TypeScript

Interview Questions and Answers

Core Concepts

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

1. Explain the difference between 'type' and 'interface' in TypeScript. When would you choose one over the other?

Key Differences

- **Declaration merging:** Interfaces support automatic merging when declared multiple times, types do not
- **Extends vs Intersection:** Interfaces use 'extends', types use '&' for composition
- **Computed properties:** Types support mapped and conditional types more naturally
- **Primitives:** Types can alias primitives and unions, interfaces cannot

When to Use Interface

- Defining object shapes, especially for public APIs
- When you need declaration merging (e.g., extending third-party libraries)
- Class contracts and object-oriented patterns

When to Use Type

- Complex type compositions with unions or intersections
- Mapped types and conditional types
- Aliasing primitives or tuples

```
// Interface - better for objects
interface User {
  id: string;
  name: string;
}

// Type - better for unions/complex types
type Status = 'pending' | 'active' | 'inactive';
type Result = { success: true; data: T } | { success: false; error: string };
```

2. What are conditional types in TypeScript? Provide a practical example of their usage.

Conditional Types

Conditional types allow you to create types that depend on a condition, following the pattern **T extends U ? X : Y**. They enable powerful type transformations and are fundamental to advanced TypeScript patterns.

Practical Example

```
// Extract promise return type
type Unwrap = T extends Promise ? U : T;

type A = Unwrap>; // string
type B = Unwrap; // number

// Flatten array types
type Flatten = T extends Array ? U : T;

type C = Flatten; // string
type D = Flatten; // number
```

Real-World Use Case

Conditional types are essential for building type-safe API clients, creating flexible utility types, and implementing type guards that preserve type information through transformations.

3. Explain mapped types and provide an example of creating a deep partial type.

Mapped Types

Mapped types transform properties of an existing type by iterating over keys. They use the syntax **[K in keyof T]** to map over type properties and apply transformations.

Deep Partial Implementation

```
type DeepPartial = {
  [K in keyof T]?: T[K] extends object
    ? DeepPartial
    : T[K];
};

interface Config {
  database: {
    host: string;
    port: number;
  };
  cache: { ttl: number; };
}

type PartialConfig = DeepPartial;
```

Common Use Cases

- **Readonly/Mutable transformations:** Making all properties readonly or writable
- **Nullable types:** Adding null/undefined to all properties
- **Property filtering:** Omitting or picking properties based on conditions
- **Type validation:** Ensuring type constraints across nested structures

4. What is the 'infer' keyword and how does it work in TypeScript?

Infer Keyword

The **infer** keyword is used within conditional types to declare a type variable that TypeScript should infer from the structure being matched. It allows extracting and naming types from complex type expressions.

Syntax and Examples

```
// Extract return type of function
type ReturnType = T extends (...args: any[]) => infer R ? R : never;

// Extract array element type
type ArrayElement = T extends (infer U)[] ? U : never;

// Extract promise value type
type PromiseValue = T extends Promise ? V : T;

type Func = () => string;
type Result = ReturnType; // string
```

Advanced Pattern

```
// Extract constructor parameters
type ConstructorParams = T extends new (...args: infer P) => any ? P : never;
```

The infer keyword is essential for creating utility types that extract information from function signatures, class constructors, and generic type parameters.

5. How do template literal types work? Provide an example of building a type-safe event system.

Template Literal Types

Template literal types use the same syntax as JavaScript template literals but operate at the type level, allowing you to construct new string literal types by combining and transforming existing ones.

Type-Safe Event System

```
type EventName = 'click' | 'focus' | 'blur';
type Handler = 'Handler' | 'Listener';
type EventHandler = `on${Capitalize}${Handler}`;
// Result: 'onClickHandler' | 'onClickListener' | ...

type PropEventKey = `on${Capitalize}`;
type ClickEvent = PropEventKey<'click'>; // 'onClick'

type HTTPMethod = 'GET' | 'POST';
type Endpoint = `/api/${string}`;
type Route = `${HTTPMethod} ${Endpoint}`;
```

Benefits

- Type-safe string manipulation at compile time
- Automatic inference of valid string combinations
- Reduces runtime errors in event handling and routing
- Excellent autocomplete support in IDEs

6. Explain variance in TypeScript. What are covariance, contravariance, and bivariance?

Variance Concepts

Variance describes how subtyping between complex types relates to subtyping between their components. It determines type safety in assignments and function parameters.

Covariance

Definition: If A is a subtype of B, then Type<A> is a subtype of Type<B>. Arrays and return types are covariant.

```
class Animal { }
class Dog extends Animal { bark() {} }

// Covariant - Arrays
let dogs: Dog[] = [new Dog()];
let animals: Animal[] = dogs; // OK
```

Contravariance

Definition: If A is a subtype of B, then Type<B> is a subtype of Type<A>. Function parameters are contravariant in strict mode.

```
type Handler = (arg: T) => void;
let animalHandler: Handler = (a) => {};
let dogHandler: Handler = animalHandler; // OK
```

Bivariance

Parameters are both covariant and contravariant (unsound). Occurs with method syntax when strictFunctionTypes is false.

7. What are discriminated unions and how do they enable exhaustive type checking?

Discriminated Unions

Discriminated unions (tagged unions) combine union types with a common literal property (discriminant) that TypeScript uses to narrow types automatically. They enable type-safe state machines and result types.

