MERN

q1) What are pure components in React? → Pure components are components that only re-render when their props or state change. They perform a shallow comparison of props and state to determine if re-rendering is necessary

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

q2) H**ow do you create class components in React?**

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

q3) How do you create functional components in React?

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

q4) diff b/w props and states

Props are immutable , passed to components similar to fuctional parameters. inputs to a react component

State→ It is a built-in object that holds property values that belong to the component. When the state object changes, the component re-renders, muttable

```
class Clock extends React.Component {
constructor(props) {
```

```
super(props);
this.state = { date: new Date() };
}
render() {
return <h1>It is {this.state.date.toLocaleTimeString()}.</h1>;
}
}
```

q5) . **How do you create components in React?**

Components in React can be created using functions or classes

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

q6) what is JSX ? Why is it used in React ?

JSX is a syntax extension for JavaScript that looks similar to XML or HTML. . It is used in React to describe what the UI should look like .

const element = <h1>Hello, world!</h1>;

q7) what is reactjs→ it is a JavaScript library for building user interfaces, primarily developed and maintained by Facebook. It allows developers to create large web applications that can update
and render efficiently in response to data changes .

q8) key features of react js ?

1)C**omponent-Based:** Build encapsulated components that manage their own state.

2)**Declarative:** Design simple views for each state in your application .

3)Virtual DOM: React creates a virtual DOM to improve performance.

4)u**nidirectional Data Flow:** Data flows in one direction, making it easier to debug.

q9) W**hat are the differences between React and Angular?**

Library vs Framework → React is library while angular is a full-fledged framework.

Data Binding: React uses one-way data binding, Angular uses two-way data binding.

DOM: React uses a virtual DOM, Angular manipulates the real DOM directly

Learning Curve: React has a simpler learning curve compared to Angular.

q10) **How does React achieve better performance with the virtual DOM?**

React achieves better performance by using the virtual DOM to batch updates and minimizing direct manipulations to the real DOM.

q11) **What are higher-order components (HOC) in React?**

HOCs are functions that take a component and return a new component with added functionality.

```
function withGreeting(WrappedComponent) {
return class extends React.Component {
render() {
return <WrappedComponent greeting="Hello" {...this.props} />;
}
};
}
```

16. **How do you handle props and state changes in React components?**

Props: Handled by the parent component and passed down to the child component.

State: Managed within the component, changes are handled with setState or useState hook.

```
/ Example: Handling state changes
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>Count: {this.state.count}</p>
<button onClick={this.increment}>Increment</button>
</div>
);
}
}
```

17. **Explain the lifecycle methods of React components.**

Lifecycle methods allow you to hook into certain points of a component's life.

Mounting: constructor(), componentDidMount() → The component is being **created** and added to the page (DOM).

Updating: shouldComponentUpdate() , componentDidUpdate() → Happens when the component **re-renders** due to `props` or `state` changes

Unmounting: componentWillUnmount() → The component is **being removed** from the page (DOM).

```
class MyComponent extends React.Component {
componentDidMount() {
console.log('Component mounted');
}
componentDidUpdate() {
```

```
console.log('Component updated');
}
componentWillUnmount() {
console.log('Component will unmount');
}
render() {
return <div>Hello</div>;
}
}
```

18. **What is the significance of shouldComponentUpdate method?**

It allows you to optimize performance by preventing unnecessary re-renders. It returns true by default, but you can override it to return false if the component does not need to update .

```
class MyComponent extends React.Component {
shouldComponentUpdate(nextProps, nextState) {
return nextProps.someValue !== this.props.someValue;
}
render() {
return
<div>{this.props.someValue}</div>;
}
}
```

19. **How do you implement default props in a React component?**

```
class Greeting extends React.Component {
static defaultProps = {
name: 'Guest'
};
render() {
return <h1>Hello, {this.props.name}</h1>;
```

```
}
}
```

20. **What are controlled and uncontrolled components in React?**

Form inputs whose values are controlled by React state. → controlled

Form inputs that maintain their own state. → uncontrolled

21.What are the different ways to manage state in React?

Local state: Managed within a component using useState or setState.

Global state: Managed across multiple components using Context API or libraries like Redux.

Derived state: State derived from props or other state values.

URL state: Managed through React Router for route parameters and query strings.

21. **Explain the useState hook in React.**

useState is a hook that allows you to add state to functional components. It returns an array with two values: the current state and a function to update it .

22. **What is the useEffect hook used for in React?**

useEffect is a hook for performing side effects like API calls, timers, etc. in functional components. It runs after the component renders.

```
function Timer() {
const [count, setCount] = useState(0);
useEffect(() => {
const timer = setInterval(() => {
setCount(prevCount => prevCount + 1);
}, 1000);
return () => clearInterval(timer);
}, []);
return <div>Timer: {count}</div>;
}
```

23.How do you manage global state in React applications?

Use Context API or state management libraries like Redux or MobX

24. **What are the differences between Redux and Context API for state management?**

Redux: Centralized store, more boilerplate, powerful debugging tools.

Context API: Simpler, less boilerplate, no middleware, better for smaller applications.

25. **Explain the Redux architecture and its core principles.**

Single Source of Truth: The state is stored in a single, centralized store.

