MERN END-TERM SOLUTION

END TERM EXAMINATION SOLUTIONS

Q1: Attempt all the questions. [3*5=15]

(a) Explain the three-tier architecture of Mern Stack with diagram.

- **Theory:** MERN stack (MongoDB, Express.js, React.js, Node.js) commonly uses a 3-tier architecture for separation of concerns, scalability, and maintainability.
- **Tiers:**

1. **Presentation Tier (Client-Side):**

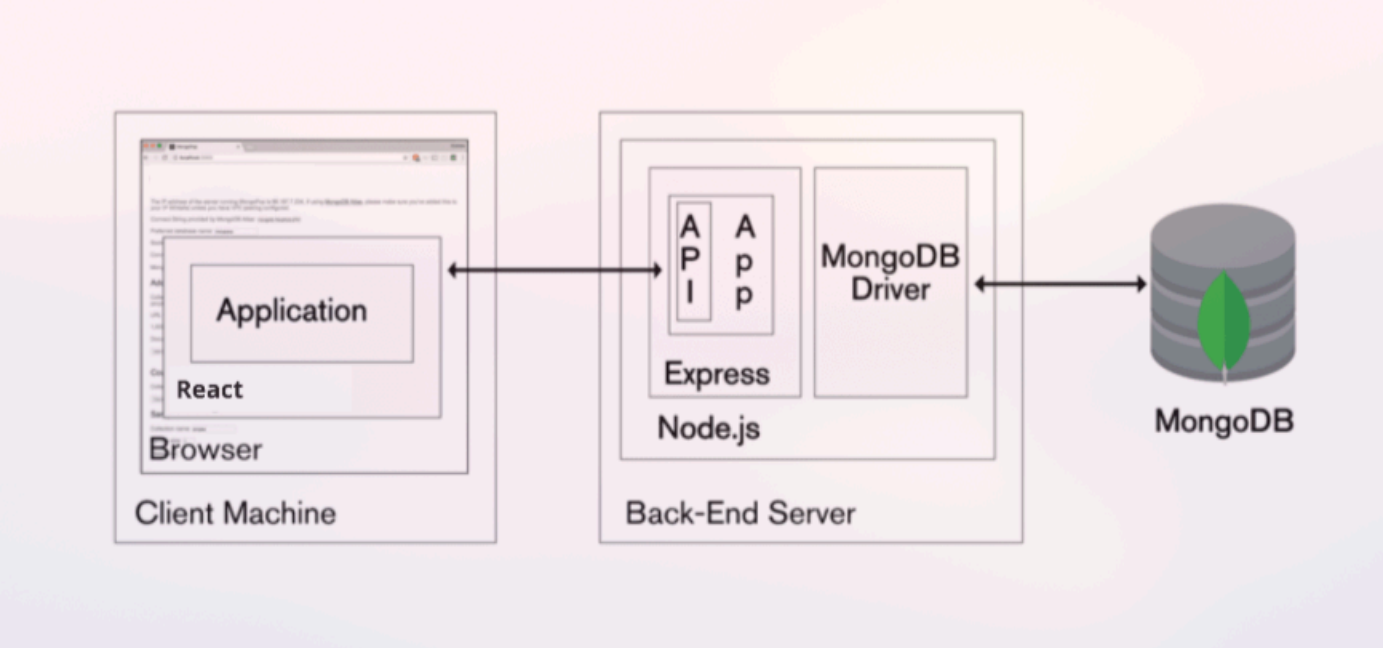
- Handled by **React.js**.
- Responsible for the User Interface (UI) and User Experience (UX).
- Runs in the user's web browser.
- Sends requests to and receives data from the application tier.

2. **Application Tier (Server-Side Logic/Middle Tier):**

- Handled by **Node.js** and **Express.js**.
- Contains the business logic, API routes, request handling, and data processing.
- Acts as an intermediary between the client and the database.

3. **Data Tier (Database Layer):**

- Handled by **MongoDB**.
- Responsible for storing, retrieving, and managing application data.
- **Diagram:**
- Client (React in Browser) <---> API (Express.js on Node.js Server) <---> Database (MongoDB)



(b) Differentiate between: Shadow Dom and Virtual Dom.

- **Virtual DOM (VDOM):**
- **Concept:** A lightweight, in-memory representation (JavaScript object) of the actual browser's Document Object Model (DOM).
- **Usage (React):** React uses the VDOM for performance optimization. When state changes, React creates a new VDOM tree, "diffs" it against the previous VDOM tree, and then efficiently updates only the necessary parts of the real DOM.
- **Benefit:** Minimizes direct manipulation of the slow real DOM, leading to faster UI updates.
- **Shadow DOM:**
- **Concept:** A browser technology designed for encapsulation in web components. It allows a hidden, separate DOM tree to be attached to an element in the main document DOM.
- **Usage:** Scopes CSS styles and JavaScript logic to a specific component, preventing them from leaking out and affecting other parts of the page, and vice-versa.
- **Benefit:** Creates reusable, encapsulated components with isolated styles and behavior.
- **Key Differences:**

Feature	Virtual DOM (React)	Shadow DOM (Browser API)
Purpose	Performance optimization via diffing	Encapsulation, style/script scoping
Nature	In-memory JS object	Actual, separate DOM tree in the browser
Manipulation	Managed by libraries like React	Standard DOM APIs within its scope
Scope	Concept within a JS library framework	A web platform feature for web components

(c) Describe the building blocks of React?

