Web Developer

Interview Questions and Answers

Core Concepts

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

1. Explain the event loop in JavaScript and how it handles asynchronous operations with the call stack, task queue, and microtask queue.

Event Loop Mechanism

The JavaScript event loop is a single-threaded mechanism that coordinates execution between the **call stack**, **task queue (macrotasks)**, and **microtask queue**.

- **Call Stack:** Executes synchronous code in LIFO order
- **Microtask Queue:** Handles Promises, queueMicrotask(), and MutationObserver callbacks
- **Task Queue (Macrotask):** Handles setTimeout, setInterval, I/O operations, and UI rendering

Execution Order

The event loop follows this priority:

1. Execute all synchronous code on the call stack
2. Process ALL microtasks in the microtask queue
3. Execute ONE macrotask from the task queue
4. Repeat from step 2

```
console.log('1');
setTimeout(() => console.log('2'), 0);
Promise.resolve().then(() => console.log('3'));
console.log('4');
// Output: 1, 4, 3, 2
// Microtasks (Promise) execute before macrotasks (setTimeout)
```

This architecture ensures non-blocking I/O while maintaining predictable execution order for asynchronous operations.

2. What are the key differences between Server-Side Rendering (SSR), Client-Side Rendering (CSR), and Static Site Generation (SSG)? When would you choose each approach?

Rendering Strategies Comparison

Client-Side Rendering (CSR):

- HTML rendered in browser via JavaScript
- Initial page load shows minimal HTML, then JS hydrates content
- Best for: Highly interactive web apps, dashboards, admin panels
- Drawbacks: Poor initial SEO, slower First Contentful Paint

Server-Side Rendering (SSR):

- HTML generated on server for each request
- Fully rendered page sent to client, then hydrated
- Best for: Dynamic content requiring fresh data, personalized experiences
- Drawbacks: Higher server load, slower Time to Interactive

Static Site Generation (SSG):

- HTML pre-rendered at build time
- Served as static files from CDN
- Best for: Blogs, marketing sites, documentation with infrequent updates
- Drawbacks: Requires rebuild for content changes, not suitable for user-specific content

Decision Framework

Choose **SSG** for content-heavy sites with predictable data. Use **SSR** when you need real-time data with SEO. Opt for **CSR** when SEO isn't critical and interactivity is paramount. Modern frameworks like Next.js support hybrid approaches, combining these strategies per-route.

3. Explain how the Critical Rendering Path works and what strategies you would use to optimize it for better performance.

Critical Rendering Path Overview

The Critical Rendering Path is the sequence of steps browsers take to convert HTML, CSS, and JavaScript into rendered pixels on screen:

1. **DOM Construction:** Parse HTML to build Document Object Model
2. **CSSOM Construction:** Parse CSS to build CSS Object Model
3. **Render Tree:** Combine DOM and CSSOM into render tree
4. **Layout:** Calculate geometry and position of elements
5. **Paint:** Rasterize pixels to screen

Optimization Strategies

1. Minimize Critical Resources:

- Inline critical CSS for above-the-fold content
- Defer non-critical CSS with media queries or async loading
- Remove unused CSS with tools like PurgeCSS

2. Reduce Critical Bytes:

- Minify HTML, CSS, and JavaScript
- Enable compression (Gzip/Brotli)
- Optimize images with modern formats (WebP, AVIF)

3. Minimize Critical Path Length:

- Use async/defer for non-critical JavaScript
- Implement resource hints: dns-prefetch, preconnect, prefetch
- Leverage HTTP/2 multiplexing

<link rel="preconnect" href="https://fonts.googleapis.com">
<link rel="dns-prefetch" href="https://api.example.com">
<script src="app.js" defer></script>

These optimizations significantly improve First Contentful Paint and Time to Interactive metrics.

4. What is the difference between debouncing and throttling? Provide implementation examples and use cases for each.

Debouncing vs Throttling

Debouncing: Delays function execution until after a specified time has elapsed since the last invocation. Resets the timer on each new call.

Use cases: Search input autocomplete, window resize handlers, form validation

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

Throttling: Ensures function executes at most once per specified time interval, regardless of how many times it's called.

Use cases: Scroll event handlers, mouse movement tracking, API rate limiting

function throttle(func, limit) {

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

Key Difference

Debouncing waits for a **pause in events** before executing. Throttling executes at **regular intervals** during continuous events. For search input, debouncing prevents API calls until user stops typing. For infinite scroll, throttling ensures scroll handler runs every 200ms, not on every pixel scrolled.

5. Explain Cross-Origin Resource Sharing (CORS) and how you would configure it properly for a production API.

CORS Fundamentals

Cross-Origin Resource Sharing (CORS) is a security mechanism that allows or restricts web applications running at one origin to access resources from a different origin. Browsers enforce the Same-Origin Policy, and CORS provides a way to relax these restrictions safely.

How CORS Works

Simple Requests: Browser sends request with Origin header. Server responds with Access-Control-Allow-Origin header.

Preflight Requests: For non-simple requests (PUT, DELETE, custom headers), browser sends OPTIONS request first to check permissions.

Production Configuration Best Practices

1. Specify Explicit Origins (Never use * in production):

```
// Express.js example
const allowedOrigins = ['https://app.example.com'];
app.use(cors({
  origin: (origin, callback) => {
    if (allowedOrigins.includes(origin) || !origin) {
      callback(null, true);
    } else callback(new Error('Not allowed by CORS'));
  },
  credentials: true
}));
```

2. Configure Credentials Properly:

- Set credentials: true only when necessary
- Cannot use wildcard origin with credentials
- Include Access-Control-Allow-Credentials: true header

3. Limit Allowed Methods and Headers:

```
Access-Control-Allow-Methods: GET, POST, PUT, DELETE
Access-Control-Allow-Headers: Content-Type, Authorization
```

4. Set Appropriate Preflight Cache: Access-Control-Max-Age: 86400 to reduce preflight requests.

6. What are Web Workers and Service Workers? How do they differ, and what are practical use cases for each?

Web Workers

Web Workers run JavaScript in background threads, separate from the main UI thread, enabling true parallelism for CPU-intensive tasks.

Characteristics:

- No access to DOM or window object
- Communicate via postMessage API
- Terminated when page closes
- Dedicated workers (one-to-one with script) or Shared workers (shared across tabs)

Use cases: Image processing, data parsing, complex calculations, encryption

```
// main.js
const worker = new Worker('worker.js');
worker.postMessage({data: largeDataset});
worker.onmessage = (e) => console.log(e.data);

// worker.js
self.onmessage = (e) => {
  const result = processData(e.data);
  self.postMessage(result);
};
```

Service Workers

Service Workers act as programmable network proxies, intercepting network requests and enabling offline functionality.

