Senior JS / TS / React / Node Interview Prep

Free preparation kit • Edition 1

A practical revision guide for a three- or four-round technical loop. Interview formats vary: confirm the actual scope with your interviewer. This guide is educational, not a promise of interview or hiring outcomes. All design scenarios are hypothetical exercises, not career achievements.

How to use this kit

1. Review the round map and identify the skills being assessed.
2. Implement one coding exercise without looking at the solution.
3. Build one timed mini-app, including loading and failure states.
4. Sketch a system design and explain two tradeoffs aloud.
5. Rehearse only your own truthful, non-confidential experience.

A possible round map

• Screen: role fit, communication, ownership and honest skill boundaries.

• Live coding: problem solving, JS/TS fundamentals, React and tests.

• Machine coding: a small working app with typed models and usable UI.

• System design: requirements, architecture, failure and tradeoffs.

Some loops combine rounds or add behavioral and frontend deep dives. A working-app exercise differs from architecture discussion. Ask what is expected rather than treating HLD, LLD and machine coding as synonyms.

Screen preparation

Structure your introduction: current strengths, one relevant project, your contribution, a decision and what you learned. Use measured results only when you can substantiate and disclose them. Do not borrow examples from a guide as personal experience. State unfamiliar stacks honestly.

Ask: What outcomes define success? How does the team review and ship? What balance of frontend, backend and architecture does this role need?

Live coding: make the reasoning visible

A repeatable approach

• Clarify inputs, outputs, constraints, runtime and allowed references.

• Work through normal, empty, duplicate, invalid and slow-network cases.

• Explain the data model and a simple approach before typing.

• Build a running happy path, then address failures and accessibility.

• Reserve time for tests, complexity and production limitations.

JavaScript and TypeScript checklist

Closures; lexical scope; reference equality; immutable updates; Map and Set; debounce versus throttle; promises; cancellation; array transforms. Practice narrowing a discriminated union rather than using loose any.

```
type Job =
  | { status: "queued" }
  | { status: "running"; progress: number }
  | { status: "failed"; reason: string }
  | { status: "succeeded"; result: string };

function label(job: Job): string {
  switch (job.status) {
    case "queued": return "Waiting";
    case "running": return `${job.progress}%`;
    case "failed": return job.reason;
    case "succeeded": return job.result;
  }
}
```

React checklist

Keep state minimal; compute derived values. Use stable list IDs when items can reorder. Clean up subscriptions and requests in effects. Use reducers for linked transitions. Memoize only for a measured cost; useMemo is an optimization, not a correctness guarantee.

Handle loading, empty, error and retry states explicitly. Include labels, keyboard behavior and visible focus. A modal needs focus management and focus restoration. Error boundaries do not catch every asynchronous error.

Practice prompts

Typeahead: debounce, abort old requests, guard against stale responses, cache deliberately and support keyboard selection. Abort alone does not make every asynchronous continuation safe; check current request identity.

Virtualized table: fixed-height window plus overscan; stable IDs; sorting and selection. Discuss dynamic heights and accessible navigation.

Nested comments: tree model, recursive UI, collapsed-ID set and rollback on failed optimistic creation. Infinite scroll: cursor, dedupe and retry.

Machine coding: deliver a small working app

Suggested 90-minute practice allocation

• 0–10: restate requirements, entities and scope boundaries.

• 10–25: model and operations; validate the core logic.

• 25–70: build one usable flow, not multiple unfinished screens.

• 70–85: validation, disabled controls, loading and failure states.

• 85–90: demo, tests, remaining risks and next steps.

Adjust the allocation to the actual session duration and rubric.

Choose one hypothetical exercise

Job board: queued/running/succeeded/failed steps; reason on failure; retry eligible steps without discarding completed work. Discuss how backend idempotency prevents duplicated side effects.

Data table: search, sort, selection and a large synthetic dataset. Measure before adding virtualization. Avoid storing duplicated totals.

Component playground: typed button, input and modal variants; shared tokens; keyboard and focus behavior. Keep abstractions easy to read.

Search widget: recent queries, keyboard selection, cancellation and a retry action. Test an older response arriving after a newer response.

Definition of a useful submission

• The primary flow runs and is easy to demonstrate.

• State transitions and validation rules are explicit.

• A double click cannot silently create duplicate local operations.

• Server-side side effects still require idempotency and authorization.

• At least one normal and one failure case are tested.

• You can explain what is deliberately out of scope.

If the round is algorithmic

Practice hash maps, sliding windows, two pointers, stacks, intervals, tree traversal, heaps and LRU cache design. State time and space costs, then check edge cases against the implementation. Do not hide uncertainty behind an unsupported complexity claim.