- The primary building blocks of React are:

1. **Components:**

- Reusable, independent pieces of UI. They can be JavaScript functions (Functional Components) or ES6 classes (Class Components).
- They accept inputs (called "props") and return React elements describing what should appear on the screen.
- Components can be composed to build complex UIs.

2. **JSX (JavaScript XML):**

- A syntax extension for JavaScript that allows writing HTML-like structures within JavaScript code.
- Describes the UI of components. It's transpiled by tools like Babel into `React.createElement()` calls.

3. **Props (Properties):**

- Read-only objects used to pass data from parent components to child components, allowing for dynamic and configurable UIs.

4. **State:**

- An object that represents parts of a component that can change over time, typically due to user interaction or network responses.
- When state changes, React re-renders the component. State is local and encapsulated within the component that owns it.

(d) What is meant by “Callback” and “Callback Hell” in Node JS?

- **Callback:**
- **Definition:** In JavaScript (and Node.js), a callback is a function that is passed as an argument to another function (the higher-order function).
- **Purpose:** The higher-order function executes the callback function at a later point in time, typically after an asynchronous operation (like reading a file, making a network request, or a timer expiring) has completed.
- *Example:*

```
fs.readFile('file.txt', 'utf8', (err, data) => { // This arrow function is a callback
  if (err) console.error(err); else console.log(data);
});
```

- **Callback Hell (Pyramid of Doom):**
- **Definition:** Refers to a situation where multiple asynchronous operations are performed sequentially, and each subsequent operation depends on the result of the previous one. This leads

to deeply nested callback functions.

- **Problem:** The code becomes difficult to read, understand, debug, and maintain due to its pyramid-like structure and rightward drift.
- *Structure:*

```
asyncOp1((err1, res1) => {
  if (err1) { /* handle error */ } else {
    asyncOp2(res1, (err2, res2) => {
      if (err2) { /* handle error */ } else {
        asyncOp3(res2, (err3, res3) => { /* ...and so on... */ });
      }
    });
  }
});
```

- **Avoidance:** Use Promises, async/await, named functions, or modularization.

(e) How to explain closures in JavaScript and when to use it?

- **Explanation:**
- **Definition:** A closure is a combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment). In simpler terms, an inner function has access to the variables and parameters of its outer (enclosing) function, even after the outer function has finished executing and returned.
- The inner function "remembers" the environment in which it was created.
- **How it Works:** When a function is defined, it creates a scope. If an inner function is defined within an outer function, the inner function forms a closure, capturing the scope of the outer function.
- **When to Use It (Use Cases):**

1. **Data Privacy / Encapsulation:** Creating private variables and methods that are not accessible from the outside, exposing only a public interface.

```
function createCounter() {
  let count = 0; // private variable
  return function increment() { count++; console.log(count); };
}
const counter1 = createCounter(); counter1(); // 1; counter1(); // 2
// `count` is not directly accessible outside createCounter
```

2. **Maintaining State in Asynchronous Operations:** Useful in event handlers or callbacks where a function needs to access some state from when it was defined.
3. **Currying and Partial Application:** Creating functions that take arguments one at a time or pre-fill some arguments.
4. **Module Pattern (older JS):** Simulating modules before ES6 modules were available.

UNIT-1

Q2 (a) In How many ways an HTML element can be accessed in JavaScript code? What is the 'this' keyword in JavaScript? [5]

- **Ways to Access HTML Elements in JS:**

1. `document.getElementById('elementId')`: Selects a single element by its unique `id`.
2. `document.getElementsByTagName('tagName')`: Returns an HTMLCollection (array-like) of all elements with the specified tag name.
3. `document.getElementsByClassName('className')`: Returns an HTMLCollection of all elements with the specified class name.
4. `document.querySelector('cssSelector')`: Returns the *first* element that matches the specified CSS selector.
5. `document.querySelectorAll('cssSelector')`: Returns a static NodeList (array-like) of all elements that match the specified CSS selector.
6. Accessing forms and form elements: `document.forms['formName']`, `document.forms[0].elements['inputName']`.

- **The `this` keyword in JavaScript:**
- **Definition:** `this` is a special keyword that refers to the object it belongs to, or the context in which the current code is executing. Its value is determined by how a function is called (invocation context).
- **Contexts:**
- **Global Context:** Outside any function, `this` refers to the global object (`window` in browsers, `global` in Node.js strict mode off).
- **Function Context (Simple Call):** In a regular function call (not as a method of an object), `this` also refers to the global object in non-strict mode. In strict mode (`'use strict';`), `this` is `undefined`.
- **Method Context:** When a function is called as a method of an object, `this` refers to the object the method is called on (e.g., `obj.myMethod()`, `this` is `obj`).
- **Constructor Context:** When a function is used as a constructor with the `new` keyword (e.g., `new MyObject()`), `this` refers to the newly created instance.
- **Event Handlers:** In HTML event handlers, `this` typically refers to the HTML element that received the event.
- **Arrow Functions:** Arrow functions do not have their own `this` binding. They lexically inherit `this` from their surrounding (enclosing) non-arrow function's scope.

