Backend 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 CAP theorem and how it influences distributed system design decisions.

CAP Theorem Overview

The **CAP theorem** states that a distributed system can only guarantee two out of three properties simultaneously:

- **Consistency (C):** All nodes see the same data at the same time
- **Availability (A):** Every request receives a response (success or failure)
- **Partition Tolerance (P):** System continues operating despite network partitions

Practical Implications

Since network partitions are inevitable in distributed systems, you must choose between **CP** (Consistency + Partition Tolerance) or **AP** (Availability + Partition Tolerance):

- **CP Systems:** MongoDB, HBase, Redis Cluster - prioritize consistency, may reject requests during partitions
- **AP Systems:** Cassandra, DynamoDB, Couchbase - remain available but may serve stale data

Design Decisions

For financial transactions or inventory management, choose CP to prevent data inconsistencies. For social media feeds or content delivery, AP systems provide better user experience with eventual consistency.

2. How does database connection pooling work, and what are the key configuration parameters to optimize?

Connection Pooling Mechanism

Connection pooling maintains a cache of reusable database connections to avoid the overhead of creating new connections for each request. When an application needs a connection, it borrows one from the pool and returns it after use.

Key Configuration Parameters

- **minPoolSize:** Minimum connections maintained (typically 5-10)
- **maxPoolSize:** Maximum connections allowed (calculate based on: concurrent users × avg connections per user)
- **connectionTimeout:** Max wait time for available connection (5-10 seconds)
- **idleTimeout:** Time before idle connections are removed (10-30 minutes)
- **maxLifetime:** Maximum connection lifetime (30 minutes to 2 hours)

Optimization Example

```
// HikariCP configuration (Java)
HikariConfig config = new HikariConfig();
config.setMaximumPoolSize(20);
config.setMinimumIdle(5);
config.setConnectionTimeout(10000);
config.setIdleTimeout(600000);
config.setMaxLifetime(1800000);
config.setLeakDetectionThreshold(60000);
```

Monitor **pool exhaustion** and **connection wait times** to tune these values based on actual load patterns.

3. What are the differences between optimistic and pessimistic locking, and when would you use each?

Optimistic Locking

Optimistic locking assumes conflicts are rare. It checks for conflicts only at update time using version numbers or timestamps.

```
// Example with version field
UPDATE products
SET price = 29.99, version = version + 1
WHERE id = 123 AND version = 5;

if (rowsAffected == 0) {
  throw new OptimisticLockException();
}
```

- **Use when:** Low contention, read-heavy workloads, distributed systems
- **Pros:** Better performance, no lock overhead, prevents deadlocks
- **Cons:** Requires retry logic, can fail transactions

Pessimistic Locking

Pessimistic locking acquires locks immediately, preventing other transactions from accessing data.

```
// SQL example
SELECT * FROM accounts
WHERE id = 123 FOR UPDATE;

// Update after lock acquired
UPDATE accounts SET balance = balance - 100
WHERE id = 123;
```

- **Use when:** High contention, critical data integrity (banking), write-heavy operations
- **Pros:** Guarantees consistency, no retry needed
- **Cons:** Performance overhead, potential deadlocks, reduced concurrency

4. Explain how database indexing works internally and the trade-offs of different index types.

Index Internal Structure

Most databases use **B-tree** or **B+ tree** structures for indexes. B+ trees store data only in leaf nodes, with internal nodes containing only keys for navigation, enabling efficient range queries.

Common Index Types

- **B-tree Index:** Default for most databases, supports equality and range queries, O(log n) lookup
- **Hash Index:** O(1) equality lookups but no range queries, used for exact matches
- **Bitmap Index:** Efficient for low-cardinality columns (gender, status), great for data warehouses
- **Full-text Index:** Inverted index for text search, supports linguistic features
- **Covering Index:** Includes all columns needed for query, avoids table lookup

Trade-offs

```
// Composite index example
CREATE INDEX idx_user_search
ON users(last_name, first_name, email);

// Efficient: uses index
SELECT * FROM users
WHERE last_name = 'Smith';
```

- **Pros:** Faster reads (10-100x), enables query optimization
- **Cons:** Slower writes (30-50% overhead), storage cost (10-20% of table size), maintenance

overhead

Index selectivity matters: high-cardinality columns benefit most from indexing.

5. How do you design an idempotent API, and why is idempotency critical in distributed systems?

Idempotency Definition

Idempotency ensures that multiple identical requests produce the same result as a single request, preventing duplicate operations from network retries or client errors.

Implementation Strategies

- **Idempotency Keys:** Client-generated unique identifiers for each operation
- **Natural Idempotency:** GET, PUT, DELETE are naturally idempotent; POST is not
- **State Checking:** Verify current state before applying changes

Implementation Example

```
// Payment API with idempotency key
POST /api/payments
Idempotency-Key: uuid-12345

// Server-side check
if (redis.exists(idempotencyKey)) {
  return redis.get(idempotencyKey);
}
result = processPayment(request);
redis.setex(idempotencyKey, 86400, result);
return result;
```

Why It's Critical

- **Network failures:** Clients retry requests without knowing if previous attempt succeeded
- **Duplicate prevention:** Prevents double-charging, duplicate records, inconsistent state
- **Distributed transactions:** Enables safe retries in saga patterns and event-driven systems

Store idempotency keys with TTL in Redis or database with unique constraints.

6. What is the N+1 query problem, and how do you solve it in ORM frameworks?

N+1 Query Problem

The **N+1 problem** occurs when an application executes one query to fetch N records, then N additional queries to fetch related data for each record, resulting in N+1 total queries.

Example Problem

```
// Fetches all users (1 query)
users = User.all()

// N additional queries (one per user)
for user in users:
  print(user.posts.count())
```

This generates: SELECT * FROM users + SELECT * FROM posts WHERE user_id = ? (N times)

Solutions

- **Eager Loading:** Load associations in a single query using JOINs
- **Batch Loading:** Load all related records in one additional query
- **DataLoader Pattern:** Batch and cache requests within a single execution context

Implementation Examples

```
// Django: select_related (JOIN)
users = User.objects.select_related('profile').all()
```

```
// Django: prefetch_related (separate query)
users = User.objects.prefetch_related('posts').all()
```

Use **select_related** for foreign keys (1-to-1, many-to-1) and **prefetch_related** for reverse foreign keys and many-to-many relationships.

