FrontEnd Developers

Interview Questions and Answers

Core Concepts

This section focuses on fundamental principles and advanced concepts that an experienced developer should master.

---

1. Explain the Virtual DOM diffing algorithm and how React optimizes reconciliation. What are the key heuristics used?

Virtual DOM diffing is React's process of comparing the new virtual DOM tree with the previous one to determine minimal DOM updates.

Key Heuristics:

- **Tree Level Comparison:** React compares trees level by level, not recursively across all nodes
- **Component Type Matching:** Different component types produce different trees; React will unmount the old tree and mount a new one
- **Keys for List Items:** Keys help React identify which items have changed, been added, or removed
- **Same-Level Siblings:** React only compares siblings at the same level

Optimization Strategies:

- **Batching:** Multiple state updates are batched into a single re-render
- **Fiber Architecture:** Enables incremental rendering, splitting work into chunks
- **shouldComponentUpdate/React.memo:** Allows developers to prevent unnecessary re-renders

```
// Example of optimization
const MemoizedComponent = React.memo(({ data }) => {
  return
{data.name}
;
}, (prevProps, nextProps) => {
  return prevProps.data.id === nextProps.data.id;
});
```

This approach makes React efficient by avoiding expensive full-tree comparisons and minimizing actual DOM manipulations.

2. What is the difference between microtasks and macrotasks in the JavaScript event loop? Provide examples of each.

Microtasks and macrotasks represent different priority queues in JavaScript's event loop, determining execution order of asynchronous operations.

Macrotasks (Task Queue):

- setTimeout, setInterval
- setImmediate (Node.js)
- I/O operations
- UI rendering

Microtasks (Job Queue):

- Promises (.then, .catch, .finally)
- queueMicrotask()
- MutationObserver
- process.nextTick (Node.js - higher priority)

Execution Order:

All microtasks are executed before the next macrotask.

```
console.log('1');
setTimeout(() => console.log('2'), 0);
Promise.resolve().then(() => console.log('3'));
queueMicrotask(() => console.log('4'));
console.log('5');
// Output: 1, 5, 3, 4, 2
```

The event loop processes: synchronous code → all microtasks → one macrotask → all microtasks → next macrotask. This is critical for understanding async behavior and avoiding race conditions in complex applications.

3. Explain Closure in JavaScript and provide a practical use case where closures solve a real-world problem.

A closure is a function that retains access to variables from its outer (enclosing) lexical scope, even after the outer function has finished executing.

How Closures Work:

When a function is created, it maintains a reference to its lexical environment. This allows inner functions to access outer variables even after the outer function returns.

Practical Use Case - Data Privacy/Encapsulation:

```
function createCounter() {
  let count = 0; // private variable
  return {
    increment: () => ++count,
    decrement: () => --count,
    getCount: () => count
  };
}
const counter = createCounter();
counter.increment(); // 1
counter.getCount(); // 1
```

Real-World Applications:

- **Module Pattern:** Creating private variables and methods
- **Event Handlers:** Maintaining state across multiple event calls
- **Callbacks:** Preserving context in asynchronous operations
- **Function Factories:** Creating specialized functions with preset parameters
- **Memoization:** Caching expensive function results

Closures enable powerful patterns like the revealing module pattern and are fundamental to functional programming in JavaScript.

4. What are Web Vitals and how do you optimize for Core Web Vitals (LCP, FID, CLS)? Provide specific techniques for each.

Core Web Vitals are Google's metrics for measuring user experience, directly impacting SEO and user satisfaction.

The Three Core Web Vitals:

- **LCP (Largest Contentful Paint):** Loading performance - should occur within 2.5s
- **FID (First Input Delay):** Interactivity - should be less than 100ms
- **CLS (Cumulative Layout Shift):** Visual stability - should be less than 0.1

LCP Optimization:

- Optimize and compress images (WebP, AVIF formats)
- Implement lazy loading for below-fold content
- Use CDN for static assets
- Preload critical resources with <link rel="preload">
- Minimize render-blocking JavaScript and CSS

- Use server-side rendering or static generation

FID Optimization:

- Break up long JavaScript tasks (use code splitting)
- Implement web workers for heavy computations
- Reduce JavaScript execution time
- Use requestIdleCallback for non-critical work

CLS Optimization:

- Always include size attributes on images and videos
- Reserve space for ad slots and embeds
- Avoid inserting content above existing content
- Use transform animations instead of properties that trigger layout

```
// Example: Reserving space for images
<img src="hero.jpg"
    width="800"
    height="600"
    alt="Hero" />
```

5. Explain the differences between CSR, SSR, SSG, and ISR. When would you choose each rendering strategy?

Modern web applications use different rendering strategies, each with specific trade-offs for performance, SEO, and user experience.

Client-Side Rendering (CSR):

- JavaScript renders content in the browser
- **Pros:** Rich interactions, reduced server load
- **Cons:** Slower initial load, poor SEO without hydration
- **Use case:** Dashboards, admin panels, authenticated apps

Server-Side Rendering (SSR):

- HTML generated on each request
- **Pros:** Better SEO, faster first contentful paint
- **Cons:** Higher server load, slower TTFB
- **Use case:** Dynamic content, personalized pages, social media feeds

Static Site Generation (SSG):

- HTML generated at build time
- **Pros:** Fastest performance, excellent SEO, low server cost
- **Cons:** Requires rebuild for updates, not suitable for dynamic content
- **Use case:** Blogs, documentation, marketing sites, portfolios

Incremental Static Regeneration (ISR):

- Static pages regenerated in background after deployment
- **Pros:** Combines SSG speed with SSR freshness
- **Cons:** Complex cache invalidation, potential stale content
- **Use case:** E-commerce product pages, news sites, content with periodic updates

```
// Next.js ISR example
export async function getStaticProps() {
  const data = await fetchData();
  return {
    props: { data },
    revalidate: 60 // Regenerate every 60s
  };
}
```

6. How does the CSS containment property work, and how can it improve rendering performance?

CSS Containment allows developers to isolate parts of the page, telling the browser that an element's subtree is independent from the rest of the page.