Characteristics:

- Run independently of web pages
- Persist across page sessions
- Enable Progressive Web App features
- Require HTTPS (except localhost)

Use cases: Offline caching, push notifications, background sync, performance optimization

```
// Register service worker
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/sw.js');
}
```

Key Difference

Web Workers handle **computational tasks** to prevent UI blocking. Service Workers manage **network requests** and enable offline experiences.

7. Explain the concept of tree shaking in modern JavaScript bundlers. What conditions must be met for it to work effectively?

Tree Shaking Overview

Tree shaking is a dead-code elimination technique that removes unused exports from JavaScript bundles, significantly reducing bundle size. It relies on static analysis of ES6 module syntax.

How It Works

Bundlers like Webpack, Rollup, and Vite analyze import/export statements to build a dependency graph, marking used code as "live" and eliminating unreferenced exports.

```
// utils.js
export function used() { return 'kept'; }
export function unused() { return 'removed'; }

// app.js
import { used } from './utils';
console.log(used());
// unused() is tree-shaken from final bundle
```

Requirements for Effective Tree Shaking

1. Use ES6 Module Syntax:

- Must use import/export, not require/module.exports
- CommonJS modules cannot be statically analyzed

2. Avoid Side Effects:

- Mark pure modules in package.json: "sideEffects": false
- Avoid top-level code execution in modules
- Use /*#__PURE__*/ comments for function calls

3. Enable Production Mode:

- Set NODE_ENV=production
- Enable minification (Terser, esbuild)

4. Configure Package.json Correctly:

```
{
  "sideEffects": ["*.css", "*.scss"],
  "module": "dist/index.esm.js"
}
```

Libraries like Lodash-es provide tree-shakeable versions. Using named imports instead of default imports improves tree shaking effectiveness.

8. What is the difference between OAuth 2.0 and JWT authentication? How would you implement a secure authentication system using both?

OAuth 2.0 vs JWT

OAuth 2.0 is an **authorization framework** that defines flows for granting third-party access to resources without sharing credentials. It's a protocol, not a token format.

JWT (JSON Web Token) is a **token format** for securely transmitting claims between parties. It's often used as the token type in OAuth 2.0 implementations.

Key Differences

- **OAuth 2.0:** Defines authorization flows (authorization code, client credentials, etc.)
- **JWT:** Defines token structure and signing mechanism
- OAuth can use JWT as access tokens, but can also use opaque tokens
- JWT can be used independently of OAuth for authentication

Secure Implementation Pattern

Combined Approach (OAuth 2.0 with JWT tokens):

```
// 1. Client requests authorization
GET /oauth/authorize?response_type=code&client_id=xxx

// 2. Server returns authorization code
// 3. Exchange code for JWT tokens
POST /oauth/token
{
  "grant_type": "authorization_code",
  "code": "auth_code",
  "client_id": "xxx",
  "client_secret": "yyy"
}
```

Security Best Practices:

- Use short-lived access tokens (15 minutes) with refresh tokens
- Store refresh tokens securely (httpOnly cookies or secure storage)
- Implement token rotation on refresh
- Use PKCE (Proof Key for Code Exchange) for public clients
- Validate JWT signature, expiration, and issuer on every request
- Implement rate limiting and token revocation

This hybrid approach leverages OAuth 2.0's robust authorization flows with JWT's stateless, self-contained token format.

9. Explain the concept of Content Security Policy (CSP). How would you configure it to prevent XSS attacks while maintaining functionality?

Content Security Policy (CSP)

CSP is a security standard that helps prevent Cross-Site Scripting (XSS), clickjacking, and other code injection attacks by declaring which content sources are trusted.

How CSP Works

CSP is implemented via HTTP header or meta tag, specifying allowed sources for scripts, styles, images, and other resources. Browsers block content from unauthorized sources.

```
Content-Security-Policy: default-src 'self';
  script-src 'self' https://trusted-cdn.com;
  style-src 'self' 'unsafe-inline';
  img-src 'self' data: https:;
```

Key Directives

- **default-src:** Fallback for all resource types
- **script-src:** Controls JavaScript sources
- **style-src:** Controls CSS sources
- **img-src:** Controls image sources
- **connect-src:** Controls fetch, XHR, WebSocket connections
- **frame-ancestors:** Prevents clickjacking

Secure Configuration Strategy

1. Start with Report-Only Mode:

```
Content-Security-Policy-Report-Only: default-src 'self';
  report-uri /csp-violation-report
```

2. Avoid Unsafe Directives:

- Never use 'unsafe-eval' in production
- Minimize 'unsafe-inline' usage
- Use nonces or hashes for inline scripts

3. Use Nonces for Inline Scripts:

```
Content-Security-Policy: script-src 'nonce-random123';
<script nonce="random123">console.log('allowed');</script>
```

4. Progressive Enhancement: Start strict, monitor violations, whitelist necessary sources incrementally. This approach blocks XSS while maintaining legitimate functionality.

10. What are the differences between HTTP/1.1, HTTP/2, and HTTP/3? How do these differences impact web application performance?

HTTP Protocol Evolution

HTTP/1.1 (1997):

- Single request per TCP connection (or head-of-line blocking with pipelining)
- Text-based protocol with verbose headers
- No header compression
- Requires multiple connections for parallelism (typically 6 per domain)

HTTP/2 (2015):

- **Multiplexing:** Multiple requests over single TCP connection
- **Binary protocol:** More efficient parsing
- **Header compression:** HPACK algorithm reduces overhead
- **Server push:** Proactively send resources
- **Stream prioritization:** Control resource loading order
- Still suffers from TCP head-of-line blocking at transport layer

HTTP/3 (2022):

- Built on **QUIC protocol** (UDP-based) instead of TCP
- Eliminates head-of-line blocking completely
- **Faster connection establishment:** 0-RTT resumption
- **Better mobile performance:** Connection migration across networks
- Built-in encryption (TLS 1.3)

Performance Impact

HTTP/2 vs HTTP/1.1:

- 30-50% faster page loads due to multiplexing
- Eliminates need for domain sharding and resource concatenation
- Reduces overhead from redundant headers

HTTP/3 vs HTTP/2:

- Improved performance on lossy networks (mobile)
- Faster recovery from packet loss
- Reduced latency for initial connections

Optimization Strategies

With HTTP/2/3, traditional optimizations like sprite sheets and domain sharding become anti-patterns. Focus on code splitting, lazy loading, and efficient caching strategies instead.