7. Explain the differences between horizontal and vertical scaling, including when to choose each approach.

Vertical Scaling (Scale Up)

Vertical scaling increases capacity by adding more resources (CPU, RAM, storage) to existing servers.

- **Pros:** Simpler implementation, no application changes, maintains data consistency, lower latency
- **Cons:** Hardware limits, single point of failure, downtime during upgrades, expensive at scale
- **Use when:** Monolithic applications, databases requiring ACID guarantees, initial growth phase

Horizontal Scaling (Scale Out)

Horizontal scaling adds more servers to distribute load across multiple machines.

- **Pros:** Virtually unlimited scaling, fault tolerance, cost-effective with commodity hardware, no downtime
- **Cons:** Complex architecture, data consistency challenges, network overhead, requires load balancing
- **Use when:** Stateless services, microservices, high availability requirements, unpredictable traffic

Hybrid Approach

```
// Load balancer config
upstream app_servers {
  server app1:8080 weight=3;
  server app2:8080 weight=2;
  server app3:8080 weight=1;
}
```

Modern architectures use both: vertically scale databases and horizontally scale application servers. Use **read replicas** and **sharding** for database horizontal scaling.

8. How does message queue processing ensure exactly-once delivery semantics?

Delivery Guarantees

Message queues provide three delivery semantics:

- **At-most-once:** Messages may be lost but never duplicated
- **At-least-once:** Messages never lost but may be duplicated (most common)
- **Exactly-once:** Each message processed exactly once (hardest to achieve)

Exactly-Once Implementation

True exactly-once requires **idempotent processing** combined with **transactional outbox pattern** or **deduplication**:

```
// Idempotent consumer with deduplication
def process_message(msg):
  if db.exists('processed', msg.id):
    return  # Already processed

  with db.transaction():
    perform_business_logic(msg)
    db.insert('processed', msg.id)
    queue.ack(msg)
```

System-Level Support

- **Kafka:** Exactly-once via transactional writes and idempotent producers
- **RabbitMQ:** Publisher confirms + consumer acknowledgments + deduplication
- **AWS SQS:** At-least-once only; requires application-level deduplication

Use **message IDs** stored in database or Redis with TTL to track processed messages. Combine with database transactions for atomicity.

9. What are the key differences between REST and GraphQL, and when would you choose GraphQL over REST?

Core Differences

- **Data Fetching:** REST returns fixed data structures; GraphQL lets clients specify exact fields needed
- **Endpoints:** REST uses multiple endpoints; GraphQL uses single endpoint with query language
- **Over/Under-fetching:** REST often returns too much or too little data; GraphQL eliminates this
- **Versioning:** REST requires version management; GraphQL evolves schema without versions

GraphQL Query Example

```
query {
  user(id: "123") {
    name
    email
    posts(limit: 5) {
      title
      createdAt
    }
  }
}
```

When to Choose GraphQL

- **Complex data requirements:** Multiple related resources, nested relationships
- **Mobile applications:** Bandwidth optimization, reduce round trips
- **Rapid frontend iteration:** Frontend teams need flexibility without backend changes
- **Multiple clients:** Web, mobile, IoT with different data needs

When to Choose REST

- **Simple CRUD operations:** Straightforward resource management
- **Caching requirements:** HTTP caching works out-of-box
- **File uploads:** Better support for multipart requests
- **Team familiarity:** REST is more widely understood

10. Explain database sharding strategies and the challenges of implementing sharding in production systems.

Sharding Strategies

- **Range-based:** Partition by key ranges (user IDs 1-1000, 1001-2000). Simple but can create hotspots
- **Hash-based:** Use hash function on shard key (user_id % num_shards). Even distribution but difficult to rebalance
- **Geographic:** Shard by region/location. Reduces latency but uneven data distribution
- **Directory-based:** Lookup table maps keys to shards. Flexible but adds lookup overhead

Implementation Example

```
// Hash-based sharding
def get_shard(user_id, num_shards):
  shard_id = hash(user_id) % num_shards
  return db_connections[shard_id]

db = get_shard(user_id, 4)
```

```
db.execute("SELECT * FROM users WHERE id = ?", user_id)
```

Production Challenges

- **Cross-shard queries:** JOINs across shards require application-level aggregation or denormalization
- **Rebalancing:** Adding/removing shards requires data migration with minimal downtime
- **Distributed transactions:** ACID guarantees difficult across shards; use saga pattern
- **Hotspot management:** Popular keys create uneven load; use consistent hashing
- **Operational complexity:** Backups, monitoring, schema changes across multiple databases

Consider **Vitess** (MySQL) or **Citus** (PostgreSQL) for managed sharding solutions.

Data Structures and Algorithms

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

1. Explain how you would 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 **doubly linked list** and a **hash map**. The hash map stores key-value pairs with pointers to nodes in the linked list, while the linked list maintains the order of usage.

- **Get operation:** Hash map lookup is O(1), then move the accessed node to the front of the list
- **Put operation:** Add new node to the front; if capacity exceeded, remove the tail node and its hash map entry
- **Space complexity:** O(capacity)

```
class LRUCache:
    def __init__(self, capacity):
        self.cache = {}
        self.capacity = capacity
        self.head = Node(0, 0)
        self.tail = Node(0, 0)
        self.head.next = self.tail
        self.tail.prev = self.head
```

2. What is the time complexity of different operations on a hash table, and what happens during collision resolution?

Hash Table Complexity

Average case time complexity:

- Insert: O(1)
- Delete: O(1)
- Search: O(1)

Worst case: O(n) when all keys hash to the same bucket

Collision resolution strategies:

- **Chaining:** Each bucket contains a linked list of entries with the same hash
- **Open addressing:** Linear probing, quadratic probing, or double hashing to find the next available slot
- **Load factor:** When it exceeds a threshold (typically 0.75), the hash table is resized and rehashed to maintain O(1) performance

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

Two Sum Problem

The optimal approach uses a **hash set** to achieve O(n) time complexity with O(n) space.

