.NET Coding Challenges

Interview Questions and Answers

Core Concepts

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

1. Create a custom value object with equality comparison

```
public readonly struct Money : IEquatable {
    public decimal Amount { get; }
    public string Currency { get; }
    public bool Equals(Money other) =>
        Amount == other.Amount &&
        Currency == other.Currency;
}
```

2. Implement a custom exception handler middleware

```
public class GlobalExceptionHandler {
    private readonly RequestDelegate _next;
    public async Task InvokeAsync(HttpContext context) {
        try { await _next(context); }
        catch (Exception ex) {
            context.Response.StatusCode = 500;
        }
    }
}
```

3. Design a custom object pool implementation

```
public class ObjectPool {
    private readonly ConcurrentBag _objects;
    private readonly Func _objectGenerator;
    public T Get() => _objects.TryTake(out T item) ?
        item : _objectGenerator();
    public void Return(T item) => _objects.Add(item);
}
```

4. Implement a thread-safe singleton pattern in C#

Solution:

Here's a thread-safe implementation using lazy initialization:

```
public sealed class Singleton {
    private static readonly Lazy instance =
        new Lazy(() => new Singleton());
    private Singleton() {}
    public static Singleton Instance => instance.Value;
}
```

Key points:

- Uses Lazy for thread-safe initialization
- Private constructor prevents direct instantiation
- Sealed class prevents inheritance

5. Write a custom generic collection that implements IEnumerable<T>

```
public class CircularBuffer : IEnumerable {
    private T[] buffer;
    private int start, count;
    public IEnumerator GetEnumerator() {
        for(int i = 0; i < count; i++)
            yield return buffer[(start + i) % buffer.Length];
    }
}
```

Important aspects:

- Generic type parameter
- Yield return implementation
- Circular buffer logic

6. Implement a custom async/await operation with cancellation support

```
public async Task ProcessDataAsync(
    IEnumerable items,
    CancellationToken token) {
    foreach(var item in items) {
        token.ThrowIfCancellationRequested();
        await Task.Delay(100, token);
    }
}
```

Key features:

- Proper cancellation token usage
- Async/await pattern
- Exception handling for cancellation

7. Create a custom attribute and reflection-based validator

```
public class MaxLengthAttribute : Attribute {
    public int Length { get; }
    public MaxLengthAttribute(int length) => Length = length;
    public bool IsValid(string value) =>
        value?.Length <= Length;
}
```

Usage:

- Custom attribute definition
- Reflection-based validation
- Parameter validation logic

8. Implement a custom middleware in ASP.NET Core

```
public class TimingMiddleware {
    private readonly RequestDelegate _next;
    public async Task InvokeAsync(HttpContext context) {
        var sw = Stopwatch.StartNew();
        await _next(context);
        context.Response.Headers.Add("X-Response-Time",
            sw.ElapsedMilliseconds.ToString());
    }
}
```

9. Design a generic caching decorator pattern

```
public class CacheDecorator {
    private readonly Func> _getter;
    private readonly IMemoryCache _cache;
    public async Task GetAsync(TKey key) =>
        await _cache.GetOrCreateAsync(key,
            async e => await _getter(key));
}
```

10. Implement a custom LINQ extension method for pagination

```
public static class QueryableExtensions {
    public static IQueryable Page(
        this IQueryable query,
        int page, int pageSize) =>
        query.Skip((page - 1) * pageSize).Take(pageSize);
}
```

Data Structures and Algorithms

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

1. Implement an LRU Cache in C# with a capacity limit. What's the time complexity for get/put operations?

Solution:

An LRU Cache can be implemented using a Dictionary and LinkedList for O(1) operations:

```
public class LRUCache {
    private Dictionary>> _cache;
    private LinkedList> _lruList;
    private int _capacity;

    public LRUCache(int capacity) {
        _cache = new Dictionary>>();
        _lruList = new LinkedList>();
        _capacity = capacity;
    }
```

Time Complexity: Both Get and Put operations are O(1)

2. Write a C# method to find all pairs of integers in an array that sum to a given target. What's the most efficient approach?

Optimal Solution using HashSet:

```
public IList<(int,int)> FindPairs(int[] nums, int target) {
    var seen = new HashSet();
    var result = new List<(int,int)>();
    foreach(var num in nums) {
        if(seen.Contains(target - num))
            result.Add((num, target - num));
        seen.Add(num);
    }
    return result;
}
```

Time Complexity: O(n) with O(n) space complexity

3. Implement a thread-safe producer-consumer queue with a maximum capacity in C#. How would you handle concurrent access?