Containment Types:

- **layout:** Element's internal layout doesn't affect external elements
- **paint:** Element's descendants won't display outside its bounds
- **size:** Element's size can be calculated without examining descendants
- **style:** Properties that can affect more than just descendants are contained

Usage:

```
.container {
  contain: layout paint;
}
.strict-container {
  contain: strict; /* layout + paint + size */
}
.content-container {
  contain: content; /* layout + paint + style */
}
```

Performance Benefits:

- **Reduced Reflow Scope:** Changes inside contained elements don't trigger layout recalculation for the entire page
- **Optimized Rendering:** Browser can skip rendering off-screen contained elements
- **Parallel Processing:** Contained elements can be rendered independently
- **Better Virtualization:** Essential for infinite scroll and virtual lists

Practical Use Cases:

- Card components in a grid layout
- List items in virtual scrolling
- Widgets and embeds
- Third-party content containers

Combined with **content-visibility: auto**, containment dramatically improves rendering performance for complex layouts.

7. Explain how React's useEffect hook works internally. What is the cleanup function and when is it called?

useEffect is React's hook for handling side effects in functional components, running after the render is committed to the screen.

Internal Mechanism:

- Effects are stored in a queue during render phase
- After DOM mutations are committed, React runs all effects
- Effects run asynchronously to avoid blocking browser paint
- Dependencies are compared using Object.is() for changes

Cleanup Function:

The cleanup function is returned from useEffect and is called:

- Before running the effect on subsequent renders (if dependencies changed)
- When the component unmounts
- This prevents memory leaks and stale subscriptions

```
useEffect(() => {
  const subscription = api.subscribe(data);

  // Cleanup function
  return () => {
    subscription.unsubscribe();
  };
```

}, [api, data]);

Common Use Cases for Cleanup:

- **Event listeners:** Remove listeners to prevent memory leaks
- **Subscriptions:** Unsubscribe from data sources
- **Timers:** Clear setTimeout/setInterval
- **Async operations:** Cancel pending requests
- **WebSocket connections:** Close connections

Execution Order:

1. Component renders → 2. DOM updates → 3. Browser paints → 4. Cleanup from previous effect runs → 5. New effect runs. Understanding this flow is critical for avoiding race conditions and memory leaks.

8. What is Tree Shaking and how does it work? What are the requirements for code to be tree-shakeable?

Tree shaking is a dead-code elimination technique that removes unused exports from JavaScript bundles, significantly reducing bundle size.

How It Works:

- Relies on ES6 module static structure (import/export)
- Build tools (Webpack, Rollup) analyze dependency graph
- Mark unused exports during bundling
- Minifiers remove marked dead code

Requirements for Tree-Shakeable Code:

- **Use ES6 modules:** import/export, not require/module.exports
- **Avoid side effects:** Pure modules without global state changes
- **Mark side effects:** Use "sideEffects": false in package.json
- **Named exports:** Prefer named exports over default exports
- **Static imports:** Avoid dynamic import() for tree-shakeable code

```
// Tree-shakeable (named exports)
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;

// Import only what's needed
import { add } from './math';
// subtract is eliminated from bundle
```

Common Pitfalls:

- CommonJS modules prevent tree shaking
- Side effects in module initialization
- Class methods are harder to tree shake
- Barrel exports can hinder optimization

Verification:

Use webpack-bundle-analyzer or source-map-explorer to verify tree shaking effectiveness in your bundles.

9. Explain the concept of Hydration in modern frameworks. What problems can occur during hydration and how do you solve them?

Hydration is the process where client-side JavaScript attaches event listeners and state to server-rendered HTML, making it interactive.

How Hydration Works:

- Server sends fully rendered HTML (fast FCP)
- Browser displays static HTML immediately
- JavaScript bundle downloads and executes

- Framework matches virtual DOM to existing DOM
- Event listeners and state are attached

Common Hydration Problems:

- **Hydration Mismatch:** Server and client render different content
- **Flash of Unstyled Content:** CSS loads after HTML
- **Large TTI Gap:** Long delay between FCP and interactivity
- **Double Data Fetching:** Fetching same data on server and client

Solutions:

```
// 1. Serialize server data
<script>
  window.__INITIAL_DATA__ = ${JSON.stringify(data)};
</script>

// 2. Use consistent rendering logic
const App = ({ data }) => {
  const date = useMemo(() => data.date, []);
  return <div>{date}</div>;
};
```

Best Practices:

- **Avoid browser-only APIs:** Check typeof window before using
- **Use suppressHydrationWarning:** For unavoidable mismatches like timestamps
- **Progressive Hydration:** Hydrate components as they become visible
- **Selective Hydration:** Only hydrate interactive components
- **Streaming SSR:** Send HTML in chunks for faster TTFB

Modern frameworks like Next.js and Remix provide built-in solutions for these challenges with streaming and selective hydration.

10. What are JavaScript Generators and how can they be used to handle asynchronous operations? Provide a practical example.

Generators are functions that can pause execution and resume later, yielding multiple values over time using the function* syntax.

Key Characteristics:

- Use function* syntax and yield keyword
- Return an iterator object
- Maintain state between yields
- Can receive values via .next(value)
- Enable lazy evaluation

Basic Syntax:

```
function* numberGenerator() {
  yield 1;
  yield 2;
  return 3;
}
const gen = numberGenerator();
gen.next(); // {value: 1, done: false}
gen.next(); // {value: 2, done: false}
gen.next(); // {value: 3, done: true}
```

Async Operations with Generators:

Generators formed the basis for async/await and can still be useful for complex async flows:

```
function* fetchUserData() {
  const user = yield fetch('/api/user');
  const posts = yield fetch(`/api/posts/${user.id}`);
  return { user, posts };
```

```
}

function runGenerator(gen) {
  const iter = gen();
  function handle(result) {
    if (result.done) return result.value;
    return result.value.then(
      res => handle(iter.next(res))
    );
  }
  return handle(iter.next());
}
```

Practical Use Cases:

- **Infinite sequences:** ID generation, pagination
- **State machines:** Complex workflow management
- **Data streaming:** Processing large datasets
- **Custom iterators:** Traversing complex data structures

Data Structures and Algorithms

Questions in this section test your understanding of how to work with and manipulate data efficiently.

1. Implement a Stack data structure in JavaScript with push, pop, peek, and isEmpty methods. What is the time complexity of each operation?