State is Read-Only: The state can only be changed by emitting actions.

Changes are Made with Pure Functions: Reducers specify how the state changes in response to actions.

26. **How do you connect Redux with a React application?**

Use the Provider component from react-redux to wrap your app and the connect function or useSelector and useDispatch hooks to connect components to the Redux store.

27. **What is an action creator in Redux?**

An action creator is a function that returns an action object.

```
// Example: Action creator
function increment() {
return { type: 'INCREMENT' };
}
```

28. E**xplain reducers in Redux ?**

Reducers are pure functions that take the current state and an action as arguments and return a new state.

29. **How does Redux store work?**

The Redux store holds the state of the application, allows access to state via getState(), dispatches actions via dispatch(action), and registers listeners via subscribe(listener).

30. **What are hooks in React? Why were they introduced?**

Hooks are functions that let you use state and other React features in functional components. They were introduced to provide a way to use state and lifecycle methods without writing class components.

31. **How do you create custom hooks in React ?**

Custom hooks are functions that use other hooks to encapsulate and reuse stateful logic.

32. **How do you optimize performance with hooks?**

Memoization is a performance optimization technique to store theresults of expensive function calls and return the cached result when the same inputs occur again.

Use useMemo to memoize expensive calculations.

Use useCallback to memoize event handlers.

Use React.memo to prevent unnecessary re-renders of functional components.

33. **How to implement routing in a react application?**

Use react-router-dom for declarative routing. React Router is a standard library for routing in React.

- It enables the navigation among views of various components in a React application, allows changing the browser URL, and keeps the UI in sync with the URL.

34. **Explain the role of react-router in React applications ?**

react-router provides a declarative way to handle routing in React applications, allowing for dynamic routing, nested routes, and route parameters.

35. **How do you handle dynamic routing in React?**

Use route parameters and useParams hook from react-router-dom.

37. **What are nested routes in React-router ?**

Nested routes allow you to render child routes within parent routes.

38. **What are the performance best practices for React applications?**

Avoid inline functions in render,Use production build, minimize re-renders, code splitting

39. **What is Context API in React?**

Context API provides a way to pass data through the component tree without having to pass props down manually at every level.

It's used for sharing state, themes, or other common properties across components.

40. **When would you use Context API over props drilling?**

Use Context API when you need to share state or data across multiple levels of the component tree, making the code cleaner and avoiding the "props drilling" problem.

Create a context using → React.createContext. Provide the context value using → Context.Provider.

Consume the context value using → useContext or Context.Consumer.

Example: Sharing a theme or user authentication.

41. **How do you test React components?**

Use libraries like Jest for testing JavaScript and React Testing Library for testing React components.

Write unit tests for individual components and integration tests for component interactions.

42.What are the popular testing libraries for React?

Jest: A JavaScript testing framework used for unit tests, snapshot tests, and more.

React Testing Library: A library for testing React components by querying the DOM.

Enzyme: A JavaScript testing utility for React, useful for shallow rendering and DOM manipulation

43.What is the useReducer hook and when would you use it?

The useReducer hook is an alternative to useState for managing complex state logic . Use it when you have state logic that involves multiple sub-values or when

the next state depends on the previous one.

44.What is the purpose of the Switch component in React Router ?

The `Switch` component is used to group `Route` components and ensures that only one
route is rendered at a time.

45. **What are some security/prevent XSS attacks best practices in React applications?**

Sanitize Inputs: Always sanitize and validate user inputs to prevent injection attacks. , **Avoid Dangerous APIs, Use HTTPS , Content Security Policy (CSP) , Security Headers, Environment Variables:** Store sensitive data (e.g., API keys) in environment variables, not in the source code.

46. **What are portals in React?**

Portals provide a way to render children into a DOM node that exists outside the DOM hierarchy of the parent component.

Useful for rendering modals, tooltips, or any content that needs to visually break out of its container.

47. **How do you handle state management without Redux?**

Context API and useReducer: Use React's Context API combined with the useReducer hook to manage global state.

Hooks: Use custom hooks to encapsulate and manage state logic.

Third-party Libraries: Use other state management libraries like MobX, Recoil or Zustand.

48. **What are lazy loading and code splitting in React?**

Lazy Loading: Technique to defer loading of non-critical resources at the initial load time. Improves performance by loading components only when they are needed.

Code Splitting: Breaking up the codebase into smaller chunks that canbe loaded on demand. Usually implemented using React.lazy and Suspense.

49. **What is Redux middleware? Give examples.** Redux middleware provides a way to interact with dispatched actions before they reach the reducer.

Examples: redux-thunk, redux-saga, and redux-logger.

50. **What is Server-side Rendering (SSR) in React?**

SSR is the process of rendering React components on the server and sending the HTML to the client.

It improves performance and SEO by delivering fully rendered pages on the first request.

51. **How do you implement SSR in a React application?**

Use a framework like Next.js that provides built-in support for SSR.

Alternatively, set up your own SSR using Node.js and libraries like Express and ReactDOMServer.

52. **What is Node.js?**

Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine that allows developers to run JavaScript on the server side. It is designed to build scalable network applications.

53. **How does Node.js differ from traditional web servers like Apache?**

Node.js uses an event-driven, non-blocking I/O model which makes it lightweight and efficient.

