timeEnd`, `assert`).
- **Concurrency Handling (Single-Threaded):** Achieved via Event Loop & Non-blocking I/O. I/O delegated to OS/libuv thread pool. Callbacks queued.
- **Worker Threads:** For CPU-intensive tasks. Run JS in parallel threads, communicate via message passing.
- **Clustering:** Multiple Node.js processes (forks) sharing same server port. Utilizes multi-core CPUs for network apps. Master process manages workers.
- **Forking (`child_process.fork`):** Spawns new Node.js instances with IPC channel.
- **Timing Features:** `setTimeout`, `setInterval`, `setImmediate`, `process.nextTick()`. `perf_hooks` for performance measurement.

IV. INTRODUCTION TO EXPRESS.JS

- **What is Express.js?** Minimal, flexible Node.js web application framework. Simplifies routing, middleware, request/response handling for APIs and web apps.
- **Why Use Express.js?** Simplifies building web servers/APIs, helpful utilities, large middleware ecosystem, unopinionated.

V. EXPRESS.JS CORE CONCEPTS

- **Basic Setup:** `npm install express`. Create `app.js` (or `server.js`), require Express, create app instance, define routes, start server (`app.listen()`).
- **Routing:** How app endpoints (URIs) respond to client requests (GET, POST, etc.).
  - `app.METHOD(PATH, HANDLER)`. `HANDLER` gets `(req, res, next)`.
  - **Route Parameters (`req.params`):** Capture dynamic URL segments (e.g., `/users/:id`).
  - **Query Parameters (`req.query`):** Key-value pairs after `?` in URL (e.g., `/search?term=node`).
  - **`express.Router`:** Modular, mountable route handlers. Group routes in separate files.
  - **Route Chaining (`app.route('/path').get(...).post(...)`):** Group handlers for same path.

HTTP METHODS (WEB STANDARD, CRUD OPERATIONS)

- Standard methods for actions on resources. CRUD: Create, Read, Update, Delete.
- **GET (Read):** Retrieve data. *Ex (Express):* `app.get('/users/:id', (req, res) => { /* find user */ });`
- **POST (Create):** Submit data to create a new resource. *Ex (Express):* `app.post('/users', (req, res) => { /* create user from req.body */ });`

- **PUT (Update/Replace):** Update/replace an existing resource completely. *Ex (Express):* `app.put('/users/:id', (req, res) => { /* replace user from req.body */ });`
- **PATCH (Update/Modify):** Apply partial modifications to a resource. *Ex (Express):* `app.patch('/users/:id', (req, res) => { /* partially update user from req.body */ });`
- **DELETE (Delete):** Delete a specified resource. *Ex (Express):* `app.delete('/users/:id', (req, res) => { /* delete user */ });`
- **HEAD:** GET without response body (headers only).
- **OPTIONS:** Describe communication options for a resource.

- **Middleware:** [PYQ Q7(a) Concept of middleware in Express.js. Give two examples.] Functions with access to `req`, `res`, `next`.
  - Can execute code, modify `req`/`res`, end cycle, or call `next()` middleware.
  - **Types:** Application-level (`app.use(logger)`), Router-level (`router.use()`), Route-specific (passed to route handler), Error-handling (`app.use((err, req, res, next) => ...)` - defined last).
  - **Examples for PYQ Q7(a):**
    1. **Logger Middleware:**

```
const logger = (req, res, next) => {
  console.log(`${req.method} ${req.url} - ${new
Date().toISOString()}`);
  next(); // Purpose: Log request details and pass control.
};
app.use(logger);
```

    2. **Authentication Middleware (Conceptual):**

```
const checkAuth = (req, res, next) => {
  if (req.session && req.session.user) {
    next(); // Purpose: Verify user is authenticated, allow access.
  } else {
    res.status(401).send('Unauthorized'); // Purpose: Block
unauthenticated access.
  }
};
app.get('/protected-route', checkAuth, (req, res) => { /* ... */ });
```

- **Built-in Middleware:** `express.json()` (parses JSON body), `express.urlencoded()` (parses URL-encoded body), `express.static('public')` (serves static files).
- **Third-Party Middleware:** `cors` (Cross-Origin Resource Sharing), `morgan` (HTTP logger), `cookie-parser`, `multer` (file uploads), `helmet` (security headers).