Implementation using BlockingCollection:

```
public class BoundedQueue {
    private BlockingCollection _queue;
    public BoundedQueue(int capacity) {
        _queue = new BlockingCollection(new ConcurrentQueue(), capacity);
    }
    public void Produce(T item) => _queue.Add(item);
    public T Consume() => _queue.Take();
}
```

Key Points:

- Thread-safe operations
- Automatic blocking when full/empty
- Built-in synchronization

4. Design a data structure that supports O(1) Push, Pop, and GetMin operations. How would you implement it?

MinStack Implementation:

```
public class MinStack {
    private Stack mainStack = new Stack();
    private Stack minStack = new Stack();
    public void Push(int x) {
        mainStack.Push(x);
        if(minStack.Count == 0 || x <= minStack.Peek()) minStack.Push(x);
    }
    public int GetMin() => minStack.Peek();
```

Time/Space Complexity: O(1) for all operations, O(n) space

5. Implement a sliding window maximum algorithm that returns the maximum element for each window of size k in an array.

Efficient Solution using Deque:

```
public int[] MaxSlidingWindow(int[] nums, int k) {
    var result = new List();
    var deque = new LinkedList();
    for(int i = 0; i < nums.Length; i++) {
        while(deque.Count > 0 && deque.First.Value < i - k + 1) deque.RemoveFirst();
        while(deque.Count > 0 && nums[deque.Last.Value] < nums[i]) deque.RemoveLast();
        deque.AddLast(i);
        if(i >= k - 1) result.Add(nums[deque.First.Value]);
    }
```

Time Complexity: O(n)

6. Implement a concurrent dictionary with automatic cleanup of expired entries. How would you handle thread safety?

Implementation:

```
public class ExpiringDictionary {
    private ConcurrentDictionary _store;
    private Timer _cleanupTimer;

    public void Add(TKey key, TValue value, TimeSpan ttl) {
        _store.AddOrUpdate(key, (value, DateTime.UtcNow.Add(ttl)),
            (k, v) => (value, DateTime.UtcNow.Add(ttl)));
    }
```

Features:

- Thread-safe operations
- Automatic cleanup
- TTL support

7. Design a rate limiter that allows n requests per second per user. How would you implement it efficiently?

Token Bucket Implementation:

```
public class RateLimiter {
    private ConcurrentDictionary Timestamps, object Lock)> _buckets;
    private readonly int _maxRequests;
    private readonly TimeSpan _interval;

    public bool AllowRequest(string userId) {
        var now = DateTime.UtcNow;
        return _buckets.AddOrUpdate(userId,
            CreateNewBucket(now),
            (key, bucket) => UpdateBucket(bucket, now));
```

Advantages:

- Thread-safe
- Memory efficient
- Accurate timing

8. Implement a custom IEnumerable that generates Fibonacci numbers up to a specified limit. How would you make it memory efficient?

Lazy Implementation:

```
public class FibonacciSequence : IEnumerable {
    private readonly long _limit;
    public FibonacciSequence(long limit) => _limit = limit;
    public IEnumerator GetEnumerator() {
        long prev = 0, curr = 1;
        while (curr <= _limit) {
            yield return curr;
            (prev, curr) = (curr, prev + curr);
        }
    }
```

Benefits:

- Memory efficient
- Lazy evaluation
- No array storage

9. Create a generic object pool implementation that's thread-safe and supports automatic return of resources.

Implementation:

```
public class ObjectPool {
    private ConcurrentBag _objects;
    private Func _objectGenerator;
    private readonly int _maxSize;

    public T Acquire() {
        return _objects.TryTake(out T item) ? item : _objectGenerator();
    }
    public void Release(T item) {
        if (_objects.Count < _maxSize) _objects.Add(item);
    }
```

Features:

- Thread-safe operations
- Resource recycling
- Configurable pool size

10. Implement a custom priority queue with support for updating priorities. What data structure would you use internally?

Implementation using Heap:

```
public class PriorityQueue {
    private SortedDictionary> _queue = new();

    public void Enqueue(T item, int priority) {
        if (!_queue.ContainsKey(priority))
            _queue[priority] = new Queue();
        _queue[priority].Enqueue(item);
    }
    public T Dequeue() => _queue.First().Value.Dequeue();
```

Time Complexity:

- Enqueue: O(log n)
- Dequeue: O(1) amortized
- Update: O(log n)

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 using .NET technologies

Key Components

- **API Layer:** ASP.NET Core Web API with rate limiting and authentication
- **Storage:** Primary SQL Server for URL mappings, Redis cache for hot URLs
- **URL Generation:** Base62 encoding or MD5 hash first 6-8 chars

Architecture

- Stateless design for horizontal scaling
- Distributed caching with Redis cluster
- Load balancer (Azure/AWS) for traffic distribution

Sample URL Generation Code:

```
public string GenerateShortUrl(string longUrl) {
    var md5Hash = MD5.Create().ComputeHash(Encoding.UTF8.GetBytes(longUrl));
    return Convert.ToBase64String(md5Hash).Substring(0, 7);
}
```

