Here is a comprehensive set of answers to all the **HTML, CSS, and JavaScript Interview Questions** you've listed:

---

1. What is HTML? Differences between HTML & HTML 5? Advantages of HTML 5?

- **HTML (HyperText Markup Language)** is the standard markup language used to create web pages.
- **Differences:**
  - HTML 5 supports multimedia elements (`<audio>`, `<video>`), while older versions require plugins like Flash.
  - HTML 5 introduces semantic tags like `<header>`, `<footer>`, `<article>`, etc.
  - HTML 5 supports SVG, canvas, and vector graphics.
  - HTML 5 has better error handling.
- **Advantages of HTML 5:**
  - Semantic structure improves SEO and accessibility.
  - Multimedia support without plugins.
  - Offline storage using AppCache, localStorage, sessionStorage.
  - Cross-browser compatibility.

---

2. Difference between HTML and XHTML?

Feature	HTML	XHTML
Syntax	Lenient (tags can be lowercase or uppercase)	Strict XML syntax
Parsing	Tolerant	Must be well-formed
File Extension	.html	.xhtml
Case Sensitivity	Not case-sensitive	Tags must be lowercase

---

3. Role of DOCTYPE in HTML?

- The `<!DOCTYPE html>` declaration defines the document type and version of HTML being used.
- It tells the browser how to render the page and triggers standards mode.

4. Difference between Head & Body in HTML? Where to place JS link reference?

- **Head:** Contains metadata, title, styles, scripts that don't affect rendering directly.
- **Body:** Contains content visible to users.
- **JS Reference Placement:**
  - Scripts that manipulate DOM should be placed at the bottom (before `</body>`).
  - Or use `defer` or `async` attributes if placed in `<head>`.

---

5. What is Title Tag in HTML?

- `<title>` tag defines the title of the webpage shown in the browser tab and used by search engines for indexing.

---

6. Different HTML heading tags?

- `<h1>` to `<h6>` — `<h1>` is the most important, `<h6>` the least.

---

7. What are Meta Tags?

- Used to provide metadata about the HTML document (e.g., character set, description, keywords, viewport settings).
- Example:

```
<meta charset="UTF-8">
<meta name="description" content="A sample HTML page">
```

---

8. HTML Elements vs Tags?

- **Element:** A complete structure including opening tag, content, and closing tag (e.g., `<p>Hello</p>`).
- **Tag:** Part of an element used to define its start and end (e.g., `<p>`, `</p>`).

---

9. Roles and uses of `<div>` element?

- `<div>` is a block-level container used for grouping and styling sections of a webpage.

---

10. Difference between `<div>` and `<span>`?

Feature	`<div>`	`<span>`
Type	Block-level	Inline
Purpose	Layout blocks	Inline text styling
Default Width	Full width	Only wraps content

---

11. Use of `<label>` Tag?

- Associates a label with a form control (like input fields).
- Improves accessibility and usability.

---

12. Roles of `<a>`, `<br>`, `<hr>`, `<em>`, `<img>`, `<input>`, & `<button>` elements

- `<a>`: Hyperlink
- `<br>`: Line break
- `<hr>`: Horizontal rule
- `<em>`: Emphasized text (italic)
- `<img>`: Embeds image
- `<input>`: Form input field
- `<button>`: Clickable button

---

13. Roles of `<header>`, `<main>`, `<section>`, `<footer>`, & `<address>`

- `<header>`: Introductory content or navigation
- `<main>`: Main content of the page
- `<section>`: Thematic grouping of content
- `<footer>`: Footer content like copyright info
- `<address>`: Contact information

---

14. Root, Parent, Child, Nested elements

- **Root**: Topmost node (`<html>`)
- **Parent**: Contains other elements
- **Child**: Inside another element
- **Nested**: Elements inside multiple levels of others

---

15. Empty Elements

- Elements without content (no closing tag), e.g., `<br>`, `<img>`, `<hr>`, `<input>`, `<link>`, `<meta>`

---

16. Semantic Elements in HTML

- Provide meaning to the structure of a webpage, e.g., `<article>`, `<nav>`, `<aside>`, `<figure>`, `<figcaption>`, `<section>`, `<header>`, `<footer>`

---

17. Can HTML tags be written in Uppercase?

- Yes, but it's recommended to use lowercase for consistency and compatibility.