Stack Implementation

A stack follows the **LIFO (Last In First Out)** principle. Here's a clean implementation:

```
class Stack {
  constructor() { this.items = []; }
  push(element) { this.items.push(element); }
  pop() { return this.items.pop(); }
  peek() { return this.items[this.items.length - 1]; }
  isEmpty() { return this.items.length === 0; }
  size() { return this.items.length; }
}
```

Time Complexity:

- push(): O(1)
- pop(): O(1)
- peek(): O(1)
- isEmpty(): O(1)

2. How would you implement an LRU (Least Recently Used) Cache with O(1) get and put operations?

LRU Cache Implementation

An **LRU Cache** requires a combination of a **Map** (for O(1) lookup) and a doubly linked list concept. JavaScript's Map maintains insertion order, making it perfect:

```
class LRUCache {
  constructor(capacity) {
    this.capacity = capacity;
    this.cache = new Map();
  }
  get(key) {
    if (!this.cache.has(key)) return -1;
    const val = this.cache.get(key);
    this.cache.delete(key);
    this.cache.set(key, val);
    return val;
  }
  put(key, value) {
    this.cache.delete(key);
    this.cache.set(key, value);
    if (this.cache.size > this.capacity) {
      this.cache.delete(this.cache.keys().next().value);
    }
  }
}
```

Time Complexity: Both get() and put() are O(1)

3. Explain the difference between Map, WeakMap, Set, and WeakSet in JavaScript. When would you use each?

Map vs WeakMap vs Set vs WeakSet

Map: Key-value pairs where keys can be any type. Maintains insertion order. Keys are strongly referenced.

WeakMap: Keys must be objects and are **weakly referenced** (can be garbage collected). No iteration methods. Use for private data or caching without memory leaks.

Set: Collection of unique values of any type. Use for deduplication and membership testing.

WeakSet: Only stores objects, weakly referenced. Use for tagging objects without preventing garbage collection.

- Use **Map** for general key-value storage
- Use **WeakMap** for metadata associated with objects
- Use **Set** for unique collections
- Use **WeakSet** for object tagging/tracking

4. Implement a function to find two numbers in an array that sum to a target value. What's the optimal time complexity?

Two Sum Problem

The optimal solution uses a **hash map** to achieve O(n) time complexity:

```
function twoSum(nums, target) {
  const map = new Map();
  for (let i = 0; i < nums.length; i++) {
    const complement = target - nums[i];
    if (map.has(complement)) {
      return [map.get(complement), i];
    }
    map.set(nums[i], i);
  }
  return null;
}
```

Time Complexity: O(n) - single pass through array

Space Complexity: O(n) - hash map storage

This is more efficient than the brute force $O(n^2)$ nested loop approach.

5. Implement a debounce function from scratch. How does it differ from throttle?

Debounce Implementation

Debounce delays function execution until after a specified time has elapsed since the last invocation:

```
function debounce(func, delay) {
  let timeoutId;
  return function(...args) {
    clearTimeout(timeoutId);
    timeoutId = setTimeout(() => {
      func.apply(this, args);
    }, delay);
  };
}
```

Debounce vs Throttle:

- **Debounce:** Executes after the event stops firing for X ms (e.g., search input)
- **Throttle:** Executes at most once every X ms regardless of how many times triggered (e.g., scroll events)

Use debounce for user input, throttle for high-frequency events.

6. What is the time complexity of common JavaScript array methods like push, pop, shift, unshift, and splice?

Array Method Time Complexities

Understanding these complexities is crucial for performance optimization:

- **push():** O(1) - adds to end
- **pop():** O(1) - removes from end
- **shift():** O(n) - removes from start, requires reindexing all elements
- **unshift():** O(n) - adds to start, requires reindexing all elements
- **splice():** O(n) - may require shifting elements
- **slice():** O(n) - creates new array
- **concat():** O(n+m) - creates new array

Performance tip: Avoid shift/unshift in loops. Use push/pop or consider a deque implementation for O(1) operations at both ends.

7. Implement a function to find the maximum sum of a sliding window of size k in an array.

Sliding Window Maximum Sum

The **sliding window technique** optimizes this to O(n) instead of O(n*k):

```
function maxSlidingWindow(arr, k) {
  if (arr.length < k) return null;
  let maxSum = 0, windowSum = 0;
  for (let i = 0; i < k; i++) windowSum += arr[i];
  maxSum = windowSum;
  for (let i = k; i < arr.length; i++) {
    windowSum = windowSum - arr[i - k] + arr[i];
    maxSum = Math.max(maxSum, windowSum);
  }
  return maxSum;
}
```

Time Complexity: O(n) - single pass after initial window

Space Complexity: O(1) - constant extra space

8. Explain how JavaScript's Set data structure works internally and when you should prefer it over an array.

JavaScript Set Internals

A **Set** is implemented using a hash table, providing fast operations:

- **add(value):** O(1) average case
- **has(value):** O(1) average case
- **delete(value):** O(1) average case
- **size:** O(1)

Use Set when:

- You need to ensure **uniqueness** of values
- You need **fast membership testing** (has())
- Order matters but duplicates don't
- Performing set operations (union, intersection, difference)

Use Array when:

- You need indexed access
- Duplicates are meaningful
- You need array methods like map, filter, reduce

9. Implement a function to deep clone an object, handling circular references.

Deep Clone with Circular Reference Handling

A robust deep clone must handle **circular references** to avoid infinite loops:

```
function deepClone(obj, hash = new WeakMap()) {
  if (obj === null || typeof obj !== 'object') return obj;
  if (hash.has(obj)) return hash.get(obj);
  const clone = Array.isArray(obj) ? [] : {};
  hash.set(obj, clone);
  for (let key in obj) {
    if (obj.hasOwnProperty(key)) {
      clone[key] = deepClone(obj[key], hash);
    }
  }
  return clone;
}
```

The **WeakMap** tracks visited objects to detect and handle circular references without preventing garbage collection.

