1000+ MERN STACK INTERVIEW QUESTIONS

Real Questions Asked by Startups, Product Companies & MNCs

(0–12+ Years)

JavaScript

React & Redux

MongoDB

System Design

Security & DevOps

Crack MERN Interviews from 0 to 12+ Years!

1000+ Complete MERN Stack Interview Question Answer Mastery Prep Kit

Real Questions Asked by Startups, Product Companies & MNCs (0–12+ Years | JavaScript • React • Redux • Node • MongoDB • System Design)

By
Sandeep Pal

This is sample Ebook. You can buy complete ebook from link: https://topmate.io/sandeeppal/1957516

📖 Preface

The MERN stack is no longer "just a tech stack." It is an ecosystem powering high-scale startups, enterprise platforms, SaaS products, fintech systems, and real-time applications. Yet, most interview preparation material focuses on:

- Theoretical definitions
- Copy-paste answers
- Surface-level explanations

This book is different.
This book is written from the perspective of a 15+ year MERN architect and active interview panelist who has:

- Conducted 1000+ interviews
- Built large-scale production systems

- Designed microservices architectures
- Mentored junior to senior engineers

Here, you will not find "Google answers."
You will find:

- Real interview questions asked by product companies and MNCs
- Production-level explanations
- Real-world examples
- Common mistakes candidates make
- Architectural thinking patterns

If you study this book deeply, you will not just "clear interviews" — you will think like a MERN architect.

⚠️ Disclaimer

This book is intended for educational and interview preparation purposes only.
The interview questions included are based on:

- Real interview experiences
- Industry-standard evaluation patterns
- Common technical screening processes across startups, MNCs, and product companies

Company names are not referenced to avoid confidentiality violations.
All code samples are simplified for learning clarity and may require adaptation for production systems.
The author assumes no responsibility for:

- Implementation errors in live systems
- Interview decisions made by organizations

This book does not guarantee job placement. It guarantees preparation quality.

🧭 Who This Book Is For

This book is designed for:
✔ Freshers entering MERN ecosystem
✔ Frontend or Backend developers switching to MERN
✔ 1–5 year developers preparing for product companies
✔ 5–12+ year engineers targeting senior or architect roles
✔ Interviewers wanting structured evaluation questions
If you want shortcut answers — this book is not for you.
If you want deep understanding — you are in the right place.

EBOOK: Question Distribution

Section	**Questions**
JavaScript	130
React + Redux	150
Node.js	155
MongoDB	155
Express & APIs	145
System Design	130
Auth & Security	75
DevOps & CI/CD	45
Testing	20
Behavioral	30
TOTAL	**1,035+ QUESTIONS** 🚀

📚 CATEGORY-WISE BREAKDOWN (CORE OF THE EBOOK)

1️⃣ JavaScript (Core + Advanced) – 140 Questions

Most MERN interviews FAIL here.

Subcategories

- JS Fundamentals (Execution, Scope, Hoisting)
- Closures, Currying, Memoization
- `this`, call/apply/bind
- Event Loop, Microtask vs Macrotask
- Promises, async/await pitfalls
- Memory leaks
- Prototype & inheritance
- ES6+ internals
- Performance optimization

🔹 **Company-style question example**

- *"Why does `setTimeout(fn, 0)` still wait?"*
- *"How does V8 handle closures in memory?"*
- *"Design a retry mechanism using Promises."*

2️⃣ React.js (Beginner → Advanced → Internals) – 200 Questions

Subcategories

- JSX & rendering behavior
- Hooks deep dive (`useEffect`, `useMemo`, `useCallback`)
- Controlled vs uncontrolled components
- State management patterns

- Reconciliation & Fiber
- Performance optimization
- Error boundaries
- React 18 concurrency
- Custom hooks design
- Anti-patterns

🔹 **Real interview questions**

- *“Why does useEffect run twice in React 18?”*
- *“When should you NOT use useMemo?”*
- *“How would you detect unnecessary re-renders?”*

3️⃣ Node.js (Runtime + Internals) – 130 Questions

Subcategories

- Event loop internals
- Streams & buffers
- Cluster vs Worker Threads
- Async patterns
- Memory management
- Scaling Node apps
- Error handling strategies
- Security risks
- Logging & monitoring

🔹 **Panel-level questions**

- *“Why Node is single-threaded but still scalable?”*
- *“How does backpressure work in streams?”*
- *“Explain a Node memory leak you faced.”*

4️⃣ Express.js & API Design – 80 Questions

Subcategories

- Middleware lifecycle
- Error handling
- API versioning
- REST vs RPC
- Authentication strategies
- Authorization models
- Rate limiting
- Validation
- Pagination & filtering

🔹 **Real-world**

- *“How would you design a versioned API without breaking clients?”*
- *“JWT vs session — when will JWT fail?”*

5️⃣ MongoDB (Beginner → Advanced → Production) – 120 Questions

Subcategories

- Data modeling
- Indexing strategies
- Aggregation pipeline
- Transactions
- Replication
- Sharding
- Performance tuning
- MongoDB vs SQL decision-making
- Atlas production issues

🔹 **Company questions**

- *“Why embedding vs referencing?”*
- *“Explain a slow query you optimized.”*
- *“How sharding impacts writes?”*

6️⃣ Authentication, Authorization & Security – 60 Questions

Subcategories

- JWT internals
- OAuth 2.0
- Refresh token strategies
- XSS, CSRF
- CORS
- Rate limiting

- Password hashing
- Secure cookie handling

🔹 **Advanced**

- *“How would you revoke JWTs?”*
- *“Explain CSRF even when using JWT.”*

7️⃣ System Design (MERN-Focused) – 90 Questions

Subcategories

- Designing scalable MERN apps
- Frontend architecture
- Backend scalability
- Database scaling
- Caching strategies
- Message queues
- Real-time apps (Socket.io)
- File uploads
- CDN usage

🔹 **Design prompts**

- *“Design Instagram-like feed using MERN.”*
- *“How would you scale a chat app?”*

8️⃣ Testing (Frontend + Backend) – 40 Questions

Subcategories

- Jest
- React Testing Library
- Supertest
- Unit vs integration testing
- Mocking strategies
- Test pyramid

9️⃣ DevOps, Deployment & Production – 40 Questions

Subcategories

- Docker basics
- CI/CD
- Environment variables
- PM2
- AWS basics
- Nginx
- Monitoring
- Logging

🔟 Behavioral + Real Project Experience – 50 Questions

Very underrated but heavily asked

Examples

- *“Tell me about a production bug you caused.”*
- *“How did you optimize frontend load time?”*
- *“How do you review PRs?”*
- *“How do you handle junior developers?”*

1️⃣ What is an Execution Context?