Implementation Example

```
type Success = { status: 'success'; data: T };
type Error = { status: 'error'; error: string };
type Loading = { status: 'loading' };

type Result = Success | Error | Loading;

function handle(result: Result) {
  switch (result.status) {
    case 'success': return result.data;
    case 'error': return result.error;
    case 'loading': return null;
  }
}
```

Exhaustive Checking

```
function assertNever(x: never): never {
  throw new Error('Unexpected value: ' + x);
}

// TypeScript errors if case is missing
switch (result.status) {
  case 'success': return result.data;
  default: return assertNever(result);
}
```

This pattern ensures compile-time verification that all cases are handled.

8. How do you implement a type-safe builder pattern in TypeScript?

Type-Safe Builder Pattern

A builder pattern that enforces required properties at compile time using conditional types and method chaining.

Implementation

```
type Builder = {
  set

(key: P, val: T[P]): Builder;
  build: K extends keyof T ? () => T : never;
};

function createBuilder(): Builder {
  const data: Partial = {};
  return {
    set(key, val) {
      data[key] = val;
      return this as any;
    },
    build: (() => data) as any
  };
}
```

Usage

```
interface User { name: string; email: string; }
const builder = createBuilder();
builder.set('name', 'John').build(); // Error: missing email
builder.set('name', 'John').set('email', 'j@ex.com').build(); // OK
```

This approach prevents runtime errors by ensuring all required properties are set before calling build().

9. Explain the difference between 'unknown' and 'any'. When should you use each?

Key Differences

- **any:** Disables type checking completely; allows any operation without validation
- **unknown:** Type-safe alternative; requires type checking before use

Unknown - Type-Safe Approach

```
function processValue(val: unknown) {
  // Error: must check type first
  // val.toUpperCase();

  if (typeof val === 'string') {
    return val.toUpperCase(); // OK
  }
  if (val instanceof Date) {
    return val.getTime(); // OK
  }
}
```

Any - Escape Hatch

```
function legacy(val: any) {
  return val.toUpperCase(); // No error, unsafe
}
```

When to Use

- **unknown:** When receiving data from external sources (API responses, user input), migrating from JavaScript, or when type is truly unknown
- **any:** Only when interfacing with untyped libraries, during gradual migration, or as a last resort (prefer unknown)

Always prefer **unknown** over any for better type safety.

10. What are const assertions and how do they differ from regular const declarations?

Const Assertions

Const assertions (using **as const**) tell TypeScript to infer the most specific type possible, making all properties readonly and literal types instead of widened types.

Comparison

```
// Regular const - type widening
const config = {
  endpoint: 'https://api.example.com',
  timeout: 5000
};
// Type: { endpoint: string; timeout: number }

// Const assertion - literal types
const strictConfig = {
  endpoint: 'https://api.example.com',
  timeout: 5000
} as const;
// Type: { readonly endpoint: 'https://api.example.com'; readonly timeout: 5000 }
```

Array Example

```
const routes = ['/', '/about', '/contact'] as const;
type Route = typeof routes[number]; // '/' | '/about' | '/contact'
```

Benefits

- Prevents accidental mutations
- Enables precise type inference for literal values
- Creates union types from array literals

- Useful for configuration objects and enums

Data Structures and Algorithms

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

1. How would you implement a generic Stack data structure in TypeScript with type safety?

A generic Stack implementation ensures type safety while maintaining LIFO behavior.

Implementation:

```
class Stack {
  private items: T[] = [];
  push(item: T): void { this.items.push(item); }
  pop(): T | undefined { return this.items.pop(); }
  peek(): T | undefined { return this.items[this.items.length - 1]; }
  isEmpty(): boolean { return this.items.length === 0; }
  size(): number { return this.items.length; }
}
```

Time Complexity: All operations are O(1). The generic type parameter ensures compile-time type checking for stack elements.

2. Implement an LRU (Least Recently Used) Cache with O(1) get and put operations in TypeScript.

An LRU Cache requires a combination of a Map for O(1) lookup and a doubly-linked structure for O(1) reordering.

Implementation:

```
class LRUCache {
  private cache = new Map();
  constructor(private capacity: number) {}
  get(key: K): V | undefined {
    if (!this.cache.has(key)) return undefined;
    const val = this.cache.get(key)!;
    this.cache.delete(key); this.cache.set(key, val);
    return val;
  }
  put(key: K, value: V): void {
    this.cache.delete(key);
    this.cache.set(key, value);
    if (this.cache.size > this.capacity) {
      this.cache.delete(this.cache.keys().next().value);
    }
  }
}
```

Key Points:

- Map maintains insertion order in JavaScript/TypeScript
- Deleting and re-inserting moves item to end (most recent)
- First key is least recently used

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

The optimal approach uses a Set for O(n) time complexity with a single pass through the array.

Implementation:

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

Complexity Analysis:

- **Time:** O(n) - single pass through array
- **Space:** O(n) - Set stores up to n elements
- Alternative sorting approach would be O(n log n)

4. Implement a type-safe Binary Search Tree with insert, search, and inorder traversal methods.

A BST maintains sorted order with average O(log n) operations for balanced trees.

Implementation:

```
class TreeNode {
  constructor(public value: T, public left: TreeNode | null = null,
    public right: TreeNode | null = null) {}
}
class BST {
  root: TreeNode | null = null;
  insert(value: T): void {
    this.root = this.insertNode(this.root, value);
  }
  private insertNode(node: TreeNode | null, value: T): TreeNode {
    if (!node) return new TreeNode(value);
    if (value < node.value) node.left = this.insertNode(node.left, value);
    else node.right = this.insertNode(node.right, value);
    return node;
  }
}
```

Time Complexity: O(log n) average, O(n) worst case for skewed trees.