10. What are the time and space complexities of common sorting algorithms, and which does JavaScript's Array.sort() use?

Sorting Algorithm Complexities

Common Algorithms:

- **Quick Sort:** O(n log n) average, O(n²) worst | Space: O(log n)
- **Merge Sort:** O(n log n) all cases | Space: O(n)
- **Heap Sort:** O(n log n) all cases | Space: O(1)
- **Bubble Sort:** O(n²) average/worst | Space: O(1)

JavaScript Array.sort():

V8 engine (Chrome/Node) uses **Timsort** - a hybrid of merge sort and insertion sort:

- Time: O(n log n) worst case
- Space: O(n)
- Stable sort (maintains relative order of equal elements)
- Optimized for real-world data with partial ordering

Note: Always provide a compare function for numeric sorting to avoid lexicographic ordering.

System Design

These questions evaluate your ability to think about the bigger picture, including architecture, scalability, and performance.

1. Design a scalable URL shortener service like bit.ly. What are the key components and how would you handle high traffic?

Key Components & Architecture

- **URL Generation Service:** Use base62 encoding with auto-incrementing IDs or hash-based approach (MD5/SHA256 truncated)
- **Database Design:** NoSQL (Cassandra/DynamoDB) for horizontal scalability with key-value pairs mapping short codes to original URLs
- **Caching Layer:** Redis/Memcached for frequently accessed URLs (80-20 rule applies)
- **Load Balancer:** Distribute requests across multiple application servers
- **CDN:** Cache redirect responses geographically

High Traffic Handling

- **Read-heavy optimization:** Cache-aside pattern with 99%+ cache hit ratio
- **Sharding:** Partition data by hash of short code or range-based sharding
- **Stateless servers:** Enable horizontal scaling without session affinity
- **Rate limiting:** Token bucket algorithm to prevent abuse

Sample Encoding Logic

```
function encodeBase62(num) {
  const chars = '0-9a-zA-Z';
  let result = '';
  while (num > 0) {
    result = chars[num % 62] + result;
    num = Math.floor(num / 62);
  }
  return result;
}
```

Capacity Estimation

- 7 characters base62 = 62^7 = 3.5 trillion URLs
- Storage: 500 bytes/URL × 1B URLs = 500GB
- Bandwidth: 100M daily writes, 10B reads = ~115K QPS reads

2. How would you design a real-time social media feed (like Twitter/Facebook) that scales to millions of users?

Architecture Approaches

- **Fan-out on Write (Push Model):** Pre-compute feeds when posts are created, write to followers' timelines
- **Fan-out on Read (Pull Model):** Compute feed on-demand by querying followees' recent posts
- **Hybrid Approach:** Push for regular users, pull for celebrities with millions of followers

Core Components

- **Post Service:** Handles post creation, stores in distributed database (Cassandra)
- **Feed Generation Service:** Aggregates posts from followees, ranks by algorithm
- **Timeline Cache:** Redis stores pre-computed feeds with TTL
- **WebSocket/SSE Server:** Real-time push notifications for new posts

- **Message Queue:** Kafka for async fan-out processing

Data Model

```
// Posts table (Cassandra)
POST_ID | USER_ID | CONTENT | TIMESTAMP

// User timeline cache (Redis)
USER:{id}:feed -> [
  {postId, timestamp, score},
  ...
]
```

Scalability Strategies

- **Sharding:** Partition users and posts by user_id hash
- **Denormalization:** Store user metadata with posts to avoid joins
- **Lazy loading:** Load initial 20 posts, infinite scroll for more
- **CDN:** Serve media content (images/videos) from edge locations

CAP Theorem Trade-off

Choose AP over CP: Eventual consistency acceptable - brief delays in feed updates preferred over unavailability

3. Design a real-time chat application supporting one-on-one and group messaging with delivery receipts and online status.

System Architecture

- **WebSocket Gateway:** Persistent bidirectional connections for real-time messaging
- **Message Service:** Handles message routing, storage, and delivery
- **Presence Service:** Tracks user online/offline status with heartbeats
- **Message Queue:** RabbitMQ/Kafka for reliable message delivery
- **Database:** Cassandra for message history, Redis for active sessions

Message Flow

```
Client A -> WebSocket -> Gateway
  -> Message Service -> Queue
  -> Gateway -> WebSocket -> Client B

// Acknowledgment flow
Client B -> ACK -> Service
  -> Update delivery status
  -> Notify Client A
```

Data Models

- **Messages:** (message_id, conversation_id, sender_id, content, timestamp, status)
- **Conversations:** (conversation_id, participant_ids[], type, last_message)
- **Presence:** Redis key USER:{id}:status with TTL of 30 seconds

Scalability Considerations

- **Consistent Hashing:** Route users to specific gateway servers for session affinity
- **Message Ordering:** Use Lamport timestamps or vector clocks for distributed ordering
- **Offline Messages:** Queue messages in database, deliver on reconnection
- **Group Chat Optimization:** Fan-out to max 256 members, use channels for larger groups

Delivery Guarantees

At-least-once delivery: Idempotent message handling with deduplication using message_id

4. How would you design a distributed caching system with high availability and consistency?

Architecture Components

- **Cache Nodes:** Distributed Redis/Memcached cluster with replication
- **Consistent Hashing:** Distribute keys across nodes, minimize rehashing on node changes
- **Replication:** Master-slave setup with async replication for availability
- **Proxy Layer:** Twemproxy/Envoy for connection pooling and routing

Consistent Hashing Implementation

```
class ConsistentHash {
  constructor(nodes, replicas = 150) {
    this.ring = new Map();
    nodes.forEach(node => {
      for(let i=0; i
```

Cache Strategies

- **Cache-Aside:** Application checks cache, loads from DB on miss, updates cache
- **Write-Through:** Write to cache and database synchronously
- **Write-Behind:** Write to cache immediately, async persist to database
- **Refresh-Ahead:** Proactively refresh cache before expiration

Handling Cache Failures

- **Circuit Breaker:** Fail fast when cache is down, fallback to database
- **Cache Stampede Prevention:** Use mutex/semaphore for popular keys
- **Thundering Herd:** Implement probabilistic early expiration

Consistency vs Availability

Trade-off: Async replication provides availability but eventual consistency. Use versioning (vector clocks) to detect conflicts.