An execution context is the environment in which JavaScript code runs.
Think of it as a “container” that holds:

- Variables
- Function declarations
- The value of this
- Scope references

Whenever JavaScript runs:

- Global code → Global Execution Context
- Function call → New Function Execution Context
- eval (rare) → Eval Execution Context

Real-world analogy:
It’s like a workspace created every time code starts running.
Without execution context, JavaScript cannot track variables or function calls.

2️⃣ What are the phases of execution context?

There are two main phases:

🔹 1. Creation Phase

Before executing code, JS engine:

- Creates memory for variables
- Sets up function declarations

- Assigns undefined to var
- Sets up scope chain
- Determines this

This is where hoisting happens.

🔹 2. Execution Phase

- Code runs line by line
- Variables get assigned actual values
- Functions are invoked

Interview tip:
If candidate cannot explain creation phase, they don't understand hoisting.

3️⃣ What is the Global Execution Context?

It is the default context created when JavaScript file starts executing.
In browser:

- this refers to window

In Node.js:

- this refers to module.exports

Only one global execution context exists per program.
All functions created inside refer back to this via scope chain.

4️⃣ What is the Call Stack?

The call stack is a data structure (LIFO – Last In First Out) that keeps track of function calls.
Example:

```
function a() {
  b();
}
function b() {
  console.log("Hello");
}
a();
```

Call stack flow:

- Global context
- a()
- b()
- b() completes → removed
- a() completes → removed

Stack behavior is critical for:

- Debugging
- Understanding recursion
- Event loop

5️⃣ What is Lexical Scope?

Lexical scope means scope is determined by where code is written, not where it is called.
Example:

```
function outer() {
  let name = "John";

  function inner() {
    console.log(name);
  }

  inner();
}
```

inner() accesses name because it was defined inside outer.
Not based on runtime location.
Based on code structure.

6️⃣ What is Scope Chain?

Scope chain is the mechanism by which JS looks for variables.
When accessing a variable:

1. Check local scope
2. If not found → check parent scope

3. Continue upward
4. If not found → ReferenceError

This chain is built during creation phase.
This is why nested functions can access outer variables.

7️⃣ Explain Block Scope with Example

Block scope applies to:

- let
- const

Example:

```
if (true) {
  let x = 10;
}
console.log(x); // Error
```

x only exists inside block.
But:

```
if (true) {
  var x = 10;
}
console.log(x); // 10
```

var ignores block scope.
In production:
Using var causes unpredictable bugs in loops.

8️⃣ What is Temporal Dead Zone (TDZ)?

TDZ is the time between:

- Entering scope
- Variable declaration

Example:

```
console.log(a);
let a = 5;
```

Throws ReferenceError.
Even though let is hoisted, it is not initialized.
This prevents accidental usage before declaration.

9️⃣ Why does let behave differently from var?

Because:

var	let
Function scoped	Block scoped
Initialized as undefined	Not initialized
Allows redeclaration	No redeclaration
Attached to window (browser)	Not attached

let was introduced to fix problems caused by var.

🔟 What happens when a function is invoked?

When function runs:

1. New execution context created
2. Memory allocated
3. Arguments object created
4. this determined
5. Scope chain established
6. Code executed
7. Context removed from stack

This process happens every time.

1️⃣1️⃣ What is Variable Shadowing?

When inner variable has same name as outer variable.
Example:

```
let x = 10;

function test() {
  let x = 20;
  console.log(x);
}
```

Inner x shadows outer x.
Used sometimes intentionally.
But can cause confusion in large systems.

1️⃣2️⃣ Can JavaScript have Function-Level Scope?

Yes.
Before ES6, only function-level scope existed using var.
Example:

```
function test() {
  var x = 10;
}
```

x only exists inside function.
But ES6 introduced block scope.

1️⃣3️⃣ How does JavaScript resolve variable lookups?

Through scope chain.
When accessing variable:

- Check current execution context's lexical environment
- If not found → check outer reference
- Continue upward

JS does NOT search downward.
This is important when debugging undefined errors.

1️⃣4️⃣ Why does typeof null return "object"?

Historical bug.
In early JS implementation, type tags were used.
null was mistakenly assigned object tag.
For backward compatibility, it was never fixed.
So:

```
typeof null // "object"
```

Interview trick question.

1️⃣5️⃣ What happens when the call stack overflows?

You get:

RangeError: Maximum call stack size exceeded

Occurs when:

- Infinite recursion
- Too many nested calls

Example:

```
function test() {
  test();
}
```

This crashes execution.

1️⃣6️⃣ What is Recursion? When is it dangerous?

Recursion = function calling itself.
Used in:

- Tree traversal
- Deep cloning
- Algorithms

Dangerous when:

- No base condition
- Very deep recursion → stack overflow

In production:
Prefer iterative solutions for large datasets.

1️⃣7️⃣ How do Closures relate to Scope?

Closure = function remembers its lexical scope even after outer function finishes.
Example:

```
function outer() {
  let count = 0;
  return function () {
    count++;
    console.log(count);
  };
}
```

count is preserved in memory.

Closure works because of lexical scope + scope chain.

---

18 How does memory allocation happen in JS?

Two major areas:

🔹 Stack

Stores:

- Primitive values
- Execution contexts
- Function calls

Fast allocation.

🔹 Heap

Stores:

- Objects
- Arrays
- Functions

Dynamic memory allocation.

---

19 When are variables garbage collected?

When no references exist to them.
JS uses:

- Mark-and-sweep algorithm

If object is unreachable → cleaned.
Example memory leak:

```
let arr = [];
setInterval(() => {
  arr.push(new Date());
}, 1000);
```

arr keeps growing → never garbage collected.