Algorithm:

- Iterate through the array once
- For each element, check if (target - element) exists in the hash set
- If found, you have a valid pair
- Add the current element to the hash set

```
def find_pairs(arr, target):
    seen = set()
    pairs = []
    for num in arr:
        complement = target - num
        if complement in seen:
            pairs.append((complement, num))
        seen.add(num)
    return pairs
```

4. Explain the difference between a stack and a queue, and provide a real-world backend use case for each.

Stack vs Queue

Stack (LIFO - Last In First Out):

- Operations: push O(1), pop O(1), peek O(1)
- **Backend use case:** Function call stack, undo/redo functionality, expression evaluation, backtracking algorithms

Queue (FIFO - First In First Out):

- Operations: enqueue O(1), dequeue O(1), peek O(1)
- **Backend use case:** Message queues (RabbitMQ, SQS), task scheduling, BFS traversal, request processing pipelines

```
// Stack example: middleware execution
stack.push(authMiddleware)
stack.push(validationMiddleware)
stack.pop() // Execute in reverse order

// Queue example: job processing
queue.enqueue(emailJob)
queue.dequeue() // Process in order
```

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

Trie (Prefix Tree)

A **Trie** is a tree-based data structure for storing strings where each node represents a character. All descendants of a node share a common prefix.

Time complexity:

- Insert: O(m) where m is the length of the string
- Search: O(m)
- Prefix search: O(p) where p is the prefix length

Backend use cases:

- **Autocomplete systems:** Search suggestions, command completion
- **IP routing tables:** Longest prefix matching
- **Spell checkers:** Dictionary lookups
- **URL routing:** Matching request paths to handlers

6. How does a Binary Search Tree differ from a Balanced BST (like AVL or Red-Black Tree)? When would you choose one over the other?

BST vs Balanced BST

Standard BST:

- Average case: O(log n) for search, insert, delete
- Worst case: O(n) when tree becomes skewed (inserting sorted data)
- No rebalancing overhead

Balanced BST (AVL/Red-Black):

- Guaranteed O(log n) for all operations
- Self-balancing through rotations
- **AVL:** Strictly balanced, faster lookups, slower insertions
- **Red-Black:** Loosely balanced, faster insertions, used in most standard libraries

Choose Balanced BST when: You need guaranteed performance, frequent insertions/deletions, or can't predict input patterns (e.g., database indexes, TreeMap implementations)

7. Implement a sliding window algorithm to find the maximum sum of k consecutive elements in an array. What's the time complexity?

Sliding Window Maximum Sum

The **sliding window technique** optimizes problems involving contiguous subarrays by maintaining a window that slides through the data.

Time complexity: O(n) - single pass through the array
Space complexity: O(1)

```
def max_sum_subarray(arr, k):
    if len(arr) < k:
        return None
    window_sum = sum(arr[:k])
    max_sum = window_sum
    for i in range(k, len(arr)):
        window_sum = window_sum - arr[i-k] + arr[i]
        max_sum = max(max_sum, window_sum)
    return max_sum
```

This avoids the O(n*k) brute force approach by reusing the previous window's sum.

8. What is a Heap data structure and how is it used in priority queues? Explain heapify operation complexity.

Heap and Priority Queue

A **Heap** is a complete binary tree that satisfies the heap property: parent nodes are always greater (max-heap) or smaller (min-heap) than their children.

Operations complexity:

- Insert: O(log n) - add at end, bubble up
- Extract min/max: O(log n) - remove root, bubble down
- Peek: O(1)
- **Heapify:** O(n) - build heap from unsorted array

Backend use cases:

- **Priority queues:** Task scheduling, event processing
- **Dijkstra's algorithm:** Finding shortest paths
- **Top K problems:** Finding k largest/smallest elements
- **Median maintenance:** Using two heaps

9. Explain how you would detect a cycle in a linked list. What are the time and space complexities?

Cycle Detection in Linked List

The optimal solution uses **Floyd's Cycle Detection Algorithm** (Tortoise and Hare).

Algorithm:

- Use two pointers: slow (moves 1 step) and fast (moves 2 steps)
- If there's a cycle, fast will eventually meet slow
- If fast reaches null, there's no cycle

Time complexity: O(n)
Space complexity: O(1)

```
def has_cycle(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:
            return True
    return False
```

Alternative: Use a hash set to track visited nodes - O(n) time, O(n) space.

10. What is the difference between BFS and DFS? Provide a backend scenario where each would be more appropriate.

BFS vs DFS

Breadth-First Search (BFS):

- Uses a **queue**, explores level by level
- Space: O(w) where w is maximum width
- **Best for:** Shortest path, nearest neighbor, level-order traversal

Depth-First Search (DFS):

- Uses a **stack** (or recursion), explores as deep as possible first
- Space: O(h) where h is height
- **Best for:** Topological sort, detecting cycles, path finding

Backend scenarios:

- **BFS:** Social network (find connections within N degrees), web crawlers (level-by-level), cache warming
- **DFS:** Dependency resolution, file system traversal, detecting deadlocks in resource allocation graphs

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

- **API Gateway:** Entry point for URL shortening and redirection requests
- **Application Servers:** Stateless servers handling business logic
- **Database:** Store mappings between short and long URLs
- **Cache Layer:** Redis/Memcached for frequently accessed URLs
- **Load Balancer:** Distribute traffic across multiple servers

URL Generation Strategy

Base62 Encoding: Convert a unique numeric ID to alphanumeric string (a-z, A-Z, 0-9) giving 62^7 = 3.5 trillion combinations for 7-character URLs.

```
function encodeBase62(num) {
  const chars = '0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ';
  let result = '';
  while (num > 0) {
    result = chars[num % 62] + result;
    num = Math.floor(num / 62);
  }
  return result;
}
```

Scalability Considerations

- **Read-heavy system:** 100:1 read-to-write ratio, optimize for reads
- **Caching strategy:** Cache popular URLs with LRU eviction, 80-90% cache hit rate
- **Database sharding:** Partition by hash of short URL for horizontal scaling
- **CDN:** Serve redirects from edge locations for global performance
- **Rate limiting:** Prevent abuse using token bucket algorithm

High Availability

- Multi-region deployment with active-active configuration
- Database replication with master-slave setup
- Health checks and automatic failover

2. How would you design a real-time news feed system like Twitter or Facebook? Discuss the architecture for both pull and push models.