Data Structures and Algorithms

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

1. Implement an LRU (Least Recently Used) Cache with O(1) time complexity for both get and put operations.

LRU Cache Implementation

An **LRU Cache** requires a combination of a **HashMap** for O(1) lookups and a **Doubly Linked List** for O(1) insertion/deletion to track access order.

```
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: O(1) for both operations. **Space Complexity:** O(capacity).

2. Explain the difference between a Stack and a Queue, and provide a real-world use case for each in web development.

Stack vs Queue

Stack (LIFO - Last In First Out):

- Elements are added and removed from the same end (top)
- Operations: push(), pop(), peek()
- **Use case:** Browser history navigation, undo/redo functionality, function call stack

Queue (FIFO - First In First Out):

- Elements are added at the rear and removed from the front
- Operations: enqueue(), dequeue(), peek()
- **Use case:** Task scheduling, message queues, breadth-first search, handling API requests

```
// Stack
const stack = [];
stack.push(1); stack.push(2);
stack.pop(); // 2

// Queue
const queue = [];
queue.push(1); queue.push(2);
queue.shift(); // 1
```

3. How would you find all pairs in an array that sum to a target value? What's the optimal time complexity?

Two Sum / Pair Sum Problem

The optimal approach uses a **Hash Set** to achieve O(n) time complexity with a single pass through the array.

```
function findPairs(arr, target) {
  const seen = new Set();
  const pairs = [];
  for (const num of arr) {
    const complement = target - num;
    if (seen.has(complement)) {
      pairs.push([complement, num]);
    }
    seen.add(num);
  }
  return pairs;
}
```

Time Complexity: O(n) - single pass through array. **Space Complexity:** O(n) - hash set storage.

Alternative: Sorting + two pointers is O(n log n) time but O(1) extra space.

4. Implement a function to detect a cycle in a linked list. What algorithm would you use?

Cycle Detection - Floyd's Cycle Algorithm

Use **Floyd's Cycle Detection Algorithm** (tortoise and hare) with two pointers moving at different speeds.

```
function hasCycle(head) {
  let slow = head;
  let fast = head;
  while (fast && fast.next) {
    slow = slow.next;
    fast = fast.next.next;
    if (slow === fast) return true;
  }
  return false;
}
```

How it works:

- Slow pointer moves 1 step at a time
- Fast pointer moves 2 steps at a time
- If there's a cycle, they will eventually meet
- If fast reaches null, no cycle exists

Time Complexity: O(n), **Space Complexity:** O(1)

5. What is a Trie data structure and when would you use it in web applications?

Trie (Prefix Tree)

A **Trie** is a tree-like data structure that stores strings character by character, enabling efficient prefix-based operations.

```
class TrieNode {
  constructor() {
    this.children = {};
    this.isEndOfWord = false;
  }
}
class Trie {
  constructor() { this.root = new TrieNode(); }
  insert(word) {
    let node = this.root;
```

```
    for (let char of word) {
      if (!node.children[char]) node.children[char] = new TrieNode();
      node = node.children[char];
    }
    node.isEndOfWord = true;
  }
}
```

Web Development Use Cases:

- Autocomplete/typeahead search features
- Spell checkers and word validators
- IP routing tables
- Dictionary implementations

Time Complexity: O(m) where m is the length of the word.

6. Explain the Sliding Window technique and solve: Find the maximum sum of k consecutive elements in an array.

Sliding Window Technique

The **Sliding Window** pattern optimizes problems involving subarrays/substrings by maintaining a window that slides through the data structure.

```
function maxSumSubarray(arr, k) {
  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;
}
```

Key Concepts:

- Avoid recalculating from scratch for each window
- Add new element, remove old element
- **Time Complexity:** O(n) instead of O(n*k)

Common applications: Longest substring problems, maximum/minimum in subarrays

7. What is the difference between a Min Heap and a Max Heap? Implement a basic Min Heap insert operation.

Heap Data Structure

Max Heap: Parent node is always greater than or equal to children (root is maximum).

Min Heap: Parent node is always less than or equal to children (root is minimum).

```
class MinHeap {
  constructor() { this.heap = []; }
  insert(val) {
    this.heap.push(val);
    this.bubbleUp(this.heap.length - 1);
  }
  bubbleUp(idx) {
    while (idx > 0) {
      let parent = Math.floor((idx - 1) / 2);
      if (this.heap[parent] <= this.heap[idx]) break;
      [this.heap[parent], this.heap[idx]] = [this.heap[idx], this.heap[parent]];
      idx = parent;
    }
  }
```

```
}
```

Use Cases: Priority queues, scheduling algorithms, finding k smallest/largest elements, Dijkstra's algorithm.

Time Complexity: Insert O(log n), Extract-Min O(log n), Peek O(1)

8. How do you implement a debounce function? What's the difference between debounce and throttle?

Debounce vs Throttle

Debounce: Delays execution until after a specified time has passed since the last invocation.

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

Throttle: Ensures function executes at most once per specified time period.

```
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, resize events, form validation
- **Throttle:** Scroll events, mouse movement, API rate limiting

9. Explain Binary Search and its time complexity. When can it be applied?

Binary Search Algorithm

Binary Search is a divide-and-conquer algorithm that efficiently searches for a target value in a **sorted array** by repeatedly dividing the search interval in half.

```
function binarySearch(arr, target) {
  let left = 0, right = arr.length - 1;
  while (left <= right) {
    const mid = Math.floor((left + right) / 2);
    if (arr[mid] === target) return mid;
    if (arr[mid] < target) left = mid + 1;
    else right = mid - 1;
  }
  return -1;
}
```

Requirements:

- Array must be **sorted**
- Random access to elements (arrays, not linked lists)

Time Complexity: O(log n) - much faster than linear search O(n)

Space Complexity: O(1) iterative, O(log n) recursive

Applications: Searching in sorted data, finding insertion points, range queries

10. What is memoization and how does it relate to dynamic programming? Provide an example with Fibonacci.

Memoization and Dynamic Programming

Memoization is an optimization technique that stores the results of expensive function calls and returns cached results when the same inputs occur again.

Fibonacci without memoization: O(2^n) time complexity

```
function fib(n, memo = {}) {
  if (n in memo) return memo[n];
  if (n <= 1) return n;
  memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
  return memo[n];
}
```

With memoization: O(n) time, O(n) space

Dynamic Programming is a broader technique that breaks problems into overlapping subproblems and stores results to avoid redundant calculations.