---

20 Explain Stack vs Heap Memory

Stack

- LIFO structure
- Stores primitives
- Fast
- Fixed size

Heap

- Stores objects
- Dynamic
- Managed by garbage collector
- Slower than stack

When you assign:
let obj = { name: "A" };

obj reference stored in stack.
 Actual object stored in heap.

SECTION 3: Closures & Functional JavaScript (2–5 Years)

---

1 What is a Closure? Explain with a real-world example.

A closure is created when:
A function remembers variables from its lexical scope even after the outer function has finished executing.
Example:

```
function createCounter() {
  let count = 0;

  return function () {
    count++;
    return count;
  };
}

const counter = createCounter();
counter(); // 1
counter(); // 2
```

Here:

- count is not destroyed.
- Inner function “closes over” it.

Real-world example:
Imagine a banking system:

```
function createAccount(initialBalance) {
  let balance = initialBalance;

  return {
    deposit(amount) {
      balance += amount;
      return balance;
    },
    withdraw(amount) {
      balance -= amount;
      return balance;
    }
  };
}
```

Balance is protected.
Only accessible via returned methods.
That’s closure in real production usage.

2️⃣ Why are Closures Used in JavaScript?

Closures are used for:

- Data privacy
- State preservation
- Functional patterns
- Event handlers

- Async callbacks
- Memoization
- React hooks internals

Closures are fundamental to:

- React
- Node async patterns
- Middleware design

Without closures, modern JavaScript architecture collapses.

3️⃣ How do Closures Help in Data Encapsulation?

Encapsulation = hiding internal state.
Before ES6 classes, closures were primary method for private variables.
Example:

```
function createUser(name) {
  let password = "12345";

  return {
    getName() {
      return name;
    }
  };
}
```

Password is inaccessible directly.

```
user.password // undefined
```

Only accessible inside closure.
This is functional-style privacy.

4️⃣ Can Closures Cause Memory Leaks? How?

Yes.
Because closure keeps reference to outer variables.
If outer variable is large and not needed anymore, it still stays in memory.
Example:

```
function heavy() {
  let largeData = new Array(1000000).fill("data");
  return function () {
    console.log("Using closure");
  };
}
```

Even if largeData is not used, it remains in memory.
Common real-world leak:

- Event listeners not removed
- Timers not cleared
- Long-living closures in Node servers

In React:
Stale closures can retain old state references.

5️⃣ What is Currying?

Currying transforms a function with multiple arguments into a sequence of functions each taking one argument.
Normal function:

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

Curried version:

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

Usage:

```
add(5)(3); // 8
```

6️⃣ Convert a Normal Function into Curried Function

Normal:

```
function multiply(a, b, c) {
  return a * b * c;
}
```

Curried:

```
function multiply(a) {
  return function (b) {
    return function (c) {
      return a * b * c;
    };
  };
}
```

More modern:

```
const multiply = a => b => c => a * b * c;
```

7️⃣ What is Function Composition?

Function composition means combining multiple functions where output of one becomes input of another.

Example:

```
const double = x => x * 2;
const square = x => x * x;

const compose = (f, g) => x => f(g(x));

const result = compose(square, double);
result(3); // square(double(3)) = 36
```

Used heavily in:

- Redux
- Middleware chains
- Functional utilities

8️⃣ What is a Higher-Order Function?

A function that:

- Takes another function as argument
  OR
- Returns a function

Examples:

- map
- filter

- reduce
- setTimeout

Example:

```
function greet(fn) {
  fn();
}
```

Higher-order functions allow abstraction.

9️⃣ What is Memoization?

Memoization = caching function results based on input.

Example:

```
function memoize(fn) {
  const cache = {};

  return function (arg) {
    if (cache[arg]) return cache[arg];

    const result = fn(arg);
    cache[arg] = result;
    return result;
  };
}
```

Improves performance for expensive computations.

🔟 Where Have You Used Memoization in Real Projects?

Real examples:

1. React expensive calculations
2. Filtering large datasets
3. Search result caching
4. API response caching

Example in React:

```
const filteredData = useMemo(() => {
  return data.filter(item => item.active);
}, [data]);
```

Prevents unnecessary recalculation.

1️⃣1️⃣ How does useCallback Relate to Closures?

useCallback returns a memoized function.
That function closes over variables in its scope.
Problem:
 If dependencies change, new closure created.
If dependency array is wrong → stale closure bug.
Example stale closure:

```
const handleClick = useCallback(() => {
  console.log(count);
}, []);
```

count is frozen at initial value.

1️⃣2️⃣ What Happens if a Closure Captures a Large Object?

Large object stays in memory as long as closure exists.
This causes:

- Increased heap usage
- Potential memory leaks
- Slower GC cycles

In Node servers:
 Long-lived closures capturing request objects can leak memory.
Always avoid capturing unnecessary references.

1️⃣3️⃣ Explain Private Variables Using Closures

Example:

```
function secretHolder(secret) {
  return {
    getSecret() {
      return secret;
    }
  };
}
```

secret cannot be accessed directly.
Used before class #private fields existed.

---

14 Why Closures Are Heavily Used in React?

React functional components rely on closures.
Every render:

- Component function executes
- New closure created
- Hooks depend on closure behavior

useState, useEffect rely on closure mechanism.
Stale closure issues are common React bugs.

---

15 What is Partial Application?

Partial application fixes some arguments of a function.
Example:

```
function multiply(a, b) {
  return a * b;
}

const double = multiply.bind(null, 2);
double(5); // 10
```

Difference from currying:
Partial application can fix multiple arguments at once.

---

16 How Would You Implement a Counter Using Closure?

```
function counter() {
  let count = 0;

  return {
    increment() {
      return ++count;
    },
    decrement() {
      return --count;
    }
  };
}
```

State persists without global variable.

1️⃣7️⃣ Difference Between Closure and Scope?

Scope:
 Defines where variables are accessible.
Closure:
 Function + preserved scope after outer function finishes.
Scope is concept.
 Closure is behavior.

1️⃣8️⃣ What is a Pure Function?

A pure function:

- Same input → same output
- No side effects
- Does not modify external state

Example:

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

1️⃣9️⃣ Why Pure Functions Are Preferred?