5. Design a rate limiting system that can handle millions of requests per second across distributed services.

Rate Limiting Algorithms

- **Token Bucket:** Tokens added at fixed rate, request consumes token. Allows bursts.
- **Leaky Bucket:** Requests processed at constant rate, excess queued/dropped
- **Fixed Window:** Count requests in fixed time windows (simple but has boundary issues)
- **Sliding Window Log:** Track timestamps, count in rolling window (memory intensive)
- **Sliding Window Counter:** Hybrid approach, weighted count from current and previous window

Token Bucket Implementation

```
class TokenBucket {
  allow(key, limit, window) {
    const now = Date.now();
    const tokens = redis.get(key) || limit;
    const lastRefill = redis.get(key+':time');
    const elapsed = now - lastRefill;
    const newTokens = Math.min(limit,
      tokens + elapsed * limit / window);
    if(newTokens >= 1) return true;
  }
}
```

Distributed Architecture

- **Redis Cluster:** Centralized counter storage with atomic INCR operations
- **Lua Scripts:** Atomic read-modify-write operations in Redis
- **Local Cache + Sync:** Each server maintains local counters, periodically syncs
- **Sticky Sessions:** Route same user to same server for accurate counting

Scalability Strategies

- **Sharding:** Partition rate limit data by user_id or API key hash
- **Approximate Counting:** Use probabilistic data structures (Count-Min Sketch) for high-scale
- **Hierarchical Limits:** Apply limits at multiple levels (user, IP, API key, global)

Edge Cases

Clock Skew: Use logical clocks or centralized time service. **Race Conditions:** Redis transactions or Lua scripts ensure atomicity

6. How would you design a video streaming platform like YouTube with support for different resolutions and adaptive bitrate streaming?

System Components

- **Upload Service:** Chunked upload to S3/object storage, generate unique video ID
- **Transcoding Pipeline:** FFmpeg workers convert to multiple formats (480p, 720p, 1080p, 4K)
- **CDN:** CloudFront/Akamai for global content delivery with edge caching
- **Metadata Service:** Store video info, thumbnails, view counts in database
- **Recommendation Engine:** ML-based system for personalized suggestions

Adaptive Bitrate Streaming

- **HLS (HTTP Live Streaming):** Break video into small chunks (2-10 sec), create manifest file
- **MPEG-DASH:** Similar to HLS, more flexible, codec-agnostic
- **Client Logic:** Monitor bandwidth, switch quality levels dynamically

HLS Manifest Example

```
#EXTM3U
#EXT-X-STREAM-INF:BANDWIDTH=800000
480p.m3u8
#EXT-X-STREAM-INF:BANDWIDTH=1400000
720p.m3u8
#EXT-X-STREAM-INF:BANDWIDTH=2800000
1080p.m3u8
```

Scalability & Performance

- **Distributed Transcoding:** Kubernetes cluster with autoscaling based on queue depth
- **Storage Optimization:** Use different storage tiers (hot/cold) based on popularity
- **Lazy Transcoding:** Generate only popular resolutions initially
- **Thumbnail Generation:** Extract keyframes at upload time, store in CDN

Analytics & Monitoring

Track: Buffering ratio, startup time, quality switches, CDN cache hit ratio, concurrent viewers

7. Design a notification system that supports multiple channels (email, SMS, push, in-app) with priority handling and retry logic.

Architecture Overview

- **Notification Service:** Central service accepting notification requests via API
- **Message Queue:** Kafka/RabbitMQ with priority queues for different urgency levels
- **Channel Workers:** Separate workers for email (SendGrid), SMS (Twilio), push (FCM), in-app
- **Template Service:** Manage notification templates with variable substitution
- **Delivery Tracking:** Store delivery status, read receipts, user preferences

Priority Queue System

```
// Message structure
{
  userId: '123',
  channels: ['email', 'push'],
  priority: 'high', // critical, high, medium, low
  template: 'order_confirmation',
  data: { orderId: '456' },
```

```
  retryPolicy: { maxAttempts: 3, backoff: 'exponential' }
}
```

Delivery Strategy

- **Fan-out:** Send to all specified channels in parallel
- **Fallback Chain:** Try push -> SMS -> email until one succeeds
- **User Preferences:** Check opt-in/opt-out settings before sending
- **Rate Limiting:** Prevent notification spam per user (max 5/hour for marketing)

Retry Logic

- **Exponential Backoff:** Retry after 1min, 5min, 15min, 1hr
- **Dead Letter Queue:** Move failed messages after max retries for manual review
- **Idempotency:** Use notification_id to prevent duplicate sends
- **Circuit Breaker:** Temporarily disable channel if provider is down

Scalability

Partitioning: Shard by user_id for parallel processing. **Batching:** Group emails for bulk sending API calls

8. How would you design a search autocomplete/typeahead system like Google's search suggestions?

System Architecture

- **Trie Data Structure:** Prefix tree storing popular search queries
- **Aggregation Service:** Collect search queries, calculate frequency/popularity scores
- **Suggestion Service:** Fast lookup of top-k suggestions for given prefix
- **Cache Layer:** Redis for frequently searched prefixes
- **Analytics Pipeline:** Real-time and batch processing of search logs

Trie Node Structure

```
class TrieNode {
  constructor() {
    this.children = new Map();
    this.topSuggestions = []; // Top 10 queries
    this.isEndOfWord = false;
    this.frequency = 0;
  }
}
```

Ranking Factors

- **Query Frequency:** How often query is searched globally
- **Personalization:** User's search history and preferences
- **Recency:** Trending queries weighted higher
- **Location:** Geographic relevance of suggestions
- **Typo Tolerance:** Fuzzy matching using edit distance

Optimization Strategies

- **Pre-computation:** Store top-k suggestions at each trie node
- **Sampling:** Process only sample of queries to reduce computation
- **Bloom Filter:** Quick check if query exists before trie lookup
- **Sharding:** Partition trie by first character or hash
- **Async Updates:** Update trie in background, serve from cache

Scale Considerations

Storage: 100M queries × 50 bytes = 5GB (fits in memory). **QPS:** 10K requests/sec, <1ms latency with cached trie

9. Design a distributed file storage system like Dropbox or Google Drive with sync capabilities across devices.

Core Components

- **Metadata Service:** Stores file/folder structure, versions, permissions in SQL database
- **Block Storage:** S3/object storage for actual file data, split into chunks (4MB blocks)
- **Sync Service:** Detects file changes, coordinates sync across devices
- **Notification Service:** WebSocket/long polling to notify clients of remote changes
- **Client Application:** Desktop/mobile app with local file watcher