Architecture Overview

Two primary approaches: Pull (Fan-out on read) vs Push (Fan-out on write)

Push Model (Fan-out on Write)

- **When user posts:** Write to all followers' feed caches immediately
- **Pros:** Fast read times, pre-computed feeds
- **Cons:** Expensive writes for users with millions of followers
- **Best for:** Users with moderate follower counts

```
// Pseudo-code for push
function createPost(userId, content) {
  const post = savePost(userId, content);
  const followers = getFollowers(userId);
  followers.forEach(followerId => {
    cache.addToFeed(followerId, post);
  });
}
```

Pull Model (Fan-out on Read)

- **When user requests feed:** Query posts from all followees and merge
- **Pros:** Efficient writes, no wasted computation
- **Cons:** Slower reads, complex merge logic
- **Best for:** Celebrity accounts with millions of followers

Hybrid Approach (Recommended)

- Use **push model** for regular users (< 100k followers)
- Use **pull model** for celebrities/influencers
- Combine both at read time with ranking algorithm
- **Feed ranking:** ML-based scoring considering recency, engagement, user preferences

Technical Components

- **Message Queue:** Kafka/RabbitMQ for asynchronous fan-out
- **Feed Cache:** Redis sorted sets for chronological ordering
- **Timeline Service:** Aggregates and ranks posts
- **WebSockets:** Real-time updates for active users

3. Design a distributed rate limiting system that works across multiple servers. How would you ensure accuracy and prevent race conditions?

Approaches to Distributed Rate Limiting

1. Token Bucket Algorithm with Redis

Centralized approach using Redis atomic operations:

```
function allowRequest(userId, limit, window) {
  const key = `rate:${userId}`;
  const current = redis.incr(key);
  if (current === 1) {
    redis.expire(key, window);
  }
  return current <= limit;
}
```

- **Pros:** Simple, accurate, atomic operations
- **Cons:** Single point of failure, Redis becomes bottleneck

2. Sliding Window Log

- Store timestamps of requests in Redis sorted set
- Remove expired entries and count remaining
- More accurate than fixed windows

```
function slidingWindow(userId, limit, window) {
  const now = Date.now();
  const key = `rate:${userId}`;
  redis.zremrangebyscore(key, 0, now - window);
  const count = redis.zcard(key);
  if (count < limit) {
    redis.zadd(key, now, now);
    return true;
  }
  return false;
}
```

3. Distributed Token Bucket with Local Counters

- Each server maintains local counters
- Periodically sync with central Redis
- **Trade-off:** Slight inaccuracy for better performance

Preventing Race Conditions

- **Lua scripts in Redis:** Execute multiple commands atomically
- **Optimistic locking:** Use WATCH/MULTI/EXEC in Redis
- **Distributed locks:** Redlock algorithm for critical sections

High Availability

- Redis Cluster with replication
- Fallback to local rate limiting if Redis unavailable
- Circuit breaker pattern to prevent cascade failures

4. Explain how you would design a distributed caching system. Discuss cache invalidation strategies, consistency models, and handling cache stampede.

Architecture Components

- **Cache Layer:** Redis/Memcached cluster with consistent hashing
- **Application Tier:** Cache-aside or write-through pattern
- **Invalidation Service:** Manages cache coherency

Cache Distribution Strategy

Consistent Hashing: Minimizes redistribution when nodes are added/removed

```
function getNode(key, nodes) {
  const hash = hashFunction(key);
  const ring = nodes.map(n => hashFunction(n));
  const sorted = ring.sort((a, b) => a - b);
  for (let nodeHash of sorted) {
    if (hash <= nodeHash) return nodeHash;
  }
  return sorted[0];
}
```

Cache Invalidation Strategies

- **TTL-based:** Set expiration time, simple but may serve stale data
- **Write-through:** Update cache on every write, strong consistency
- **Write-behind:** Async updates, better performance but eventual consistency
- **Event-driven:** Publish invalidation events via message queue

Handling Cache Stampede

Problem: Multiple requests simultaneously query DB when cache expires**Solutions:**

- **Probabilistic early expiration:** Refresh before actual expiry
- **Locking mechanism:** First request locks, others wait

```
async function getWithStampedePrevention(key) {
  let value = cache.get(key);
  if (!value) {
    const lock = await acquireLock(key);
    if (lock) {
      value = await db.query(key);
      cache.set(key, value, TTL);
      releaseLock(key);
    } else {
      await waitForLock(key);
      value = cache.get(key);
    }
  }
  return value;
}
```

Consistency Models

- **Strong consistency:** Synchronous invalidation, higher latency
- **Eventual consistency:** Async propagation, better performance
- **Read-your-writes:** User sees their own updates immediately

5. Design a real-time chat application supporting millions of concurrent users. How would you handle message delivery, presence, and scaling?

Core Architecture

- **WebSocket Servers:** Maintain persistent connections with clients
- **Message Queue:** Kafka for message persistence and routing
- **Presence Service:** Track online/offline status
- **Message Store:** Cassandra/MongoDB for message history
- **API Gateway:** REST APIs for non-real-time operations

Message Delivery Flow

```
// WebSocket server handling
function onMessageReceived(userId, message) {
  message.id = generateId();
  message.timestamp = Date.now();
  kafka.publish('messages', message);
  const recipient = message.recipientId;
  if (isOnline(recipient)) {
    sendViaWebSocket(recipient, message);
  }
  db.saveMessage(message);
}
```

Scaling WebSocket Connections

- **Connection distribution:** Use consistent hashing to route users to specific WS servers
- **Server discovery:** Service registry (Consul/etcd) tracks which users are on which servers
- **Message routing:** Redis pub/sub or message queue for inter-server communication
- **Horizontal scaling:** Add more WS servers behind load balancer

Presence Service

- Heartbeat mechanism: clients send periodic pings
- Redis for fast presence lookups with TTL
- Broadcast presence changes to friends/contacts only

Reliability & Delivery Guarantees

- **At-least-once delivery:** Store messages in queue before ACK
- **Message ordering:** Sequence numbers per conversation
- **Offline messages:** Store in DB, deliver on reconnection
- **Read receipts:** Separate acknowledgment system

Optimizations

- **Message batching:** Combine multiple messages in single transmission
- **Connection pooling:** Reuse database connections
- **CDN for media:** Images/videos served from edge locations

6. How would you design a distributed task scheduler that executes jobs at specific times across multiple workers? Address fault tolerance and exactly-once execution.