Unlike Apache, which creates a new thread for each request, Node.js handles multiple requests on a single thread using asynchronous callbacks.

54. **Explain the characteristics of Node.js that make it**

suitable for building scalable applications. Non-blocking I/O: Handles many connections

concurrently. Event-driven architecture: Efficiently handles asynchronous operations.

Single-threaded: Uses fewer resources and is easier to manage

55. **What is npm? How do you use it in Node.js projects?**

npm (Node Package Manager) is a package manager for JavaScript, included with Node.js. It helps install, update, and manage packages

/ Initialize a new Node.js project
npm init -y
// Install a package (e.g., Express)
npm install express

56. **What is event-driven programming in Node.js?**

Event-driven programming in Node.js means that the flow of the program is determined by events such as user actions, sensor outputs, or messages from other programs.

57. **How do you include external libraries in Node.js?**

External libraries are included using the require function.

58. **What is the difference between require and import in Node.js?**

require is used in CommonJS module system, while import is used in ES6 modules. Node.js natively supports CommonJS, but ES6 modules can be used with the .mjs extension or with specific configurations .

59. **How do you create and publish your own npm package?**

Create a package.json file, write your code, and use npm publish to publish the package.

60. **What are the built-in modules in Node.js?**

Some built-in modules include http, fs, path, url, crypto, and events.

61. **Explain the concept of non-blocking I/O in Node.js.**

Non-blocking I/O means that operations like reading from a file or database do not block the execution of other operations. Instead, they are executed asynchronously, allowing other code to run concurrently.

62. **How does Node.js handle asynchronous operations?**

Node.js uses callbacks, Promises, and async/await to handle asynchronous operations.

63. **what are callbacks in Node.js? How do you handle errors with callbacks?**

Callbacks are functions passed as arguments to other functions to be executed after the

completion of an operation.

64. **What are async/await in Node.js? How do they work?**

async/await is syntactic sugar built on top of Promises, making asynchronous code look
and behave more like synchronous code Promises represent the eventual completion
(or failure) of an asynchronous operation and its resulting value .

65. H**ow do you perform file operations in Node.js?**

Using the fs module for synchronous and asynchronous file operations.

66.Explain the difference between synchronous and asynchronous file operations in Node.js.

Synchronous operations block the event loop until the operation completes, while asynchronous operations do not block the event loop and use callbacks, Promises, or async/await for the result.

67. **How do you handle streams in Node.js?**

Streams are used to handle reading and writing of large data efficiently. They are

instances of EventEmitter.

68. **How do you create a simple HTTP server in Node.js?**

Using HTTP module.

69. **What is Express.js? How do you use it in Node.js applications?**

Express.js is a minimal and flexible Node.js web application framework that provides a robust set of features for building web and mobile applications.

70. **Explain middleware in the context of Express.js.**

Middleware functions are functions that have access to the request object (req), the response object (res), and the next middleware function in the application’s request-response cycle.

71. **How do you handle routing in Express.js?**

By defining route handlers for different HTTP methods and URL paths

72. **How do you create RESTful APIs using Express.js?**

By defining routes that correspond to CRUD operations (Create, Read, Update, Delete)

73. **How do you connect Node.js with databases like MongoDB or MySQL?**

Using database drivers or ORMs.(object relational models).

74. **What is Mongoose? How do you use it with MongoDB in Node.js?**

Mongoose is an ODM (Object Data Modeling) library for MongoDB and Node.js. It manages

relationships between data, provides schema validation, and translates between objects in

code and the MongoDB documents.

75. **How do you handle errors in Node.js applications?**

By using try/catch blocks, callbacks with error arguments, Promises, and centralized error handling middleware in Express.

76. **What tools and techniques do you use for debugging Node.js applications?**

Tools: node inspect, Chrome DevTools, Visual Studio Code debugger, and libraries like debug

77. **What are some common security concerns in Node.js applications?**

Injection attacks (SQL, NoSQL, Command), cross-site scripting (XSS), cross-site request

forgery (CSRF), and insecure dependencies.

78. **How do you prevent common security vulnerabilities in Node.js applications?**

Use parameterized queries, sanitize user input, use security headers, validate data, keep dependencies up to date, and use tools like Helmet for setting HTTP headers.

79. **How do you deploy Node.js applications? What considerations are important for deployment?**

Deploy to platforms like Heroku, AWS, orDigitalOcean. Use process managers like PM2, and configure environments properly.

```
// Install PM2 globally
npm install pm2 -g
// Start application with PM2
pm2 start app.js
```

30. **What is the event loop in Node.js? How does it work?**

The event loop is a mechanism that allows Node.js to perform non-blocking I/O operations by offloading operations to the system kernel whenever possible. It processes asynchronous callbacks and executes them in a single-threaded event-driven loop.

31. **What are process.nextTick() and setImmediate()? How do they differ?**

process.nextTick() schedules a callback function to be invoked in the next iteration of the event loop, before any I/O operations. setImmediate() schedules a callback to be executed after I/O events callbacks and before timers.

82. **Explain the concept of clustering in Node.js. How does it improve performance?**

Clustering allows Node.js to create multiple processes that share the same server port. This is useful for taking advantage of multi-core systems to handle more concurrent connections.