File Chunking Strategy

```
// Split file into blocks
function chunkFile(file) {
  const CHUNK_SIZE = 4 * 1024 * 1024;
  const chunks = [];
  for(let i=0; i
```

Sync Algorithm

- **File Watcher:** Monitor local filesystem changes (inotify/FSEvents)
- **Delta Sync:** Upload only modified chunks, not entire file
- **Conflict Resolution:** Last-write-wins or merge strategies with version vectors
- **Deduplication:** Content-addressable storage using chunk hashes

Metadata Schema

- **Files:** (file_id, name, parent_folder_id, owner_id, size, version, modified_time)
- **Chunks:** (chunk_hash, storage_path, size)
- **File_Chunks:** (file_id, chunk_hash, sequence_number)

Scalability & Reliability

- **Sharding:** Partition metadata by user_id
- **Replication:** Store chunks in multiple availability zones
- **Caching:** CDN for frequently accessed files
- **Compression:** Gzip chunks before storage

10. How would you design a ride-sharing platform like Uber with real-time location tracking and driver-rider matching?

System Architecture

- **Location Service:** Receive GPS updates from drivers every 4 seconds
- **Matching Service:** Find optimal driver-rider pairs based on proximity and ETA
- **Trip Service:** Manage trip lifecycle (request, accept, start, complete)
- **Payment Service:** Handle fare calculation and payment processing
- **Notification Service:** Real-time updates via WebSocket/push notifications

Geospatial Indexing

- **QuadTree:** Divide map into hierarchical grid, store drivers in leaf nodes
- **Geohash:** Encode lat/long into string, nearby locations share prefix
- **Redis Geo:** Built-in geospatial commands (GEOADD, GEORADIUS)

Driver Matching Algorithm

```
function findDrivers(riderLat, riderLng, radius) {
  // Query Redis Geo
  const nearby = redis.georadius(
    'drivers:active',
    riderLng, riderLat, radius, 'km'
  );
  // Score by distance, rating, acceptance rate
  return nearby.sort((a,b) =>
    score(a) - score(b)
  ).slice(0, 5);
}
```

Real-time Location Updates

- **WebSocket Connection:** Persistent connection for driver location streaming
- **Message Queue:** Kafka topic for location events, consumed by matching service
- **Database:** Redis for active driver locations (TTL 30 sec), Cassandra for trip history

Scalability Considerations

- **Sharding:** Partition by geographic region (city-level)
- **Load Balancing:** Route requests based on rider location to regional servers
- **Caching:** Cache driver availability, surge pricing multipliers
- **ETA Calculation:** Pre-computed road network graph with traffic data

CAP Theorem

AP System: Availability over consistency - brief location staleness acceptable for better user experience

Coding and Debugging

This section presents practical coding challenges and questions about debugging techniques.

1. Write a function to flatten a nested array to any depth without using built-in flat() method.

Flattening Nested Arrays

Here's an efficient recursive solution:

```
function flattenArray(arr) {
  const result = [];
  for (let item of arr) {
    if (Array.isArray(item)) {
      result.push(...flattenArray(item));
    } else {
      result.push(item);
    }
  }
  return result;
}
```

Key Points:

- Uses recursion to handle arbitrary nesting depth
- Spread operator efficiently concatenates results
- Time complexity: O(n) where n is total elements
- Alternative: Use reduce for a more functional approach

2. How would you implement debounce and throttle functions from scratch? Explain the difference.

Debounce vs Throttle

Debounce delays execution until after a pause in events:

```
function debounce(func, delay) {
  let timeoutId;
  return function(...args) {
    clearTimeout(timeoutId);
    timeoutId = setTimeout(() => func.apply(this, args), delay);
  };
}
```

Throttle ensures execution at regular intervals:

```
function throttle(func, limit) {
  let inThrottle;
  return function(...args) {
    if (!inThrottle) {
      func.apply(this, args);
      inThrottle = true;
      setTimeout(() => inThrottle = false, limit);
    }
  };
}
```

Use Cases:

- Debounce: Search input, window resize
- Throttle: Scroll events, mouse movement tracking

3. Write a function to deep clone an object including handling circular references.

Deep Clone with Circular Reference Handling

```
function deepClone(obj, hash = new WeakMap()) {
  if (obj === null || typeof obj !== 'object') return obj;
  if (hash.has(obj)) return hash.get(obj);
  const clone = Array.isArray(obj) ? [] : {};
  hash.set(obj, clone);
  for (let key in obj) {
    if (obj.hasOwnProperty(key)) {
      clone[key] = deepClone(obj[key], hash);
    }
  }
  return clone;
}
```

Key Features:

- WeakMap tracks visited objects to prevent infinite loops
- Handles arrays and objects
- Preserves prototype chain considerations
- For production, consider structuredClone() API

4. How do you debug memory leaks in a React application? What tools and techniques do you use?

Debugging Memory Leaks in React

Common Causes:

- Event listeners not removed in cleanup
- Subscriptions not cancelled
- Closures holding references
- Detached DOM nodes

Tools and Techniques:

- **Chrome DevTools Memory Profiler:** Take heap snapshots, compare allocations
- **Performance Monitor:** Track DOM nodes, JS heap size over time
- **React DevTools Profiler:** Identify unnecessary re-renders
- **useEffect cleanup:** Always return cleanup functions

```
useEffect(() => {
  const handler = () => {};
  window.addEventListener('scroll', handler);
  return () => window.removeEventListener('scroll', handler);
}, []);
```

Use **allocation timeline** to find where memory grows.

5. Explain the event loop and microtask queue. What's the output of this code and why?

Event Loop and Microtask Queue

```
console.log('1');
setTimeout(() => console.log('2'), 0);
Promise.resolve().then(() => console.log('3'));
queueMicrotask(() => console.log('4'));
console.log('5');
```

Output: 1, 5, 3, 4, 2

Explanation:

- Synchronous code runs first: 1, 5
- Microtasks (Promises, queueMicrotask) execute before macrotasks: 3, 4
- Macrotasks (setTimeout) run last: 2
- **Event Loop Order:** Call Stack → Microtask Queue → Macrotask Queue

Understanding this is crucial for async operations, rendering optimization, and preventing blocking.