Key Differences:

- **Memoization:** Top-down approach (recursion + caching)
- **DP:** Can be bottom-up (iterative tabulation)

Use cases: Optimization problems, path finding, resource allocation

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 billions of URLs?

Key Components

- **API Gateway:** Handle incoming requests for URL creation and redirection
- **Application Servers:** Stateless services for business logic
- **Database:** Store URL mappings (short code → original URL)
- **Cache Layer:** Redis/Memcached for frequently accessed URLs
- **Load Balancer:** Distribute traffic across application servers

Architecture Approach

Hash Generation Strategy: Use base62 encoding (a-z, A-Z, 0-9) to generate short codes. With 7 characters, you get 62^7 = ~3.5 trillion unique URLs.

```
function generateShortCode(id) {
  const chars = '0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ';
  let code = '';
  while (id > 0) {
    code = chars[id % 62] + code;
    id = Math.floor(id / 62);
  }
  return code.padStart(7, '0');
}
```

Database Design: Use a distributed database (Cassandra/DynamoDB) with short_code as partition key for horizontal scaling. Schema: {short_code, original_url, created_at, expiry, user_id}.

Scalability Considerations

- **Write-heavy optimization:** Use auto-incrementing ID generators (Twitter Snowflake) across multiple servers
- **Read-heavy optimization:** Cache popular URLs with 80/20 rule (80% traffic to 20% URLs)
- **CDN:** Serve redirects from edge locations for global users
- **Rate Limiting:** Prevent abuse using token bucket algorithm per user/IP

CAP Theorem: Choose AP (Availability + Partition Tolerance) over Consistency. Eventual consistency is acceptable for URL creation.

2. How would you design a real-time chat application supporting millions of concurrent users? Discuss WebSocket management, message delivery guarantees, and scalability.

Architecture Components

- **WebSocket Servers:** Maintain persistent connections with clients
- **Message Queue:** Kafka/RabbitMQ for reliable message delivery
- **Presence Service:** Track online/offline status using Redis
- **Message Store:** Cassandra for chat history (time-series data)
- **Push Notification Service:** For offline users

WebSocket Management

Connection Handling: Use a connection manager to track user-to-server mappings in Redis.

```
// Store connection mapping
```

```
redis.hset('user:connections', userId, serverId);

// Route message to correct server
const serverId = await redis.hget('user:connections', recipientId);
if (serverId) {
  messageQueue.publish(serverId, message);
}
```

Horizontal Scaling: Use multiple WebSocket servers behind a load balancer with sticky sessions or consistent hashing.

Message Delivery Guarantees

- **At-least-once delivery:** Store messages in queue before sending, acknowledge after delivery
- **Idempotency:** Use message IDs to prevent duplicates on client side
- **Offline handling:** Queue messages in database, deliver on reconnection
- **Acknowledgment system:** Implement sent/delivered/read receipts

Scalability Patterns

- **Sharding:** Partition users across WebSocket servers by user_id hash
- **Message fanout:** For group chats, use pub/sub pattern with Redis or Kafka
- **Backpressure:** Implement rate limiting and message throttling
- **Database:** Use Cassandra with partition key (conversation_id, timestamp) for efficient queries

CAP Trade-off: Prioritize Availability and Partition tolerance. Accept eventual consistency for presence updates but ensure message ordering per conversation.

3. Design a news feed system like Twitter or Facebook. How would you generate personalized feeds for millions of users efficiently?

Feed Generation Approaches

1. Fanout-on-Write (Push Model): Pre-compute feeds when content is created

- When user posts, write to all followers' feeds immediately
- Read is fast (just query user's pre-built feed)
- Write is expensive for users with millions of followers

2. Fanout-on-Read (Pull Model): Compute feed at request time

- Fetch posts from all followees when user requests feed
- Write is fast, read is expensive
- Good for users following many people

Hybrid Approach: Use fanout-on-write for regular users, fanout-on-read for celebrities.

```
function getFeed(userId) {
  if (isRegularUser(userId)) {
    return cache.get(`feed:${userId}`);
  } else {
    const followees = getFollowees(userId);
    return fetchAndMerge(followees);
  }
}
```

Architecture Components

- **Feed Service:** Manages feed generation and ranking
- **Post Service:** Handles post creation and storage
- **Graph Service:** Manages follower/following relationships
- **Ranking Service:** Applies ML models for personalization
- **Cache Layer:** Redis for hot feeds and recent posts

Scalability Strategies

- **Feed Storage:** Store feed items in Redis sorted sets with timestamp scores
- **Pagination:** Use cursor-based pagination instead of offset
- **Denormalization:** Store author info with posts to avoid joins

- **Lazy Loading:** Load feed in batches, fetch details on-demand

Ranking Algorithm

Consider: recency, engagement (likes/comments), user interests, content type. Use machine learning models deployed as microservices.

4. How would you design a distributed rate limiter that works across multiple servers? Explain different algorithms and their trade-offs.

Rate Limiting Algorithms

1. Token Bucket: Tokens added at fixed rate, requests consume tokens

- Allows bursts up to bucket capacity
- Smooth traffic over time
- Most flexible algorithm

```
class TokenBucket {
  constructor(capacity, refillRate) {
    this.capacity = capacity;
    this.tokens = capacity;
    this.refillRate = refillRate;
    this.lastRefill = Date.now();
  }
  allow() {
    this.refill();
    if (this.tokens >= 1) { this.tokens--; return true; }
    return false;
  }
}
```

2. Leaky Bucket: Requests processed at constant rate

- Smooths out bursts completely
- Strict rate enforcement
- May reject valid bursts

3. Fixed Window Counter: Count requests per time window

- Simple to implement
- Boundary issue: 2x rate at window edges

4. Sliding Window Log: Track timestamp of each request

- Most accurate
- Memory intensive

5. Sliding Window Counter: Hybrid of fixed window and sliding log

- Approximates sliding window with less memory
- Good balance of accuracy and efficiency

Distributed Implementation

Using Redis:

```
// Sliding window counter
const key = `rate:${userId}:${currentWindow}`;
const count = await redis.incr(key);
await redis.expire(key, windowSize);
if (count > limit) return false;
```

Challenges:

- **Race conditions:** Use Redis Lua scripts for atomic operations
- **Synchronization:** Eventual consistency acceptable, use local cache + Redis
- **High availability:** Redis cluster with replication

Advanced Patterns

- **Hierarchical limits:** Per-user, per-IP, per-API-key, global
- **Dynamic limits:** Adjust based on system load
- **Distributed coordination:** Use gossip protocol for soft limits across nodes

5. Design a video streaming platform like YouTube. How would you handle video upload, encoding, storage, and adaptive streaming?