(b) Discuss the different CSS link states? Can elements be overlapped in CSS. Justify this. [5]

- **CSS Link States (Pseudo-classes):**

1. `:link`: Styles an unvisited link.
   `a:link { color: blue; }`
2. `:visited`: Styles a link that the user has already visited.
   `a:visited { color: purple; }`
3. `:hover`: Styles a link when the user mouses over it.
   `a:hover { color: red; text-decoration: underline; }`
4. `:active`: Styles a link when it is being clicked (activated).
   `a:active { color: orange; }`

- *Note:* For these to work correctly, they are often defined in the LVHA-order: `:link`, `:visited`, `:hover`, `:active` because of CSS specificity.
- **Overlapping Elements in CSS:**
- Yes, elements can be overlapped in CSS.
- **Justification/Methods:**

1. **`position` Property:**

- `position: absolute;` or `position: fixed;`: Takes the element out of the normal document flow. You can then use `top`, `right`, `bottom`, `left` properties to position it, potentially overlapping other elements.
- `position: relative;` on a parent can be used as a containing block for absolutely positioned children.

2. **`z-index` Property:**

- Used with positioned elements (`absolute`, `relative`, `fixed`, `sticky`).
- Controls the stacking order of overlapping elements. An element with a higher `z-index` value will appear in front of an element with a lower `z-index`.

3. **Negative Margins:**

- Applying negative margins (e.g., `margin-top: -20px;`) can pull an element over another.

4. **CSS Transforms (`translate`):**

- Using `transform: translate(x, y);` can move an element, potentially causing it to overlap, without affecting the layout of other elements.
- *Example:*

```
.box1 { position: absolute; top: 10px; left: 10px; width: 100px; height: 100px; background: red; z-index: 1; }
.box2 { position: absolute; top: 30px; left: 30px; width: 100px; height: 100px; background: blue; z-index: 2; } /* box2 overlaps box1 */
```

(c) Differentiate between: [2*2.5=5]

(i) Block elements and inline elements

Feature	Block-level Elements	Inline Elements
New Line	Always start on a new line.	Do not start on a new line; flow with content.
Width	Take up the full width available by default.	Take up only as much width as necessary.
Height/Width	`height` and `width` properties apply.	`height` and `width` properties generally do not apply directly (padding/margin horizontal only).
Examples	`<div>`, `<p>`, `<h1>`-`<h6>`, `<ul>`, `<li>`	`<span>`, `<a>`, `<img>`, `<strong>`, `<em>`
Can Contain	Can contain other block or inline elements.	Typically contain only data or other inline elements.

(ii) `<em>` tag and `<i>` tag

Feature	`<em>` (Emphasis) Tag	`<i>` (Idiomatic Text/Icon) Tag
Semantic Meaning	Indicates text that should be emphasized (semantically important). Browsers usually render as italic.	Represents text in an alternate voice or mood, or for icons. Historically used for italics, but its semantic meaning is now broader.
Default Styling	Typically rendered as italic by browsers.	Typically rendered as italic by browsers.
Purpose	To stress emphasis on a word or phrase.	For terms, technical names, foreign words, thoughts, ship names, or when an icon font is used.
Accessibility	Screen readers may convey emphasis (e.g., change in voice tone).	Screen readers generally do not announce any special emphasis.
Usage Example	`You <em>must</em> complete this.`	`The term is <i>HTML</i>.` or `<i class="fas fa-home"></i>` (for Font Awesome icon)

Q3 (a) How to handle JavaScript Events in HTML? Explain with example. [7]

- **Event Handling Theory:** Events are actions that occur in the browser (user clicks, page loads, key presses). JavaScript can "listen" for these events and execute code (an event handler/listener function) in response.
- **Methods to Handle JS Events in HTML:**

1. **Inline Event Handlers (HTML Attributes):**

- JavaScript code is directly embedded within HTML event attributes (e.g., `onclick`, `onmouseover`).
- Simple for small tasks but generally discouraged for larger applications due to mixing HTML and JS, and poor maintainability.
- *Example:*

```
<button onclick="alert('Button was clicked!')">Click Me</button>
<p onmouseover="this.style.color='red';"
onmouseout="this.style.color='black';">Hover over me!</p>
```

2. **DOM Property Event Handlers:**

- Assign a JavaScript function to an element's DOM property (e.g., `element.onclick = myFunction;`).
- Cleaner than inline, but you can only assign one handler per event type per element.
- *Example:*

```
<button id="myBtn">Click Me Too</button>
<script>
  const btn = document.getElementById('myBtn');
  btn.onclick = function() {
    alert('Button (DOM property) clicked!');
  };
</script>
```

3. **`addEventListener()` Method (Recommended):**

- Attaches an event listener function to an element without overwriting existing event handlers.
- Allows multiple listeners for the same event type on one element.
- Provides more control (e.g., event capturing/bubbling phase).
- *Syntax:* `element.addEventListener('eventName', functionName, useCaptureBoolean);`
- *Example:*

```
<button id="listenerBtn">Click Me (Listener)</button>
<script>
  const listenerBtn = document.getElementById('listenerBtn');
  function showAlert() {
    alert('Button (addEventListener) clicked!');
  }
  listenerBtn.addEventListener('click', showAlert); // No 'on' prefix
  // To remove: listenerBtn.removeEventListener('click', showAlert);
</script>
```

