NestJS Coding Challenges

Interview Questions and Answers

Core Concepts

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

1. Implement a custom ParamDecorator that validates and transforms a UUID parameter in NestJS

Solution:

Here's how to create a custom UUID parameter decorator with validation:

```
import { createParamDecorator, ExecutionContext, BadRequestException } from '@nestjs/common';

export const ValidUUID = createParamDecorator((data: unknown, ctx: ExecutionContext) => {
  const request = ctx.switchToHttp().getRequest();
  const uuid = request.params.id;
  const uuidRegex = /^[0-9a-f]{8}-[0-9a-f]{4}-4[0-9a-f]{3}-[89ab][0-9a-f]{3}-[0-9a-f]{12}$/i;
  if (!uuidRegex.test(uuid)) throw new BadRequestException('Invalid UUID format');
  return uuid;
});
```

2. Create a custom ExceptionFilter that handles and logs database connection errors

Solution:

```
@Catch(TypeORMError)
export class DatabaseExceptionFilter implements ExceptionFilter {
  catch(exception: TypeORMError, host: ArgumentsHost) {
    const ctx = host.switchToHttp();
    const response = ctx.getResponse();
    Logger.error(`Database Error: ${exception.message}`, exception.stack);
    response.status(503).json({ message: 'Database service unavailable' });
  }
}
```

3. Implement a custom interceptor that caches HTTP responses with Redis

Solution:

```
@Injectable()
export class RedisCacheInterceptor implements NestInterceptor {
  constructor(private readonly redis: Redis) {}
  async intercept(context: ExecutionContext, next: CallHandler) {
    const key = context.switchToHttp().getRequest().url;
    const cached = await this.redis.get(key);
    if (cached) return of(JSON.parse(cached));
    return next.handle().pipe(
      tap(async response => await this.redis.set(key, JSON.stringify(response), 'EX', 3600))
    );
  }
}
```

4. Create a custom middleware that rate-limits API requests based on IP address

Solution:

```
@Injectable()
export class RateLimitMiddleware implements NestMiddleware {
  private readonly store = new Map();
  use(req: Request, res: Response, next: Function) {
    const ip = req.ip;
    const now = Date.now();
    const record = this.store.get(ip) || { count: 0, timestamp: now };
    if (now - record.timestamp > 60000) record.count = 0;
    if (record.count >= 100) throw new HttpException('Too Many Requests', 429);
    record.count++; this.store.set(ip, record); next();
  }
}
```

5. Implement a custom Guard that validates JWT tokens and assigns user roles

Solution:

```
@Injectable()
export class RoleGuard implements CanActivate {
  constructor(private readonly reflector: Reflector) {}
  canActivate(context: ExecutionContext): boolean {
    const requiredRoles = this.reflector.get('roles', context.getHandler());
    const request = context.switchToHttp().getRequest();
    const user = request.user;
    return requiredRoles.some(role => user.roles?.includes(role));
  }
}
```

6. Create a custom Pipe that validates and transforms GraphQL input

Solution:

```
@Injectable()
export class GraphQLValidationPipe implements PipeTransform {
  constructor(private schema: Joi.ObjectSchema) {}
  transform(value: any) {
    const { error, value: validated } = this.schema.validate(value);
    if (error) throw new UserInputError(error.message);
    return validated;
  }
}
```

7. Implement a custom decorator that measures method execution time

Solution:

```
export function Measure() {
  return function (target: any, propertyKey: string, descriptor: PropertyDescriptor) {
    const original = descriptor.value;
    descriptor.value = async function (...args: any[]) {
      const start = process.hrtime();
      const result = await original.apply(this, args);
      const [seconds, nanoseconds] = process.hrtime(start);
      console.log(`${propertyKey} took ${seconds}s ${nanoseconds/1000000}ms`);
      return result;
    };
  };
}
```