Because they are:

- Predictable
- Testable
- Easier to debug
- Safe in concurrent systems

React prefers pure components for performance.

2️⃣0️⃣ How Do Side Effects Affect Predictability?

Side effects:

- Modify global state
- Change DOM

- Make network requests
- Mutate variables

They make debugging harder.
Example:

```
let total = 0;

function add(x) {
  total += x;
}
```

Output depends on external state.
Harder to test.

SECTION 1: React Basics (0–1 Years) – 25 Questions

1️⃣ What is React and Why Is It Used?

React is a JavaScript library for building user interfaces, primarily for single-page applications (SPAs).
It focuses only on:

- The View layer
- Component-based architecture
- Efficient rendering

Why it's used:

- Reusable UI components
- Fast rendering via Virtual DOM
- Strong ecosystem
- Predictable state management
- Large community support

In production:
React allows building scalable frontends with modular structure.

2️⃣What Problems Does React Solve Compared to Vanilla JS?

Before React:

- Manual DOM manipulation
- Complex UI state synchronization
- Hard-to-maintain large apps
- Event spaghetti code

Example in vanilla JS:
Updating UI requires:

- Query selectors
- Manual DOM updates
- Managing side effects manually

React solves this by:

- Declarative rendering
- State-driven UI
- Component reusability
- Automatic DOM diffing

Instead of saying:
"Update this DOM node"
You say:
"Here's the state — render UI accordingly."

3️⃣What is JSX?

JSX = JavaScript XML
It allows writing HTML-like syntax inside JavaScript.
Example:

```
const element = <h1>Hello</h1>;
```

JSX improves readability and developer experience.
But JSX is not HTML.

4️⃣ Why Can’t Browsers Understand JSX Directly?

Browsers understand:

- HTML
- CSS
- JavaScript

JSX must be transpiled (converted) into JavaScript.
Example:

```
<h1>Hello</h1>
```

Becomes:

```
React.createElement("h1", null, "Hello");
```

This transformation is done by:

- Babel
- Vite
- Webpack

JSX is just syntactic sugar.

5️⃣ What is a Component?

A component is a reusable, independent UI unit.
Example:

```
function Button() {
  return <button>Click</button>;
}
```

Components:

- Accept props
- Manage state
- Render UI
- Can be nested

In large apps:
Everything is a component.

6️⃣ Difference Between Functional and Class Components?

Functional Component

- Simple function
- Uses hooks
- No this

Example:
function App() {}

Class Component

- Uses ES6 class
- Has lifecycle methods
- Uses this

Example:
class App extends React.Component {}

Modern React prefers functional components + hooks.
Class components are legacy in most new projects.

7️⃣ What Are Props?

Props = properties passed from parent to child.
Example:
<Profile name="John" />

Inside component:

```
function Profile(props) {
  return <h1>{props.name}</h1>;
}
```

Props allow:

- Data passing
- Component reuse

8️⃣ Are Props Mutable?

No.
Props are read-only.
You should never modify props inside a component.
Wrong:
props.name = "Alex"; // ❌

React enforces one-way data flow.

9️⃣ What is State?

State is internal data that controls component behavior.
Example:
const [count, setCount] = useState(0);

When state changes:
Component re-renders.
State represents:
Dynamic UI data.

🔟 Difference Between State and Props?

Props	State
Passed from parent	Managed internally
Immutable	Mutable via setter
Used for configuration	Used for dynamic UI

Props = input
State = internal memory

1️⃣1️⃣ Why State Updates Are Asynchronous?

React batches state updates for performance.
Example:
setCount(count + 1);
setCount(count + 1);

May not increment twice.
Because:
React groups updates and schedules re-render.
In React 18:
Automatic batching applies to more scenarios.

1️⃣2️⃣ What is Virtual DOM?

Virtual DOM is a lightweight JavaScript representation of real DOM.
React:

- Creates virtual copy
- Compares with previous version
- Updates only changed parts

This reduces expensive DOM operations.

1️⃣3️⃣ How Virtual DOM Improves Performance?

Direct DOM manipulation is slow.
Virtual DOM:

- Performs diff in memory
- Updates minimal DOM nodes
- Batches updates

Result:
 Faster UI updates.
Important:
 React is fast not because of Virtual DOM alone,
 but because of optimized reconciliation.

1️⃣4️⃣ What is Reconciliation?

Reconciliation is React's process of:
Comparing previous virtual DOM with new virtual DOM to determine changes.
This diffing process decides:

- What to update
- What to remove
- What to create

Efficient reconciliation prevents full re-render.

1️⃣5️⃣What is key Prop and Why Is It Important?

Keys help React identify list items uniquely.
Example:
items.map(item => <li key={item.id}>{item.name}</li>)

Keys:

- Help React track elements
- Improve diffing efficiency
- Prevent unnecessary re-renders

SECTION 2: Hooks Deep Dive (1–3 Years) – 30 Questions

1️⃣Why Were Hooks Introduced?

Hooks were introduced in React 16.8 to:

- Use state in functional components
- Avoid complex class components
- Share logic without HOCs or render props
- Improve code readability
- Reduce lifecycle confusion

Before hooks:

- Classes were mandatory for state
- this binding issues
- Lifecycle duplication

Hooks simplified mental model.

2️⃣Rules of Hooks – Why Do They Exist?

Two main rules:

1. Only call hooks at top level.
2. Only call hooks inside React functions.

Why?
Because React relies on call order to map hooks to internal state slots.
Example bad:

```
if (condition) {
  useState(0);
}
```

This breaks hook ordering.
React uses an internal linked list to track hooks per component.
Changing order = broken state mapping.

3️⃣ What is useState?

useState allows functional components to manage state.

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

Returns:

- Current state
- Setter function

Each render:
 React reads state from internal hook storage.

4️⃣ Why Should We Not Mutate State Directly?

Wrong:

```
state.value = 10;
```

React won't detect mutation.
React re-renders only when setter function triggers update.
Mutation causes:

- No re-render
- Unexpected UI bugs
- State inconsistency

Always use setter:

```
setState(newValue);
```

5️⃣ What is Functional Update in useState?