2. Design a real-time chat system using SignalR and .NET

Core Components

- **Backend:** ASP.NET Core with SignalR Hub
- **Storage:** MongoDB for messages, Redis for user sessions
- **Scaling:** Azure SignalR Service for multi-instance support

Key Features

- Message persistence and history
- Presence detection
- Group chat support

Sample Hub Code:

```
public class ChatHub : Hub {
    public async Task SendMessage(string user, string message) {
        await Clients.All.SendAsync("ReceiveMessage", user, message);
        await _messageRepository.SaveMessageAsync(user, message);
    }
}
```

3. Design a social media feed system with .NET

Architecture Components

- **Feed Generation:** Background workers with Hangfire
- **Storage:** Cassandra for posts, Redis for feed cache
- **Queue:** Azure Service Bus for async processing

Feed Types

- Push model for celebrities (fan-out on write)
- Pull model for regular users (fan-out on read)
- Hybrid approach for optimal performance

Sample Feed Query:

```
public async Task> GetUserFeed(string userId) {
```

```
    var cachedFeed = await _redis.GetAsync($"feed:{userId}");
    return cachedFeed ?? await _postRepo.GetLatestPostsAsync(userId);
}
```

4. Design a distributed caching system using .NET

Architecture

- **Primary Cache:** Redis cluster
- **Secondary Cache:** In-memory with Microsoft.Extensions.Caching
- **Consistency:** Write-through caching

Features

- Cache aside pattern implementation
- Circuit breaker for cache failures
- Automatic cache invalidation

Sample Implementation:

```
public async Task GetOrSetAsync(string key, Func> factory) {
    if (await _cache.ExistsAsync(key)) return await _cache.GetAsync(key);
    var value = await factory();
    await _cache.SetAsync(key, value, TimeSpan.FromMinutes(30));
    return value;
}
```

5. Design a job scheduling system using .NET

Components

- **Scheduler:** Quartz.NET for job scheduling
- **Queue:** RabbitMQ for job distribution
- **Storage:** SQL Server for job history

Features

- Cron-based scheduling
- Job retry policies
- Distributed execution

Sample Job Definition:

```
public class DataProcessingJob : IJob {
    public async Task Execute(IJobExecutionContext context) {
        var jobData = context.JobDetail.JobDataMap;
        await _processor.ProcessDataAsync(jobData["dataId"].ToString());
    }
}
```

6. Design a distributed logging system using .NET

Components

- **Collector:** Serilog with structured logging
- **Storage:** Elasticsearch for searchable logs
- **Queue:** Kafka for log streaming

Features

- Correlation IDs for request tracking
- Log aggregation and filtering
- Real-time log analytics

Sample Configuration:

```
Log.Logger = new LoggerConfiguration()
    .WriteTo.Elasticsearch(new ElasticsearchSinkOptions(url))
    .WriteTo.Kafka(kafkaConfig)
    .Enrich.WithCorrelationId()
    .CreateLogger();
```

7. Design a rate limiting system for a .NET API

Implementation Approaches

- **Token Bucket Algorithm**
- **Sliding Window Counter**
- **Distributed Rate Limiting with Redis**

Features

- Per-user and per-IP limiting
- Custom rate limit rules
- Burst handling

Sample Middleware:

```
public async Task IsRateLimitExceeded(string key) {
    var counter = await _redis.IncrementAsync($"ratelimit:{key}");
    await _redis.ExpireAsync($"ratelimit:{key}", TimeSpan.FromMinutes(1));
    return counter > _limit;
}
```

8. Design a notification system using .NET

Components

- **Push Notifications:** Azure Notification Hubs
- **Email:** SendGrid integration
- **SMS:** Twilio integration

Architecture

- Template-based notifications
- Delivery tracking
- Retry mechanism

Sample Notification:

```
public async Task SendNotification(NotificationType type, string userId) {
    var template = await _templateService.GetTemplate(type);
    var channels = await _prefService.GetUserChannels(userId);
    await Task.WhenAll(channels.Select(c => _sender.SendAsync(c, template)));
}
```

9. Design a distributed configuration system using .NET

Components

- **Storage:** Azure App Configuration
- **Secret Management:** Azure Key Vault
- **Change Notification:** Event Grid

Features

- Feature flags
- Configuration versioning
- Real-time updates

Sample Implementation:

```
public async Task GetConfigAsync(string key) {
    var config = await _configClient.GetConfigurationSettingAsync(key);
    await _cache.SetAsync(key, config.Value, TimeSpan.FromMinutes(5));
    return JsonSerializer.Deserialize(config.Value);
}
```