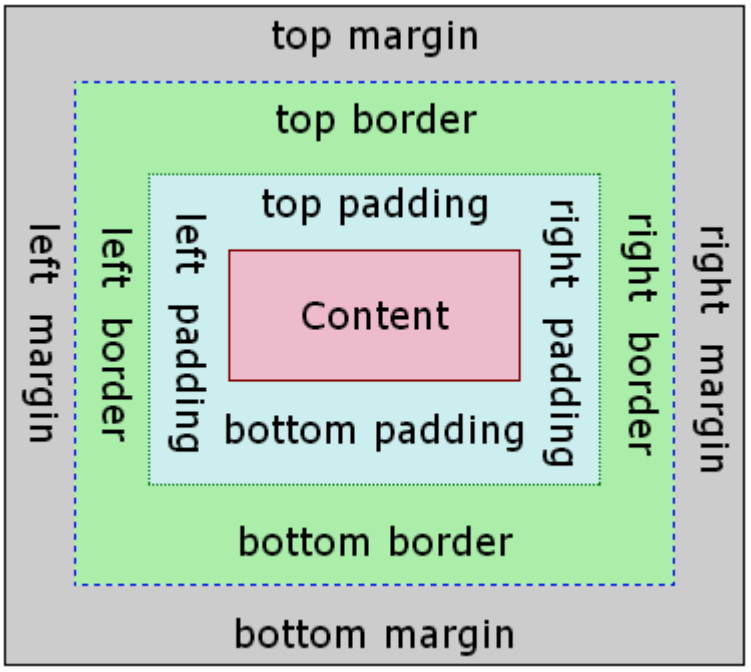
(b) Write short notes on (any two): [2*4=8]

(i) Call Method( ) and Apply Method( )

- **Purpose:** Both `call()` and `apply()` are built-in JavaScript function methods used to invoke a function with a specified `this` value and arguments. They allow you to borrow methods from other objects or explicitly set the `this` context for a function call.
- **`call(thisArg, arg1, arg2, ...)`:**
- Invokes the function with a given `this` value and arguments provided individually (comma-separated).
- *Structure:* `functionName.call(objectForThis, param1, param2);`
- **`apply(thisArg, [argsArray])`:**
- Invokes the function with a given `this` value and arguments provided as an array (or an array-like object).
- *Structure:* `functionName.apply(objectForThis, [param1, param2]);`
- **Key Difference:** How they accept arguments (individual for `call`, array for `apply`).
- *Example:*

```
function greet(greeting, punctuation) {
  console.log(greeting + ', ' + this.name + punctuation);
}
const person = { name: 'Alice' };
greet.call(person, 'Hello', '!');   // Output: Hello, Alice!
greet.apply(person, ['Hi', '.']);    // Output: Hi, Alice.
```