When new state depends on previous state.
Example:
setCount(prev => prev + 1);

Why needed?
Because state updates are asynchronous and batched.
Wrong pattern:
setCount(count + 1);
setCount(count + 1);

May not increment twice.
Functional update ensures correct previous value.

6️⃣ What is useEffect?

useEffect performs side effects after render.
Examples:

- API calls
- Subscriptions
- Timers
- DOM manipulation

It runs after component renders.

7️⃣ How Does useEffect Dependency Array Work?

Dependency array controls when effect runs.
useEffect(() => {
  // effect
}, [dependency]);

React compares previous dependency values with current ones.
If changed → effect runs.
Comparison uses shallow equality.

8️⃣ What Happens If Dependency Array is Empty?

```
useEffect(() => {
  console.log("Runs once");
}, []);
```

Runs only once (after first render).
Equivalent to componentDidMount behavior.
But in React 18 Strict Mode (dev), runs twice.

9️⃣ What Happens If Dependency Array is Omitted?

```
useEffect(() => {
  console.log("Runs every render");
});
```

Runs after every render.
This can cause infinite loops if state updated inside effect.

SECTION 6: Redux Async, Architecture & Scaling (5–9 Years)

1️⃣ What is Redux Thunk?

Redux Thunk is middleware that allows action creators to return a function instead of a plain object.
Normal action:

```
dispatch({ type: "INCREMENT" });
```

With Thunk:

```
dispatch((dispatch, getState) => {
  // async logic
});
```

It enables:

- Async API calls
- Delayed dispatch
- Conditional dispatch

2️⃣ How Does Thunk Work Internally?

Thunk middleware checks:
If action is a function → call it.
Simplified internal logic:

```
if (typeof action === "function") {
  return action(dispatch, getState);
}
```

Otherwise:
 Pass action to reducer.
Thunk is simple but powerful.

3️⃣ What is Redux Saga?

Redux Saga is middleware that handles side effects using generator functions.
It uses:

- yield
- watchers
- effects like call, put, takeLatest

Example:

```
function* fetchUserSaga() {
  const data = yield call(apiCall);
  yield put(setUser(data));
}
```

Saga runs separately from reducers.

4️⃣ Difference Between Thunk and Saga?

Thunk	Saga
Simple	Powerful
Uses functions	Uses generators
Easier to learn	Steeper learning curve
Basic async	Complex workflows

Harder for cancellation	Built-in cancellation

Thunk = lightweight
Saga = enterprise-level async orchestration

Advanced React, Architecture & Interviews (7–12+ Years)

1️⃣ How Would You Architect a Large React App?

A large React app must be:

- Modular
- Scalable
- Testable
- Performance-aware
- Team-friendly

My architectural approach:

1. Feature-based modular structure

Each feature owns:

- Components
- Hooks
- Slice
- API logic
- Tests

```
/features
  /auth
  /dashboard
  /billing
```

2. Clear layering

- UI layer
- State layer
- API layer
- Utilities

3. State strategy

- Local state by default
- Global state only when needed
- RTK Query for server state

4. Performance strategy

- Code splitting
- Lazy loading
- Memoization
- Virtualization

5. Strict linting + testing + CI

Architecture must evolve with team size.

2️⃣ Feature-Based vs Layer-Based Structure?

Layer-based (old pattern):

/components
/actions
/reducers

Problems:

- Scattered logic
- Hard to scale

- Tight coupling

Feature-based (modern):

```
/features
  /users
    userSlice.js
    UserList.jsx
```

Benefits:

- Encapsulation
- Easier refactoring
- Better team ownership

For large apps → always feature-based.

3️⃣ How Do You Manage Shared State at Scale?

Strategy:

1. Local state first
2. Lift state only when necessary
3. Context for lightweight global data
4. Redux for complex global workflows
5. RTK Query for server state

Avoid globalizing everything.
Global state should be:

- Rare
- Predictable
- Minimal

4️⃣ How Do You Decide Redux vs Local State?

Decision factors:

Scenario	Use
Shared across distant components	Redux
UI-only state	Local state
Temporary form data	Local
Authentication	Redux
Server data caching	RTK Query

If 2 components need it → maybe lift state.
If 10 components need it → Redux.

5️⃣ How Do You Migrate Legacy Redux Code?

Legacy Redux often has:

- Action types everywhere
- Switch-based reducers
- Deep nesting

Migration strategy:

1. Incrementally move to Redux Toolkit
2. Replace reducers with createSlice
3. Introduce RTK Query for APIs
4. Normalize state gradually
5. Add tests before refactor

Never rewrite entire app at once.

6️⃣ How Do You Handle Micro-Frontends?

Micro-frontends split large frontend into independent modules.
Approaches:

- Module Federation (Webpack)
- Iframes (legacy)

- Independent deployments

Key challenges:

- Shared dependencies
- Version conflicts
- State isolation
- Auth consistency

Best practice:

- Shared design system
- Shared authentication layer
- Clear API contracts

7️⃣ How Do You Manage Permissions?

Permission strategy:

1. Backend enforces real security
2. Frontend hides UI based on role

Example:

```
if (user.role === "admin") {
  showAdminPanel();
}
```

Better:

Centralized permission utility:

```
canAccess("billing.read");
```

Never trust frontend for actual security.

Node.js Fundamentals (0–1 Years) – 25 Questions

1️⃣ What is Node.js?

Node.js is a JavaScript runtime built on Chrome’s V8 engine that allows JavaScript to run outside the browser.
It enables:

- Backend development
- CLI tools
- Real-time applications
- APIs
- Microservices

Important:
Node.js is not a framework.
It is a runtime environment.

2️⃣ Is Node.js a Language or a Runtime?

Node.js is a runtime.
Language = JavaScript
Runtime = Environment where JS executes
Node provides:

- V8 engine
- Standard libraries
- OS access
- File system access
- Network capabilities

3️⃣ Why Node.js is Popular for Backend Development?

Reasons:

- Same language for frontend & backend (JavaScript)
- Non-blocking I/O
- High concurrency

- Fast development
- Large ecosystem (npm)
- Great for real-time apps
- Lightweight and scalable

Used heavily in:

- APIs
- Chat apps
- Streaming platforms
- SaaS dashboards

4️⃣ What is V8 Engine?