33. **What are worker threads in Node.js? How do they differ from clustering?**

Worker threads allow the execution of JavaScript code in parallel using multiple threads. Unlike clustering, worker threads can share memory and are suitable for CPU-intensive tasks.

34. **How do you write tests for Node.js applications? What frameworks do you use?**

Use testing frameworks like Mocha, Jest, and Chai to write and run tests.

35. **How do you handle asynchronous testing in Node.js?**

Use done callback, Promises, or async/await to handle asynchronous code in tests.

36. **What is the role of environment variables in Node.js applications?**

Environment variables are used to store configuration and secrets, allowing applications to be configured differently in various environments (development, testing, production).

37. **How do you handle configuration management in Node.js?**

Use libraries like dotenv for loading environment variables from a .env file, and config for managing configuration settings.

38. **What are some best practices for structuring Node.js applications?**

```
yapp/
├── controllers/
│   └── userController.js
├── models/
│   └── userModel.js
├── routes/
│   └── userRoutes.js
├── services/
│   └── userService.js
├── app.js
└── package.json
```

39. **How do you handle static files in Express.js?**

Use the express.static middleware to serve static files.

90. **How do you implement authentication and authorization in Node.js applications?**

Use libraries like Passport.js for authentication and implement role-based access control for authorization.

91. **Explain the differences between SQL and NoSQL databases.**

SQL databases are relational, use structured query language (SQL) for defining and manipulating data, and have a predefined schema. Examples include MySQL, PostgreSQL, and Oracle.

NoSQL databases are non-relational, can store unstructured data, and have dynamic schemas. They are designed for distributed data stores with large-scale data storage needs. Examples include MongoDB, Cassandra, and Redis.

db.users.find({ _id: 1 }); || SELECT * FROM users WHERE id = 1

92.W**hat are the advantages and disadvantages of using a NoSQL database like MongoDB?**

Advantages: → Flexible schema, Scalability , Performance: Can handle large volumes of unstructured data.

Disadvantages: Consistency: May not be strongly consistent (depends on the implementation).

Complex queries: Limited support for complex queries and joins compared to SQL.

Data redundancy: Denormalization may lead to data redundancy.

93. How do you model one-to-many relationships in MongoDB?

Use embedded documents or references. Embedded documents are suitable for small, bounded data, while references are better for large or frequently updated data.

```
//embedded documents:
const blogPost = {
title: "Post Title",
content: "Post content...",
comments: [
{ user: "Alice", text: "Great post!" },
{ user: "Bob", text: "Thanks for sharing!" }
]
};
db.posts.insert(blogPost);
//References:
const post = { title: "Post Title", content: "Post content..." };
db.posts.insert(post);
const comment = { postId: post._id, user: "Alice", text: "Great post!" };
db.comments.insert(comment);
```

94. **How do you model many-to-many relationships in MongoDB?**

Use an array of references to represent many-to-many relationships

```
onst student = { name: "John Doe", courses: [course1_id, course2_id] };
const course = { title: "Math 101", students: [student1_id, student2_id] };
db.students.insert(student);
db.courses.insert(course);
```

95. **What is the purpose of indexing in MongoDB?**

Indexing improves the performance of queries by allowing the database to quickly locate documents that match the query criteria.

```
db.users.createIndex({ name: 1 });
```

96. **How do you analyze and optimize query performance in MongoDB?**

Use the explain() method to analyze query execution and identify performance bottlenecks.

```
db.users.find({ age: { $gt: 25 }
}).explain("executionStats");
```

97.What are MongoDB transactions? How do you use them?

MongoDB transactions provide ACID (Atomicity, Consistency, Isolation, Durability) guarantees, allowing multiple operations to be executed in an all-or-nothing manner.

> MongoDB 4.0+ supports multi-document ACID transactions.

```
const session = client.startSession();
session.startTransaction();
try {
// multiple ops here
session.commitTransaction();
} catch (e) {
session.abortTransaction();
}
```

98. **What is replication in MongoDB? How do you set up a replica set?**

Replication is the process of synchronizing data across multiple servers to ensure high availability and data redundancy.

99. **What is sharding in MongoDB? How do you set up sharding?**

Sharding is the process of distributing data across multiple servers to handle large datasets and high throughput operations.

00. **What are some best practices for securing MongoDB installations?**

Enable authentication, use strong passwords, enforce role-based access control, enable SSL/TLS for encrypted connections, and regularly update MongoDB to the latest version.

01. **What is a document and a collection in MongoDB?**

> A document is a key-value pair structure (like a JSON object). A collection is a group of documents, like a table in SQL.

02. **What is MongoDB?**

> MongoDB is a NoSQL, document-oriented database that stores data in flexible, JSON-like documents (BSON format), making it schema-less and scalable.

03. **Update vs Replace in MongoDB?**

- `updateOne()` or `updateMany()` updates **specific fields**.
- `replaceOne()` **replaces the entire document**.