System Components

- **Scheduler Service:** Manages job definitions and triggers
- **Job Queue:** RabbitMQ/SQS for task distribution
- **Worker Pool:** Stateless workers executing jobs
- **Coordination Service:** ZooKeeper/etcd for distributed locking
- **Metadata Store:** PostgreSQL for job state and history

Job Scheduling Mechanism

```
class DistributedScheduler {
  scheduleJob(job, cronExpr) {
    const nextRun = calculateNext(cronExpr);
    db.saveJob({...job, nextRun, status: 'PENDING'});
    delayQueue.enqueue(job, nextRun - Date.now());
  }

  async executeJob(jobId) {
    const lock = await acquireLock(jobId);
    if (lock) {
      await processJob(jobId);
      releaseLock(jobId);
    }
  }
}
```

Ensuring Exactly-Once Execution

- **Distributed locking:** Acquire lock before execution using Redlock or ZooKeeper
- **Idempotency keys:** Each job execution has unique ID
- **Database transactions:** Update job status atomically
- **Optimistic locking:** Version number in job record

Fault Tolerance

- **Worker health checks:** Monitor worker heartbeats
- **Job timeout:** Automatically reassign stuck jobs
- **Retry mechanism:** Exponential backoff with max attempts
- **Dead letter queue:** Failed jobs after max retries

Scalability Patterns

- **Sharding:** Partition jobs by hash of job ID
- **Priority queues:** Multiple queues for different priorities
- **Auto-scaling workers:** Scale based on queue depth

Implementation Example

```
async function processWithLock(jobId) {
  const lockKey = `lock:job:${jobId}`;
  const acquired = await redis.set(lockKey, 'locked', 'NX', 'EX', 300);
  if (!acquired) return false;
  try {
    await executeJob(jobId);
    await db.updateJobStatus(jobId, 'COMPLETED');
  } finally {
    await redis.del(lockKey);
  }
}
```

7. Design a content delivery network (CDN) from scratch. Explain edge server placement, cache hierarchy, and content invalidation strategies.

CDN Architecture

- **Origin Servers:** Primary content source
- **Edge Servers:** Geographically distributed caching nodes
- **DNS System:** GeoDNS for routing users to nearest edge
- **Control Plane:** Configuration and monitoring
- **Purge System:** Content invalidation mechanism

Edge Server Placement Strategy

- **Geographic distribution:** Deploy in major internet exchange points (IXPs)
- **Criteria for placement:** User density, network latency, peering agreements

- **Anycast routing:** Same IP announced from multiple locations
- **Traffic analysis:** Add capacity where request volume is high

Cache Hierarchy

Three-tier caching model:

- **Browser cache:** Client-side with Cache-Control headers
- **Edge cache:** First CDN tier, serves most requests
- **Regional cache:** Mid-tier aggregation point
- **Origin shield:** Protects origin from cache misses

```
function handleRequest(url, edgeServer) {
  let content = edgeServer.cache.get(url);
  if (!content) {
    content = regionalCache.get(url);
    if (!content) {
      content = originShield.fetch(url);
    }
    edgeServer.cache.set(url, content);
  }
  return content;
}
```

Content Invalidation Strategies

- **Time-based (TTL):** Content expires after set duration
- **Purge API:** Immediate invalidation via API call
- **Tag-based purging:** Invalidate groups of related content
- **Versioned URLs:** Change URL when content updates (cache busting)

Cache Key Design

```
function generateCacheKey(request) {
  const base = request.url;
  const vary = [
    request.headers['accept-encoding'],
    request.headers['accept-language'],
    request.device.type
  ];
  return hash(base + vary.join(':'));
}
```

Performance Optimizations

- **TCP optimization:** Keep-alive connections, TCP Fast Open
- **Compression:** Gzip/Brotli on the fly
- **HTTP/2 & HTTP/3:** Multiplexing and QUIC protocol
- **Smart routing:** Real-time latency measurement between nodes

8. Explain how you would design a search engine like Elasticsearch. Discuss indexing, query processing, and relevance ranking at scale.

Core Architecture

- **Indexing Pipeline:** Ingest, analyze, and store documents
- **Inverted Index:** Map terms to document IDs
- **Distributed Storage:** Sharded indices across cluster
- **Query Coordinator:** Route and aggregate search requests
- **Ranking Engine:** Score and sort results

Indexing Process

```
class SearchIndex {
  indexDocument(docId, content) {
    const tokens = this.analyze(content);
    tokens.forEach(token => {
      this.invertedIndex[token] =
        this.invertedIndex[token] || [];
      this.invertedIndex[token].push(docId);
    });
  }

  analyze(text) {
    return text.toLowerCase().split(/\s+/);
  }
}
```

Inverted Index Structure

- **Term dictionary:** All unique terms with metadata
- **Posting lists:** Document IDs containing each term
- **Term frequency:** Count of term occurrences per document
- **Position information:** For phrase queries

Query Processing

- **Query parsing:** Tokenize and analyze search terms
- **Boolean retrieval:** AND/OR/NOT operations on posting lists
- **Distributed execution:** Query each shard in parallel
- **Result merging:** Combine and sort results from all shards

Relevance Ranking (TF-IDF & BM25)

TF-IDF formula: Score = TF × IDF

- **Term Frequency (TF):** How often term appears in document
- **Inverse Document Frequency (IDF):** Rarity of term across all documents
- **BM25:** Improved ranking with saturation function

Scaling Strategies

- **Horizontal sharding:** Partition index by document hash
- **Replication:** Multiple copies for fault tolerance and read scaling
- **Segment merging:** Periodically merge small index segments
- **Caching:** Query results and filter caches

Advanced Features

- **Fuzzy matching:** Levenshtein distance for typos
- **Faceting:** Aggregate results by categories
- **Highlighting:** Show matching text snippets
- **Auto-complete:** Prefix trees (tries) for suggestions

9. Design a distributed file storage system like Amazon S3 or Google Cloud Storage. Address data durability, consistency, and efficient retrieval.