5. Implement a sliding window algorithm to find the maximum sum of k consecutive elements in an array.

The sliding window technique optimizes from O(n*k) brute force to O(n) by reusing calculations.

Implementation:

```
function maxSumSubarray(arr: number[], k: number): number {
  if (arr.length < k) return 0;
  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:

- Calculate initial window sum
- Slide by subtracting left element and adding right element
- **Time:** O(n), **Space:** O(1)

6. How would you implement a Queue using two Stacks in TypeScript? What are the time complexities?

Using two stacks simulates FIFO behavior with amortized O(1) operations.

Implementation:

```
class QueueWithStacks {
  private stack1: T[] = [];
  private stack2: T[] = [];
  enqueue(item: T): void { this.stack1.push(item); }
  dequeue(): T | undefined {
    if (this.stack2.length === 0) {
      while (this.stack1.length > 0) this.stack2.push(this.stack1.pop()!);
    }
    return this.stack2.pop();
  }
}
```

Complexity Analysis:

- **Enqueue:** O(1) - simple push
- **Dequeue:** Amortized O(1) - each element moved once
- Stack1 holds new elements, Stack2 holds reversed elements for dequeue

7. Implement a function to detect if a linked list has a cycle using Floyd's algorithm. Include TypeScript types.

Floyd's Cycle Detection (tortoise and hare) uses two pointers moving at different speeds to detect cycles in O(n) time with O(1) space.

Implementation:

```
class ListNode {
  constructor(public value: T, public next: ListNode | null = null) {}
}
function hasCycle(head: ListNode | null): boolean {
  let slow = head, fast = head;
  while (fast && fast.next) {
    slow = slow!.next;
    fast = fast.next.next;
    if (slow === fast) return true;
  }
  return false;
}
```

Why it works: If there's a cycle, the fast pointer will eventually catch up to the slow pointer inside the cycle.

8. How do you implement a Trie (Prefix Tree) for efficient string searching? Include insert and search methods.

A Trie provides O(m) search time where m is the key length, ideal for autocomplete and

prefix matching.

Implementation:

```
class TrieNode {
  children = new Map();
  isEndOfWord = false;
}
class Trie {
  private root = new TrieNode();
  insert(word: string): void {
    let node = this.root;
    for (const char of word) {
      if (!node.children.has(char)) node.children.set(char, new TrieNode());
      node = node.children.get(char)!;
    }
    node.isEndOfWord = true;
  }
  search(word: string): boolean {
    let node = this.root;
    for (const char of word) {
      if (!node.children.has(char)) return false;
      node = node.children.get(char)!;
    }
    return node.isEndOfWord;
  }
}
```

Space: O(ALPHABET_SIZE * N * M) where N is number of words and M is average length.

9. Implement a Min Heap with insert and extractMin operations. What are the time complexities?

A Min Heap maintains the smallest element at the root with O(log n) insertions and deletions.

Implementation:

```
class MinHeap {
  private heap: number[] = [];
  insert(val: number): void {
    this.heap.push(val);
    this.bubbleUp(this.heap.length - 1);
  }
  private bubbleUp(idx: number): void {
    while (idx > 0) {
      const 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;
    }
  }
}
```

Complexities:

- **Insert:** O(log n) - bubble up at most log n levels
- **ExtractMin:** O(log n) - bubble down after removing root
- **GetMin:** O(1) - root element

10. How would you implement a function to find the longest substring without repeating characters? Optimize for time complexity.

Use a sliding window with a Map to track character positions for O(n) time complexity.

Implementation:

```
function longestUniqueSubstring(s: string): number {
  const charIndex = new Map();
  let maxLen = 0, start = 0;
  for (let end = 0; end < s.length; end++) {
    if (charIndex.has(s[end])) {
      start = Math.max(start, charIndex.get(s[end])! + 1);
    }
    charIndex.set(s[end], end);
    maxLen = Math.max(maxLen, end - start + 1);
  }
  return maxLen;
}
```

Algorithm:

- Expand window by moving end pointer
- When duplicate found, move start past previous occurrence
- **Time:** O(n), **Space:** O(min(n, m)) where m is charset size

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:** Handle incoming requests for creating and resolving short URLs
- **Application Servers:** Stateless servers for business logic
- **Database:** Store URL mappings (SQL or NoSQL)
- **Cache Layer:** Redis/Memcached for frequently accessed URLs
- **Load Balancer:** Distribute traffic across app servers

Architecture Approach

URL Generation: Use base62 encoding with auto-incrementing ID or hash-based approach. For distributed systems, use a distributed ID generator like Twitter Snowflake.

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

Read-Heavy Optimization: Implement multi-layer caching (CDN, Redis) since reads far exceed writes. Use cache-aside pattern with TTL.

Scalability Considerations

- **Database Sharding:** Partition by hash of short URL
- **Replication:** Master-slave for read scaling
- **Rate Limiting:** Prevent abuse using token bucket algorithm
- **Analytics:** Use message queue (Kafka) for async click tracking

CAP Theorem Trade-off: Favor availability and partition tolerance (AP) over strict consistency for read operations.

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

Core Architecture

- **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/MongoDB for chat history
- **API Gateway:** REST APIs for non-real-time operations

Message Delivery Strategy

Connection Management:

```
interface WebSocketConnection {
  userId: string;
  socketId: string;
  serverId: string;
  lastSeen: Date;
}

class ConnectionManager {
  async registerConnection(conn: WebSocketConnection) {
    await redis.hset(`user:${conn.userId}`, 'server', conn.serverId);
  }
}
```

Message Flow:

1. Client sends message via WebSocket
2. Server publishes to message queue with recipient ID
3. Message broker routes to appropriate WebSocket server
4. Server delivers to recipient or stores for offline delivery

Scalability Solutions

- **Horizontal Scaling:** Stateless WebSocket servers behind load balancer with sticky sessions
- **Pub/Sub Pattern:** Redis Pub/Sub or Kafka for inter-server communication
- **Database Partitioning:** Shard by user ID or conversation ID
- **Presence Updates:** Batch updates every 30s to reduce load

Consistency Model: Eventual consistency acceptable for presence; strong consistency for message ordering within conversations.

3. Design a distributed rate limiter that can handle 100k requests per second. Discuss algorithms, data structures, and implementation challenges.

Rate Limiting Algorithms

- **Token Bucket:** Best for burst traffic, allows temporary exceeding
- **Leaky Bucket:** Smooths traffic, processes at constant rate
- **Fixed Window:** Simple but has boundary issues
- **Sliding Window Log:** Accurate but memory intensive
- **Sliding Window Counter:** Balance of accuracy and efficiency

Token Bucket Implementation

```
class TokenBucket {
  async allowRequest(key: string, limit: number, window: number): Promise {
    const now = Date.now();
    const tokens = await redis.get(key) || limit;
    if (tokens > 0) {
      await redis.decr(key);
      await redis.expire(key, window);
      return true;
    }
    return false;
  }
}
```

Distributed Challenges

- **Race Conditions:** Use Redis Lua scripts for atomic operations
- **Clock Synchronization:** Use centralized timestamp service or logical clocks
- **Network Latency:** Local rate limiting + eventual consistency

Architecture

Centralized: Redis cluster as single source of truth. Pros: accurate. Cons: single point of failure, latency.**Distributed:** Local counters + gossip protocol for synchronization. Pros: low latency. Cons: eventual consistency.**Hybrid Approach:** Local cache with periodic sync to Redis. Allow 10% overrun for performance.

- Store counters in Redis with expiry

- Use sorted sets for sliding window
- Implement circuit breaker for Redis failures

4. Design a news feed system like Twitter or Facebook. How would you generate, rank, and deliver personalized feeds to millions of users?

Feed Generation Approaches

- **Fanout on Write (Push):** Pre-compute feeds when post is created
- **Fanout on Read (Pull):** Compute feed when user requests
- **Hybrid:** Push for active users, pull for inactive/celebrities

Architecture Components

- **Post Service:** Handle post creation and storage
- **Feed Generation Service:** Build personalized feeds
- **Ranking Service:** ML-based content ranking
- **Feed Cache:** Redis for pre-computed feeds
- **Graph Database:** Store social relationships

Feed Generation Implementation

```
class FeedGenerator {
  async generateFeed(userId: string, limit: number) {
    const following = await getFollowing(userId);
    const posts = await Promise.all(
      following.map(id => getRecentPosts(id, 100))
    );
    const ranked = await this.rankPosts(posts.flat(), userId);
    return ranked.slice(0, limit);
  }
}
```

Ranking Strategy

- **Signals:** Recency, engagement (likes, comments), user affinity, content type
- **ML Model:** Train on user interactions to predict engagement probability
- **Real-time Updates:** Stream processing for trending content

Scalability Optimizations

- **Fanout Optimization:** Limit fanout for users with millions of followers
- **Materialized Views:** Pre-compute feeds for active users
- **Pagination:** Cursor-based pagination for infinite scroll
- **CDN:** Cache static content and media

Consistency: Eventual consistency acceptable; prioritize availability over immediate consistency.

5. Design a distributed cache system. How would you handle cache invalidation, consistency, and high availability?

Cache Architecture

- **Cache Topology:** Distributed hash ring (consistent hashing)
- **Replication:** Master-slave or multi-master replication
- **Partitioning:** Shard data across nodes for horizontal scaling
- **Client Library:** Smart client with server discovery

Consistent Hashing Implementation

```
class ConsistentHash {
  private ring: Map = new Map();

  addNode(node: string, virtualNodes: number = 150) {
    for (let i = 0; i < virtualNodes; i++) {
      const hash = this.hash(`${node}:${i}`);
      this.ring.set(hash, node);
    }
  }
}
```

Cache Invalidation Strategies

- **TTL-based:** Set expiration time for each key
- **Write-through:** Update cache and database synchronously
- **Write-behind:** Update cache immediately, database asynchronously
- **Cache-aside:** Application manages cache explicitly
- **Event-driven:** Invalidate via message queue on data changes

Consistency Models

- **Strong Consistency:** Synchronous replication (high latency)
- **Eventual Consistency:** Asynchronous replication (better performance)
- **Read-your-writes:** Route user requests to same replica

High Availability

- **Replication:** 3+ replicas across availability zones
- **Health Checks:** Automatic failover on node failure
- **Circuit Breaker:** Fallback to database on cache failure
- **Monitoring:** Track hit rate, latency, eviction rate

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

System Components

- **Upload Service:** Handle large file uploads with resumable uploads
- **Transcoding Pipeline:** Convert videos to multiple formats/resolutions
- **Storage:** Object storage (S3) for video files
- **CDN:** Distribute content globally
- **Metadata Service:** Store video information in database
- **Streaming Service:** Deliver adaptive bitrate streaming

Upload Flow