- **Request (`req`) Object:** Represents incoming HTTP request. Properties: `req.params`, `req.query`, `req.body`, `req.headers`, `req.method`, `req.url`, `req.cookies`.
- **Response (`res`) Object:** Represents HTTP response. Methods: `res.send()`, `res.json()`, `res.status()`, `res.sendStatus()`, `res.redirect()`, `res.render()` (templates), `res.sendFile()`, `res.set()` (headers), `res.cookie()`.
- **Template Engines (View Engines):** Embed dynamic data into HTML (e.g., EJS, Pug, Handlebars). Configure with `app.set('view engine', 'ejs')`, `app.set('views', './views')`. Render with `res.render('templateName', data)`.
- **File Uploads (`multer`):** Middleware for `multipart/form-data`. Configure storage (disk/memory), file filters. Access files via `req.file` (single) or `req.files` (multiple).
- **Cookies (`cookie-parser`):** Parse `Cookie` header (`req.cookies`), set cookies (`res.cookie()`).
- **Express Generator (Scaffolding):** [PYQ Q7(b)(ii) Express.js Scaffolding] CLI tool (`express-generator`) to quickly create basic Express app structure (routes, views, public folders, `app.js`). `npm install -g express-generator`, then `express my-app --view=ejs`.
- **Common Project Structure (MVC-like):** Folders for `config`, `controllers`, `models`, `routes`, `views`, `public`, `middleware`, `utils`. `app.js` for setup, `server.js` for starting.

VI. NODE.JS DATABASE INTEGRATION

- Node.js connects to DBs via specific driver libraries (npm packages).
- **MongoDB with Mongoose (ODM - Object Data Modeling):** Recommended for MongoDB.
  - Install: `npm install mongoose`.
  - Connect: `mongoose.connect('mongodb_uri')`.
  - Define Schema: `const mySchema = new mongoose.Schema({ ... });`.
  - Create Model: `const MyModel = mongoose.model('ModelName', mySchema);`.
  - CRUD: `MyModel.create()`, `MyModel.find()`, `MyModel.findById()`, `MyModel.findByIdAndUpdate()`, `MyModel.findByIdAndDelete()`.
- **PostgreSQL with `pg` driver:**
  - Install: `npm install pg`.
  - Connect using `Pool` or `Client`.
  - Execute SQL queries: `pool.query('SELECT * FROM users')`.

VII. ADVANCED TOPICS / FAQ

- **First-Class Functions:** Functions treated as variables (assigned, passed as args, returned).
- **Callback Hell & Avoidance:** Deeply nested callbacks. Avoid with Promises, Async/Await, Named Functions, Modularization. [PYQ Q1(d)]
- **Promises vs. Callbacks:** Promises offer better readability, error handling (`.catch()`), composition.

- **`process.nextTick()` vs. `setImmediate()`:** Both async. `nextTick` runs immediately after current op (before next event loop phase, higher priority). `setImmediate` runs in "check" phase (after I/O, safer for deferring).
- **Node.js Exit Codes:** Indicate process termination status (0: success, 1: uncaught fatal exception).
- **Stubs in Testing:** Replace real dependencies with simulated objects/functions.
- **Reactor Pattern:** Architectural pattern for concurrent requests. Node.js event loop is an implementation.
- **`perf_hooks`:** Measure async operation performance (`performance.mark`, `performance.measure`).
- **WASI (WebAssembly System Interface):** API for WebAssembly to interact with OS outside browser.
- **Package Management (`package.json`):** Records metadata, `dependencies`, `devDependencies`, `scripts`. `package-lock.json` locks exact versions.
  - Commands: `npm install <pkg>`, `npm install -D <pkg>`, `npm uninstall <pkg>`, `npm update`, `npm list`.

UNIT 4: MONGODB

1. INTRODUCTION TO MONGODB

- **What is MongoDB?** [PYQ Q8(a) features of MongoDB, how it differs from RDBMS] Open-source NoSQL, document-oriented database. Written in C++. Uses JSON-like BSON documents with optional/dynamic schemas. Scalable, cross-platform.
  - **Concepts:** Collection (like table), Document (like row/JSON object).
- **Key Features:** [PYQ Q8(a)]
  - **Indexing:** Supports various indexes (secondary, unique, compound, geospatial, full-text) for query performance.
  - **Aggregation Framework:** Data processing pipelines for complex analysis/transformation.
  - **Special Collections/Indexes:** TTL (Time-To-Live) collections for auto-expiring data.
  - **File Storage (GridFS):** Protocol for storing large files (images, videos).
  - **Sharding:** [PYQ Q9(a) process of sharding, high availability] Horizontal scaling by splitting data across multiple machines.
    - **Process:** Data distributed based on a shard key. Query router (`mongos`) directs queries. Config servers store metadata.
    - **High Availability:** [PYQ Q9(a)] Achieved via **Replica Sets**. Multiple copies of data on different servers. Automatic failover if primary node fails. Sharding can also incorporate replica sets for each shard.