04. **What is aggregation in MongoDB?**

> Like GROUP BY in SQL. Used for data processing and transformation.

```
db.orders.aggregate([
{ $match: { status: "delivered" } },
{ $group: { _id: "$customerId", total: { $sum: "$amount" } } }
```

```
])
$lookup – for performing joins.
$project – reshaping the document.
$facet, $bucket, $unwind – for reporting, dashboards, and analytics.
```

105. **Difference:** `find()` **vs** `findOne()`

- `find()` returns a **cursor** (multiple docs).
- `findOne()` returns the **first matched doc**.

106 **CAP Theorem: How MongoDB fits?**

> MongoDB is CP (Consistency + Partition Tolerance) by default. But can be tuned for Availability with replica sets.

107 **How do you handle large documents (16MB limit)?**

> Use GridFS to split and store large files (images, videos) across multiple documents.

108 what is axios and cors?

> Axios is a promise-based HTTP client used to send requests (GET, POST, etc.) to a server and handle responses easily in JavaScript/React.

> CORS (Cross-Origin Resource Sharing) is a security feature that allows or blocks requests between different origins (like frontend & backend on different ports).

🌐 MERN CRUD Overview

Operation	HTTP	Frontend (React)	Backend (Express + Node)	DB (MongoDB)

Create	POST	Form Input	`app.post()`	`insertOne()`
Read	GET	Fetch & Display	`app.get()`	`find()`
Update	PUT	Edit & Submit	`app.put()`	`updateOne()`
Delete	DELETE	Button/Action	`app.delete()`	`deleteOne()`

```
mern-crud/
├── backend/
│   ├── controllers/
│   │   └── userController.js
│   ├── models/
│   │   └── User.js
│   ├── routes/
│   │   └── userRoutes.js
│   ├── server.js
├── frontend/
│   ├── App.js
│   └── index.js
```

1️⃣ server.js

```
const express = require('express');
const mongoose = require('mongoose');
const cors = require('cors');
const userRoutes = require('./routes/userRoutes');

const app = express();
app.use(cors());
app.use(express.json());

// MongoDB Connection
mongoose.connect('mongodb://127.0.0.1:27017/mern-crud')
  .then(() => console.log('MongoDB connected'))
  .catch((err) => console.error(err));

// Routes
```

```
app.use('/api/users', userRoutes);
app.listen(5000, () ⇒ console.log('Server running on http://localhost:5000'));
```

2️⃣ models/User.js

```
const mongoose = require('mongoose');
const userSchema = new mongoose.Schema({
  name: String,
  email: String
});
module.exports = mongoose.model('User', userSchema);
```

3️⃣ controllers/userController.js

```
const User = require('../models/User');

// @desc    Get all users
exports.getUsers = async (req, res) ⇒ {
  const users = await User.find();
  res.json(users);
};

// @desc    Create a user
exports.createUser = async (req, res) ⇒ {
  const user = new User(req.body);
  await user.save();
  res.json(user);
};

// @desc    Update a user
exports.updateUser = async (req, res) ⇒ {
  const updatedUser = await User.findByIdAndUpdate(req.params.id, req.bod
y, { new: true });
  res.json(updatedUser);
```

```
};

// @desc    Delete a user
exports.deleteUser = async (req, res) => {
  await User.findByIdAndDelete(req.params.id);
  res.json({ message: 'User deleted' });
};
```

4 routes/userRoutes.js

```
const express = require('express');
const router = express.Router();
const {
  getUsers,
  createUser,
  updateUser,
  deleteUser
} = require('../controllers/userController');

router.get('/', getUsers);
router.post('/', createUser);
router.put('/:id', updateUser);
router.delete('/:id', deleteUser);

module.exports = router;
```

5 App.js (Frontend)

```
import React, { useEffect, useState } from 'react';
import axios from 'axios';
function App() {
  const [users, setUsers] = useState([]);
  const [name, setName] = useState('');
  const [email, setEmail] = useState('');
  const [editId, setEditId] = useState(null); // null means no edit
```

```
  // Get users
  const fetchUsers = async () => {
    const res = await axios.get('http://localhost:5000/api/users');
    setUsers(res.data);
  };
  useEffect(() => {
    fetchUsers();
  }, []);
  // Add or Update user
  const handleSubmit = async () => {
    if (editId) {
      await axios.put(`http://localhost:5000/api/users/${editId}`, { name, email
});
      setEditId(null);
    } else {
      await axios.post('http://localhost:5000/api/users', { name, email });
    }
    setName('');
    setEmail('');
    fetchUsers();
  };
  // Edit user
  const handleEdit = (user) => {
    setName(user.name);
    setEmail(user.email);
    setEditId(user._id);
  };
  // Delete user
  const handleDelete = async (id) => {
    await axios.delete(`http://localhost:5000/api/users/${id}`);
    fetchUsers();
  };
  return (
    <div style={{ padding: 20 }}>
      <h2>MERN CRUD (Beginner)</h2>
      <inputplaceholder="Name"
```

```
        value={name}
        onChange={(e) => setName(e.target.value)}
      />
      <inputplaceholder="Email"
        value={email}
        onChange={(e) => setEmail(e.target.value)}
      />
      <button onClick={handleSubmit}>
        {editId ? 'Update' : 'Add'}
      </button>
      <hr />
      {users.map(user => (
        <div key={user._id}>
          <p>{user.name} - {user.email}</p>
          <button onClick={() => handleEdit(user)}>Edit</button>
          <button onClick={() => handleDelete(user._id)}>Delete</button>
        </div>
      ))}
    </div>
  );

export default App;
```

HTML (25 Questions)

1. **HTML Definition and Purpose:** Standard markup language for creating web pages using tags to structure content.
2. **HTML Tags and Attributes:** Tags mark up elements, attributes provide additional info (e.g., `<a href="url">`).
3. **Block-Level vs Inline Elements:** Block elements start new line, take full width. Inline elements don't start new line.
4. **Semantic HTML Elements:** Elements with meaning (e.g., `<header>`, `<nav>`) not just presentation.