```
interface UploadChunk {
  videoId: string;
  chunkIndex: number;
  totalChunks: number;
  data: Buffer;
}

async function handleChunkUpload(chunk: UploadChunk) {
  await s3.putObject(`${chunk.videoId}/${chunk.chunkIndex}`);
  if (chunk.chunkIndex === chunk.totalChunks - 1) {
    await triggerTranscoding(chunk.videoId);
  }
}
```

Transcoding Architecture

- **Message Queue:** Kafka/SQS for transcoding jobs
- **Worker Pool:** Auto-scaling workers process jobs
- **Multiple Outputs:** 1080p, 720p, 480p, 360p + audio tracks
- **Parallel Processing:** Split video into segments for faster transcoding
- **Priority Queue:** Popular creators get higher priority

Adaptive Streaming

- **HLS/DASH:** Segment videos into small chunks (2-10s)
- **Manifest File:** Contains URLs for different quality levels
- **Client Logic:** Switch quality based on bandwidth

Scalability

- **Storage:** Petabyte-scale object storage with lifecycle policies
- **CDN:** Multi-tier caching (edge, regional, origin)
- **Database:** Shard by video ID or user ID
- **Analytics:** Stream view events to data warehouse

7. Design an e-commerce inventory management system that handles high concurrency. How would you prevent overselling and maintain consistency?

Core Challenges

- **Race Conditions:** Multiple users buying last item simultaneously
- **Distributed Transactions:** Coordinating inventory across services
- **High Throughput:** Flash sales with 10k+ requests/second
- **Consistency:** Inventory accuracy across replicas

Pessimistic Locking Approach

```
async function reserveInventory(productId: string, qty: number) {
  return await db.transaction(async (trx) => {
    const product = await trx('inventory')
      .where('id', productId)
      .forUpdate()
      .first();
    if (product.stock >= qty) {
      await trx('inventory').where('id', productId)
        .decrement('stock', qty);
      return true;
    }
    return false;
  });
}
```

Optimistic Locking Approach

```
async function reserveWithVersion(productId: string, qty: number) {
  const product = await getProduct(productId);
  const updated = await db('inventory')
    .where({ id: productId, version: product.version })
    .where('stock', '>=', qty)
    .update({ stock: db.raw('stock - ?', [qty]), version: product.version + 1 });
  return updated > 0;
}
```

Distributed Solutions

- **Redis Atomic Operations:** Use DECR for stock updates with Lua scripts
- **Reservation System:** Reserve inventory for 10 minutes, release if unpaid
- **Event Sourcing:** Store all inventory changes as events
- **CQRS:** Separate read and write models

Flash Sale Optimization

- **Queue System:** Buffer requests in queue
- **Pre-allocation:** Distribute inventory to regional warehouses
- **Rate Limiting:** Limit requests per user

8. Design a notification system that supports push notifications, emails, and SMS across millions of users. How would you ensure delivery and handle failures?

System Architecture

- **Notification Service:** Orchestrates notification delivery
- **Channel Handlers:** Separate services for push, email, SMS
- **Message Queue:** Kafka/RabbitMQ for reliable delivery
- **User Preference Service:** Store notification preferences

- **Template Service:** Manage notification templates
- **Analytics Service:** Track delivery and engagement

Notification Flow

```
interface Notification {
  userId: string;
  type: 'push' | 'email' | 'sms';
  priority: 'high' | 'normal' | 'low';
  template: string;
  data: Record;
}

class NotificationService {
  async send(notification: Notification) {
    await queue.publish('notifications', notification);
  }
}
```

Delivery Guarantees

- **At-least-once:** Use message queue acknowledgments
- **Idempotency:** Deduplicate using notification ID
- **Retry Logic:** Exponential backoff for failed deliveries
- **Dead Letter Queue:** Move failed messages after max retries

Priority Handling

- **Multiple Queues:** Separate queues for different priorities
- **Rate Limiting:** Throttle low-priority notifications
- **Batching:** Combine multiple notifications for efficiency

Scalability Patterns

- **Fan-out:** Parallel processing across multiple workers
- **Circuit Breaker:** Prevent cascade failures to third-party providers
- **Provider Fallback:** Switch to backup provider on failure
- **User Segmentation:** Shard users across notification servers

Monitoring

- Track delivery rate, latency, failure rate per channel
- Alert on abnormal patterns
- A/B test notification content

9. Design a ride-sharing service like Uber. Focus on the matching algorithm, location tracking, and handling high concurrency during peak hours.

Core Services

- **Location Service:** Track driver and rider locations in real-time
- **Matching Service:** Pair riders with nearby drivers
- **Trip Service:** Manage trip lifecycle
- **Pricing Service:** Dynamic pricing based on demand
- **Payment Service:** Handle transactions
- **Notification Service:** Real-time updates

Location Tracking

```
interface Location {
  userId: string;
  lat: number;
  lng: number;
  timestamp: Date;
}

class LocationService {
  async updateLocation(loc: Location) {
    await redis.geoadd('drivers', loc.lng, loc.lat, loc.userId);
    await redis.expire(`driver:${loc.userId}`, 300);
  }
}
```

Matching Algorithm

- **Geospatial Indexing:** Use geohash or quadtree for efficient proximity search
- **Redis GEO:** GEORADIUS to find drivers within radius
- **Scoring:** Consider distance, driver rating, ETA, acceptance rate
- **Reservation:** Lock driver for 30s while waiting for acceptance

Matching Flow