6. How would you implement a custom Promise.all that handles errors differently?

Custom Promise.all Implementation

```
function promiseAll(promises) {
  return new Promise((resolve, reject) => {
    const results = [];
    let completed = 0;
    promises.forEach((promise, index) => {
      Promise.resolve(promise)
        .then(value => {
          results[index] = value;
          if (++completed === promises.length) resolve(results);
        })
        .catch(reject);
    });
  });
}
```

Enhanced Version with Error Collection:

- Use Promise.allSettled pattern to collect all results
- Return both successes and failures
- Useful for batch operations where partial failure is acceptable
- Consider timeout handling for hung promises

7. What are the best practices for exception handling in async/await code?

Exception Handling in Async/Await

Best Practices:

- **Always use try-catch:** Wrap await calls to prevent unhandled rejections
- **Error boundaries:** Use React error boundaries for component errors
- **Centralized error handling:** Create error handler utilities
- **Specific error types:** Catch and handle different error types

```
async function fetchData() {
  try {
    const response = await fetch('/api/data');
    if (!response.ok) throw new Error('HTTP error');
    return await response.json();
  } catch (error) {
    console.error('Fetch failed:', error);
    throw error;
  }
}
```

Use **finally** for cleanup, avoid silent failures, and log errors appropriately.

8. Explain monkey patching in JavaScript. When is it appropriate and what are the risks?

Monkey Patching

Definition: Modifying or extending built-in objects or third-party code at runtime.

```
Array.prototype.last = function() {
  return this[this.length - 1];
};
const arr = [1, 2, 3];
console.log(arr.last()); // 3
```

Appropriate Use Cases:

- Polyfills for missing browser features
- Testing and mocking
- Temporary debugging hooks

Risks:

- Conflicts with future native implementations
- Hard to track and maintain
- Can break third-party libraries
- Non-enumerable properties recommended

Better Alternative: Use composition, wrapper functions, or utility libraries instead.

9. How do you debug performance issues in a large React application?

Debugging React Performance

Tools and Techniques:

- **React DevTools Profiler:** Record and analyze component render times
- **Chrome Performance Tab:** Identify long tasks, layout thrashing
- **React.memo:** Prevent unnecessary re-renders
- **useMemo/useCallback:** Memoize expensive computations and functions
- **Code splitting:** Use React.lazy and Suspense
- **Virtualization:** Use react-window for long lists

```
const MemoizedComponent = React.memo(({data}) => {
  const computed = useMemo(() => expensiveOp(data), [data]);
  return
{computed}
;
});
```

Monitor **bundle size**, lazy load routes, and use production builds for accurate profiling.

10. Write a function to implement a simple LRU (Least Recently Used) cache.

LRU Cache Implementation

```
class LRUCache {
  constructor(capacity) {
    this.capacity = capacity;
    this.cache = new Map();
  }
  get(key) {
    if (!this.cache.has(key)) return -1;
    const value = this.cache.get(key);
    this.cache.delete(key);
    this.cache.set(key, value);
    return value;
  }
  put(key, value) {
    this.cache.delete(key);
    this.cache.set(key, value);
    if (this.cache.size > this.capacity) {
      this.cache.delete(this.cache.keys().next().value);
    }
  }
}
```

Key Points:

- Map maintains insertion order in JavaScript
- Delete and re-insert to mark as recently used
- O(1) time complexity for get and put
- Useful for caching API responses, computed values

Behavioral Questions

These questions assess your soft skills, problem-solving approach, and how you work in a team.

1. Tell me about a time when you had to optimize a poorly performing web application. What was your approach?

Situation: At my previous company, our e-commerce platform was experiencing slow page load times (5-7 seconds), causing a 30% cart abandonment rate.

Task: I was assigned to identify performance bottlenecks and reduce load times to under 2 seconds within one sprint.

Action: I conducted a comprehensive audit using Lighthouse and Chrome DevTools. I implemented code splitting with React.lazy(), optimized images using WebP format with lazy loading, reduced bundle size by 40% through tree-shaking unused dependencies, implemented service workers for caching, and moved to a CDN for static assets.

Result: Page load time decreased to 1.8 seconds, cart abandonment dropped by 18%, and we saw a 12% increase in conversion rates within the first month.

2. Describe a situation where you had a disagreement with a team member about a technical approach. How did you handle it?

Situation: During a redesign project, my colleague wanted to use Redux for state management in a relatively simple React application, while I advocated for Context API to reduce complexity.

Task: We needed to align on a state management solution that wouldn't delay the project timeline and would be maintainable long-term.

Action: I organized a technical discussion where we both presented our cases with specific metrics: bundle size impact, learning curve for junior developers, and scalability considerations. I created a small proof-of-concept for both approaches, measuring performance and developer experience. We also consulted our team lead for a third perspective.

Result: After reviewing the data, we agreed to start with Context API for the current scope with a clear migration path to Redux if complexity increased. This decision saved us 2 days of setup time and reduced our bundle size by 45KB. Six months later, Context API still met our needs perfectly.

3. Tell me about a time when you had to learn a new technology or framework quickly to meet a project deadline.

Situation: Our client requested a real-time collaborative feature similar to Google Docs, but our team had no experience with WebSockets or operational transformation algorithms.

Task: I volunteered to lead the implementation and had three weeks to deliver a working prototype.

Action: I dedicated the first three days to intensive learning: studying WebSocket protocols, exploring libraries like Socket.io and Yjs, and building small test applications. I created a knowledge-sharing document for the team, scheduled daily 30-minute pair programming sessions to transfer knowledge, and broke down the feature into smaller, manageable tasks. I also reached out to developers in online communities for best practices.

Result: We delivered the prototype on time with real-time text synchronization for up to 50 concurrent users. The feature became a key differentiator for the client, leading to a contract extension. Three team members gained proficiency in WebSocket development, expanding our technical capabilities.

4. Describe a situation where you identified a critical bug in production. How did you handle it?

Situation: On a Friday evening, our monitoring system alerted us that users were unable to

complete payments on our checkout page, affecting approximately 200 transactions per hour.