V8 is Google’s JavaScript engine written in C++.
It:

- Parses JS
- Compiles JS to machine code
- Executes optimized code

Node.js embeds V8 to execute JavaScript outside browser.
V8 includes:

- JIT compiler
- Garbage collector
- Optimization engine

5️⃣ How Does Node.js Execute JavaScript Outside the Browser?

Node uses:

- V8 engine for execution
- libuv for event loop and async I/O

- OS APIs for file and network operations

When you write:
console.log("Hello");

V8 executes it.
When you write:
fs.readFile();

libuv handles the async I/O.
Node combines:

- JS engine
- C++ bindings
- OS-level capabilities

6️⃣ What is Non-Blocking I/O?

Non-blocking I/O means:
Node does not wait for an operation to complete before continuing execution.
Example:
fs.readFile("file.txt", callback);

Node:

- Delegates file read to OS
- Continues executing next lines
- Executes callback when file read completes

This allows handling thousands of concurrent requests efficiently.

7️⃣ What is Event-Driven Architecture?

Event-driven architecture means:
Code execution responds to events.
Examples:

- HTTP request received
- File read completed
- Timer expired

Node listens for events and triggers callbacks.
It relies heavily on:

- Event loop
- Event emitters

8️⃣ Why Node.js is Single-Threaded?

Node uses a single-threaded event loop to avoid:

- Race conditions
- Complex thread management
- Locking issues

However:
Node handles concurrency via:

- Non-blocking I/O
- libuv thread pool (for heavy operations)

Single-threaded does NOT mean single-operation-at-a-time.

9️⃣ Difference Between Node.js and Browser JavaScript?

Node.js	Browser
Runs on server	Runs in browser
No DOM	Has DOM
Has fs module	No file system access
Uses CommonJS modules	Uses ES modules
Has process object	Has window object

Browser JS focuses on UI.
Node focuses on backend logic.

🔟 What Are Built-in Node.js Modules?

Core modules include:

- fs (file system)
- http
- path
- os
- crypto
- events
- stream
- util

These are built into Node.
No installation required.

1️⃣1️⃣ What is require()?

require() is CommonJS module loader.
Example:
const fs = require("fs");

It:

- Loads module
- Caches module
- Returns exported object

Used in older Node module system.

1️⃣2️⃣ Difference Between require and import?

require	import
CommonJS	ES Modules
Synchronous	Static
Dynamic	Statically analyzable
Used in older Node	Modern standard

Modern Node supports ES modules:

```
import fs from "fs";
```

1️⃣3️⃣ What is package.json?

package.json is project metadata file.
Contains:

- Project name
- Version
- Dependencies
- Scripts
- Entry point
- License

Example:

```
{
  "name": "app",
  "version": "1.0.0"
}
```

Central file for Node project management.

1️⃣4️⃣ What is node_modules?

node_modules is folder where:
All installed packages are stored.
It can contain:

- Direct dependencies
- Nested dependencies

Usually large in size.
Should not be committed to Git.

1️⃣5️⃣ Difference Between dependencies and devDependencies?

dependencies:

- Required for production

devDependencies:

- Required only for development

Example:

- express → dependency
- nodemon → devDependency

1️⃣6️⃣ What is npm?

npm = Node Package Manager.
Used for:

- Installing packages
- Managing dependencies
- Running scripts
- Publishing packages

It is the largest software registry in the world.

1️⃣7️⃣ What is npx?

npx runs packages without installing globally.
Example:
npx create-react-app app

It:

- Downloads temporarily
- Executes
- Removes after

Useful for CLI tools.

1️⃣8️⃣ What is process Object?

process is global object in Node.
Provides:

- Environment info
- Process ID
- Memory usage
- Exit control

Example:
console.log(process.pid);

1️⃣9️⃣ What Are Environment Variables?

Environment variables store configuration outside code.
Examples:

- DB_URL
- PORT
- API_KEY

Used to separate config from logic.

Transactions, Consistency & Reliability (4–8 Years)

1️⃣ What Are Transactions in MongoDB?

Transactions allow multiple operations to execute atomically.
Either:

- All succeed
 OR
- All fail (rollback)

Mongo supports multi-document ACID transactions.

2️⃣ When Were Transactions Introduced?

- MongoDB 4.0 → Multi-document transactions in replica sets
- MongoDB 4.2 → Transactions across sharded clusters

Before 4.0:
Only single-document atomicity was supported.

3️⃣ How Do Transactions Work Internally?

Internally:

- Mongo uses snapshot isolation
- Operations are staged
- Write-ahead logging ensures durability
- Changes committed only when transaction commits

Transactions use:

- Logical session
- Transaction number
- Commit/abort protocol

4️⃣ What Is Single-Document Atomicity?

Mongo guarantees atomicity at document level.
Example:

```
db.users.updateOne({ _id: 1 }, { $inc: { balance: -100 } });
```

Even if document has multiple fields,
Update is atomic.
Single document operations are safe by default.

5️⃣ When Do You Need Multi-Document Transactions?

You need them when:

- Updating multiple collections

- Money transfer between accounts
- Complex relational-like updates
- Ensuring cross-collection consistency

Example:
Debit account A → Credit account B
Without transaction:
Risk of partial update.

6️⃣ Performance Impact of Transactions?

Transactions:

- Add overhead
- Increase locking
- Reduce throughput
- Consume more memory
- Increase replication cost

Mongo is optimized for document atomicity.
Use multi-document transactions carefully.

7️⃣ What Is Write Concern?

Write concern controls acknowledgment level of write operations.
Options:

- w:1 → Acknowledge by primary
- w:"majority" → Acknowledge by majority of replica set
- w:0 → No acknowledgment

Higher write concern = more durability, slower writes.

8️⃣ What Is Read Concern?

Read concern controls data consistency during reads.
Levels:

- local
- majority
- linearizable
- snapshot

Higher read concern = stronger consistency, slower reads.

9️⃣ What Is Read Preference?

Determines which replica to read from.
Options:

- primary
- primaryPreferred
- secondary
- secondaryPreferred
- nearest

Use secondary reads for analytics.
Use primary for strong consistency.

🔟 What Is Eventual Consistency?