System design: scope, tradeoffs and failure

Suggested 45-minute discussion

- 5 minutes: functional and non-functional requirements.
- 2 minutes: estimates with explicitly stated assumptions.
- 8 minutes: clients, APIs, stores, queues and data flow.
- 15 minutes: one deep dive chosen with the interviewer.
- 10 minutes: failure handling, scaling and observability.
- 5 minutes: alternatives, tradeoffs and questions.

These are practice allocations, not a universal interview format.

Hypothetical designs

Live widget: isolate styles where appropriate, use short-lived session tokens and an explicit connection state machine. Handle permissions, reconnect, cleanup and tenant isolation. Shadow DOM isolates styles; it is not a security boundary. Separate session control from media flow.

Server-driven UI: version schemas, validate payloads, restrict allowed components and provide fallbacks. Compare flexibility with compatibility, accessibility, caching and rollout complexity.

Analytics dashboard: paginate data, virtualize rendering if justified, isolate expensive charts and avoid duplicated derived aggregates. Network payload and DOM work are separate bottlenecks.

Async export: API stores a durable job; workers process bounded steps; UI observes status via polling or events. Persist outputs and failure reasons. Distinguish transient failure, permanent failure and lost worker.

Payment-event processing: verify signatures on the raw request body, durably record events, dedupe and apply ledger changes transactionally. Plan reconciliation and replay. An event ID alone does not dedupe two different events representing the same business operation.

Senior-level discussion prompts

What happens if the worker crashes after a provider succeeds but before local commit? How do you reconcile an unknown result? Which data is the source of truth? What do retries cost? What is safe to log? How will you roll back a bad release? Which simpler design did you reject, and why?

Use hypothetical estimates as assumptions, never as claimed experience.

Promises and scheduling

A promise settles once: pending to fulfilled or rejected. Resolving to a thenable can leave it pending until that thenable settles. A then call returns a new promise; values fulfill it, thrown errors reject it, and returned thenables are assimilated.

Calling asynchronous operations starts work; Promise.all aggregates results. It does not itself start tasks, cancel siblings or create threads. Sequential await is appropriate when later work depends on earlier work.

Combinators

• all: input-order results; rejects on a rejection; empty input fulfills [].

• allSettled: settlement records; empty input fulfills [].

• race: adopts the first settlement; empty input stays pending.

• any: first fulfillment; all rejected or empty rejects with AggregateError.

These settlement summaries assume valid iteration. Iteration errors can reject combinator promises, including allSettled.

Browser or ordinary top-level CommonJS exercise

```
console.log("A");
setTimeout(() => console.log("B"), 0);
Promise.resolve().then(() => console.log("C"));
console.log("D");
// A D C B
```

Synchronous work completes first; promise reactions run as microtasks before the next timer task. Do not generalize this to every Node callback ordering or module context.

Bounded immediate retry exercise

```
async function retry(fn, attempts = 3) {
  if (!Number.isInteger(attempts) || attempts < 1) {
    throw new RangeError("attempts must be positive");
  }
  for (let i = 0; i < attempts; i++) {
    try { return await fn(); }
    catch (error) {
      if (i === attempts - 1) throw error;
    }
  }
}
```

Exercise only: no backoff, jitter, cancellation or error classification. Production retries need deadlines, retryability rules and observability. Side effects need durable idempotency or reconciliation; retry is not an exactly-once guarantee.

Polyfill-style interview exercises

A polyfill supplies a missing runtime API; a ponyfill is a standalone helper without global patching. Syntax transformation is different from providing a missing method. The helpers here do not patch native globals. They are interview exercises, not spec-complete polyfills.

Aggregate promises in input order

```
function promiseAll(items) {
  return new Promise((resolve, reject) => {
    const list = Array.from(items);
    const out = new Array(list.length);
    let left = list.length;
    if (left === 0) return resolve([]);
    list.forEach((item, i) => {
      Promise.resolve(item).then(value => {
        out[i] = value;
        if (--left === 0) resolve(out);
      }, reject);
    });
  });
}
```

Preserves result order and handles values, rejection and empty input. Not native-equivalent: Array.from accepts array-like objects beyond the native iterable contract; subclassing and iterator semantics differ.

Ordinary-call binding only

```
function bindCall(fn, thisArg, ...bound) {
  if (typeof fn !== "function") {
    throw new TypeError("expected a function");
  }
  return function (...args) {
    return fn.apply(thisArg, [...bound, ...args]);
  };
}
```

This models receiver and partial arguments for ordinary calls. It does not implement native bound-function construction, prototype behavior, name or length semantics. Do not use it as a native bind replacement.