---

18. 3 Differences between Block-Level & Inline Elements

Feature	Block-Level	Inline
Line Break	Starts on new line	Does not start on new line
Width	Takes full width	Only wraps content
Nesting	Can contain inline and block elements	Can only contain other inline elements

---

19. List of Block-Level & Inline Elements

- **Block-Level**: `<div>`, `<p>`, `<h1>–<h6>`, `<ul>`, `<ol>`, `<li>`, `<table>`, `<form>`, `<section>`, `<article>`
- **Inline**: `<span>`, `<a>`, `<strong>`, `<em>`, `<b>`, `<i>`, `<input>`, `<label>`, `<img>`, `<br>`

---

20. HTML Attributes

- Modify or provide additional info about an HTML element.
- Syntax: `<tag attribute="value">`
- Examples: `id`, `class`, `src`, `href`, `alt`, `type`

---

21. Id, style, class attributes

- **id**: Unique identifier for an element (used once per page)
- **style**: Applies inline CSS
- **class**: Reusable group of styles (can be applied to multiple elements)

---

22. Same IDs for two elements?

- Invalid HTML; IDs must be unique. May cause unexpected behavior in CSS/JavaScript.

---

23. Data Attributes in HTML

- Custom attributes prefixed with `data-` to store extra data.
- Example: `<div data-role="user"></div>`

---

24. 5 Types of Links in HTML

1. Internal Link
2. External Link
3. Email Link (`mailto:`)
4. Anchor Link (to section on same page)
5. Download Link (`download` attribute)

---

25. Absolute vs Relative URLs

- **Absolute**: Full URL (e.g., `https://example.com/page`)
- **Relative**: Path relative to current page (e.g., `/page`, `images/photo.jpg`)

---

26. Purpose of `<nav>` element

- Defines a section of navigation links.

---

27. Add External Stylesheet

```
<link rel="stylesheet" href="styles.css">
```

---

28. Open Link in New Tab

```
<a href="https://example.com" target="_blank">Link</a>
```

---

29. Create Email Link

```
<a href="mailto:someone@example.com">Email Me</a>
```

---

30. Types of Lists in HTML

- **Ordered List** `<ol>`
- **Unordered List** `<ul>`
- **Definition List** `<dl>`, `<dt>`, `<dd>`

---

31. Nested List

- A list inside another list.
- Example:

```
<ul>
  <li>Item 1
    <ul>
      <li>Subitem 1</li>
    </ul>
  </li>
</ul>
```

---

32. Table Elements

- `<table>`: Creates table
- `<tr>`: Table row
- `<th>`: Header cell
- `<td>`: Standard cell

---

33. `colspan` Attribute

- Merges cells horizontally.

```
<td colspan="2">Merged Cell</td>
```

---

34. Best way to add border to table

Use CSS:

```css
table, th, td {
  border: 1px solid black;
}
```

---

35. What is CSS? 3 ways to Implement CSS

- **CSS (Cascading Style Sheets)**: Controls layout and appearance of HTML documents.

- **Ways:**

    1. Inline Style
    2. Internal Stylesheet (`<style>` in `<head>`)
    3. External Stylesheet (`<link>`)

---

36. Inline Style in CSS

- Applied directly to HTML elements via `style` attribute.
- Use sparingly (for quick fixes or dynamic styling).

---

37. Internal Stylesheet

- Defined within `<style>` tag in `<head>`.
- Useful for single-page styling.

---

38. External Stylesheet

- Linked using `<link>` tag.
- Ideal for multi-page websites (centralized styling).

---

39. CSS Selectors

Used to select HTML elements for styling.

- **Types:**
  - Element selector (`p`)
  - Class selector (`.class`)
  - ID selector (`#id`)
  - Universal selector (`*`)
  - Attribute selector (`[type="text"]`)
  - Pseudo-class (`:hover`)
  - Pseudo-element (`::before`)

---

40. Include CSS in Webpage

Using `<link>` for external:

```
<link rel="stylesheet" href="styles.css">
```

Or `<style>` for internal:

```
<style>
  p { color: red; }
</style>
```

---

41. Box Model in CSS

Every HTML element is a box made up of:

- Content
- Padding
- Border
- Margin

---

42. Padding, Margin, Border

- **Padding:** Space inside the element around content
- **Border:** Line surrounding padding and content
- **Margin:** Space outside the element between borders of adjacent elements

---

43. Data Types in JavaScript

- **Primitive types:**
  - `string`
  - `number`
  - `boolean`
  - `null`
  - `undefined`
  - `symbol` (ES 6)
  - `bigint`

- **Examples:**

```
let str = "Hello"; // string
let num = 100; // number
let bool = true; // boolean
let n = null; // null
let u; // undefined
let sym = Symbol('id'); // symbol
let big = 123456789n; // bigint
```

---

44. Var, let, const

Keyword	Scope	Hoisted	Can Reassign	Can Redeclare
`var`	Function scope	✅	✅	✅
`let`	Block scope	❌	✅	❌
`const`	Block scope	❌	❌	❌

---

45. Hoisting in JavaScript

- Variable declarations are moved to the top of their scope during compilation.
- `var`: Hoisted and initialized as `undefined`
- `let` / `const`: Hoisted but not initialized (TDZ – Temporal Dead Zone)

---

46. Callback Function

- A function passed as an argument to another function and called later.
- Example:

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

greet("John", function() {
  console.log("Callback executed");
});
```

---

47. Arrow Functions

- Shorter syntax introduced in ES 6.
- Do not have their own `this`, arguments, super, or new. Target.
- Example:

```
const add = (a, b) ⇒ a + b;
```

---

48. Callback Hell

- Multiple nested callbacks make code hard to read and maintain.

- Occurs when asynchronous operations depend on each other.

---

49. Solutions to Avoid Callback Hell

- Use Promises
- Use `async/await`
- Modularize functions

Example with Promises:

```
fetchData()
  .then(data ⇒ process(data))
  .catch(error ⇒ console.error(error));
```

---

50. Scope in JavaScript

- **Global Scope:** Accessible anywhere
- **Function Scope:** Declared inside function
- **Block Scope:** Declared inside `{}` (with `let` and `const`)
- `var`: Function scoped
- `let` / `const`: Block scoped

---

51. Asynchronous Operations in JS

- Handled using:
  - Callbacks
  - Promises
  - `async/await`
- **Event Loop:** Manages execution of asynchronous code by checking the call stack and message queue.

---

52. Synchronous vs Asynchronous Code

- **Synchronous:**

  ```
  console.log("Start");
  console.log("Middle");
  console.log("End");
  ```

  Output: Start → Middle → End

- **Asynchronous:**

  ```
  console.log("Start");
  setTimeout(() => console.log("Middle"), 1000);
  console.log("End");
  ```

  Output: Start → End → Middle

Here is a **comprehensive set of answers** for the **WPT Interview Questions Set-2** (covering **JavaScript, DOM, JSON, and Advanced Concepts**). This list builds on your previous one and dives deeper into modern JavaScript and web development.

---

1. What is JavaScript, and what are its primary uses?

- **JavaScript** is a high-level, interpreted programming language used primarily to add interactivity and dynamic behavior to websites.
- **Primary Uses:**
  - Manipulate HTML/CSS (DOM manipulation)
  - Handle events
  - Validate forms
  - Build single-page applications (SPAs)
  - Backend development with Node. Js
  - Mobile app development with React Native, Ionic, etc.

---

2. Differences between `let`, `const`, and `var`. And their scope.

Feature	`var`	`let`	`const`
Scope	Function-scoped	Block-scoped	Block-scoped
Hoisting	✅ (initialized as `undefined`)	✅ (not initialized)	✅ (not initialized)

Feature	`var`	`let`	`const`
Reassignment	✅	✅	❌
Redeclaration	✅ in same scope	❌ in same block	❌ in same block

3. What is hoisting in JavaScript, and how does it work?

- **Hoisting** is JavaScript's default behavior of moving declarations to the top of the current scope (global or function).
- Only **declarations**, not initializations, are hoisted.
- Example:

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

console.log(y); // ReferenceError
let y = 10;
```

4. Explain closures and provide an example of how they are used

- A **closure** gives you access to an outer function's scope from an inner function.
- Closures preserve variables even after the outer function has returned.