8. Create a custom provider that implements a circuit breaker pattern

Solution:

```
@Injectable()
export class CircuitBreakerProvider {
  private failures = 0;
  private lastFailure: number = 0;
  async executeWithBreaker(fn: () => Promise): Promise {
    if (this.isOpen()) throw new ServiceUnavailableException();
    try {
      const result = await fn();
      this.reset(); return result;
    } catch (error) {
      this.recordFailure(); throw error;
    }
  }
}
```

9. Implement a custom module that provides websocket authentication

Solution:

```
@Module({
  imports: [JwtModule.register({ secret: 'secret' })],
  providers: [
    {
      provide: 'WS_GUARD',
      useFactory: (jwtService: JwtService) => {
        return new WsAuthGuard(jwtService);
      },
      inject: [JwtService]
    }
  ]
})
export class WebsocketAuthModule {}
```

10. Create a custom scheduler that implements retry logic with exponential backoff

Solution:

```
@Injectable()
export class RetryScheduler {
  @Cron('*/5 * * * *')
  async handleRetry() {
    const failedTasks = await this.taskRepository.findFailed();
    for (const task of failedTasks) {
      const backoff = Math.pow(2, task.attempts) * 1000;
      if (Date.now() - task.lastAttempt >= backoff) {
        await this.processTask(task);
      }
    }
  }
}
```

Data Structures and Algorithms

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

1. Implement a thread-safe singleton pattern in NestJS

Thread-Safe Singleton Implementation

```
@Injectable()
export class ThreadSafeSingleton {
  private static instance: ThreadSafeSingleton;
  private constructor() {}

  static getInstance(): ThreadSafeSingleton {
    if (!ThreadSafeSingleton.instance) {
      ThreadSafeSingleton.instance = new ThreadSafeSingleton();
    }
    return ThreadSafeSingleton.instance;
  }
}
```

2. How would you implement a rate limiter in NestJS using a sliding window algorithm?

Sliding Window Rate Limiter Implementation

A sliding window rate limiter can be implemented as a custom decorator using Redis:

```
@Injectable()
class RateLimiterGuard {
  constructor(@Inject('REDIS') private redis: Redis) {}
  async canActivate(context: ExecutionContext): Promise {
    const now = Date.now();
    const windowSize = 60000; // 1 minute window
    const key = context.switchToHttp().getRequest().ip;
    const count = await this.redis.zcount(key, now - windowSize, now);
    return count < 100; // limit to 100 requests per minute
  }
}
```

3. Implement a caching decorator that stores results in a LRU cache with TypeScript

LRU Cache Decorator Implementation

```
function LRUCache(size: number = 100) {
  const cache = new Map();
  return function(target: any, propertyKey: string, descriptor: PropertyDescriptor) {
    const original = descriptor.value;
    descriptor.value = function(...args) {
      const key = args.join('-');
      if (cache.has(key)) return cache.get(key);
      const result = original.apply(this, args);
      if (cache.size >= size) cache.delete(cache.keys().next().value);
      cache.set(key, result);
      return result;
    };
  };
}
```

4. How would you implement a custom priority queue for handling background jobs in NestJS?

Priority Queue Implementation

```
class JobPriorityQueue {
  private queue: {job: any, priority: number}[] = [];

  enqueue(job: any, priority: number): void {
    this.queue.push({job, priority});
    this.queue.sort((a, b) => b.priority - a.priority);
  }

  dequeue(): any {
    return this.queue.shift()?.job;
  }
}
```

Key points:

- O(n log n) time complexity for enqueue
- O(1) time complexity for dequeue
- Can be enhanced with binary heap for O(log n) enqueue

5. Design a consistent hashing implementation for distributing cache keys across multiple Redis instances

Consistent Hashing Implementation

```
class ConsistentHashing {
  private ring: {hash: number, node: string}[] = [];
  constructor(nodes: string[], replicas: number = 3) {
    nodes.forEach(node => {
      for(let i = 0; i < replicas; i++) {
        const hash = this.hash(`${node}-${i}`);
        this.ring.push({hash, node});
      }
    });
    this.ring.sort((a, b) => a.hash - b.hash);
  }
}
```