System Architecture

- **API Gateway:** RESTful interface for upload/download
- **Metadata Service:** Tracks file locations and properties
- **Storage Nodes:** Actual file storage across data centers
- **Replication Manager:** Ensures data durability
- **Load Balancer:** Distributes requests across nodes

Data Organization

Object storage model:

- **Buckets:** Logical containers for objects
- **Objects:** Files with unique keys (paths)
- **Metadata:** Content-type, size, timestamps, custom tags

```
class ObjectStore {
  putObject(bucket, key, data, metadata) {
    const objectId = generateId();
    const chunks = this.splitIntoChunks(data);
    const locations = [];
    chunks.forEach(chunk => {
      const nodes = this.selectNodes(3);
      nodes.forEach(n => n.store(chunk));
      locations.push(nodes);
    });
    this.metadata.save({bucket, key, objectId, locations});
  }
}
```

Data Durability & Replication

- **Replication factor:** Store 3+ copies across different availability zones
- **Erasure coding:** Split data into fragments with parity (e.g., 10+4 scheme)
- **Checksum verification:** MD5/SHA256 to detect corruption
- **Background scrubbing:** Periodic integrity checks

Consistency Model

- **Strong consistency:** For metadata operations (PUT, DELETE)
- **Eventual consistency:** For replicas after initial write
- **Read-after-write consistency:** Guaranteed for single client

Efficient Retrieval

- **Content addressing:** Hash-based lookup for deduplication
- **CDN integration:** Cache frequently accessed objects at edge
- **Range requests:** HTTP byte-range for partial downloads
- **Multipart downloads:** Parallel chunk retrieval

Metadata Management

```
// Metadata stored in distributed database
const metadata = {
  bucket: 'my-bucket',
  key: 'photos/image.jpg',
  objectId: 'uuid-1234',
  size: 1048576,
  chunks: [
    {id: 'c1', nodes: ['n1', 'n2', 'n3']},
    {id: 'c2', nodes: ['n4', 'n5', 'n6']}
  ],
  created: '2024-01-01T00:00:00Z'
};
```

Scalability

- **Horizontal scaling:** Add storage nodes dynamically
- **Consistent hashing:** Distribute objects evenly
- **Metadata sharding:** Partition by bucket or key prefix

10. How would you design a payment processing system handling millions of transactions? Discuss idempotency, consistency, and handling failures.

System Architecture

- **Payment Gateway:** Entry point for payment requests
- **Transaction Service:** Orchestrates payment flow
- **Ledger System:** Double-entry bookkeeping for all transactions
- **Payment Processor Integration:** Third-party APIs (Stripe, PayPal)
- **Fraud Detection:** Real-time risk assessment
- **Notification Service:** Async updates to users

Transaction Flow

```
async function processPayment(request) {
  const idempotencyKey = request.idempotencyKey;
  const existing = await checkDuplicate(idempotencyKey);
  if (existing) return existing;

  const txn = await createTransaction(request);
  await fraudCheck(txn);
  const result = await chargePaymentMethod(txn);
  await updateLedger(txn, result);
  await notifyUser(txn);
  return result;
}
```

Ensuring Idempotency

- **Idempotency keys:** Client-provided unique identifier per request
- **Database constraints:** Unique index on idempotency key
- **State machine:** Track transaction states (pending, processing, completed, failed)
- **TTL on keys:** Expire old idempotency records after 24-48 hours

Consistency Guarantees

Two-Phase Commit for distributed transactions:

- **Phase 1:** Reserve funds, validate all preconditions
- **Phase 2:** Commit or rollback based on all participants

Saga Pattern for long-running transactions:

- Break into smaller local transactions
- Compensating transactions for rollback

Handling Failures

- **Retry with exponential backoff:** Transient failures from payment processor
- **Circuit breaker:** Stop calling failing service temporarily
- **Timeout configuration:** Prevent indefinite waits
- **Dead letter queue:** Manual review for persistent failures

Double-Entry Ledger

```
function recordTransaction(from, to, amount) {
  db.transaction(() => {
    db.insert('ledger', {
      account: from,
      debit: amount,
      txnId: txnId
    });
    db.insert('ledger', {
      account: to,
      credit: amount,
      txnId: txnId
    });
  });
}
```

Security & Compliance

- **PCI DSS compliance:** Never store raw card data
- **Tokenization:** Replace sensitive data with tokens
- **Encryption:** TLS in transit, AES-256 at rest
- **Audit logging:** Immutable log of all operations
- **Reconciliation:** Daily balance checks with payment processors

Coding and Debugging

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

1. Write a function to flatten a nested list/array of arbitrary depth without using built-in flatten methods.

Solution

Here's an efficient recursive approach to flatten a nested list:

```
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
- Checks each element with Array.isArray()
- Spreads recursive results into the result array
- Time complexity: O(n) where n is total number of elements

2. Implement a function to check if a string is a palindrome, considering only alphanumeric characters and ignoring case.

Solution

An optimized two-pointer approach:

```
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;
}
```

Optimizations:

- Preprocesses string once to remove non-alphanumeric characters
- Two-pointer technique for O(n) time complexity
- Space complexity: O(n) for cleaned string
- Early return on mismatch for better average performance

3. How would you debug a memory leak in a Node.js application? What tools and techniques would you use?

Debugging Memory Leaks

Tools and Techniques:

- **Node.js --inspect flag:** Enable Chrome DevTools for heap snapshots and memory profiling

- **process.memoryUsage():** Monitor heapUsed, heapTotal, and external memory programmatically
- **Heap Snapshots:** Take multiple snapshots over time and compare to identify growing objects
- **clinic.js:** Use clinic doctor and clinic heapprofiler for automated analysis
- **Common causes:** Event listeners not removed, global variables accumulating data, closures retaining references, timers not cleared

```
// Monitor memory usage
setInterval(() => {
  const used = process.memoryUsage();
  console.log(`Heap: ${Math.round(used.heapUsed / 1024 / 1024)}MB`);
}, 5000);
```

4. Explain exception handling best practices in backend applications. How do you handle errors in async/await code?

Exception Handling Best Practices

For Async/Await:

```
async function fetchUserData(userId) {
  try {
    const user = await db.users.findById(userId);
    if (!user) throw new NotFoundError('User not found');
    return user;
  } catch (error) {
    logger.error('fetchUserData failed', { userId, error });
    throw error;
  }
}
```

Key Practices:

- **Custom error classes:** Create specific error types (NotFoundError, ValidationError) for better error handling
- **Centralized error handler:** Use middleware to catch all errors in one place
- **Logging:** Always log errors with context before re-throwing
- **Never swallow errors:** Always propagate or handle appropriately
- **Graceful degradation:** Return meaningful responses to clients

5. Write a function to reverse a string without using built-in reverse methods. Optimize for Unicode characters.

Solution

Handling Unicode properly requires considering grapheme clusters:

```
function reverseString(str) {
  const graphemes = [...str];
  let left = 0, right = graphemes.length - 1;
  while (left < right) {
    [graphemes[left], graphemes[right]] = [graphemes[right], graphemes[left]];
    left++; right--;
  }
  return graphemes.join('');
}
```

Why this approach:

- Spread operator [...str] properly handles Unicode surrogate pairs
- In-place swapping using destructuring for efficiency
- Handles emojis and special characters correctly
- Time: O(n), Space: O(n)

6. What is monkey patching and when would you use it? Provide an example and discuss the risks.

Monkey Patching

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

```
// Example: Adding a method to Array prototype
const original = Array.prototype.map;
Array.prototype.map = function(...args) {
  console.log('Map called with', this.length, 'items');
  return original.apply(this, args);
};
```

Valid Use Cases:

- Polyfills for missing browser/runtime features
- Testing: mocking dependencies or adding debug hooks
- Hot-patching critical bugs in production (temporary)
- Adding instrumentation for monitoring

Risks:

- Breaking changes when libraries update
- Hard to debug and maintain
- Conflicts with other patches
- Violates principle of least surprise

7. How would you profile and optimize a slow database query? Walk through your debugging process.

Database Query Optimization Process

Step-by-step approach:

- **1. Enable query logging:** Use slow query logs to identify problematic queries
- **2. Analyze execution plan:** Use EXPLAIN/EXPLAIN ANALYZE to understand query execution
- **3. Check indexes:** Ensure appropriate indexes exist on WHERE, JOIN, and ORDER BY columns
- **4. Look for N+1 queries:** Use query counters or APM tools to detect multiple queries in loops
- **5. Optimize query structure:** Avoid SELECT *, use appropriate JOINs, add LIMIT clauses
- **6. Consider caching:** Redis or application-level caching for frequently accessed data
- **7. Monitor metrics:** Track query time, rows examined vs returned, and cache hit rates

```
// Example: Detecting N+1 in Node.js
const posts = await Post.findAll();
for (let post of posts) {
  post.author = await User.findById(post.authorId); // N+1!
}
// Fix: Use eager loading
const posts = await Post.findAll({ include: 'author' });
```

8. Implement a debounce function from scratch. Explain when you would use it in backend development.

Debounce Implementation

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

Backend Use Cases:

- **Search APIs:** Debounce autocomplete requests to reduce database load
- **Logging/Analytics:** Batch log writes instead of writing on every event
- **Rate limiting:** Delay expensive operations like email sending
- **File system watchers:** Debounce file change events to avoid multiple rebuilds
- **Webhooks:** Prevent duplicate webhook deliveries from rapid events

Note: For backend, consider using a queue system (Bull, RabbitMQ) for more robust debouncing/throttling.

9. What debugging tools and techniques do you use to diagnose performance issues in production?

Production Performance Debugging

Tools:

- **APM Solutions:** New Relic, DataDog, Dynatrace for end-to-end tracing
- **Distributed Tracing:** Jaeger, Zipkin for microservices
- **Profilers:** Node.js --prof, Python cProfile, Java JProfiler
- **Metrics:** Prometheus + Grafana for real-time monitoring
- **Logging:** ELK Stack (Elasticsearch, Logstash, Kibana) for log analysis

Techniques:

- **CPU profiling:** Identify hot code paths consuming CPU cycles
- **Flame graphs:** Visualize stack traces to find bottlenecks
- **Request tracing:** Track requests across services with correlation IDs
- **Database query analysis:** Monitor slow queries and connection pool usage
- **Load testing:** Use k6, JMeter to reproduce issues under load

10. Write a function to implement deep cloning of an object, handling circular references. What are the edge cases?

Deep Clone with Circular References