System Architecture

- **Upload Service:** Handle large file uploads with resumable upload protocol
- **Transcoding Pipeline:** Convert videos to multiple formats and resolutions
- **CDN:** Distribute video content globally
- **Metadata Service:** Store video information, comments, likes
- **Recommendation Engine:** Suggest videos based on user behavior

Video Upload Flow

1. Chunked Upload: Split large files into chunks for reliability

```
// Client-side chunked upload
async function uploadVideo(file) {
  const chunkSize = 5 * 1024 * 1024; // 5MB
  for (let i = 0; i < file.size; i += chunkSize) {
    const chunk = file.slice(i, i + chunkSize);
    await uploadChunk(chunk, i / chunkSize);
  }
}
```

2. Storage: Upload to object storage (S3/GCS) for durability

- Store original video in cold storage
- Store transcoded versions in hot storage

Video Processing Pipeline

Transcoding: Use distributed workers (FFmpeg) to encode videos

- Multiple resolutions: 360p, 480p, 720p, 1080p, 4K
- Multiple formats: MP4 (H.264), WebM (VP9)
- Audio tracks: AAC, Opus
- Generate thumbnails at different timestamps

Queue-based Processing: Use message queue (SQS/Kafka) for async transcoding jobs

Adaptive Streaming

HLS/DASH Protocol: Break video into small segments (2-10 seconds)

- Create manifest file listing all quality levels
- Client dynamically switches quality based on bandwidth

Scalability Considerations

- **CDN:** Cache video segments at edge locations
- **Database:** Shard by video_id, use Cassandra for metadata
- **Search:** Elasticsearch for video discovery
- **Analytics:** Stream view events to data pipeline (Kafka → Spark)

Cost Optimization: Use different storage tiers, compress older videos, remove unpopular content.

6. Design a distributed cache system. Discuss cache eviction policies, consistency strategies, and handling cache invalidation.

Cache Architecture

- **Client-side cache:** Browser cache, service worker
- **CDN cache:** Edge locations for static assets
- **Application cache:** In-memory cache (Redis/Memcached)
- **Database cache:** Query result cache

Cache Eviction Policies

1. LRU (Least Recently Used): Evict least recently accessed items

```
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;
  }
}
```

2. LFU (Least Frequently Used): Evict least frequently accessed

- Better for long-term access patterns
- More complex to implement

3. TTL (Time To Live): Expire after fixed duration

- Predictable memory usage
- May evict hot data prematurely

4. FIFO: First in, first out - simple but ineffective

Cache Consistency Strategies

Write-Through: Write to cache and database simultaneously

- Strong consistency
- Higher write latency

Write-Back (Write-Behind): Write to cache, async write to database

- Lower latency
- Risk of data loss

Write-Around: Write directly to database, invalidate cache

- Avoids cache pollution
- Cache miss on subsequent reads

Cache Invalidation

- **Time-based:** Set TTL on cache entries
- **Event-based:** Invalidate on data updates using pub/sub
- **Version-based:** Include version in cache key

Distributed Cache Challenges

- **Cache stampede:** Use probabilistic early expiration or locks
- **Hot keys:** Replicate hot data across nodes
- **Thundering herd:** Use request coalescing
- **Consistency:** Accept eventual consistency or use distributed locks

7. How would you design a search autocomplete system like Google's search suggestions? Consider latency, relevance, and scale.

System Requirements

- **Low latency:** Suggestions within 100-200ms
- **High availability:** Always return results
- **Relevance:** Personalized and trending suggestions
- **Scale:** Handle millions of queries per second

Data Structure: Trie (Prefix Tree)

Efficient prefix matching: Store all terms in a trie for O(k) lookup where k is prefix length

```
class TrieNode {
  constructor() {
    this.children = {};
    this.suggestions = [];
    this.isEnd = false;
  }
}

function search(prefix) {
  let node = root;
  for (let char of prefix) {
    if (!node.children[char]) return [];
    node = node.children[char];
  }
  return node.suggestions.slice(0, 10);
}
```

Ranking Algorithm

Factors to consider:

- Query frequency/popularity
- User's search history
- Trending searches
- Geographic location
- Time of day
- Typo tolerance (fuzzy matching)

Store top-k suggestions at each node: Pre-compute and cache most popular completions

Architecture Components

- **Trie Service:** In-memory trie for fast lookups
- **Data Collection:** Log all search queries
- **Analytics Pipeline:** Process logs to update frequencies
- **Trie Builder:** Periodic rebuild of trie with updated data
- **Cache Layer:** Redis for recent/popular queries

Scalability Strategies

- **Sharding:** Partition trie by first few characters
- **Replication:** Multiple read replicas per shard
- **CDN:** Cache suggestions at edge for common prefixes
- **Sampling:** Don't log every query, sample statistically

Personalization

Combine global trie with user-specific history stored in user profile service. Merge results with weighted ranking.

8. Design a notification system that supports multiple channels (email, SMS, push, in-app). How would you ensure reliability and prevent duplicate notifications?

System Architecture

- **Notification Service:** Central orchestrator for all notifications
- **Channel Handlers:** Email (SMTP), SMS (Twilio), Push (FCM/APNS), In-app
- **Message Queue:** Kafka/RabbitMQ for reliable delivery
- **Template Service:** Manage notification templates
- **User Preference Service:** Store user channel preferences
- **Delivery Tracker:** Track notification status

Notification Flow

```
async function sendNotification(userId, event, data) {
  const preferences = await getPreferences(userId);
  const template = await getTemplate(event);

  for (let channel of preferences.channels) {
    const message = renderTemplate(template, data, channel);
    await queue.publish(channel, {
      id: generateId(),
      userId,
      message,
      priority: event.priority
    });
  }
}
```

Reliability Mechanisms

1. Idempotency: Use unique notification IDs to prevent duplicates

- Store sent notification IDs in Redis with TTL
- Check before sending: if exists, skip

2. Retry Logic: Exponential backoff for failed deliveries

- Max retry attempts: 3-5
- Dead letter queue for permanently failed messages

3. Circuit Breaker: Stop sending to failing channels temporarily**4. Acknowledgment:** Confirm delivery from channel providers

Preventing Duplicates

- **Deduplication window:** Track recent notifications per user
- **Rate limiting:** Max N notifications per user per time window
- **Batching:** Combine similar notifications (e.g., "10 new likes")

Priority Queue

Use priority-based message queue:

- Critical: OTP, security alerts (immediate)
- High: Transaction confirmations (seconds)
- Medium: Social updates (minutes)
- Low: Marketing emails (hours)

Scalability

- **Horizontal scaling:** Multiple workers per channel
- **Partitioning:** Shard by user_id or notification_type
- **Monitoring:** Track delivery rates, latency, failures

User Preferences

Allow granular control: channel selection, quiet hours, frequency caps, opt-out options.