Eventual consistency means:
Secondary nodes may not immediately reflect latest writes.
Given time, they become consistent.
In distributed systems:
Trade-off between availability and consistency.

1️⃣1️⃣ How MongoDB Ensures Durability?

Mongo uses:

- Journaling
- Write-ahead logging
- Replica sets

When write is acknowledged with majority:
 Data is persisted across nodes.

1️⃣2️⃣ What Happens If Transaction Fails?

If error occurs:

- Transaction is aborted
- No changes are committed
- Client receives error

Developer must retry or handle failure.

1️⃣3️⃣ How Do You Retry Transactions?

Use retryable writes.
Wrap transaction in retry logic:

- Catch transient errors
- Retry with backoff
- Ensure idempotency

Mongo drivers support retryable writes.

1️⃣4️⃣ What Is Optimistic Concurrency?

Optimistic concurrency assumes conflicts are rare.
Strategy:

- Include version field
- Update only if version matches
- Increment version

Example:
updateOne({ _id: 1, version: 2 }, { $set: {...}, $inc: { version: 1 } });

Prevents lost updates.

1️⃣5️⃣ How Do You Handle Race Conditions?

Strategies:

- Atomic update operators ($inc, $push)
- Transactions
- Optimistic locking
- Proper indexing
- Application-level locking

Avoid read-modify-write patterns without protection.

1️⃣6️⃣ How Does MongoDB Handle Locks?

Mongo uses:

- Document-level locking (WiredTiger)
- Intent locks for collection

Older versions had collection-level locking.
Modern Mongo supports document-level concurrency.

1️⃣7️⃣ Collection-Level vs Document-Level Locking?

Collection-level locking:

- Entire collection locked
- Poor concurrency

Document-level locking:

- Only affected documents locked
- Higher throughput
- Better concurrency

Modern Mongo uses document-level.

1️⃣8️⃣ How Do You Ensure Idempotency?

Idempotency ensures repeated operation yields same result.

Techniques:

- Unique request IDs
- Upserts
- Conditional updates
- Idempotency keys (payments)

Example:
Store transactionId and prevent duplicate processing.

1️⃣9️⃣ Explain a Consistency Issue You Faced

Strong example:
"In a wallet system, users reported incorrect balances.
Root cause:
Two concurrent requests reading same balance and updating separately.
Fix:
Used atomic $inc instead of read-modify-write.
Later implemented transactions for transfer flow.
Result:
Race condition resolved."
Shows real concurrency awareness.

2️⃣0️⃣ When NOT to Use Transactions?

Avoid when:

- Single document update
- High-performance critical path
- Simple CRUD
- Large-scale bulk writes
- High throughput systems

Transactions add overhead.
Use only when required.

SECTION 6: Replication, Sharding & Scaling (6–10 Years)

1️⃣ What is Replication?

Replication is the process of copying data across multiple servers.
Purpose:

- High availability
- Fault tolerance
- Disaster recovery
- Read scalability

If one server fails, another can take over.

2️⃣ What is Replica Set?

Replica set is a group of MongoDB servers maintaining the same data.
Typically consists of:

- 1 Primary
- Multiple Secondaries
- Optional Arbiter

Replica set ensures automatic failover.

3️⃣ How Failover Works in MongoDB?

When primary goes down:

1. Heartbeat failure detected
2. Election process begins
3. One secondary becomes new primary
4. Clients reconnect automatically

Election usually completes within seconds.

4️⃣ What is Primary, Secondary, Arbiter?

Primary

- Accepts writes
- Replicates changes to secondaries

Secondary

- Replicates data from primary
- Can serve reads (depending on read preference)

Arbiter

- Participates in election
- Does NOT store data
- Used to maintain odd number of votes

5️⃣ What Happens When Primary Goes Down?

- Election triggered
- One secondary promoted
- Temporary write downtime
- Reads may continue (depending on config)

Application should handle transient errors.

6️⃣ How Replication Affects Reads?

Using read preference:

- primary → strong consistency
- secondary → eventual consistency

Reading from secondary improves scalability but may return stale data.

7️⃣ What is Oplog?

Oplog (Operations Log) is a capped collection in primary.
It stores:

- All write operations

Secondaries read oplog and replicate operations.
If secondary falls behind beyond oplog window:
 It must resync fully.

8️⃣ What is Sharding?

Sharding distributes data across multiple machines.
Used for:

- Horizontal scaling
- Handling massive datasets
- Scaling write throughput

Each shard holds subset of data.

9️⃣ Why Sharding is Needed?

Reasons:

- Dataset too large for single server
- Write throughput exceeds single machine
- Memory limitations
- Geographical distribution

Sharding solves storage and write scaling issues.

🔟 How Does MongoDB Distribute Data?

Mongo uses:

- Shard key
- Chunk ranges

Data is partitioned based on shard key value.
Each shard stores specific chunk ranges.

1️⃣1️⃣ What is Shard Key?

Shard key determines how data is distributed.
Example:
{ userId: 1 }

Data is split based on userId value.
Choosing shard key is critical.

1️⃣2️⃣ How Do You Choose a Shard Key?

Good shard key:

- High cardinality
- Even distribution
- Frequently used in queries
- Avoids hotspots
- Immutable

Example good shard key:
userId in large multi-user system.

Performance, Security & Production (8–12+ Years)

1️⃣ How Do You Monitor MongoDB in Production?

Monitoring must cover:

- CPU usage
- Memory usage
- Disk I/O
- Oplog replication lag

- Query latency
- Index usage
- Slow queries
- Connections
- Lock percentages

Tools:

- MongoDB Atlas monitoring
- mongostat
- mongotop
- Prometheus + Grafana
- Datadog / New Relic

You should always monitor:

- Replication lag
- Cache eviction rate
- Page faults
- Slow query logs

Monitoring is proactive, not reactive.

2️⃣What Causes Slow MongoDB Performance?

Common causes:

- Missing indexes
- Poor shard key
- Large $lookup
- Unbounded arrays
- Excessive writes
- Large working set not fitting in RAM

- Lock contention
- Disk bottleneck
- Poor schema design

Most performance issues trace back to bad modeling or indexing.

3️⃣ How Do You Handle MongoDB Memory?