Example:

```
function outer() {
  let count = 0;
  return function inner() {
    count++;
    return count;
  }
}
const counter = outer();
console.log(counter()); // 1
console.log(counter()); // 2
```

5. What is the event loop in JavaScript, and how does it manage asynchronous operations?

- The **event loop** manages the execution of code, collects and processes events, and executes queued sub-tasks.
- It ensures that JavaScript remains non-blocking by handling asynchronous operations using:
  - Call Stack
  - Callback Queue
  - Microtask Queue

---

6. Difference between `=` and `==` in JavaScript

Operator	Purpose
`=`	Assignment operator (assigns value)
`==`	Equality operator (compares values with type coercion)
`===`	Strict equality operator (compares value and type)

---

7. How can you check the type of a variable in JavaScript?

Use:

- `typeof` – returns primitive types (`number`, `string`, `boolean`, `undefined`, `object`, `function`)
- `instanceof` – checks if an object is an instance of a constructor
- `Array.isArray()` – specifically checks arrays

Examples:

```
typeof 123; // "number"
typeof {}; // "object"
Array.isArray([1,2]); // true
```

---

8. Use of the `this` keyword in JavaScript

- Refers to the context in which a function is executed.
- Varies depending on how the function is called:
  - In method: refers to the object

    - In function: refers to global object (`window` in browser)
    - With `new`: refers to newly created object
    - With `call`, `apply`, `bind`: explicitly defined

---

9. Difference between function declarations and function expressions

Feature	Function Declaration	Function Expression
Syntax	`function foo() {}`	`const foo = function() {}`
Hoisting	✅ Fully hoisted	❌ Not hoisted
Usage	Can be called before definition	Must be defined before calling

---

10. How does `setTimeout` work in JavaScript?

- Schedules a function to run after a specified delay (in milliseconds).
- Asynchronous, doesn't block the main thread.

Syntax:

```
setTimeout(() => {
  console.log("Executed after 2 seconds");
}, 2000);
```

---

11. What is asynchronous JavaScript, and why is it important?

- Allows non-blocking execution of long-running tasks like API calls, timers, file reading, etc.
- Prevents UI freezing and improves performance and responsiveness.

---

12. What is a callback function, and what is callback hell? How can it be avoided?

- A **callback** is a function passed as an argument and executed later.
- **Callback Hell**: Nested callbacks leading to unreadable and unmanageable code.

Avoiding Callback Hell:

- Use **Promises**
- Use **async/await**
- Modularize functions

---

13. Promises and `.then()` / `.catch()`

- A **Promise** represents the eventual completion (or failure) of an asynchronous operation.

States:

- Pending
- Fulfilled
- Rejected

Example:

```
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

---

14. What is async/await, and how does it simplify working with promises?

- `async` defines an asynchronous function.
- `await` pauses execution until a Promise resolves.

Example:

```
async function getData() {
  try {
    const response = await fetch('https://api.example.com/data');
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error(error);
  }
}
```

15. Higher-order function with example

- A function that either:
  - Accepts another function as an argument
  - Returns a function

Example:

```
function multiplier(factor) {
  return function(number) {
    return number * factor;
  };
}

const double = multiplier(2);
console.log(double(5)); // 10
```

16. Arrow functions vs traditional function expressions

Feature	Traditional Function	Arrow Function
`this` binding	Dynamically bound	Lexical `this`
`arguments` object	Available	Not available
Can be used as constructors	✅	❌
Method shorthand	❌	✅

17. Pure function in JavaScript

- A function that:
  - Always returns the same output for the same input
  - Has no side effects (doesn’t modify external state)

Example:

```
function add(a, b) {
  return a + b;
}
```

---

18. Prototypal inheritance in JavaScript

- Objects inherit properties and methods from other objects via a prototype chain.
- Every object has an internal link to another object called its **prototype**.

---

19. How can you create an object in JavaScript?

- Object literal:

```
let obj = { name: 'John' };
```

- Constructor function:

```
function Person(name) {
  this.name = name;
}
let p = new Person('John');
```

- `Object.create()`:

```
let newObj = Object.create(protoObj);
```

---

20. Purpose of the prototype property in JavaScript

- Used to add properties and methods to all instances of a constructor function.
- Shared among all instances, saving memory.

---

21. Object destructuring with example

- Extracts values from objects into variables.

Example:

```
const user = { name: 'Alice', age: 25 };
const { name, age } = user;
console.log(name); // Alice
```

---

22. Shallow copy vs Deep copy

Type	Description	Example
Shallow Copy	Copies reference	`Object.assign({}, obj)`
Deep Copy	Copies value recursively	Using libraries like Lodash, or recursive functions

---

23. Value types vs Reference types

Type	Examples	Stored By
Value Types	`number`, `string`, `boolean`, `null`, `undefined`	Value
Reference Types	`object`, `array`, `function`	Reference (address)

---

24. Lexical scope and variable access

- Functions have access to variables defined in the scope where they are declared.
- Determined at **compile time**, not runtime.

---

25. Immediately Invoked Function Expression (IIFE)

- A function that runs immediately after being defined.

Example:

```
(function() {
  console.log("IIFE executed");
})();
```

---

26. Call stack in JavaScript

- Manages function calls in a last-in-first-out (LIFO) manner.
- When a function is called, it's added to the stack; when returned, it's removed.

---

27. Function currying with example

- Transforms a function with multiple arguments into a sequence of functions each taking a single argument.

Example:

```
function add(a) {
  return function(b) {
    return a + b;
  }
}
const add5 = add(5);
console.log(add5(3)); // 8
```

---

28. `null` vs `undefined`

Feature	`null`	`undefined`
Assigned	Explicitly set to nothing	Variable declared but not assigned
Type	`object` (bug)	`undefined`
Intention	Empty value	Absence of value

---

29. Document Object Model (DOM)

- A programming interface for HTML documents.
- Represents the page so programs can change structure, style, and content.

---

30. Select elements in DOM using JS

```
document.getElementById('id');
document.getElementsByClassName('class');
document.getElementsByTagName('tag');
document.querySelector('.class');
document.querySelectorAll('div');
```

---

31. Event delegation

- Handling events at a higher level in the DOM instead of attaching them to individual elements.
- Useful for dynamic content and performance.

Example:

```
document.body.addEventListener('click', e ⇒ {
  if (e.target.matches('.btn')) {
    console.log('Button clicked');
  }
});
```

---

32. Create and remove elements in DOM

- Create:

```
const div = document.createElement('div');
div.textContent = 'Hello';
document.body.appendChild(div);
```

- Remove:

```
div.remove();
// or
document.body.removeChild(div);
```

---

33. `addEventListener` vs inline handlers

Feature	`addEventListener`	Inline
Multiple handlers	✅	❌
Separation of concerns	✅	❌
Flexibility	✅	❌

---

34. Prevent default event behavior

```
element.addEventListener('submit', function(e) {
  e.preventDefault();
});
```

---

35. `innerHTML` vs `textContent`

Feature	`innerHTML`	`textContent`
Parses HTML	✅	❌
Security Risk	✅ (XSS)	❌
Performance	Slower	Faster

---

36. JSON and its use in JavaScript

- **JSON (JavaScript Object Notation)** is a lightweight format for storing and transmitting data.
- Used for exchanging data between server and client.

---

37. Parse JSON data

```
const jsonStr = '{"name": "John"}';
const obj = JSON.parse(jsonStr);
```

---

38. Convert object to JSON string

```
const obj = { name: "John" };
const jsonStr = JSON.stringify(obj);
```

---

39. Benefits of JSON over XML

- Lightweight
- Easier to read/write
- Natively supported by JavaScript
- Better integration with APIs

---

40. Handle errors when parsing JSON

```
try {
  const data = JSON.parse(invalidJson);
} catch (error) {
  console.error("Invalid JSON", error);
}
```

---

41. JavaScript modules

- Help organize code into reusable pieces.
- Use `export` and `import`.

Example:

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

// app.js
import { add } from './math.js';
```

---

42. Synchronous vs Asynchronous execution

Type	Behavior
Synchronous	Code runs line-by-line, blocking further execution
Asynchronous	Non-blocking, allows parallel execution (e.g., `setTimeout`, `fetch`)

---

43. `bind`, `call`, `apply`

Method	Description
`call()`	Calls a function with a given `this` and arguments
`apply()`	Same as `call()`, but takes arguments as array
`bind()`	Creates a new function with a fixed `this`

---

44. Closure scope chain

- Each closure has access to:
  - Its own scope
  - Outer function scope
  - Global scope

---

45. Memory management and garbage collection

- JavaScript automatically allocates and deallocates memory.
- **Garbage collector** frees memory that is no longer referenced.