10. Design a microservices-based e-commerce system using .NET

Core Services

- **Catalog Service:** Products and inventory
- **Order Service:** Order processing
- **Payment Service:** Payment processing

Architecture

- API Gateway pattern
- Event-driven communication
- CQRS pattern

Sample Service Communication:

```
public async Task CreateOrder(OrderRequest request) {
    var order = await _orderService.CreateAsync(request);
    await _eventBus.PublishAsync(new OrderCreatedEvent(order.Id));
    return new OrderResult(order);
}
```

Coding and Debugging

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

1. Write a C# method to flatten a nested list of integers into a single list.

Solution:

```
public static List FlattenList(IEnumerable list) {
    var result = new List();
    foreach (var item in list) {
        if (item is IEnumerable subList)
            result.AddRange(FlattenList(subList));
        else if (item is int number)
            result.Add(number);
    }
    return result;
}
```

Key Points:

- Uses recursion to handle nested lists
- Handles mixed types using pattern matching
- Returns a flat List

2. Implement a thread-safe singleton pattern in C#.

Solution:

```
public sealed class Singleton {
    private static readonly Lazy instance =
        new Lazy(() => new Singleton());
    private Singleton() {}
    public static Singleton Instance => instance.Value;
}
```

Key Features:

- Thread-safe using Lazy
- Private constructor prevents direct instantiation
- Sealed class prevents inheritance

3. Write a method to check if a string is a palindrome, ignoring case and non-alphanumeric characters.

Solution:

```
public static bool IsPalindrome(string text) {
    var cleaned = new string(text.Where(char.IsLetterOrDigit).ToArray()).ToLower();
    return cleaned.SequenceEqual(cleaned.Reverse());
}
```

Explanation:

- Filters non-alphanumeric characters
- Case-insensitive comparison
- Uses LINQ for concise implementation

4. Implement a custom memory cache with expiration in C#.

Solution:

```
public class Cache {
   private ConcurrentDictionary _cache = new();
   public void Set(TKey key, TValue value, TimeSpan expiry) =>
      _cache[key] = (value, DateTime.UtcNow.Add(expiry));
   public bool TryGet(TKey key, out TValue value) {
      if (_cache.TryGetValue(key, out var entry) && entry.Expiry > DateTime.UtcNow) {
         value = entry.Value; return true;
      }
      value = default; return false;
   }}
```

5. Create a method to find the first non-repeating character in a string.

Solution:

```
public static char? FirstNonRepeating(string str) {
   return str.GroupBy(c => c)
             .Where(g => g.Count() == 1)
             .Select(g => g.Key)
             .FirstOrDefault();
```

Features:

- Uses LINQ for efficient grouping
- Returns nullable char
- O(n) time complexity

6. Implement a custom attribute to validate method parameters.

Solution:

```
[AttributeUsage(AttributeTargets.Parameter)]
public class NotNullAttribute : Attribute {
   public static void Validate(MethodInfo method, object[] args) {
      var parameters = method.GetParameters();
      for (int i = 0; i < parameters.Length; i++)
         if (parameters[i].GetCustomAttribute() != null && args[i] == null)
            throw new ArgumentNullException(parameters[i].Name);
   }}
```

7. Write a method to detect a cycle in a linked list.

Solution:

```
public bool HasCycle(ListNode head) {
   if (head == null) return false;
   var slow = head;
   var fast = head.Next;
```

```
    while (fast != null && fast.Next != null) {
        if (slow == fast) return true;
        slow = slow.Next;
        fast = fast.Next.Next;
    }
    return false;
}
```

Technique:

- Floyd's Cycle-Finding Algorithm
- O(n) time complexity
- O(1) space complexity

8. Implement a custom exception handler attribute for logging.

Solution:

```
public class LogExceptionAttribute : HandleErrorAttribute {
    public override void OnException(ExceptionContext context) {
        Logger.Error($"Exception in {context.ActionDescriptor.ActionName}: " +
                     $"{context.Exception.Message}");
        context.ExceptionHandled = true;
        context.Result = new JsonResult(new {
            error = "An error occurred"
        });
    }}
```

9. Create a generic method to swap two values.

Solution:

```
public static void Swap(ref T first, ref T second) {
    T temp = first;
    first = second;
    second = temp;
}
```

Key Points:

- Uses generics for type safety
- Uses ref parameters for actual value swap
- Works with any value type or reference type

10. Implement a custom async lock mechanism.

Solution:

```
public class AsyncLock {
    private readonly SemaphoreSlim _semaphore = new SemaphoreSlim(1);
    public async Task LockAsync() {
        await _semaphore.WaitAsync();
        return new AsyncLockReleaser(_semaphore);
    }}
```

Benefits:

- Prevents deadlocks in async code
- Implements IDisposable pattern
- Thread-safe implementation