MongoDB uses RAM primarily for:

- WiredTiger cache
- Indexes
- Working set

Best practices:

- Keep working set in memory
- Avoid unnecessary indexes
- Monitor cache eviction
- Scale RAM if dataset grows
- Use sharding to distribute load

If working set > RAM → performance drops drastically.

4️⃣ What is WiredTiger?

WiredTiger is MongoDB's default storage engine.
Features:

- Document-level locking
- Compression
- Journaling
- Efficient caching
- Snapshot isolation

Replaced older MMAPv1 engine.

Improves concurrency and performance.

5️⃣ How Does Compression Work?

WiredTiger supports compression:

- Snappy (default)
- Zlib
- Zstd

Compression:

- Reduces disk usage
- Reduces I/O
- Improves performance

Trade-off:
More CPU usage for compression/decompression.
Most production systems use Snappy.

6️⃣ How Do You Secure MongoDB?

Security layers:

1. Enable authentication
2. Use role-based access control
3. Restrict network access
4. Use TLS encryption
5. Disable public exposure
6. Enable audit logs
7. Use IP whitelisting
8. Regularly update Mongo version

Never expose MongoDB directly to internet.

7️⃣ What is Authentication in MongoDB?

Authentication verifies identity.
Methods:

- SCRAM (username/password)
- X.509 certificates
- LDAP
- Kerberos

Without authentication:
 Anyone can access data.
Always enable auth in production.

SECTION 1: Express.js Fundamentals (0–1 Years)

1️⃣ What is Express.js?

Express.js is a minimal and flexible web framework built on top of Node.js.
It helps you:

- Build APIs
- Handle HTTP requests
- Manage routing
- Use middleware
- Structure backend apps easily

Node provides runtime.
 Express provides web framework.

2️⃣ Why is Express Called a Minimal Framework?

Express is minimal because:

- It does not enforce structure

- No built-in ORM
- No built-in authentication
- No built-in validation
- Only core routing & middleware

You build what you need.
It gives flexibility.

3️⃣ How Express Works Internally?

Flow:

1. Client sends HTTP request
2. Express receives request
3. Request passes through middleware chain
4. Route handler executes
5. Response sent back

Internally, Express:

- Wraps Node’s http module
- Uses middleware stack
- Matches routes based on method & path

It is essentially a layered request handler.

4️⃣ What is an Express Application?

An Express app is created using:
const express = require("express");
const app = express();

app represents:

- Entire web server
- Entry point for routing

- Middleware manager

It handles all incoming requests.

5️⃣ What is Middleware?

Middleware is a function that has access to:

- req (request)
- res (response)
- next (next function)

It runs before the final route handler.
Example:

```
app.use((req, res, next) => {
  console.log("Request received");
  next();
});
```

Middleware can:

- Modify request
- Validate data
- Authenticate users
- Log requests

6️⃣ Types of Middleware in Express?

1. Application-level middleware
2. Router-level middleware
3. Built-in middleware
4. Error-handling middleware
5. Third-party middleware

Each serves different purpose.

7️⃣ What is req and res?

req → Represents incoming request
res → Represents outgoing response
req contains:

- Body
- Params
- Query
- Headers

res is used to:

- Send JSON
- Send status
- Send file
- End response

SECTION 2: Middleware, Routing & Request Lifecycle (1–3 Years)

1️⃣ Explain Express Request Lifecycle

Request lifecycle:

1. Client sends HTTP request
2. Express receives request
3. Request passes through middleware stack (in order registered)
4. Matching route handler executes
5. Response is sent
6. If error occurs → error middleware executes

Express processes middleware sequentially.
Flow:

Incoming Request → Middleware 1 → Middleware 2 → Route → Response → End

---

2️⃣ Order of Middleware Execution?

Middleware runs:

- In the order it is defined
- Top to bottom

Example:
app.use(m1);
app.use(m2);
app.get("/users", handler);

Execution order:
m1 → m2 → handler
Order matters significantly.

3️⃣ What is Application-Level Middleware?

Middleware attached to app object.
Example:
app.use((req, res, next) => { next(); });

Applies to all routes unless path specified.
Used for:

- Logging
- JSON parsing
- Authentication

4️⃣ What is Router-Level Middleware?

Middleware attached to router instance.
Example:
const router = express.Router();
router.use(authMiddleware);

Applies only to routes in that router.
Improves modularity.

5️⃣ What is Error-Handling Middleware?

Middleware specifically designed to handle errors.
Example:

```
app.use((err, req, res, next) => {
  res.status(500).json({ message: "Server error" });
});
```

Must be defined after all routes.

SECTION 3: REST API Design & Best Practices (2–5 Years)

1️⃣ What Are REST Principles?

REST principles include:

1. Stateless communication
2. Resource-based architecture
3. Uniform interface
4. Use of HTTP methods
5. Standard status codes
6. Layered system
7. Cacheability

REST is an architectural style, not a strict protocol.

2️⃣ What Makes an API Truly RESTful?

An API is RESTful when:

- URLs represent resources, not actions
- Uses proper HTTP methods
- Uses proper status codes

- Stateless
- Consistent structure
- Supports caching

Example good:
GET /users
POST /users
DELETE /users/123
Bad:
GET /deleteUser?id=123

Performance, Scaling & Production (5–9 Years)

1️⃣ How Do You Improve Express API Performance?

Steps:

1. Profile before optimizing
2. Avoid blocking code
3. Add proper DB indexes
4. Use caching
5. Reduce response size
6. Enable compression
7. Optimize middleware chain
8. Use connection pooling
9. Scale horizontally

Performance tuning is data-driven, not guesswork.

2️⃣ What Causes Slow Express APIs?

Common causes:

- Slow database queries
- Missing indexes
- Blocking synchronous code
- Large payloads
- Excessive middleware
- Memory pressure
- Network latency
- Poor pagination

Most slowness is DB-related.

3️⃣ How Do You Profile Express Apps?

Tools:

- Node --inspect
- Chrome DevTools
- Clinic.js
- APM tools (New Relic, Datadog)
- Logging slow queries

Measure:

- Response time
- Event loop lag
- Memory usage
- CPU usage

Always measure before optimizing.