```
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;
}
```

Edge Cases Handled:

- **Circular references:** WeakMap tracks visited objects
- **Primitives:** Return directly without cloning
- **Arrays:** Initialize as array, not object
- **Null:** Handled separately (typeof null === 'object')

Not Handled (advanced cases):

- Functions, Dates, RegExp, Map, Set (require special handling)
- Non-enumerable properties
- Prototype chain

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 slow-performing backend system. What was your approach?

Situation: Our e-commerce API was experiencing response times of 3-5 seconds during peak traffic, causing cart abandonment rates to increase by 15%.

Task: I was tasked with identifying bottlenecks and reducing response times to under 500ms within two weeks.

Action: I implemented a three-pronged approach:

- Added database query profiling and identified N+1 queries in product listings
- Implemented Redis caching for frequently accessed product data with a 5-minute TTL
- Introduced database connection pooling and optimized indexes on high-traffic tables

Result: Response times dropped to an average of 300ms, cart abandonment decreased by 22%, and the system handled 3x more concurrent users without degradation.

2. Describe a situation where you had to make a critical technical decision under pressure. How did you handle it?

Situation: During a production deployment, our payment processing service started failing with a 40% error rate, affecting real customer transactions.

Task: As the lead backend engineer on-call, I needed to decide whether to rollback immediately or attempt a forward fix while minimizing financial impact.

Action:

- Quickly analyzed logs and identified a database migration had locked critical tables
- Consulted with the team lead via Slack within 5 minutes
- Decided to rollback the deployment while implementing a circuit breaker pattern to queue failed transactions
- Documented the incident and created a post-mortem

Result: System was restored in 12 minutes, only 23 transactions were affected (all successfully reprocessed), and we implemented better migration testing protocols that prevented similar issues.

3. Give an example of when you had to deal with conflicting requirements from different stakeholders. How did you resolve it?

Situation: The product team wanted to add real-time notifications requiring WebSocket connections, while the infrastructure team mandated we reduce server costs by 30%.

Task: I needed to find a solution that satisfied both teams without compromising system reliability.

Action:

- Organized a technical discovery meeting with both stakeholders to understand core requirements
- Proposed a hybrid solution using Server-Sent Events (SSE) for notifications with a fallback to polling
- Presented cost analysis showing SSE would use 60% fewer resources than WebSockets for our use case
- Built a proof-of-concept demonstrating the approach

Result: Both teams approved the solution, we reduced infrastructure costs by 35%, and delivered real-time notifications with 99.9% uptime. The product team was satisfied with the user experience.

4. Tell me about a time when you identified a major technical debt issue. How did you convince leadership to prioritize it?

Situation: Our monolithic authentication service was becoming a single point of failure, causing 3-4 outages per month affecting all products.

Task: I needed to convince leadership to allocate two sprints for refactoring into a distributed architecture despite aggressive feature deadlines.

Action:

- Collected six months of incident data showing authentication failures cost 40+ engineering hours monthly
- Created a cost-benefit analysis demonstrating the refactor would pay for itself in 3 months
- Proposed a phased migration approach that wouldn't block feature development
- Presented a risk assessment showing potential revenue loss from continued outages

Result: Leadership approved the initiative. Post-refactor, authentication-related incidents dropped by 90%, and the team saved approximately 35 hours per month in incident response.

5. Describe a situation where you mentored a junior developer through a challenging technical problem.

Situation: A junior developer on my team was struggling to implement a complex data aggregation feature that required joining data from multiple microservices.

Task: I needed to help them complete the feature within the sprint while building their confidence and skills.

Action:

- Scheduled daily 30-minute pairing sessions to work through the problem together
- Taught them about the saga pattern and event-driven architecture through whiteboarding
- Encouraged them to propose solutions first, then guided them through tradeoffs
- Reviewed their code with constructive feedback focused on patterns, not just syntax

Result: They successfully delivered the feature on time with 95% test coverage. Six months later, they independently designed and implemented a similar system for another team. They later mentioned this experience as pivotal to their growth.

6. Tell me about a time when you had to debug a critical production issue with limited information.

Situation: Our order processing system started silently dropping 5% of orders with no error logs or alerts, discovered only when customers complained.

Task: I needed to identify the root cause and fix it immediately while we had minimal logging and no reproduction steps.

Action:

- Added temporary verbose logging to production with sampling to avoid performance impact
- Analyzed database transaction logs and discovered deadlock patterns
- Correlated timing with a recent deployment that changed transaction isolation levels
- Implemented proper retry logic with exponential backoff and dead letter queues
- Added comprehensive monitoring and alerting for queue depths

Result: Identified and fixed the issue within 4 hours. Recovered all dropped orders from event logs. Implemented monitoring that now alerts us within 30 seconds of similar issues.

7. Describe a time when you had to learn a new technology quickly to solve a business problem.

Situation: Our company needed to implement GDPR-compliant data deletion across 15 microservices within 6 weeks, requiring event sourcing patterns I hadn't used before.

Task: As the technical lead, I needed to design the solution and guide the team despite having no prior event sourcing experience.

Action:

- Spent the first week intensively studying event sourcing, CQRS, and Kafka through courses and documentation
- Built a small proof-of-concept to validate the approach
- Created architectural diagrams and documentation for the team
- Led daily knowledge-sharing sessions to bring the team up to speed
- Implemented the core framework and delegated service-specific implementations

Result: Delivered the GDPR compliance system 3 days ahead of schedule. The event sourcing architecture later became our standard pattern, improving system auditability and debugging capabilities across the platform.

8. Give an example of when you had to compromise on technical excellence to meet a business deadline.

Situation: We had a contractual obligation to integrate with a major partner's API within 3 weeks, but building it properly with full error handling and testing would take 5 weeks.

Task: I needed to deliver a working integration on time while managing technical debt responsibly.

Action:

- Identified the minimum viable scope with the product team: core happy path only
- Implemented the integration with clear TODO comments and technical debt tickets
- Added feature flags to enable quick rollback if issues arose
- Created a detailed technical debt backlog with effort estimates
- Negotiated with leadership to allocate 2 sprints post-launch for hardening

Result: Delivered on time, integration worked for 95% of cases. Secured the partnership worth $2M annually. Completed the hardening work as planned, bringing test coverage to 90% and adding comprehensive error handling.

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

Situation: My team decided to use MongoDB for a new financial transactions service, but I believed a relational database was more appropriate due to ACID requirements.

Task: I needed to voice my concerns professionally while respecting the team's decision-making process.

Action:

- Prepared a written analysis comparing both approaches with specific examples relevant to our use case
- Requested a technical review meeting and presented my concerns with data on transaction consistency requirements
- Listened to the team's rationale about scalability concerns
- Proposed a compromise: use PostgreSQL with JSONB columns for flexibility and proper ACID guarantees

Result: The team agreed to the PostgreSQL approach. Six months later, we avoided a critical data consistency bug that affected a competitor using a similar MongoDB setup. The team appreciated my constructive approach to disagreement.

10. Describe a situation where you had to handle a security vulnerability in production.

Situation: A security researcher reported an SQL injection vulnerability in our API that could expose user email addresses and hashed passwords.

Task: As the backend lead, I needed to coordinate an immediate response, patch the vulnerability, and assess the damage.

Action:

- Immediately assembled a response team including security, DevOps, and legal
- Analyzed server logs to determine if the vulnerability had been exploited (found no evidence)
- Deployed a hotfix within 2 hours using parameterized queries
- Conducted a comprehensive audit of all endpoints for similar vulnerabilities
- Implemented automated SQL injection testing in our CI/CD pipeline
- Coordinated with legal on disclosure requirements

Result: Vulnerability patched with zero confirmed exploits. Discovered and fixed 3 additional potential injection points. Implemented security training for all engineers and established a bug bounty program. The researcher praised our response time and transparency.