(ii) CSS Box Model

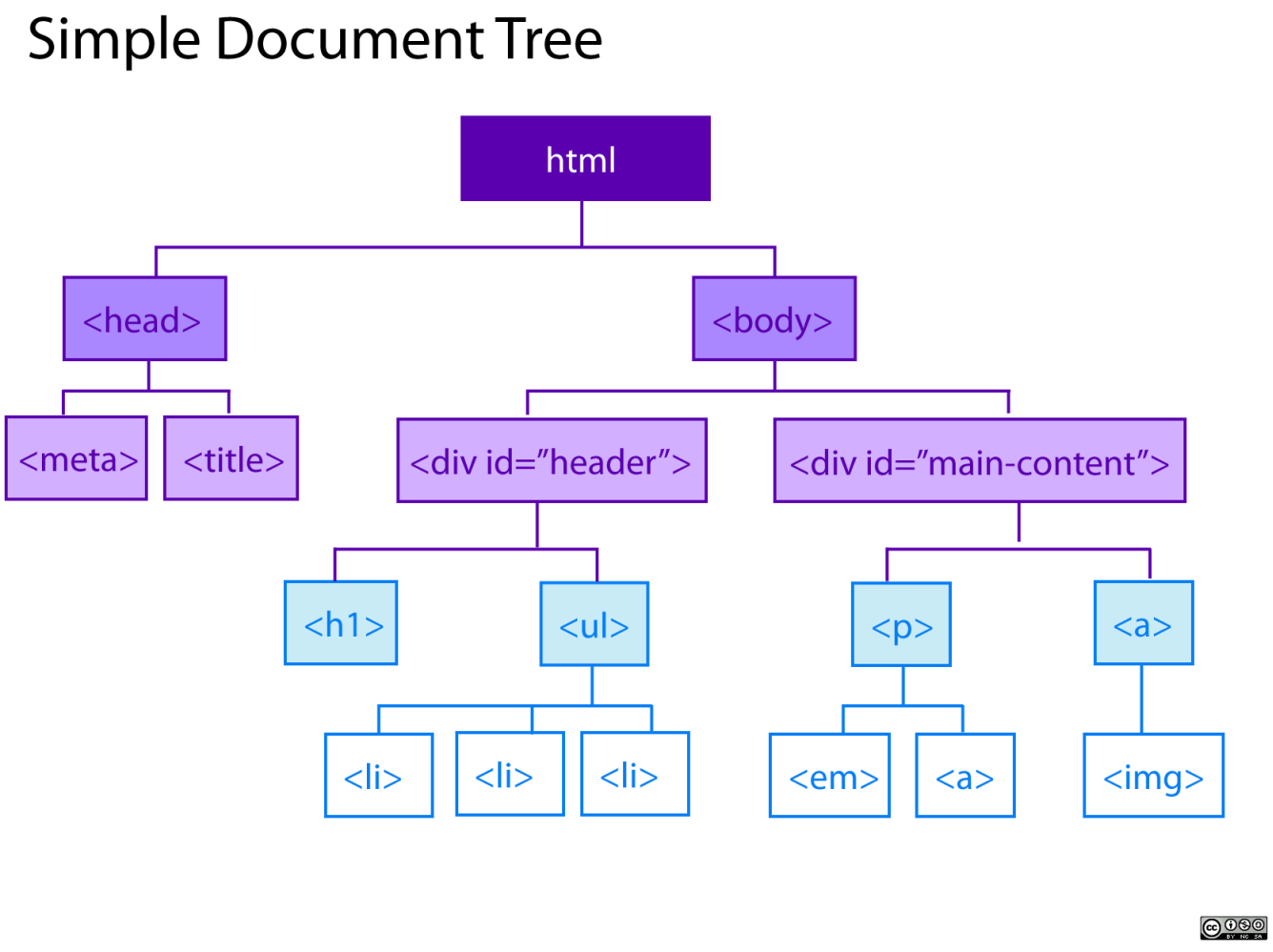


- **Definition:** A fundamental CSS concept describing how HTML elements are rendered as rectangular boxes on a web page. Each box has distinct layers that contribute to its overall size and spacing relative to other elements.
- **Layers (from innermost to outermost):**

1. **Content:** The actual content of the element (text, images, etc.). Its dimensions can be set by `width` and `height` properties.
2. **Padding:** Transparent space around the content, *inside* the border. Adds to the element's visible size. Controlled by `padding` properties (e.g., `padding-top`, `padding`).
3. **Border:** A line that goes around the padding and content. Has thickness, style, and color. Controlled by `border` properties (e.g., `border-width`, `border-style`).
4. **Margin:** Transparent space *outside* the border. Separates the element from other elements. Controlled by `margin` properties (e.g., `margin-bottom`, `margin`). Margins can collapse vertically.

- The total width/height of an element is the sum of its content, padding, border, and margin (though `box-sizing: border-box;` changes this so width/height include padding and border).

(iii) Document Trees (DOM)



- **Definition:** The Document Object Model (DOM) is a programming interface (API) for HTML and XML documents. It represents the logical structure of a document as a hierarchical tree of nodes.
- **Tree Structure:**
- The browser parses an HTML document and creates this tree.
- The root node is typically `document`.
- Each HTML tag becomes an **Element Node**.

- Text within elements becomes **Text Nodes**.
- HTML attributes become **Attribute Nodes** (properties of element nodes).
- Comments become **Comment Nodes**.
- **Purpose:** Allows programs (especially JavaScript) to dynamically access and manipulate the content, structure, and style of a web page.
- **Navigation:** JS can navigate this tree using properties like `parentNode`, `childNodes`, `firstChild`, `nextSibling`, etc., to find and modify specific parts of the page.

UNIT-2

Q4 (a) Explain the components in React JS. Write the differences between class and functional components with example? [8]

- **React Components:**
- Reusable, independent UI building blocks. Return React elements describing UI.
- **Functional Components:** JavaScript functions. Accept `props`, return JSX. Simpler, often used for presentational UI. Can use Hooks for state/lifecycle.
- *Example:* `function Welcome(props) { return <h1>Hello, {props.name}</h1>; }`
- **Class Components:** ES6 classes extending `React.Component`. Must have a `render()` method. Can have `this.state` and lifecycle methods.
- *Example:* `class Welcome extends React.Component { render() { return <h1>Hello, {this.props.name}</h1>; } }`
- **Differences: Class vs. Functional Components:**

Feature	Class Components	Functional Components (with Hooks)
Syntax	ES6 `class`	JavaScript `function` or arrow function
State	`this.state`, `this.setState()`	`useState()` Hook
Lifecycle	Lifecycle methods (e.g., `componentDidMount`)	`useEffect()` Hook (for side effects)
`this` Keyword	Used to access props, state, methods	Not used; props accessed via arguments
Simplicity	More boilerplate	Generally more concise

(b) Define React Hooks. Demonstrate the useState hook and useEffect hook in react? [7]

- **React Hooks Definition:** 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.
- **`useState` Hook:**

- **Purpose:** Adds state to functional components.
- **Syntax:** `const [stateVariable, setStateFunction] = useState(initialState);`
- **Demonstration:**

```
import React, { useState } from 'react';
function Counter() {
  const [count, setCount] = useState(0); // Initialize count state to 0
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
```