---

46. JavaScript data types

- Primitive:
  - `number`, `string`, `boolean`, `null`, `undefined`, `symbol`, `bigint`
- Non-primitive:
  - `object`, `array`, `function`

---

47. Optimize JavaScript code

- Minify and bundle files
- Defer scripts

- Avoid global variables
- Use debouncing/throttling
- Use efficient algorithms and reduce DOM access

---

48. Service workers

- Background scripts that enable features like offline support, push notifications, caching.
- Run separately from the main browser thread.

---

49. Fetch API

- Modern way to make network requests.

Example:

```
fetch('https://api.example.com/data')
  .then(res ⇒ res.json())
  .then(data ⇒ console.log(data));
```

---

50. WebSockets vs HTTP requests

Feature	HTTP	WebSockets
Connection	Request-response	Full-duplex
Latency	High	Low
Use Case	REST APIs	Real-time apps (chat, games)

Here is a **comprehensive set of answers** for **CDAC Mumbai – WPT Interview Questions Set-3**, covering **Node. Js, Express. Js, AJAX, ReactJS, and Redux**.

🔷 Node. Js

1. What is Node. Js, and how does it differ from traditional server-side technologies?

- **Node. Js** is a JavaScript runtime built on Chrome's V 8 engine that allows developers to run JavaScript on the server.
- **Differences:**
  - Built on **JavaScript**, allowing full-stack JS development.
  - Uses an **event-driven, non-blocking I/O model** (asynchronous).
  - Lightweight and efficient for real-time applications.
  - Traditional servers like PHP use multi-threaded blocking I/O.

---

2. Explain the event-driven, non-blocking I/O model in Node. Js and why it's beneficial.

- **Event-driven**: Based on events like HTTP requests or file reads.
- **Non-blocking I/O**: Operations don't wait for each other; they are asynchronous.
- **Benefits**:
  - High throughput
  - Scalable for concurrent connections
  - Efficient resource usage

---

3. How does Node. Js handle multiple requests with single-threaded architecture?

- Uses an **event loop** to manage asynchronous operations.
- Requests are processed in a **non-blocking way** using callbacks or promises.
- Heavy tasks are offloaded to system threads via **libuv**.

---

4. Core modules in Node. Js and their uses

Module	Use
`fs`	File system operations (read/write files)
`http`	Create HTTP servers/clients
`path`	Manipulate file paths

Module	Use
`os`	Get OS-related info
`events`	Handle custom events
`util`	Utility functions

---

5. What is npm, and how does it help in managing dependencies?

- **npm (Node Package Manager)** manages packages (libraries) used in Node. Js apps.
- Helps install, update, and version control third-party modules.
- Stores metadata in `package.json`.

---

6. Create a basic HTTP server in Node. Js

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

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

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

---

7. Purpose of package. Json

- Metadata about the project.
- Contains:
  - Project name & version
  - Dependencies (`dependencies`, `devDependencies`)
  - Scripts (`npm run dev`)
  - Entry point (`main`)
  - Author & license

---

8. Asynchronous operations in Node. Js

- **Callbacks**: Pass function as argument to be called later.
- **Promises**: Cleaner way to handle async logic (`then()`, `catch()`).
- **async/await**: Syntactic sugar over promises.

---

9. Streams in Node. Js

- Used to handle large data (e.g., reading/writing files).
- Types:
  - Readable
  - Writable
  - Duplex
  - Transform

Example:

```
const fs = require('fs');
const readStream = fs.createReadStream('input.txt');
const writeStream = fs.createWriteStream('output.txt');
readStream.pipe(writeStream);
```

---

10. Error handling in Node. Js

- Use `try/catch` with `async/await`.
- Use `.catch()` with promises.
- For global error handling in Express, use middleware.

🔷 Express. Js

11. What is ExpressJS, and why is it commonly used?

- A minimal and flexible Node. Js web application framework.
- Simplifies routing, middleware, and request/response handling.

---

12. Basic ExpressJS server example

```
const express = require('express');
const app = express();

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

app.listen(3000, () => {
  console.log('Express server running on port 3000');
});
```

---

13. Middleware in ExpressJS

- Functions that have access to the request and response objects.
- Perform actions before sending a response.
- Applied using `app.use()` or route-specific handlers.