Task: As the on-call frontend lead, I needed to quickly identify the root cause, implement a fix, and restore service with minimal revenue loss.

Action: I immediately checked our error tracking (Sentry) and identified a TypeError in our payment validation logic caused by a recent deployment. I rolled back the deployment within 10 minutes to restore service, then analyzed the problematic code change. The issue was a null check missing for an optional field. I created a hotfix with comprehensive unit tests, conducted a thorough code review with another senior developer, and deployed to production with enhanced monitoring.

Result: Service was restored within 15 minutes, limiting lost transactions to approximately 50. I documented the incident in a post-mortem, which led to implementing stricter pre-deployment checks and better test coverage for payment flows. We also improved our feature flag strategy to enable safer rollouts.

5. Tell me about a time when you had to mentor a junior developer. What was your approach?

Situation: A junior developer joined our team with limited React experience and was struggling with component architecture and state management patterns.

Task: I was assigned as their mentor to help them become productive within two months while working on real project tasks.

Action: I created a structured mentorship plan: weekly 1-on-1s to discuss concepts and challenges, assigned progressively complex tasks starting with UI components then moving to state logic, conducted code reviews with detailed explanations rather than just corrections, paired programmed on challenging features, and shared curated learning resources. I also encouraged them to ask questions publicly in team channels to normalize learning.

Result: Within six weeks, the junior developer was independently implementing features and contributing meaningful code reviews. They successfully built a complex data visualization dashboard that became a client favorite. The mentorship framework I created was adopted company-wide, and I've since mentored four more developers using this approach.

6. Describe a project where you had to balance technical debt with feature development. How did you manage it?

Situation: Our codebase had accumulated significant technical debt with a legacy jQuery application while the business was pushing for new React-based features.

Task: I needed to advocate for addressing technical debt while continuing to deliver business value and new features.

Action: I quantified the impact of technical debt by measuring: bug frequency in legacy code (3x higher), development velocity (40% slower for features touching old code), and onboarding time for new developers. I proposed a hybrid approach: allocating 30% of each sprint to incremental refactoring, implementing a strangler fig pattern to gradually migrate to React, and ensuring all new features were built with modern architecture. I created a visual dashboard showing debt reduction progress to maintain stakeholder buy-in.

Result: Over six months, we reduced legacy code by 60%, decreased bug reports by 45%, and improved feature delivery speed by 35%. The business saw the value as we could deliver features faster, and the development team's satisfaction scores increased significantly.

7. Tell me about a time when you had to make a difficult trade-off between code quality and meeting a deadline.

Situation: We were building a product launch page for a major client event in two weeks, but implementing all accessibility features and comprehensive testing would require three weeks.

Task: I needed to deliver a functional, professional page on time while maintaining our quality standards and not accruing unmanageable technical debt.

Action: I conducted a risk assessment with the team, identifying must-have vs. nice-to-have features. We prioritized core functionality and WCAG AA compliance for keyboard navigation and screen readers, but deferred advanced animations and edge case testing. I documented all deferred items as tickets with clear acceptance criteria and negotiated with the product manager to schedule

a hardening sprint immediately after launch. We also implemented feature flags to disable problematic features quickly if needed.

Result: We launched on time with 95% of planned functionality and zero critical bugs. The page handled 50,000 visitors on launch day successfully. We completed the hardening sprint two weeks later, addressing all deferred items. This approach became our template for handling time-constrained projects while maintaining quality standards.

8. Describe a situation where you had to advocate for a better user experience against business pressure for quick implementation.

Situation: The business team wanted to add multiple promotional pop-ups and banners to increase conversion, but I believed this would harm user experience and potentially decrease overall engagement.

Task: I needed to convince stakeholders to reconsider the approach while respecting their conversion goals.

Action: I gathered data from UX research showing that intrusive pop-ups increase bounce rates by 35%. I proposed an A/B test comparing their approach against a less intrusive alternative: a single, well-timed slide-in notification with personalized content. I created prototypes of both approaches and presented user testing results showing the subtle approach had better reception. I also cited industry studies and competitor analysis supporting user-centric design.

Result: The stakeholders agreed to the A/B test. The subtle approach resulted in 8% higher conversion and 22% lower bounce rate compared to multiple pop-ups. This data-driven approach established credibility, and I was subsequently included in early-stage feature planning to provide UX input before development began.

9. Tell me about a time when you had to work with a difficult stakeholder or client. How did you manage the relationship?

Situation: A key stakeholder frequently changed requirements mid-sprint, submitted unclear specifications, and expected immediate implementation without considering technical constraints.

Task: I needed to establish a more productive working relationship while delivering on project commitments.

Action: I scheduled a one-on-one meeting to understand their underlying concerns and business pressures. I learned they felt disconnected from the development process and worried about missing market opportunities. I proposed weekly demo sessions to show progress, implemented a structured change request process with impact assessments (timeline, resources, trade-offs), and created a shared roadmap with clear milestones. I also educated them on agile principles and why mid-sprint changes were costly.

Result: The relationship improved significantly. Mid-sprint change requests decreased by 70%, and when changes were necessary, they came with proper context and priority. The stakeholder became one of our strongest advocates, praising the team's transparency and reliability in executive meetings. Project delivery predictability increased from 60% to 90%.

10. Describe a time when you failed to meet a deadline or deliverable. What did you learn from it?

Situation: I committed to delivering a complex data visualization dashboard in two weeks but underestimated the complexity of integrating with multiple third-party APIs and handling edge cases.

Task: I needed to communicate the delay, manage stakeholder expectations, and ensure successful delivery.

Action: As soon as I realized we'd miss the deadline (at the one-week mark), I immediately informed the project manager and stakeholders with full transparency about the challenges: API rate limiting issues, data inconsistencies requiring additional transformation logic, and performance problems with large datasets. I provided a revised timeline with buffer, broke remaining work into smaller milestones with daily updates, and brought in a colleague to pair program on the most complex parts.

Result: We delivered four days late but with a robust, well-tested solution. The transparency maintained trust with stakeholders. I learned crucial lessons about estimation: now I always add 30-40% buffer for integration work, break down tasks more granularly, and communicate risks earlier. I

also implemented a personal practice of mid-point check-ins on all commitments. Since then, my on-time delivery rate improved to 95%.