5. **Document Object Model (DOM):** Tree structure of HTML document that JavaScript can manipulate.
6. **HTML5 vs HTML4 Features:** HTML5 added semantic elements, multimedia support, canvas, better forms.
7. **Creating Hyperlinks (`<a>` tag):** `<a href="url">` creates links to other pages, files, or locations.
8. **HTML Forms and Form Elements:** `<form>` with `<input>`, `<textarea>`, `<button>` for user data input.
9. **Embedding Multimedia Elements:** `<audio>`, `<video>`, `<embed>` for media content.
10. **Data Attributes (`data-*` attributes):** `data-*` attributes store custom data accessible via JavaScript.
11. **Creating Tables (`<table>`, `<tr>`, `<td>`, `<th>`):** for tabular data.
12. `<meta>` **Tag and Its Purpose:** Provide metadata like charset, viewport, descriptions for SEO.(search engine optimization)
13. `<div>` **vs** `<span>`**:** `<div>` is block-level, `<span>` is inline.
14. `<iframe>` **Tag Usage:** Embeds another HTML page within current page.
15. **Including External Resources (CSS, JS):** `<link>` for CSS, `<script>` for JavaScript.
16. **alt Attribute in Images:** Alternative text for images (accessibility).
17. **Creating Lists (`<ul>`, `<ol>`, `<dl>`):** for unordered, ordered, definition lists.
18. **New HTML5 Form Elements:** `<datalist>`, `<keygen>`, `<output>` for advanced forms.
19. `<canvas>` **Element for Graphics:** HTML5 element for drawing graphics with JavaScript.
20. `<script>`**,** `<noscript>`**, and** `<template>` **Tags:** `<script>` for JavaScript, `<noscript>` for fallback, `<template>` for reusable content.
21. **Global Attributes (`id`, `class`, `style`, `title`):** `id`, `class`, `style`, `title` can be used on any element.
22. `<section>` **vs** `<div>`**:** `<section>` has semantic meaning, `<div>` is generic container.

23. **Responsive Layouts with HTML (meta viewport):** Use meta viewport tag for mobile-friendly design.
24. `<!DOCTYPE html>` **Declaration:** `<!DOCTYPE html>` specifies HTML5 and triggers standards mode.
25. **Handling Broken Images (`onerror` attribute):** Use `onerror` attribute to handle image loading failures.

CSS (25 Questions)

1. **CSS Definition and Purpose:** Language for styling web pages (colors, layout, fonts).
2. **CSS Selectors:** Patterns to select elements (element, class, ID selectors).
3. **CSS Specificity and Inheritance:** Determines which rules apply; child elements inherit parent styles.
4. **CSS Box Model:** Content, padding, border, margin around elements.
5. **Differences Between Classes and IDs:** Classes reusable, IDs unique. IDs have higher specificity.
6. **Applying CSS (Inline, Internal, External):** Inline (style attribute), internal (`<style>`), external (separate file).
7. **CSS Positioning (Static, Relative, Absolute, Fixed, Sticky):** Static, relative, absolute, fixed, sticky positioning.
8. **CSS Flexbox Layout:** One-dimensional layout system for responsive design.
9. **CSS Grid Layout:** Two-dimensional layout system for complex layouts.
10. **CSS Display Property:** Controls how element is displayed (block, inline, flex, grid).
11. **CSS Pseudo-classes and Pseudo-elements:** Style elements based on state or position (`:hover`, `::first-line`).
12. **CSS Transitions and Animations:** Smooth property changes and keyframe animations.

13. **CSS Media Queries for Responsive Design:** Apply styles based on device characteristics (screen size).
14. **Differences Between `display: none` and `visibility: hidden`:** None removes from layout, hidden preserves space.
15. **CSS `float` Property and Clearfix:** Float positions elements left/right; clearfix clears floats.
16. **CSS Z-index and Stacking Context:** Controls stacking order of positioned elements.
17. **CSS Units (px, em, rem, %, vw, vh):** px (pixels), em/rem (relative), % (percentage), vw/vh (viewport).
18. **CSS Variables (Custom Properties):** Custom properties for reusable values (`--variable-name`).
19. **CSS `box-sizing` Property:** Controls width/height calculation (content-box vs border-box).
20. **CSS Combinators (Descendant, Child, Adjacent Sibling, General Sibling):** Select elements by relationship.
21. **CSS Sprites:** Combine multiple images into one to reduce HTTP requests.
22. **Centering Elements Horizontally and Vertically:** Use flexbox, grid, or absolute positioning with transform.
23. **CSS Preprocessors (SASS (Syntactically Awesome Style Sheets), LESS (Leaner Style Sheets)):** SASS/LESS extend CSS with variables, nesting, mixins.
24. **Including Custom Fonts (`@font-face`, Google Fonts):** `@font-face` for local fonts, Google Fonts for web fonts.
25. **Handling Browser Compatibility (Vendor Prefixes, Feature Queries, Polyfills):** Use vendor prefixes, feature queries, polyfills.