---

14. Define routes in ExpressJS

```
app.get('/users', (req, res) => {
  res.send('Get all users');
});
```

---

15. Difference between app.Get (), app.Post (), etc.

Method	Purpose
`get`	Retrieve data
`post`	Submit data
`put`	Update entire resource
`patch`	Partially update resource
`delete`	Delete resource

---

16. Error handling in ExpressJS

Use middleware:

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

---

17. Static file server in Express

```
app.use(express.static('public'));
```

Serves static assets from `public` folder.

---

18. `req.params` vs `req.query`

Type	Example URL	Description
`params`	`/user/:id`	Route parameters
`query`	`/search?q=react`	Query string parameters

---

19. Environment variables in Express

Use `process.env.VAR_NAME`. Load with `.env` file using `dotenv`.

---

20. Secure an Express app – best practices

- Use HTTPS
- Sanitize user input
- Use Helmet middleware
- Rate limiting
- Use JWT for authentication
- Validate inputs with Joi or express-validator

🔷 AJAX

21. What is AJAX, and how is it different from traditional HTTP requests?

- **AJAX (Asynchronous JavaScript And XML)** allows partial page updates without reloading the whole page.
- Traditional requests reload the page on every request.

22. Steps in making an AJAX request

1. Create `XMLHttpRequest` object or use `fetch()`.
2. Open connection.
3. Send request.
4. Handle response asynchronously.

23. Role of XMLHttpRequest

- API for transferring data between client and server asynchronously.

24. AJAX request with jQuery

```
$.ajax({
  url: '/api/data',
  method: 'GET',
  success: function(data) {
    console.log(data);
  },
  error: function(err) {
    console.error(err);
  }
});
```

25. States of XMLHttpRequest

State	Meaning
0	UNSENT
1	OPENED
2	HEADERS_RECEIVED
3	LOADING
4	DONE

---

26. JSON in AJAX

- Lightweight data format.
- Easier to parse than XML.
- Native support in JS → preferred for APIs.

---

27. Fetch () in JavaScript

Modern replacement for `XMLHttpRequest`.

```
fetch('/api/data')
  .then(res ⇒ res.json())
  .then(data ⇒ console.log(data));
```

---

28. Handle errors with fetch

```
fetch(url)
  .then(res ⇒ {
    if (!res.ok) throw new Error("Network response was not ok");
    return res.json();
  })
  .catch(error ⇒ console.error("Error:", error));
```

---

29. CORS (Cross-Origin Resource Sharing)

- Browser mechanism that allows or blocks cross-origin HTTP requests.
- Prevents malicious scripts from accessing resources from another domain.

---

30. Synchronous vs Asynchronous AJAX calls

Type	Behavior
Sync	Blocks code until response received
Async	Continues execution while waiting for response

🔹 ReactJS

31. What is ReactJS and what problem does it solve?

- **React** is a front-end library by Facebook for building UIs.
- Solves issues with complex UI state management and performance.

---

32. JSX vs HTML

- **JSX**: JavaScript XML syntax extension.
- Allows writing HTML-like code inside JavaScript.

---

33. React lifecycle methods

- Mounting: `constructor()`, `render()`, `componentDidMount()`
- Updating: `shouldComponentUpdate()`, `render()`, `componentDidUpdate()`
- Unmounting: `componentWillUnmount()`

---

34. Props in React

- Input passed to components.
- Immutable within the component.

---

35. State in React

- Internal data managed by the component.
- Can change over time and triggers re-render.

---

36. Functional vs Class Components

Feature	Functional	Class
Syntax	Simple	Complex
Lifecycle Methods	❌ (use Hooks)	✅
State	✅ (useState Hook)	✅

---

37. Virtual DOM

- In-memory copy of real DOM.
- Improves performance by minimizing direct DOM manipulation.

---

38. React Hooks

Introduced in React 16.8 to use state and lifecycle features in functional components.

Examples:

- `useState`
- `useEffect`
- `useContext`
- `useReducer`
- `useRef`
- `useMemo`
- `useCallback`

---

39. UseState Hook

Used to add state to functional components.

```
const [count, setCount] = useState(0);
```

40. UseEffect Hook

Performs side effects like data fetching or subscriptions.

```
useEffect(() ⇒ {
  document.title = `You clicked ${count} times`;
}, [count]);
```

41. Controlled Components

- Form elements whose value is controlled by React state.
- Ensures one source of truth.