Benefits:

- Minimizes key redistribution when scaling
- O(log n) lookup time
- Ensures even distribution

6. Implement a Bloom filter for efficient cache miss detection in NestJS

Bloom Filter Implementation

```
class BloomFilter {
  private bitArray: boolean[];
  private hashFunctions: ((item: string) => number)[];

  add(item: string): void {
    this.hashFunctions.forEach(hash => {
      const index = hash(item) % this.bitArray.length;
      this.bitArray[index] = true;
    });
  }

  mightContain(item: string): boolean {
    return this.hashFunctions.every(hash =>
      this.bitArray[hash(item) % this.bitArray.length]);
  }
}
```

7. How would you implement a distributed lock mechanism using Redis in NestJS?

Distributed Lock Implementation

```
@Injectable()
class DistributedLock {
  constructor(@Inject('REDIS') private redis: Redis) {}

  async acquire(key: string, ttl: number): Promise {
    const token = uuid();
    const acquired = await this.redis.set(key, token, 'NX', 'PX', ttl);
    return acquired === 'OK';
  }

  async release(key: string): Promise {
    await this.redis.del(key);
  }
}
```

8. Implement a circular buffer for handling real-time metrics in NestJS

Circular Buffer Implementation

```
class CircularBuffer {
  private buffer: T[];
  private writePointer = 0;

  constructor(private capacity: number) {
    this.buffer = new Array(capacity);
  }

  write(item: T): void {
    this.buffer[this.writePointer] = item;
    this.writePointer = (this.writePointer + 1) % this.capacity;
  }
}
```

Use cases:

- Real-time metrics tracking
- Rolling window calculations
- Stream processing

9. Design a trie data structure for autocomplete functionality in NestJS

Trie Implementation for Autocomplete

```
class TrieNode {
  children: Map = new Map();
  isEndOfWord = false;
}

class Trie {
  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;
  }
}
```

10. Design a custom event emitter with debounce functionality in NestJS

Debounced Event Emitter Implementation

```
class DebouncedEventEmitter {
  private timers = new Map();
  emit(event: string, data: any, delay: number = 1000): void {
    if (this.timers.has(event)) clearTimeout(this.timers.get(event));
    this.timers.set(event, setTimeout(() => {
      super.emit(event, data);
      this.timers.delete(event);
    }, delay));
  }
}
```

System Design

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

1. How would you design a scalable URL shortener service using NestJS?

Key Components:

- **API Gateway**: NestJS application handling incoming requests
- **Database**: Redis for caching, PostgreSQL for persistence
- **URL Generation**: Base62 encoding or MD5 hash first 8 chars

Implementation Approach:

```
@Injectable()
export class UrlService {
  async shortenUrl(longUrl: string): Promise {
    const hash = createHash('md5').update(longUrl).digest('hex').substr(0, 8);
    await this.cache.set(hash, longUrl, 'EX', 86400);
    return `domain.com/${hash}`;
  }
}
```

2. Design a real-time chat system using NestJS and WebSockets

Architecture Components:

- **WebSocket Gateway**: Handle real-time connections
- **Redis Pub/Sub**: For horizontal scaling
- **MongoDB**: Store chat history

```
@WebSocketGateway()
export class ChatGateway {
  @SubscribeMessage('message')
  handleMessage(@MessageBody() data: string): void {
    this.server.emit('newMessage', data);
  }
}
```

3. How would you implement a distributed rate limiter in NestJS?

Solution Components:

- **Redis**: Track request counts
- **Sliding Window**: Time-based tracking
- **Custom Decorator**: Rate limit annotation

```
@Injectable()
export class RateLimiterGuard {
  async canActivate(context: ExecutionContext) {
    const key = this.getRequestKey(context);
    return this.redis.incr(key) <= this.limit;
  }
}
```

4. Design a social media feed system with NestJS

Key Features:

- **Feed Generation**: Fan-out on write vs read

- **Caching Strategy**: Redis for hot data
- **Database**: Cassandra for posts

```
@Injectable()
export class FeedService {
  async getFeed(userId: string): Promise {
    const cachedFeed = await this.cache.get(`feed:${userId}`);
    return cachedFeed || this.generateFeed(userId);
  }
}
```

5. How would you implement a distributed job queue system using NestJS?

Components:

- **Bull Queue**: Redis-based job queue
- **Worker Processes**: Handle jobs
- **Dead Letter Queue**: Failed jobs

```
@Processor('email-queue')
export class EmailProcessor {
  @Process()
  async handleJob(job: Job) {
    await this.emailService.send(job.data);
  }
}
```

6. Design a notification service using NestJS

Architecture:

- **Push Notifications**: Firebase Cloud Messaging
- **Email Service**: AWS SES
- **SMS**: Twilio integration

```
@Injectable()
export class NotificationService {
  async notify(user: User, notification: Notification) {
    await Promise.all([
      this.pushNotify(user), this.sendEmail(user)
    ]);
  }
}
```

7. How would you implement a distributed caching system in NestJS?

Components:

- **Redis Cluster**: Distributed cache
- **Cache-Aside Pattern**: Write-through
- **Consistent Hashing**: Key distribution

```
@Injectable()
export class CacheService {
  async get(key: string): Promise {
    const cached = await this.redis.get(key);
    return cached || this.loadFromDb(key);
  }
}
```

8. Design a microservices-based payment processing system using NestJS

Services:

- **Payment Gateway**: Stripe integration
- **Transaction Service**: Handle payments
- **Notification Service**: Status updates

```
@Injectable()
export class PaymentService {
  @MessagePattern({ cmd: 'process_payment' })
  async processPayment(data: PaymentDto) {
    return this.stripeService.charge(data);
  }
}
```

9. How would you implement a search service with ElasticSearch in NestJS?

Components:

- **ElasticSearch Client**: Search engine
- **Index Management**: Data indexing
- **Query Builder**: Search DSL

```
@Injectable()
export class SearchService {
  async search(query: string): Promise {
    return this.elasticClient.search({
      index: 'products',
      body: { query: { match: { title: query } } }
    });
  }
}
```

10. Design a real-time analytics system using NestJS

Components:

- **Event Collection**: Kafka streams
- **Processing**: Apache Spark
- **Storage**: ClickHouse

```
@Injectable()
export class AnalyticsService {
  @Cron('*/5 * * * * *')
  async processEvents() {
    const events = await this.kafka.consume('events');
    await this.clickhouse.insert(events);
  }
}
```

Coding and Debugging

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

1. Implement a custom NestJS exception filter that logs errors to both console and file

Solution:

Here's how to create a custom exception filter:

```
@Catch(HttpException)
export class CustomExceptionFilter implements ExceptionFilter {
  catch(exception: HttpException, host: ArgumentsHost) {
    const ctx = host.switchToHttp();
    const response = ctx.getResponse();
    const status = exception.getStatus();
    const message = exception.message;

    Logger.error(`${status}: ${message}`);
    fs.appendFileSync('error.log', `${new Date()}: ${status} - ${message}
`);

    response.status(status).json({ status, message });
  }
}
```

Key points:

- Implements ExceptionFilter interface
- Uses @Catch decorator to specify exception type
- Logs to both console and file
- Returns structured error response

2. Create a custom NestJS decorator that measures execution time of a method

Solution:

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

Usage:

```
@MeasureTime()
async findAll() {
  return this.userService.findAll();
}
```