- **`useEffect` Hook:**
- **Purpose:** Lets you perform side effects in functional components (e.g., data fetching, subscriptions, manually changing the DOM).
- **Syntax:** `useEffect(() => { /* side effect code */; return () => { /* cleanup (optional) */ }; }, [dependenciesArray (optional)]);`
- **Demonstration (updates document title):**

```
import React, { useState, useEffect } from 'react';
function DocumentTitleChanger() {
  const [name, setName] = useState('User');
  useEffect(() => {
    document.title = `Welcome, ${name}!`; // Side effect: update document title
  }, [name]); // Only re-run effect if 'name' state changes

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

Q5 (a) Explain State and Props in React JS. Give an example to update the state of component. [7]

- **Props (Properties):**
- Data passed from a parent component to a child component.
- Read-only within the child component (unidirectional data flow).
- Used to configure and customize child components.
- *Example:* `<Child name="Alice" />` (Child receives `props.name`).

- **State:**
- Data that is private to and managed by a component itself.
- Can change over time (mutable) in response to user actions or network events.
- When state changes, the component re-renders.
- Class components: `this.state` and `this.setState()`. Functional components: `useState()` hook.
- **Example to Update State (Functional Component with `useState`):**

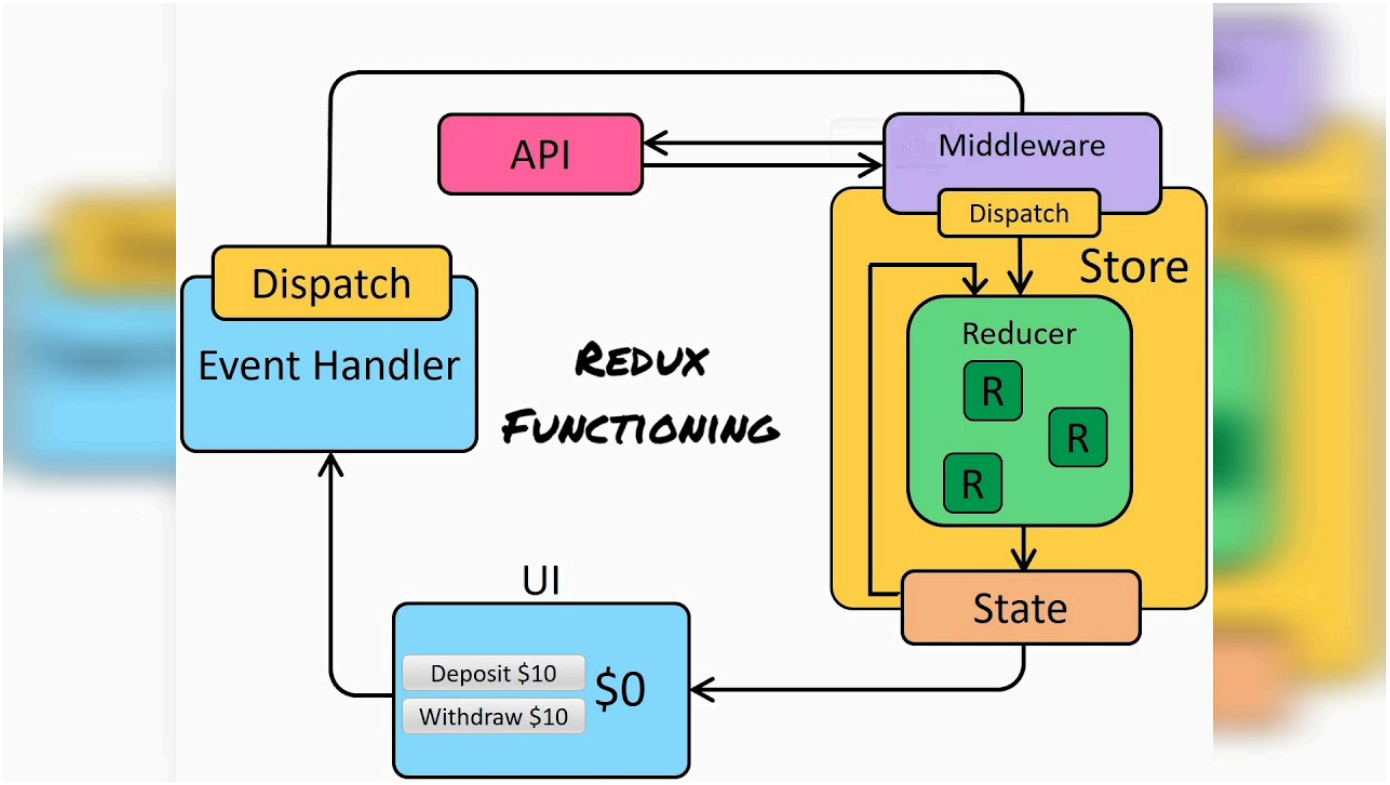
```
import React, { useState } from 'react';
function LightSwitch() {
  const [isOn, setIsOn] = useState(false); // Initial state: off

  const toggleLight = () => {
    setIsOn(prevState => !prevState); // Update state based on previous
state
  };