```
async function findDriver(riderLoc: Location, radius: number) {
  const drivers = await redis.georadius(
    'drivers', riderLoc.lng, riderLoc.lat, radius, 'km'
  );
  const available = drivers.filter(d => !isReserved(d));
  const scored = await scoreDrivers(available, riderLoc);
  return scored[0];
}
```

High Concurrency Solutions

- **Distributed Locking:** Redis SETNX for driver reservation
- **Event Sourcing:** Store all state changes as events
- **CQRS:** Separate read model for driver availability
- **Sharding:** Partition by geographic region

Scalability

- **WebSocket Servers:** Maintain persistent connections
- **Load Balancing:** Geographic routing
- **Caching:** Cache driver states and pricing rules

10. Design a search engine autocomplete system. How would you handle billions of queries, provide sub-100ms latency, and update suggestions in real-time?

System Components

- **Trie Data Structure:** Efficient prefix matching
- **Suggestion Service:** Return top-k suggestions
- **Analytics Pipeline:** Process query logs for popularity
- **Cache Layer:** Redis for hot queries
- **Storage:** Distributed database for trie nodes

Trie Implementation

```
class TrieNode {
  children: Map = new Map();
  topSuggestions: string[] = [];
  frequency: number = 0;
}

class Autocomplete {
  async getSuggestions(prefix: string, limit: number = 10) {
    const cached = await redis.get(`suggest:${prefix}`);
    if (cached) return JSON.parse(cached);
    return this.searchTrie(prefix, limit);
  }
}
```

Ranking Strategy

- **Frequency:** Number of times query was searched

- **Recency:** Recent queries weighted higher
- **Personalization:** User's search history and location
- **Trending:** Queries with sudden spike in popularity

Real-time Updates

- **Batch Processing:** Update trie every 5-10 minutes with top queries
- **Stream Processing:** Kafka + Flink for real-time trending detection
- **Incremental Updates:** Update only affected trie branches
- **Version Control:** Multiple trie versions for A/B testing

Scalability Optimizations

- **Trie Sharding:** Partition by first character or hash
- **Multi-level Cache:** Browser cache → CDN → Redis → Database
- **Compression:** Store only top-k suggestions at each node
- **Async Loading:** Load trie segments on-demand

Latency Optimization

- **Pre-computation:** Store top suggestions at each trie node
- **Geographic Distribution:** Deploy in multiple regions
- **Client-side Caching:** Cache recent searches locally

Coding and Debugging

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

1. Write a TypeScript function to flatten a nested array of any depth with proper type safety.

Solution

Use recursive type definitions and runtime recursion to flatten arrays while maintaining type safety:

```
type Flatten = T extends Array ? Flatten : T;

function flatten(arr: T[]): Flatten[] {
  return arr.reduce((acc, item) => {
    return acc.concat(Array.isArray(item) ? flatten(item) : item);
  }, [] as any);
}

const result = flatten([1, [2, [3, [4]]]]);
// result: number[]
```

Key Points:

- The **Flatten** type recursively unwraps nested arrays
- Runtime function uses reduce and recursion
- Type inference ensures compile-time safety

2. How do you implement a type-safe event emitter in TypeScript with strict event typing?

Type-Safe Event Emitter

Use mapped types and generics to enforce event name and payload type matching:

```
type Events = {
  login: { userId: string };
  logout: void;
};

class EventEmitter> {
  private listeners = new Map();

  on(event: K, fn: (data: T[K]) => void) {
    const fns = this.listeners.get(event) || [];
    this.listeners.set(event, [...fns, fn]);
  }
}
```

Benefits:

- Autocomplete for event names
- Type checking for event payloads
- Prevents runtime errors from mismatched data

3. Write a function to check if a string is a palindrome, optimized for performance with TypeScript.

Optimized Palindrome Check

Use two-pointer technique for O(n/2) time complexity:

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

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

Optimizations:

- Single pass normalization
- Two-pointer avoids string reversal
- Early exit on mismatch
- No extra array allocation

4. How do you debug memory leaks in TypeScript applications? What tools and techniques do you use?

Memory Leak Debugging Strategy

Tools:

- **Chrome DevTools:** Heap snapshots, allocation timeline, memory profiler
- **Node.js:** --inspect flag with Chrome DevTools, heapdump package
- **clinic.js:** For Node.js performance profiling

Common Causes in TypeScript:

- Uncleaned event listeners
- Closure capturing large objects
- Global variable accumulation
- Detached DOM nodes
- Uncleared timers/intervals

Technique: Take heap snapshots before/after operations, compare retained size, track detached nodes, use weak references (WeakMap/WeakSet) for caches.

5. Implement a debounce function in TypeScript with proper typing for any function signature.

Generic Debounce Implementation

```
function debounce any>(
  fn: T,
  delay: number
): (...args: Parameters) => void {
  let timeoutId: NodeJS.Timeout | null = null;

  return (...args: Parameters) => {
    if (timeoutId) clearTimeout(timeoutId);
    timeoutId = setTimeout(() => fn(...args), delay);
  };
}
```

Type Safety Features:

- **Parameters:** Extracts parameter types
- **ReturnType:** Could preserve return type
- Works with any function signature
- Maintains argument type checking

6. What are TypeScript decorators and how would you implement a method execution timer decorator?

Method Timer Decorator

Decorators are functions that modify classes, methods, or properties at design time:

```
function Timer(target: any, key: string, descriptor: PropertyDescriptor) {
  const original = descriptor.value;
  
  descriptor.value = async function(...args: any[]) {
    const start = performance.now();
    const result = await original.apply(this, args);
    console.log(`${key} took ${performance.now() - start}ms`);
    return result;
  };
}
```

Usage:

```
class Service {
  @Timer
  async fetchData() { /* ... */ }
}
```

Requires **experimentalDecorators** in tsconfig.json

7. How do you handle and debug 'Type instantiation is excessively deep' errors in TypeScript?

Debugging Deep Type Instantiation

Causes:

- Recursive conditional types without base case
- Complex mapped types with circular dependencies
- Deep object nesting in generic constraints

Solutions:

- **Add depth limits:** Use counter types to limit recursion
- **Simplify types:** Break complex types into smaller pieces
- **Use type assertions:** Strategic 'as' casts to break cycles
- **Tail recursion:** Restructure recursive types

```
type Limited = 
  D extends 0 ? T : 
  T extends Array ? Limited>[] : T;
```

8. Implement a type-safe deep partial utility type that works with nested objects and arrays.

Deep Partial Type

```
type DeepPartial = T extends object ? {
  [P in keyof T]?: T[P] extends Array
    ? Array>
    : T[P] extends ReadonlyArray
    ? ReadonlyArray>
    : T[P] extends object
    ? DeepPartial
    : T[P];
} : T;
```

Features:

- Recursively makes all properties optional
- Handles arrays and readonly arrays
- Preserves primitive types
- Works with nested object structures

9. What techniques would you use to debug TypeScript compilation errors in large codebases?

Debugging Compilation Errors

Strategies:

- **Isolate the error:** Use --noEmit and binary search to find problematic files
- **Enable verbose logging:** --listFiles, --traceResolution flags
- **Use skipLibCheck:** Temporarily to identify if issue is in dependencies
- **Incremental compilation:** --incremental flag to speed up iterations
- **Type checking tools:** tsc --noEmit --pretty for readable output

Advanced Techniques:

- Use ts-node with --transpile-only for quick testing
- Leverage IDE hover information for type inference
- Check generated .d.ts files for type mismatches
- Use @ts-expect-error with descriptions to document known issues

10. How would you implement monkey patching safely in TypeScript while maintaining type safety?

Safe Monkey Patching

Use module augmentation and interface merging to extend existing types:

```
// Extend Array prototype safely
declare global {
  interface Array {
    last(): T | undefined;
  }
}

Array.prototype.last = function() {
  return this[this.length - 1];
};