2. CRUD OPERATIONS IN MONGODB (MONGOSH SHELL) [PYQ Q8(b) Perform with code (any two): Add data, Delete, Update]

- Use `mongosh` shell.
- **Create (Add/Insert Data):** [PYQ Q8(b)(i)]
  - `db.collectionName.insertOne({ field: "value", ... })`
  - `db.collectionName.insertMany([{...}, {...}])`
- **Read (Query Data):**
  - `db.collectionName.find({ query_filter })` (e.g., `db.users.find({ age: { $gt: 25 } })`)
  - `db.collectionName.findOne({ query_filter })`
  - Empty filter `{}` finds all.
- **Update Data:** [PYQ Q8(b)(iii)]
  - `db.collectionName.updateOne({ filter }, { $set: { fieldToUpdate: newValue } })`
  - `db.collectionName.updateMany({ filter }, { $set: { fieldToUpdate: newValue } })`
  - `db.collectionName.replaceOne({ filter }, { new_document_structure })` (replaces entire doc except `_id`)
- **Delete Data:** [PYQ Q8(b)(ii)]
  - `db.collectionName.deleteOne({ filter })`
  - `db.collectionName.deleteMany({ filter })`

3. SQL VS NOSQL: KEY DIFFERENCES [PYQ Q8(a) How MongoDB differs from traditional relational databases]

Feature	SQL (Relational)	NoSQL (Non-Relational, e.g., MongoDB)
Data Model	Tables (Rows & Columns)	Documents (MongoDB), Key-Value, Graph, Columnar
Schema	Fixed Schema (Structured)	Dynamic/Flexible Schema (MongoDB is schema-less)
Query Lang.	SQL (SELECT, JOIN, etc.)	Varies (MongoDB uses JSON-like MQL queries)
Scalability	Vertical (More CPU/RAM)	Horizontal (Distributed systems, Sharding)
Transactions	ACID-compliant (strong consistency)	BASE (Eventual Consistency - often tunable)
Best For	Complex relationships, structured data	Big Data, real-time apps, unstructured data

4. MANAGING MONGODB

- **Installation:** Download from MongoDB official site (MSI for Windows, Homebrew for Mac, package manager for Linux). Start server: `mongod --dbpath /path/to/data`.
- **Connect:** Use `mongosh` shell.
- **Create/Switch Database:** `use myDatabase` (implicitly creates on first data write).
- **Collections:** Create explicitly `db.createCollection("users")` or implicitly on first insert.
- **Drop Collection:** `db.collectionName.drop()`.
- **Drop Database:** `use targetDb; db.dropDatabase()`.
- **Connect MongoDB to Node.js:** Use `mongodb` Node.js driver.

```
const { MongoClient } = require("mongodb");
const url = "mongodb://localhost:27017";
const client = new MongoClient(url);
async function run() { /* ... connect, perform ops, client.close() ... */
}
```

5. DATA MIGRATION INTO MONGODB

- **From JSON/CSV:** Use `mongoimport` CLI tool (`--jsonArray`, `--type csv --headerline`).
- **From SQL (MySQL, PostgreSQL):**
  1. Export SQL data to CSV/JSON.
  2. Transform data to fit MongoDB document schema (embedding vs. referencing).
  3. Import using `mongoimport` or custom script (Node.js, Python).
- **From Firebase (Firestore):** Use Firebase Admin SDK to export, then custom script (Node.js) with MongoDB driver to import.
- **From Excel:** Convert to CSV/JSON first, or use Node.js library like `xlsx` to read Excel and import.

6. MONGODB CONCEPTS (IN CONTEXT OF PYQ Q9(b))

- **Document:** [PYQ Q9(b)(i)] A record in a MongoDB collection, stored in BSON (binary JSON) format. A set of key-value pairs. Schemaless.

```
{ "_id": ObjectId("..."), "name": "Alice", "age": 30, "city": "New York"
}
```

- **Collection:** [PYQ Q9(b)(ii)] A grouping of MongoDB documents. Equivalent to a table in RDBMS, but doesn't enforce a schema. Documents within a collection can have different fields.
- **Databases:** [PYQ Q9(b)(iii)] 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.

7. MONGODB SERVICES (KEY ONES)

- **MongoDB Atlas:** Fully managed global cloud database service (DBaaS) on AWS, Azure, GCP. Handles infrastructure, scaling, backups.
- **MongoDB Enterprise Advanced:** Self-managed, commercial version with advanced security, monitoring (Ops Manager), support.
- **MongoDB Community Edition:** Free, open-source, self-managed version.
- **MongoDB Realm:** Mobile database (offline-first) and backend application development platform with sync to Atlas.
- **MongoDB Charts:** Data visualization tool for Atlas data.
- **MongoDB Compass:** GUI tool for interacting with MongoDB (Community, Enterprise, Atlas).
- **MongoDB Atlas Search:** Managed full-text search engine integrated with Atlas.
- **MongoDB Atlas Data Lake:** Query data in cloud object storage (S3, Azure Blob) using MQL.