  return (
    <div>
      <p>The light is {isOn ? 'ON' : 'OFF'}</p>
      <button onClick={toggleLight}>Toggle Light</button>
    </div>
  );
}
```

(b) Describe in brief (any two): [2*4=8]

(i) React Router

- A standard library for routing in React applications, enabling navigation between different views or components within a Single Page Application (SPA) without full page reloads.
- Key components: `<BrowserRouter>` (or `<HashRouter>`), `<Routes>` (v6) / `<Switch>` (v5), `<Route path="/path" element={<Component />} />` (v6), and `<Link to="/path">Link Name</Link>`.
- Manages UI rendering based on the URL.

(ii) Redux



- A predictable state container for JavaScript applications, widely used with React for managing complex, global application state.
- **Core Concepts:**

1. **Store:** Single object holding the entire application state.
2. **Action:** Plain object describing "what happened" (e.g., `{ type: 'ADD_ITEM', payload: item }`).
3. **Reducer:** Pure function `(previousState, action) => newState` that specifies how state changes in response to an action.
4. **Dispatch:** Method to send actions to the store (`store.dispatch(action)`).

(iii) Unit Testing

- A software testing method where individual units of source code (e.g., React components, functions) are tested in isolation to verify they work correctly.
- **In React:** Focuses on testing if a component renders correctly for given props/state, handles user interactions, and implements its logic as expected.
- **Tools:** Jest (test runner, assertions), React Testing Library (RTL) (for user-centric testing of component behavior).

UNIT-3

Q6 (a) Explain the significance of Node.js in backend development. How Node.js differs from traditional server-side technologies. [5]

- **Significance of Node.js in Backend:**

1. **JavaScript Full-Stack:** Allows using JavaScript for both frontend and backend, improving developer productivity and code sharing.
2. **High Performance for I/O-Bound Apps:** Its non-blocking, event-driven architecture makes it excellent for applications with many concurrent connections and I/O operations (e.g., APIs, real-time apps, microservices).
3. **Scalability:** Lightweight and efficient, making it easier to scale applications horizontally.
4. **Large Ecosystem (npm):** Access to a vast number of open-source libraries and tools.
5. **Fast Development Cycle:** Rapid prototyping and development.

- **Node.js vs. Traditional Server-Side Technologies (e.g., Java/Spring, PHP/Laravel, Python/Django):**

Feature	Node.js	Traditional Server-Side
Language	JavaScript	Java, PHP, Python, Ruby, C#, etc.
Concurrency Model	Single-threaded, event-driven, non-blocking I/O	Often multi-threaded or multi-process, blocking I/O by default
CPU Usage	Efficient for I/O; less suited for heavy CPU-bound tasks on main thread (use worker_threads)	Can better utilize multiple cores for CPU-bound tasks directly via threads
Development Paradigm	Primarily asynchronous programming	Can be synchronous or asynchronous
Ecosystem	npm (JavaScript focus)	Varied (Maven, Composer, Pip, Gems, etc.)

(b) Illustrate the process of handling HTTP requests and responses using Express.js. [5]

- **Process Overview:** Express.js simplifies handling HTTP requests by providing a routing system and request/response objects.

1. **Client Request:** A client (e.g., browser) sends an HTTP request (e.g., GET, POST) to a specific URL endpoint on the server.
2. **Express Routing:** Express matches the request's HTTP method and URL path to a defined route handler.
3. **Middleware Execution (if any):** Functions that execute sequentially before the route handler. They can modify `req`/`res`, perform logging, authentication, etc.
4. **Route Handler Execution:** The function associated with the matched route is executed. It receives:

- `req` (Request object): Contains information about the incoming request (e.g., headers, URL parameters (`req.params`), query strings (`req.query`), request body (`req.body` - requires body-

parsing middleware like `express.json()`).

- `res` (Response object): Used to send a response back to the client.

5. **Sending Response:** The handler uses methods on the `res` object to send the response (e.g., `res.send()`, `res.json()`, `res.status()`, `res.render()`).

- **Illustration (Code Snippet):**

```
const express = require('express');
const app = express();
app.use(express.json()); // Middleware to parse JSON request bodies

// 1. Route definition for GET request
app.get('/users/:id', (req, res) => { // 2. Express matches path and
method
  // 4. Route handler executes
  const userId = req.params.id; // Accessing route parameter from 'req'
  const user = { id: userId, name: 'Example User' }; // Dummy data
  if (user) {
    res.json(user); // 5. Sending JSON response using 'res'
  } else {
    res.status(404).send('User not found');
  }
});

app.post('/users', (req, res) => {
  const newUser = req.body; // Accessing request body
  res.status(201).json({ message: 'User created', user: newUser });
});
app.listen(3000);
```

(c) Explain the Node.js event loop mechanism. How does it help in handling asynchronous operations efficiently? [5]

- **Event Loop Mechanism:**
- The core of Node.js's non-blocking, asynchronous, single-threaded concurrency model.
- It continuously checks and processes events from queues (timers, I/O, `setImmediate`).
- **Process:** Async ops are offloaded (OS/thread pool). On completion, callbacks are queued. Event loop picks up and executes these callbacks on the main thread.
- **Phases (Simplified):** Timers -> Pending I/O -> Poll -> Check (`setImmediate`) -> Close Callbacks. `process.nextTick()` runs between phases.
- **Efficiency in Handling Asynchronous Operations:**

1. **Non-Blocking:** Main thread doesn't wait for slow I/O.
2. **Concurrency:** Handles many connections with a single thread by managing callbacks.
3. **Resource Efficiency:** Avoids overhead of many threads.
4. **Responsiveness:** Application remains responsive during I/O tasks.

Q7 (a) Discuss the concept of middleware in Express.js. Give two examples of middleware functions and write their purposes. [7]

- **Concept of Middleware:**
- Functions with access to `req` (request), `res` (response), and `next` function in the request-response cycle.
- Can: execute code, modify `req`/`res`, end cycle, or call `next()` to pass to the next middleware.
- Executed sequentially. Used for logging, auth, parsing, error handling.
- **Example 1: Request Logger Middleware**
- **Purpose:** Log details of incoming requests for monitoring/debugging.
- **Function:**

```
const requestLogger = (req, res, next) => {
  console.log(`${new Date().toISOString()} - ${req.method}
${req.originalUrl}`);
  next(); // Pass control
};
// app.use(requestLogger);
```

- **Example 2: Authentication Check Middleware (Conceptual)**
- **Purpose:** Verify user authentication before allowing access to protected routes.
- **Function:**