9. Design an e-commerce inventory management system. How would you handle concurrent orders, stock updates, and prevent overselling?

Core Challenges

- **Race conditions:** Multiple users buying last item simultaneously
- **Consistency:** Inventory must be accurate across services
- **Performance:** High read/write throughput during sales
- **Distributed transactions:** Order + Payment + Inventory update

Architecture Components

- **Inventory Service:** Manage stock levels

- **Order Service:** Process customer orders
- **Reservation Service:** Temporarily hold inventory
- **Payment Service:** Process payments
- **Cache Layer:** Redis for real-time inventory counts

Preventing Overselling

Approach 1: Pessimistic Locking

```
BEGIN TRANSACTION;
SELECT quantity FROM inventory
  WHERE product_id = ? FOR UPDATE;

IF quantity >= requested_quantity THEN
  UPDATE inventory
    SET quantity = quantity - requested_quantity
    WHERE product_id = ?;
  COMMIT;
ELSE
  ROLLBACK;
END IF;
```

Approach 2: Optimistic Locking with Versioning

```
UPDATE inventory
  SET quantity = quantity - ?, version = version + 1
  WHERE product_id = ?
    AND version = ?
    AND quantity >= ?;
```

Check affected rows: if 0, retry or fail.**Approach 3: Reservation System**

- Reserve inventory when user adds to cart (with TTL)
- Convert reservation to sale on payment
- Release reservation on timeout or cart abandonment

Distributed Approach

Using Redis for High Concurrency:

```
// Atomic decrement
const remaining = await redis.decrby(`stock:${productId}`, quantity);
if (remaining < 0) {
  await redis.incrby(`stock:${productId}`, quantity);
  throw new Error('Out of stock');
}
```

Saga Pattern for Distributed Transactions

- Step 1: Reserve inventory
- Step 2: Process payment
- Step 3: Create order
- Compensating transactions: Rollback on failure

Scalability Strategies

- **Sharding:** Partition inventory by product category or warehouse
- **CQRS:** Separate read (query inventory) and write (update stock) models
- **Event Sourcing:** Store all inventory changes as events
- **Cache invalidation:** Update cache on stock changes via pub/sub

10. Explain how you would design a system to handle distributed transactions across microservices. Discuss patterns like Saga, 2PC, and their trade-offs.

The Problem

In microservices, each service has its own database. Coordinating transactions across multiple services requires distributed transaction management.

Pattern 1: Two-Phase Commit (2PC)

How it works:

- **Phase 1 (Prepare):** Coordinator asks all participants to prepare transaction
- **Phase 2 (Commit):** If all agree, coordinator tells all to commit; otherwise, abort

Pros: Strong consistency, ACID guarantees
Cons: Blocking protocol, single point of failure (coordinator), poor performance, violates microservice autonomy
Verdict: Avoid in microservices; use only when strong consistency is absolutely required

Pattern 2: Saga Pattern

Choreography-based Saga: Services communicate via events

```
// Order Service
await createOrder(order);
emit('OrderCreated', order);

// Payment Service listens
on('OrderCreated', async (order) => {
  try {
    await processPayment(order);
    emit('PaymentProcessed', order);
  } catch (error) {
    emit('PaymentFailed', order);
  }
});
```

Orchestration-based Saga: Central orchestrator controls flow

```
class OrderSaga {
  async execute(order) {
    try {
      await inventoryService.reserve(order);
      await paymentService.charge(order);
      await shippingService.create(order);
    } catch (error) {
      await this.compensate(order);
    }
  }
}
```

Compensating Transactions: Each step has a rollback action

- Reserve inventory → Release inventory
- Charge payment → Refund payment
- Create shipment → Cancel shipment

Pattern 3: Event Sourcing

Store all state changes as immutable events. Rebuild state by replaying events.**Benefits:** Audit trail, temporal queries, easier debugging

Trade-offs Comparison

- **2PC:** Strong consistency, but blocking and slow
- **Saga:** Eventual consistency, non-blocking, complex error handling
- **Event Sourcing:** Complete history, complex to implement

Choosing the Right Pattern

- **Financial transactions:** Saga with orchestration for better control
- **Long-running processes:** Saga with choreography for loose coupling
- **High consistency needs:** Consider keeping services together or using 2PC sparingly

Best Practices

- Design for idempotency in all operations
- Use unique transaction IDs for deduplication
- Implement timeouts and retries
- Monitor saga execution with distributed tracing
- Accept eventual consistency where possible

Coding and Debugging

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

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

Solution

This recursive approach handles arrays of any nesting level:

```
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 concepts:

- Recursive traversal of nested structures
- Type checking with Array.isArray()
- Spread operator for array concatenation
- Time complexity: O(n) where n is total elements

2. How would you reverse a string in JavaScript while handling Unicode characters correctly?

Unicode-Safe String Reversal

Standard reverse methods can break emoji and surrogate pairs. Here's the correct approach:

```
function reverseString(str) {
  return [...str].reverse().join('');
}

// Or using Array.from
function reverseStringAlt(str) {
  return Array.from(str).reverse().join('');
}
```

Why this works:

- Spread operator [...str] correctly handles Unicode code points
- Preserves emoji, accented characters, and surrogate pairs
- str.split('') would break multi-byte characters
- Works with emojis like '□□□□'

3. Write an efficient function to check if a string is a palindrome, ignoring spaces and punctuation.

Optimized Palindrome Check

```
function isPalindrome(str) {
  const cleaned = str.toLowerCase().replace(/[^a-z0-9]/g, '');
  let left = 0, right = cleaned.length - 1;
  while (left < right) {
```