→ Sass (Syntactically Awesome Style Sheets) - A CSS preprocessor that adds power and elegance to plain CSS.

- Variables ✅

- Nesting ✅
- Mixins & Functions ✅
- Inheritance ✅
- Cleaner, DRY-er code ✅

LESS is a **CSS preprocessor**, just like **Sass**, used to extend CSS with: **Variables, Mixins ,Functions, Nested rules, Operations .**

JAVASCIPT

nonpremitive →array. list, char, int

1.What are the data types in JavaScript?

JavaScript has primitive data types such as number, string, boolean, null, undefined, and symbol (added in ES6). Objects (object) and functions (function) are also types in JavaScript.

2 **Explain the difference between undefined and null.**

undefined means a variable has been declared but not assigned. null is a deliberate non-value.

```
let a; // undefined
  let b = null; // null
```

3. **How does JavaScript handle types?**

JavaScript is dynamically typed, meaning variables can hold values of any type without explicit type declaration.

4. **What is the difference between == and ===?**

== checks for equality with type conversion. === checks for strict equality without type conversion.

5. **Explain hoisting in JavaScript.**

Hoisting moves var declarations (but not initializations) to the top of their scope before execution.

6. **What is strict mode?**

A restricted variant of JavaScript that catches common coding mistakes. Enable with 'use strict'; at the top of a script or function.

7. **What are global variables?**

Variables accessible from any part of the program. In non-strict mode, they can be created by assigning to a variable that has not been declared.

```
globalVariable = 10;
```

8. **Explain the difference between var, let, and const.**

var is function-scoped. let and const are block-scoped. const cannot be reassigned

9. **How do you define constants?**

Using the const keyword. They must be initialized and cannot be reassigned.

10. **How can you create functions?**

Using function declarations, function expressions, or arrow functions.

```
function greet1() {}          // Declaration
  const greet2 = function() {}; // Expression
  const greet3 = () => {};      // Arrow
```

11. **What are arrow functions?**

A concise syntax for writing functions (⇒). They do not have their own this or arguments bindings.

12. **Explain function closures.**

A closure is a function that remembers the variables from its outer scope, even after the outer function has finished executing.

```
function outer() {
  let count = 0;
  return () => ++count; // inner function has closure over count
}
```

13. . **What is a callback function?**

A function passed as an argument to another function, to be executed later.

```
setTimeout(() ⇒ {
  console.log("2 seconds have passed");
}, 2000);
```

14. **Explain higher-order functions.**

Functions that operate on other functions, either by taking them as arguments or by returning them.

```
[1, 2, 3].map(num ⇒ num * 2); // .map is a higher-order function
```

15. **How does the this keyword work?**

this refers to the context in which a function is called. Its value depends on how the function is invoked.

```
const person = {
  name: 'John',
  greet() { console.log(`Hello, ${this.name}`); }
};
person.greet(); // `this` refers to `person`
```

16.. **Difference between call, apply, and bind?**

All are used to set the this context. call and apply invoke the function immediately. bind returns a new function with a fixed this.

17. **Explain prototypal inheritance.**

Objects can inherit properties and methods from other objects through their prototype.

18. **What is a prototype chain?**

A series of linked objects used to implement inheritance. If a property is not found on an object, the search continues up the chain.

19. **How do you add properties to an object?**

Using dot notation (person.name = 'Alice') or bracket notation (person['age'] = 30).

20 **What are object methods?**

Functions that are properties of an object.

```
const person = {
  greet() { console.log('Hello!'); }
};
```

21 **Explain object destructuring.**

A syntax for extracting properties from objects into variables.

```
const { name, age } = { name: 'Alice', age: 25 };
```

22. **How do you clone an object?**

Using the spread syntax (...) or Object.assign() for a shallow clone.

```
const clone = { ...originalObject };
```

23 **What are ES6 classes?**

Syntactic sugar over JavaScript's prototype-based inheritance, providing a cleaner class-based syntax.

```
class Person {
  constructor(name) { this.name = name; }
}
```

24 **How do you inherit from a class?**

Using the extends keyword to create a subclass.

javascript

```
class Student extends Person { /* ... */ }
```

25 **Difference between hasOwnProperty and the in operator?**

hasOwnProperty checks if a property is on the object itself. The in operator checks if it exists in the object or its prototype chain.

26 How do you create an array?

```
const fruits = ['apple', 'banana'];
```

Using array literals ([]) or the Array constructor.

27. **Methods to add/remove elements from an array?**

Add: push (end), unshift (start). Remove: pop (end), shift (start). splice can add and remove anywhere.

28 What are map, filter, and reduce?

map: creates a new array by transforming each element. filter: creates a new array with elements that pass a test. reduce: reduces an array to a single value.

29. **Difference between slice and splice?**

slice returns a shallow copy of a portion of an array; it does not modify the original. splice changes the original array.