Array mapping and debounce exercises

Map a normal array, preserving holes

```
function mapArray(array, callback, thisArg) {
  if (!Array.isArray(array)) {
    throw new TypeError("expected an array");
  }
  if (typeof callback !== "function") {
    throw new TypeError("expected a callback");
  }
  const length = array.length;
  const out = new Array(length);
  for (let i = 0; i < length; i++) {
    if (i in array) {
      out[i] = callback.call(thisArg, array[i], i, array);
    }
  }
  return out;
}
```

The original length is captured and trailing holes are preserved. This exercise deliberately supports arrays only, not the native generic array-like contract or Array species/subclass semantics.

Trailing-only debounce

```
function debounce(fn, wait) {
  let timer;
  return function (...args) {
    clearTimeout(timer);
    timer = setTimeout(() => fn.apply(this, args), wait);
  };
}
```

Keeps the last receiver and arguments. It has no cancel, flush, maxWait or leading-edge option and does not return the eventual function result. In React, keep a stable debounced instance when needed and provide cancellation on unmount in a production implementation.

Throttle limits execution rate; debounce waits for a quiet interval. Always clarify leading/trailing behavior and test with controlled timers.

Node.js and reliable queues

Runtime discussion

JavaScript usually runs on one event-loop thread per Node isolate. Some filesystem, DNS and crypto operations use libuv's worker pool; network I/O is generally coordinated through operating-system facilities. CPU-heavy JavaScript and large synchronous parsing still block the loop. Move heavy compute to worker threads, processes or a separate service.

Know timers, poll, check and close callbacks, but avoid a universal phase-order slogan. Timer behavior depends on runtime version and execution context. setImmediate runs in the check phase. Recursive process.nextTick can starve I/O. CommonJS and ES module evaluation can differ in nextTick versus promise ordering. Measure the actual scenario.

Use streams and respect backpressure for large payloads. Handle errors, cleanup, timeouts and child-process exit codes. Bound subprocess output and avoid building shell commands from untrusted input.

Queue vocabulary

- At-least-once: delivery may repeat, so handlers must tolerate duplicates.
- Idempotency: repeating the same logical operation has one intended effect.
- Lease or visibility timeout: work may be redelivered after expiry.
- Backoff and jitter: spread retries; set attempt and deadline limits.
- Dead letter: retain exhausted work with a reason and a replay procedure.
- Concurrency: cap in-flight work against downstream and memory limits.
- Ordering: define whether ordering is global, per key or unnecessary.

A lock reduces simultaneous claims; it does not eliminate duplicate processing after crashes or lease expiry. In a database-backed queue, SKIP LOCKED skips rows locked by other transactions; correctness still needs transactional claiming and safe state transitions.

Hypothetical worker flow

Persist job -> claim with lease -> process step -> persist output -> acknowledge. Renew leases when appropriate. On uncertain external success, reconcile using provider identifiers before retrying the effect.

Compare a dedicated queue with a database-backed work ledger using throughput, durability configuration, transaction needs and operational cost. Neither technology choice automatically guarantees no data loss.

Network delivery and final revision

Optimize measured bottlenecks

Fewer bytes: route-level splitting, tree shaking, compression, correctly sized media and immutable caching for content-hashed public assets.

Fewer round trips: batch when appropriate, avoid dependent waterfalls, and prefetch only when likely useful. HTTP/2 multiplexes streams but can still suffer TCP head-of-line blocking; HTTP/3 uses QUIC streams. Protocol improvements do not fix excessive payloads or server latency.

Request correctness: debounce search; abort obsolete work; guard stale responses. Use cursor pagination where appropriate. Keep earlier pages visible when loading another page fails.

Caching: use Cache-Control and validators such as ETag deliberately. Separate public cacheable assets from personalized responses. Verify cache keys, authorization boundaries and invalidation rules.

Realtime: compare polling, server-sent events and WebSockets against latency, directionality, reconnection and infrastructure needs. On reconnect, recover missed events and avoid duplicate session creation.

Retries: exponential backoff, jitter and a deadline. A timeout does not prove the server did nothing. Do not repeat unsafe effects blindly.

Render cost: virtualization reduces mounted UI, not bytes transferred. Use profiling to distinguish network latency from main-thread work.

Twenty-minute revision drill

- 3 minutes: explain your own truthful introduction and one decision.
- 5 minutes: implement debounce; explain stale-response protection.
- 8 minutes: sketch a job pipeline, one failure and recovery path.
- 4 minutes: compare two architecture options and ask two questions.

Self-check before the interview

Can you explain Promise.all versus sequential await? Preserve array holes? Model mutually exclusive UI states? Test an out-of-order response? Explain why a queue lease is not exactly-once? Design retry-safe effects? Separate network optimization from rendering optimization?

Keep answers specific, honest and scoped. A simple design with explicit failure handling is more useful than unsupported technology name-dropping.