const arr = [1, 2, 3];
const lastItem = arr.last(); // Type-safe!
```

Best Practices:

- Always use declaration merging
- Document side effects clearly
- Consider namespacing to avoid conflicts
- Prefer composition over modification when possible

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 refactor a large JavaScript codebase to TypeScript. What was your approach?

Situation: Our team inherited a 50K+ line JavaScript codebase with frequent runtime errors and poor maintainability. Management wanted to migrate to TypeScript to improve code quality.

Task: I was tasked with leading the migration strategy while ensuring zero downtime and maintaining feature development velocity.

Action: I implemented a phased approach: (1) Configured TypeScript with allowJs: true and strict mode disabled, (2) Renamed critical files from .js to .ts starting with utility modules, (3) Gradually enabled strict checks file-by-file, (4) Created shared type definitions for common patterns, (5) Set up pre-commit hooks to prevent new JavaScript files.

Result: Completed migration in 4 months with 40% reduction in production bugs. The team reported 60% faster onboarding for new developers due to better type documentation.

2. Describe a situation where TypeScript's type system prevented a critical bug. How did you handle it?

Situation: During a payment processing feature implementation, our API contract changed but the frontend wasn't updated, which could have caused failed transactions in production.

Task: I needed to ensure type safety between frontend and backend while maintaining rapid development cycles.

Action: I implemented a shared types package using tRPC for end-to-end type safety. When the backend changed the payment response structure, TypeScript immediately flagged 15+ locations in the frontend code. I also set up automated type generation from OpenAPI specs and added CI checks to validate type consistency.

Result: Caught the breaking change during development instead of production. This approach prevented an estimated $50K+ in failed transactions and reduced integration bugs by 75% over the next quarter.

3. Give an example of when you had to convince your team to adopt advanced TypeScript features like generics or conditional types.

Situation: Our team was using any types extensively in our API client library, leading to runtime errors and poor developer experience.

Task: I needed to improve type safety without overwhelming the team with complex TypeScript concepts.

Action: I created a working prototype using generic constraints and conditional types for our API client, demonstrating autocomplete and compile-time validation. I conducted a 30-minute lunch-and-learn showing real examples of bugs it would prevent. I documented patterns with clear examples and created reusable utility types. Started with one high-traffic module as proof of concept.

Result: Team adopted the patterns within 2 sprints. Developer satisfaction scores increased 35%, and we saw 50% fewer API-related runtime errors. The patterns became our standard for new API integrations.

4. Tell me about a time when you had to balance type safety with development speed. What trade-offs did you make?

Situation: During a critical product launch with a 2-week deadline, enforcing strict TypeScript checks was causing significant development delays, but we couldn't compromise on code quality.

Task: I needed to deliver features on time while maintaining acceptable type safety standards.

Action: I established a pragmatic approach: (1) Used unknown instead of any for third-party integrations, (2) Created TODO comments with Jira tickets for proper typing post-launch, (3) Maintained strict typing for business logic while relaxing checks for UI components, (4) Implemented runtime validation using zod where types were uncertain, (5) Scheduled a 1-week hardening sprint post-launch.

Result: Launched on time with zero critical bugs. Completed type hardening in the following sprint, improving type coverage from 75% to 95%. Established this as our standard approach for high-pressure situations.

5. Describe a challenging debugging experience involving TypeScript's type inference. How did you resolve it?

Situation: We had a complex Redux store with deeply nested state where TypeScript was inferring types incorrectly, causing false positives in our IDE and masking real issues.

Task: I needed to fix the type inference without rewriting the entire state management system.

Action: I systematically debugged by: (1) Using the TypeScript Compiler API to inspect inferred types with tsc --noEmit --explainFiles, (2) Identified that circular type references were causing inference to fall back to any, (3) Introduced explicit type annotations at key boundaries using utility types like ReturnType and Parameters, (4) Refactored deeply nested types into smaller, composable interfaces, (5) Added @ts-expect-error comments with explanations where TypeScript limitations existed.

Result: Resolved all inference issues within 3 days. IDE performance improved significantly, and we caught 8 real bugs that were previously hidden. Documented the patterns for team reference.

6. Tell me about a time when you improved the developer experience using TypeScript's tooling or configuration.

Situation: Our monorepo had inconsistent TypeScript configurations across 12 packages, causing confusion and build failures when developers switched contexts.

Task: I was asked to standardize the TypeScript setup and improve the overall developer experience.

Action: I implemented: (1) A base tsconfig.base.json with shared compiler options, (2) Project references for faster incremental builds, (3) Path aliases configured in both TypeScript and Jest for cleaner imports, (4) Custom ESLint rules to enforce typing standards, (5) VS Code workspace settings with recommended extensions, (6) Pre-commit hooks to run type checks only on changed files.

Result: Build times reduced by 60%, from 5 minutes to 2 minutes. Developer satisfaction survey showed 80% improvement in tooling experience. New team members became productive 40% faster due to consistent setup.

7. Describe a situation where you had to work with poorly typed third-party libraries. What was your solution?

Situation: A critical analytics library we depended on had outdated type definitions that didn't match the actual API, causing runtime crashes in production.

Task: I needed to safely integrate the library while maintaining type safety for our team.

Action: I created a typed wrapper layer: (1) Declared a custom module in types/analytics.d.ts with accurate types based on the library's documentation and runtime testing, (2) Built a facade pattern wrapper exposing only the methods we used, (3) Added runtime validation using zod to ensure the library responses matched our types, (4) Contributed the corrected types back to DefinitelyTyped, (5) Set up tests to validate type assumptions against library updates.

Result: Eliminated all runtime errors related to the library. Our type definitions were merged into DefinitelyTyped, helping the broader community. Established this pattern as our standard for handling poorly-typed dependencies.

8. Give an example of when you had to mentor junior developers on TypeScript best practices. What was your approach?

Situation: Three junior developers joined our team with JavaScript backgrounds but limited TypeScript experience, leading to code reviews taking 3x longer due to type-related issues.

Task: I needed to upskill the team on TypeScript while maintaining our delivery commitments.

Action: I created a structured mentorship program: (1) Weekly 1-hour TypeScript workshops covering topics from basics to advanced patterns, (2) Pair programming sessions focusing on real codebase examples, (3) Created a team wiki with common patterns and anti-patterns, (4) Implemented a "TypeScript champion" rotation where each person presented a TypeScript feature, (5) Established gentle code review feedback focusing on learning, not criticism.

Result: Within 6 weeks, junior developers were independently writing well-typed code. Code review time decreased by 60%. Two juniors became go-to resources for TypeScript questions. The workshop materials were adopted company-wide.

9. Tell me about a time when you had to optimize TypeScript compilation performance in a large project.

Situation: Our TypeScript compilation time had grown to 8 minutes, severely impacting developer productivity and CI/CD pipeline efficiency.

Task: I was assigned to reduce compilation time without compromising type safety or code quality.

Action: I implemented multiple optimizations: (1) Enabled incremental: true and project references, (2) Used skipLibCheck: true to avoid re-checking node_modules, (3) Split large type files into smaller, focused modules, (4) Configured exclude patterns to ignore test files during main build, (5) Implemented tsc --build for better caching, (6) Analyzed build with --extendedDiagnostics to identify bottlenecks, (7) Moved type-only imports to use import type syntax.

Result: Reduced compilation time from 8 minutes to 90 seconds (81% improvement). CI/CD pipeline became 5x faster. Developer hot-reload time improved from 12s to 3s, significantly boosting productivity.

10. Describe a situation where you had to make architectural decisions involving TypeScript in a microservices environment.

Situation: Our company was transitioning to microservices, and each team was implementing their own TypeScript patterns, leading to inconsistent contracts and integration issues.

Task: I was appointed to establish TypeScript standards across 8 microservices teams to ensure consistency and type safety at service boundaries.

Action: I designed and implemented: (1) A shared types monorepo published as internal npm packages, (2) Automated type generation from Protocol Buffers and OpenAPI specs, (3) Contract testing using TypeScript to validate service interfaces, (4) Established a TypeScript style guide and architectural decision records (ADRs), (5) Created CLI tools to scaffold new services with correct TypeScript setup, (6) Quarterly architecture review meetings to evolve standards.

Result: Reduced cross-service integration bugs by 70%. Onboarding new services became 50% faster using standardized templates. All teams adopted the shared types approach within one quarter. The pattern was featured in our engineering blog.