```
    if (cleaned[left] !== cleaned[right]) return false;
    left++;
    right--;
  }
  return true;
}
```

Optimization highlights:

- Two-pointer technique: O(n) time, O(n) space for cleaned string
- Regex removes non-alphanumeric characters efficiently
- Early return on mismatch
- No need to reverse entire string

4. What are the key differences between Chrome DevTools Memory Profiler types: Heap Snapshot, Allocation Timeline, and Allocation Sampling?

Memory Profiling Tools Comparison

Heap Snapshot:

- Captures complete memory state at a specific moment
- Shows all objects, their sizes, and references
- Best for: Finding memory leaks by comparing snapshots
- Use case: Detect detached DOM nodes, retained objects

Allocation Timeline:

- Records allocations over time with timestamps
- Shows when and where objects are created
- Best for: Identifying allocation patterns and spikes
- Higher overhead, use for shorter recordings

Allocation Sampling:

- Statistical sampling of allocations (lower overhead)
- Shows call stacks for allocations
- Best for: Long-running profiling sessions
- Trade-off: Less detailed but more performant

5. Explain the difference between throw, throw new Error(), and custom error handling strategies in production applications.

Error Throwing Best Practices

throw vs throw new Error():

```
// Avoid: loses stack trace info
throw 'Error message';

// Better: proper Error object
throw new Error('Descriptive message');

// Best: custom error classes
class ValidationError extends Error {
  constructor(message, field) {
    super(message);
    this.name = 'ValidationError';
    this.field = field;
  }
}
```

Production strategies:

- Always throw Error objects to preserve stack traces
- Use custom error classes for different error types
- Include context: error codes, user IDs, request IDs
- Implement centralized error handling middleware
- Log errors with severity levels and structured data

- Never expose internal errors to clients

6. How do you debug memory leaks in a Node.js application? What tools and techniques would you use?

Node.js Memory Leak Debugging

Tools and techniques:

- **Process monitoring:** Track process.memoryUsage() over time
- **Heap dumps:** Use --inspect flag and Chrome DevTools
- **heapdump module:** Programmatically capture heap snapshots
- **clinic.js:** Comprehensive performance profiling suite

Common leak sources:

```
// Global variables accumulating data
const cache = {}; // Never cleared

// Event listeners not removed
emitter.on('event', handler); // Missing .off()

// Closures retaining references
setInterval(() => {
  const data = largeObject; // Retained forever
}, 1000);
```

Detection strategy:

- Compare heap snapshots before/after operations
- Look for growing arrays, maps, or object counts
- Check retained size vs shallow size
- Use --max-old-space-size flag to test limits

7. What is monkey patching and when is it appropriate to use it? Provide an example.

Monkey Patching in JavaScript

Definition: Dynamically modifying or extending objects, classes, or modules at runtime.

```
// Example: Adding a method to Array prototype
Array.prototype.last = function() {
  return this[this.length - 1];
};

// Example: Patching a library function
const original = SomeLibrary.method;
SomeLibrary.method = function(...args) {
  console.log('Called with:', args);
  return original.apply(this, args);
};
```

Appropriate use cases:

- Polyfills for missing browser features
- Testing: mocking dependencies temporarily
- Hot-fixing third-party library bugs
- Adding instrumentation/logging

Risks and alternatives:

- Can break code expecting original behavior
- Conflicts with other patches
- Better: Use composition, decorators, or wrapper functions
- In production: Prefer proper inheritance or middleware patterns

8. How would you implement debounce and throttle functions? Explain the difference and use cases.

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: Limits execution to once per time period

```
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, form validation
- **Throttle:** Scroll events, mouse movement, API rate limiting

9. What techniques would you use to debug intermittent race conditions in asynchronous code?

Debugging Race Conditions

Detection techniques:

- **Add logging:** Timestamp all async operations with unique IDs
- **Increase load:** Use stress testing to reproduce more frequently
- **Add artificial delays:** Insert random setTimeout() calls
- **Use Promise inspection:** Track pending/resolved states

Prevention patterns:

```
// Use proper synchronization
let pending = null;
async function fetchData(id) {
  const request = fetch(`/api/${id}`);
  pending = request;
  const response = await request;
  if (pending === request) {
    return response.json();
  }
}
```

Tools:

- Chrome DevTools async stack traces
- Node.js --trace-warnings flag
- async_hooks module for tracking async resources
- Mutex libraries like async-mutex
- State machines to enforce ordering

10. Explain how you would use Chrome DevTools Performance tab to identify and fix a performance bottleneck in a web application.

Performance Profiling Workflow

Step-by-step process:

- **Record:** Start recording, perform the slow operation, stop
- **Analyze FPS:** Look for frame drops (red bars) below 60fps
- **Check Main Thread:** Identify long tasks (yellow/red blocks)
- **Examine Call Tree:** Find functions consuming most time
- **Review Network:** Check for blocking resources

Key metrics to investigate:

- **Scripting time:** JavaScript execution (yellow)
- **Rendering time:** Style calculations, layout (purple)
- **Painting time:** Composite layers (green)
- **Loading time:** Network requests (blue)

Common fixes:

- Debounce expensive operations
- Use requestAnimationFrame for animations
- Virtualize long lists
- Move heavy computation to Web Workers
- Optimize DOM manipulation (batch updates)
- Use CSS transforms instead of layout-triggering properties
- Lazy load images and components

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 debug a critical production issue under pressure.

Situation: Our e-commerce platform experienced a sudden 500% increase in failed checkout transactions during Black Friday, resulting in significant revenue loss.

Task: I was responsible for identifying and resolving the issue within 30 minutes to minimize business impact.

Action: I immediately checked our monitoring dashboards and logs, identified a database connection pool exhaustion issue caused by a recent deployment. I rolled back the problematic code, increased connection pool limits temporarily, and implemented proper connection handling with timeouts.

Result: Restored full functionality within 22 minutes, recovered 95% of abandoned carts through automated email campaigns, and implemented better load testing protocols to prevent future occurrences.

2. Describe a situation where you had to refactor legacy code. How did you approach it?

Situation: I inherited a 5-year-old Node.js application with no tests, inconsistent patterns, and performance issues affecting 100K+ daily users.

Task: Modernize the codebase while maintaining zero downtime and ensuring no regression bugs.

Action: I implemented a phased approach:

- Added comprehensive unit and integration tests for existing functionality
- Refactored modules incrementally using the strangler fig pattern
- Introduced TypeScript gradually for type safety
- Documented architectural decisions and created coding standards

Result: Reduced technical debt by 60%, improved response times by 40%, and decreased bug reports by 55% over 4 months. The team's velocity increased by 30% due to improved code maintainability.

3. Tell me about a time when you disagreed with a technical decision made by your team or manager.

Situation: My team lead proposed migrating our REST API to GraphQL for a project with tight deadlines, despite the team having no GraphQL experience.