42. New Features in React 18

- Automatic batching
- New Suspense SSR
- Streaming server rendering
- Selective hydration
- UseId, useSyncExternalStore, useTransition, useDeferredValue

43. Passing Data Between Components

- Parent → Child: props
- Child → Parent: callback props
- Global: Context API, Redux, Zustand

44. React Forms Handling

- Use controlled components and `onChange` handler.
- Track form state using `useState`.

45. UseContext Hook

Access context values anywhere in the component tree without prop drilling.

---

46. UseReducer Hook

For complex state logic with nested values or when one depends on others.

```
const [state, dispatch] = useReducer((state, action) => {
  switch(action.type) {
    case 'increment':
      return { count: state.count + 1 };
    default:
      return state;
  }
}, { count: 0 });
```

---

47. React Router

Enables navigation between views in SPAs.

```
<Route path="/about" element={<About />} />
<BrowserRouter>
```

---

48. Conditional Rendering

Using ternary operators or &&.

```
{isLoggedIn ? <Dashboard /> : <Login />}
```

---

49. Prop Drilling

Passing props through many layers unnecessarily.

Minimize using:

- Context API
- Redux
- Zustand

---

50. React.Memo ()

Prevents unnecessary re-renders of functional components.

```
const MyComponent = React.memo(({ data }) => (
  <div>{data}</div>
));
```

---

51. React. ForwardRef

Pass ref to child component.

```
const FancyButton = React.forwardRef((props, ref) => (
  <button ref={ref}>{props.children}</button>
));
```

---

52. React. Lazy and Suspense

Support lazy loading and code-splitting.

```
const LazyComponent = React.lazy(() => import('./LazyComponent'));

<Suspense fallback="Loading...">
  <LazyComponent />
</Suspense>
```

---

53. Optimize React App Performance

- Use `React.memo()`
- Avoid unnecessary renders
- Code splitting
- Memoize callbacks with `useCallback`
- Use virtualization for long lists

---

54. Higher-Order Component (HOC)

Takes a component and returns a new component.

```
function withLogger(WrappedComponent) {
  return function(props) {
    console.log("Rendering HOC");
    return <WrappedComponent {...props} />;
  }
}
```

---

55. Error Boundaries

Catch JavaScript errors anywhere in component tree.

```
class ErrorBoundary extends React.Component {
  constructor() {
    super();
    this.state = { hasError: false };
  }

  componentDidCatch(error, info) {
    this.setState({ hasError: true });
  }

  render() {
    if (this.state.hasError) return <h1>Something went wrong.</h1>;
    return this.props.children;
  }
}
```

---

56. Lazy Loading in React

Using `React.lazy()` and `Suspense`.

---

57. React Context API

Provides global state across component tree.

```
const ThemeContext = React.createContext('light');
```

---

58. Redux

State management tool for predictable state container.

Used for large-scale apps with complex shared states.

---

59. Actions, Reducers, Store in Redux

- **Action**: Object describing what happened.
- **Reducer**: Function that returns new state.
- **Store**: Holds the state.

---

60. UseSelector Hook

Reads data from Redux store.

```
const count = useSelector(state ⇒ state.counter.value);
```

---

61. Middleware in Redux

Extends Redux functionality (e.g., async actions).

Example: `redux-thunk`, `redux-saga`

---

62. UseDispatch vs useSelector

Hook	Purpose
`useDispatch`	Dispatches actions
`useSelector`	Gets state from store

---

63. Connect () function in Redux

Connects React components to Redux store.

```
connect(mapStateToProps, mapDispatchToProps)(MyComponent);
```

---

64. Async operations in Redux

Use middleware like `redux-thunk` or `redux-saga`.

---

65. Combine reducers in Redux

```
import { combineReducers } from 'redux';
const rootReducer = combineReducers({ user, posts });
```

---

66. Redux DevTools

Tool for debugging Redux state changes.

Setup:

```
npm install --save redux-devtools-extension
```

```
import { composeWithDevTools } from 'redux-devtools-extension';

const store = createStore(rootReducer, composeWithDevTools(applyMiddleware(thunk)));
```