30 **How do you iterate over an array?**

Using for loops, forEach(), map(), or for...of.

```
for (const fruit of fruits) { console.log(fruit); }
```

31. **How does forEach differ from a for loop?**

forEach is a more concise array method. You cannot break or continue from a forEach loop.

32 **How do you check if a variable is an array?**

Using Array.isArray().

```
Array.isArray([1, 2, 3]); // true
```

33 What are typed arrays?

Array-like objects that provide a mechanism for handling raw binary data, offering better performance for numerical data.

34. **Explain the event loop.**

A mechanism that handles async operations by moving tasks from a queue to the call stack when the stack is empty.

35 What are promises?

Objects representing the eventual completion (or failure) of an asynchronous operation and its resulting value.

```
fetch('api/data')
  .then(res ⇒ console.log('Success!'))
  .catch(err ⇒ console.error('Failed!'));
```

36 What is asynchronous programming?

Allows operations to be executed without blocking the main thread, so the program can remain responsive.

```
console.log('Start');
setTimeout(() ⇒ console.log('Timeout done'), 1000);
console.log('End');
```

37. **Explain async and await.**

async defines an asynchronous function. await pauses execution inside an async function until a promise settles.

```
async function getData() {
  const response = await fetch('api/data');
  const data = await response.json();
}
```

38 **Difference between callbacks and promises?**

Callbacks are functions passed as arguments. Promises are objects that represent a future value and help avoid "callback hell".

39 How do you handle errors in asynchronous code?

Using the .catch() method on promises, or try...catch blocks with async/await.

40 Explain promise chaining.

Executing asynchronous operations sequentially by returning a new promise from a .then() or .catch() handler.

41 . **What are generator functions?**

Special functions (function*) that can be paused and resumed, producing a sequence of values using the yield keyword.

42. **How do you handle multiple asynchronous operations?**

Using Promise.all() to wait for all promises to settle, or Promise.race() to get the result of the first one that settles.

43. **How do you access and manipulate the DOM?**

Access with getElementById(), querySelector(), etc. Manipulate with innerHTML, setAttribute(), appendChild().

44.. **Explain event handling.**

Attaching event listeners to DOM elements to respond to user interactions or other events

```
button.addEventListener('click', () ⇒ { console.log('Clicked!'); });
```

45. **What are event bubbling and capturing?**

Bubbling: events propagate from the target up to the root. Capturing: events propagate from the root down to the target.

46. **How do you add and remove event listeners?**

Using addEventListener() to add and removeEventListener() to remove.

47 **How do you prevent default behavior in an event?**

Using event.preventDefault() inside the event handler.

48 Explain event delegation.

Attaching a single event listener to a parent element to manage events for all its child elements. It's efficient and handles dynamically added elements.

49. **How do you create and trigger custom events?**

Create with new CustomEvent() and trigger with element.dispatchEvent().

50 **getElementById vs querySelector?**

getElementById selects an element by its id. querySelector uses a CSS selector and returns the first matching element.

51 What are new ES6 features?

let/const, arrow functions, classes, template literals, destructuring, promises, modules.

52 **Explain destructuring assignment.**

Allows you to extract values from arrays or properties from objects into variables.

```
const [a, b] = [1, 2];
const { name } = { name: 'Bob' };
```

53 . **What are template literals?**

String literals allowing embedded expressions and multi-line strings, enclosed by backticks (`` ). ``

const greeting = Hello, ${name}!;

54 What are some common debugging techniques in JavaScript?

Common debugging techniques include using console.log(), console.error(), breakpoints in browser developer tools,

55 How do you use localStorage and sessionStorage inJavaScript ?

LocalStorage and sessionStorage provide mechanisms to store key-value pairs in the browser. localStorage persists data across browser sessions, while sessionStorage persists data within the session.

56 How do you make HTTP requests in JavaScript? Explain fetch API.

Fetch() API is used to make HTTP requests in JavaScript, supporting modern promises and providing a more flexible and powerful interface for fetching resources from the network

57 How do you handle CORS issues in JavaScript?

Cross-Origin Resource Sharing (CORS) issues can be handled by configuring server-side headers to allow requests from specific origins (Access-Control-Allow-Origin) and methods (Access-Control-Allow-Methods)

58 How do you optimize JavaScript code for better performance?

JavaScript code can be optimized for performance by minimizing DOM manipulation, using efficient algorithms, caching variables, reducing function calls, and leveraging browser developer tools for profiling and optimization.

59 What are the benefits of using immutable data structures in JavaScript?

Immutable data structures, where data cannot be changed after creation, offer several benefits:

Consistency, Concurrency, Predictability , Performance

60 How do you handle routing in a single-page application (SPA) using JavaScript?

Routing in SPAs is typically handled using client-side routing libraries like React Router, Vue Router, or by implementing custom routing logic using the History API

61 How do you implement animations in JavaScript?

Animations can be implemented in JavaScript using CSS animations/transitions or JavaScript libraries like GSAP (GreenSock Animation Platform) for more complex animation

62 how do you integrate third-party libraries and plugins in JavaScript applications? -

Third-party libraries andplugins can be integrated into JavaScript applications by including their scripts or modules using <script> tags, import statements (for ES modules), or using package managers like npm or yarn.