```
const isAuthenticated = (req, res, next) => {
  if (req.session && req.session.user) { // Example check
    next(); // Authenticated, proceed
  } else {
    res.status(401).json({ message: 'Unauthorized' }); // Block
  }
};
// app.get('/profile', isAuthenticated, (req, res) => { /* ... */ });
```

(b) Give brief description on following: [2*4=8]
(i) Streams in Node JS

- **Definition:** Objects for efficiently handling sequential data (e.g., files, network traffic) in chunks, without loading everything into memory at once.

- **Nature:** Instances of `EventEmitter` (emit `data`, `end`, `error`).
- **Types:** Readable (source), Writable (destination), Duplex (both), Transform (modify data).
- **Piping (`pipe()`):** Connects Readable to Writable streams (e.g., `readable.pipe(writable)`), managing data flow and backpressure.

(ii) Express.js Scaffolding

- **Definition:** Quickly generating a basic application structure for a new Express.js project.
- **Tool: `express-generator`:** A command-line utility.
- **Purpose:** Creates standard directory layout (routes, views, public), basic `app.js`, view engine setup, sample routes.
- **Usage:** `npm install -g express-generator`, then `express myapp --view=ejs`, then `cd myapp && npm install`, `npm start`.
- **Benefits:** Saves setup time, provides conventional structure.

UNIT-4

Q8 (a) Write the features of MongoDB. How does MongoDB differ from traditional relational databases? [7]

- **Features of MongoDB:**

1. **Document-Oriented:** Stores data in flexible, JSON-like BSON documents.
2. **Dynamic Schema:** Documents in a collection can have varying fields.
3. **Horizontal Scalability:** Scales out using sharding.
4. **Indexing:** Rich indexing capabilities for query performance.
5. **Aggregation Framework:** For complex data processing pipelines.
6. **Replication:** Replica sets for high availability and data redundancy.
7. **GridFS:** For storing large files.

- **MongoDB (NoSQL) vs. Traditional RDBMS (SQL):**

Feature	MongoDB (NoSQL)	Traditional RDBMS (SQL)
Data Model	Documents (BSON)	Tables (Rows & Columns)
Schema	Dynamic/Flexible	Fixed/Predefined
Scalability	Horizontal (Sharding)	Primarily Vertical
Joins	Denormalization/Embedding, `$lookup` (aggregation)	SQL JOINs
Consistency	Tunable (often eventual)	Strong (ACID)

(b) Perform with code (any two): [2*4=8]

(i) Add data in MongoDB?

- Using `mongosh`:

```
// Insert a single document into 'users' collection
db.users.insertOne({ name: "Alice Wonderland", age: 30, city: "New York"
});
// Insert multiple documents
db.users.insertMany([{ name: "Bob The Builder", age: 35 }, { name:
"Charlie Brown", age: 8 }]);
```

(ii) Delete a Document?

- Using `mongosh`:

```
// Delete the first document matching the filter
db.users.deleteOne({ name: "Alice Wonderland" });
// Delete all documents where age is less than 10
db.users.deleteMany({ age: { $lt: 10 } });
```

(iii) Update a Document?

- Using `mongosh`:

```
// Update the first document matching the filter, set new age
db.users.updateOne({ name: "Bob The Builder" }, { $set: { age: 36,
status: "Active" } });
// Increment age for all users in "New York"
db.users.updateMany({ city: "New York" }, { $inc: { age: 1 } });
```

Q9 (a) Describe the process of sharding in MongoDB. How does MongoDB ensure high availability? [7]

- **Process of Sharding in MongoDB:**

1. **Concept:** Horizontal scaling method to distribute data across multiple servers (shards). Each shard stores a subset of data.
2. **Components:**

- **Shards:** Individual `mongod` instances or replica sets holding data portions.
- **Mongos (Query Routers):** Route client queries to appropriate shards.
- **Config Servers:** Store cluster metadata (mapping of data to shards); deployed as a replica set.

3. **Shard Key:** An indexed field(s) in documents used to partition data.

4. **Chunking & Distribution:** Data divided into "chunks" based on shard key ranges. A balancer process migrates chunks for even distribution.

- **How MongoDB Ensures High Availability:**

1. **Replica Sets (Primary Mechanism):**

- A group of `mongod` instances maintaining the same data (one primary, multiple secondaries).
- **Automatic Failover:** If primary fails, secondaries elect a new primary, minimizing downtime.

2. **Sharding with Replica Sets:**

- **Each shard is typically deployed as a replica set.** This provides HA for each data portion.
- **Config servers are also deployed as a replica set,** ensuring metadata HA.

3. **Redundancy:** Multiple data copies protect against single-server failures.

(b) Explain with code (any two): [2*4=8]
(i) Document in MongoDB

- **Explanation:** Basic unit of data in MongoDB, a BSON (Binary JSON) object. Contains field-value pairs. Schemaless. Has a unique `_id`.
- **Code Example (BSON/JSON-like):**

```
{
  "_id": ObjectId("someGeneratedId"),
  "item": "journal",
  "qty": 25,
  "tags": ["blank", "red"],
  "size": { "h": 14, "w": 21, "uom": "cm" }
}
```

(ii) Collection in MongoDB

- **Explanation:** A grouping of MongoDB documents, analogous to a table in RDBMS but without a fixed schema. Documents within can have different fields.
- **Code Example (`mongosh`):**

```
// use myInventoryDB
// Insert into 'items' collection (creates if not exists)
db.items.insertOne({ product_name: "Pen", stock: 100, category:
"Stationery" });
// db.items.find(); // To view documents in the collection
```

(iii) Databases in MongoDB

- **Explanation:** A container for collections. A single MongoDB server can host multiple databases. Each database has its own files on the server. Switch with `use <dbName>`.
- **Code Example (`mongosh`):**

```
// show dbs // List all databases
// use myBusinessDB // Switch to or create 'myBusinessDB'
// db.clients.insertOne({ name: "Client A", industry: "Tech" }); // Data makes DB persist
// db // Show current database
```