Task: I needed to voice my concerns constructively while respecting the lead's authority and finding the best solution for the project.

Action: I requested a meeting to discuss the proposal, prepared a comparison analysis showing implementation timelines, learning curves, and risks. I suggested a hybrid approach: keep REST for the current sprint and pilot GraphQL on a non-critical feature to evaluate feasibility. I emphasized data showing our timeline risks.

Result: The team lead appreciated the thorough analysis. We adopted the hybrid approach, successfully delivered the project on time, and later migrated to GraphQL after the team gained experience through the pilot, resulting in a smoother transition.

4. Describe a time when you had to learn a new technology or framework quickly for a project.

Situation: Our client requested a real-time collaborative document editing feature similar to Google

Docs, requiring WebSocket implementation and operational transformation algorithms—technologies I hadn't used before.

Task: Deliver a working prototype within 2 weeks while learning WebSockets, Socket.io, and conflict resolution strategies.

Action: I created a structured learning plan:

- Spent 2 days studying WebSocket fundamentals and Socket.io documentation
- Built small proof-of-concept examples to test concepts
- Researched existing solutions like Yjs and ShareDB
- Consulted with developers experienced in real-time systems
- Implemented an MVP using Socket.io and a simplified CRDT approach

Result: Delivered the prototype in 12 days with core functionality working for up to 10 concurrent users. The client was impressed, and the feature became a key differentiator, increasing customer retention by 25%.

5. Tell me about a time when you improved the performance of a web application significantly.

Situation: Our SaaS dashboard was experiencing slow load times (8-12 seconds) and high bounce rates, with users complaining about poor user experience.

Task: Reduce initial load time to under 3 seconds and improve overall application responsiveness.

Action: I conducted a comprehensive performance audit:

- Implemented code splitting and lazy loading for routes
- Optimized bundle size by removing unused dependencies (reduced from 2.5MB to 800KB)
- Added Redis caching layer for frequently accessed data
- Optimized database queries and added proper indexing
- Implemented CDN for static assets and image optimization
- Added service workers for offline functionality

Result: Reduced initial load time to 2.1 seconds (82% improvement), decreased bounce rate from 45% to 18%, and improved Lighthouse performance score from 42 to 94. User satisfaction scores increased by 40%.

6. Describe a situation where you had to mentor or help a junior developer overcome a challenge.

Situation: A junior developer on my team was struggling with understanding asynchronous JavaScript, causing delays in their feature development and visible frustration.

Task: Help them build confidence and competence with async patterns without making them feel inadequate.

Action: I scheduled weekly pairing sessions focused on async concepts, starting with callbacks, then Promises, and finally async/await. I created simple, relatable examples and encouraged them to explain concepts back to me. I also:

- Shared curated learning resources
- Reviewed their code with constructive feedback
- Encouraged questions without judgment
- Gradually increased complexity of assigned tasks

Result: Within 6 weeks, the junior developer became proficient with async patterns, completed their feature independently, and even helped another team member with Promise chaining. They later thanked me for the patient approach, and their confidence improved dramatically.

7. Tell me about a time when you had to balance technical debt with feature development.

Situation: Our product roadmap was packed with new features, but our authentication system had accumulated significant technical debt, causing security concerns and maintenance overhead.

Task: Convince stakeholders to prioritize refactoring the auth system while not completely halting feature development.

Action: I prepared a presentation for stakeholders showing:

- Quantified risks of not addressing the debt (potential security vulnerabilities, compliance issues)
- Cost analysis of current maintenance time vs. refactoring investment
- Proposed a phased approach: 60% feature work, 40% refactoring over 2 sprints
- Demonstrated quick wins by refactoring one critical auth flow as proof of concept

Result: Stakeholders approved the plan. We successfully refactored the auth system using OAuth 2.0 and JWT, reduced auth-related bugs by 75%, improved security posture, and still delivered 2 major features. Development velocity actually increased by 20% due to reduced maintenance burden.

8. Describe a time when you made a mistake that affected production. How did you handle it?

Situation: I deployed a database migration script that accidentally deleted a non-null constraint, allowing invalid data to enter our production database, affecting data integrity for approximately 200 customer records.

Task: Fix the immediate issue, recover data integrity, and prevent similar incidents.

Action: I immediately:

- Notified my team lead and stakeholders transparently
- Rolled back the migration and restored the constraint
- Wrote a data cleanup script to identify and fix affected records
- Verified data integrity with the QA team
- Conducted a post-mortem to identify root causes
- Implemented migration review process and staging environment testing requirements

Result: Recovered all affected data within 3 hours with zero data loss. Implemented new safeguards: mandatory peer review for migrations, automated migration testing, and rollback procedures. I documented the incident transparently, which management appreciated, and used it as a learning opportunity for the entire team.

9. Tell me about a time when you had to work with a difficult stakeholder or client.

Situation: A key stakeholder constantly changed requirements mid-sprint, insisted on unrealistic deadlines, and frequently bypassed our project manager to make direct requests to developers.

Task: Establish healthy boundaries and communication patterns while maintaining a positive relationship and delivering quality work.

Action: I scheduled a one-on-one meeting to understand their underlying concerns and business pressures. I:

- Listened empathetically to their priorities and constraints
- Explained the impact of frequent changes on quality and timelines
- Proposed a structured change request process with impact assessments
- Offered weekly demos to provide visibility and early feedback
- Collaborated with PM to create a prioritization framework

Result: The stakeholder appreciated being heard and understood the technical constraints better. Mid-sprint changes decreased by 70%, and when changes were necessary, they followed the proper process. Our relationship improved significantly, and project delivery became more predictable.

10. Describe a time when you had to advocate for better development practices or tools within your team.

Situation: Our team lacked automated testing, leading to frequent production bugs, lengthy manual QA cycles, and developer anxiety during deployments.

Task: Introduce a testing culture and automated testing infrastructure despite initial resistance due to perceived time investment.

Action: I built a business case showing:

- Cost of bugs reaching production vs. time investment in testing
- Demonstrated ROI by adding tests to one problematic module, catching 5 bugs pre-deployment
- Conducted a lunch-and-learn on testing best practices
- Set up Jest and React Testing Library with examples
- Proposed gradual adoption: test new features, add tests when fixing bugs
- Made testing part of code review checklist

Result: Within 3 months, test coverage increased from 0% to 65%, production bugs decreased by 60%, and deployment confidence improved dramatically. The team embraced TDD for complex features, and our deployment frequency increased from weekly to daily.