3. Implement a custom NestJS guard that rate limits requests based on IP address

Solution:

```
@Injectable()
export class RateLimitGuard implements CanActivate {
  private requests = new Map();

  canActivate(context: ExecutionContext): boolean {
    const req = context.switchToHttp().getRequest();
    const ip = req.ip;
    const now = Date.now();
    const windowMs = 60000;
    const maxRequests = 100;

    this.requests.set(ip, (this.requests.get(ip) || []).filter(time => now - time < windowMs));
    if (this.requests.get(ip).length >= maxRequests) throw new HttpException('Too Many Requests', 429);
    this.requests.get(ip).push(now);
    return true;
  }
}
```

4. Create a custom NestJS pipe that validates and transforms MongoDB ObjectId strings

Solution:

```
@Injectable()
export class ObjectIdPipe implements PipeTransform {
  transform(value: string, metadata: ArgumentMetadata) {
    if (!Types.ObjectId.isValid(value)) {
      throw new BadRequestException('Invalid ObjectId');
    }
    return new Types.ObjectId(value);
  }
}
```

Usage:

```
@Get(':id')
findOne(@Param('id', ObjectIdPipe) id: ObjectId) {
  return this.service.findById(id);
}
```

5. Implement a custom NestJS interceptor that caches responses with Redis

Solution:

```
@Injectable()
export class RedisCacheInterceptor implements NestInterceptor {
  constructor(private readonly redis: Redis) {}

  async intercept(context: ExecutionContext, next: CallHandler): Observable {
    const key = context.switchToHttp().getRequest().url;
    const cached = await this.redis.get(key);
    if (cached) return of(JSON.parse(cached));

    return next.handle().pipe(
      tap(response => this.redis.set(key, JSON.stringify(response), 'EX', 3600))
    );
  }
}
```

6. Create a custom NestJS middleware that adds correlation IDs to requests

Solution:

```
@Injectable()
export class CorrelationIdMiddleware implements NestMiddleware {
  use(req: Request, res: Response, next: Function) {
    const correlationId = req.headers['x-correlation-id'] || uuid();
    req['correlationId'] = correlationId;
    res.set('X-Correlation-ID', correlationId);
    next();
  }
}
```

Setup in module:

```
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer.apply(CorrelationIdMiddleware).forRoutes('*');
  }
}
```

7. Implement a custom NestJS validation pipe that ensures array elements are unique

Solution:

```
@Injectable()
export class UniqueArrayPipe implements PipeTransform {
  transform(value: any[], metadata: ArgumentMetadata) {
    if (!Array.isArray(value)) {
      throw new BadRequestException('Value must be an array');
    }
    const unique = [...new Set(value)];
    if (unique.length !== value.length) {
      throw new BadRequestException('Array elements must be unique');
    }
    return value;
  }
}
```

8. Create a custom NestJS decorator that retries failed operations

Solution:

```
export function Retry(attempts: number = 3, delay: number = 1000) {
  return function (target: any, propertyKey: string, descriptor: PropertyDescriptor) {
    const original = descriptor.value;
    descriptor.value = async function (...args: any[]) {
      for (let i = 0; i < attempts; i++) {
        try {
          return await original.apply(this, args);
        } catch (error) {
          if (i === attempts - 1) throw error;
          await new Promise(resolve => setTimeout(resolve, delay));
        }
      }
    };
  };
}
```

9. Implement a custom NestJS interceptor for request/response logging

Solution:

```
@Injectable()
export class LoggingInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler): Observable {
    const req = context.switchToHttp().getRequest();
    const now = Date.now();
    return next.handle().pipe(
      tap(data => console.log({
        timestamp: new Date().toISOString(),
        duration: `${Date.now() - now}ms`,
        method: req.method,
        url: req.url,
        body: req.body,
        response: data
      }))
    );
  }
}
```

10. Create a custom NestJS health check that monitors database connectivity

Solution:

```
@Injectable()
export class DatabaseHealthIndicator extends HealthIndicator {
  constructor(private connection: Connection) {
    super();
  }

  async isHealthy(key: string): Promise {
    try {
      await this.connection.db.admin().ping();
      return this.getStatus(key, true);
    } catch (e) {
      return this.getStatus(key, false, { message: e.message });
    }
  }
}
```