.NET Mentoring Program Intermediate+ Practice_Q2_2024

CACHING AND MULTITHREADING

How ASP.NET API handles multiple requests?

- Concurrency Model: ASP.NET Web API uses a multi-threaded concurrency model by default, which means that multiple requests can be processed simultaneously on different threads. This allows for efficient utilization of system resources and improves the responsiveness of the application.

- Asynchronous Processing: ASP.NET Web API supports asynchronous programming patterns, such as async/await, allowing long-running I/O-bound operations (like accessing a database or calling an external service) to be executed asynchronously without blocking the thread. This frees up threads to handle other requests while waiting for the asynchronous operations to complete.

- Thread Pool Management: ASP.NET Web API relies on the .NET Thread Pool to manage threads efficiently. When a request arrives, it is typically handled by a thread from the thread pool. If the request requires asynchronous processing, the thread can be returned to the pool while waiting for the asynchronous operation to complete, allowing it to be used for other requests in the meantime.

What are the benefits and downsides of caching? When should we consider applying caching?

Benefits of Caching:

- Improved Performance: Caching allows frequently accessed data or computations to be stored in memory or a faster storage layer. This reduces the need to regenerate or retrieve the data from its original source, leading to faster response times and improved user experience.
- Scalability: Caching helps distribute the load on backend systems by offloading repetitive requests. This allows applications to handle more concurrent users or requests without overloading the infrastructure.
- Lower Resource Consumption: Since cached data can be reused for multiple requests, caching reduces the need for redundant processing or database queries, resulting in lower resource consumption (CPU, memory, and database I/O).

Downsides of Caching:

- Stale Data: Caching can lead to serving stale or outdated data if the cache is not properly invalidated or refreshed. This can result in inconsistencies and incorrect information being displayed to users.
- Cache Invalidation Overhead: Managing cache invalidation can be complex and introduce overhead. Ensuring that cached data remains consistent and up-to-date often requires implementing cache invalidation strategies, which can add complexity to the codebase.
- Increased Memory Usage: Caching data in memory consumes additional memory resources. In scenarios where large volumes of data are cached or the cache size is not properly managed, it can lead to increased memory usage and potential memory pressure on the system.

When to Consider Applying Caching:

- Frequent Data Access: Consider caching data that is frequently accessed but rarely changes. Examples include reference data, configuration settings, or static content.
- Expensive Computations: Cache the results of expensive computations or database queries to avoid repeating the same computations for identical input parameters.
- Highly Accessed Resources: Cache resources that are accessed by a large number of users or requests, such as popular web pages, images, or API responses.
- Reducing Load on Backend Systems: Use caching to reduce the load on backend systems, such as databases or external APIs, by serving cached content for repetitive requests.

- **Improving Response Time:** Apply caching to improve response times and reduce latency, especially for latency-sensitive applications or services.
- **Scalability and Performance**: Consider caching as a scalability and performance optimization technique to improve the overall throughput and responsiveness of the application.

What are the differences between In-memory, Distributed or Request caching options?

Aspect	In-Memory Caching	Distributed Caching	Request Caching
Storage Location	Application memory or server memory	Across multiple nodes or servers	Local to individual requests
Scope	Local to application instance	Shared and synchronized across multiple instances	Specific to each request
Access Speed	Very fast due to direct memory access	Slightly slower due to network overhead	Fast, as response is stored locally
Use Cases	- Small to medium-sized datasets - Reference data, session data	- Large datasets or frequently accessed data - Web farms, cloud environments, microservices	- Idempotent and read-only requests - API responses, web page fragments, static assets

Summary:

- **In-Memory Caching:** Fastest access speed, suitable for single-application instances.
- **Distributed Caching:** Shared and synchronized across multiple nodes, suitable for distributed applications requiring scalability and high availability.
- **Request Caching:** Caches individual HTTP request or API responses, suitable for improving the performance of idempotent and read-only requests.

What does ‘session affinity’ and ‘thread affinity’ mean? When do we have to consider session affinity?

"Session affinity" and "thread affinity" are concepts related to load balancing and concurrency management in distributed systems. Let's define each term and discuss when they are relevant:

Session Affinity:

- **Definition:** Session affinity, also known as sticky sessions or client affinity, is a technique used in load balancing to route requests from the same client to the same server or instance across multiple requests. This ensures that all requests from a particular client are handled by the same server during a session.
- Use Cases:
  - **Stateful Applications:** Applications that maintain session state or user context (e.g., shopping carts, user sessions) may require session affinity to ensure consistent user experience.
  - **Caching**: Applications with in-memory caching may benefit from session affinity to maximize cache hit rates by ensuring that subsequent requests from the same client are served by the same cache instance.

Thread Affinity:

- **Definition:** Thread affinity refers to the association of a thread with a specific CPU core or processing unit in a multi-core system. Thread affinity allows the operating system or runtime environment to schedule threads more efficiently by reducing cache misses and improving CPU cache utilization.
- Use Cases:
  - Performance Optimization: Thread affinity is commonly used in high-performance computing applications, such as scientific simulations or real-time systems, where maximizing CPU cache utilization and minimizing cache contention are critical for performance.
  - Concurrency Control: In some cases, thread affinity may be used to control concurrency and

resource access, ensuring that certain threads have exclusive access to specific resources or data structures.

For better understanding:

Session Affinity: Imagine you're visiting a busy restaurant with multiple waiters. If one waiter serves you initially, session affinity means that every time you return to the restaurant during the same visit, the restaurant ensures that the same waiter serves you. This consistency ensures that your dining experience remains smooth and seamless.

Thread Affinity: Think of a group of workers assigned to different tasks in a factory. Thread affinity is like assigning each worker to a specific workstation and task. Once a worker starts working at a particular station, they stick to it for efficiency. This ensures that each worker becomes more familiar with their task and workspace, leading to smoother operations and less confusion.

In essence, session affinity ensures that your interactions with a system remain consistent across multiple visits, while thread affinity optimizes how tasks are assigned and executed within a system for better performance and efficiency.

What are the race conditions and deadlocks? Do they possible in a single threaded application?

Race Conditions:

- Definition: A race condition occurs when the behavior of a program depends on the relative timing or interleaving of multiple threads or processes. It arises when multiple threads access shared resources or variables concurrently, and the outcome of the program becomes non-deterministic because the order of execution is unpredictable.
- Example: Suppose two threads are incrementing a shared counter variable. If the increment operation is not atomic and both threads read the counter value simultaneously, they may both increment it and write it back, resulting in a loss of updates.
- Possibility in Single-Threaded Applications: Race conditions do not occur in single-threaded applications because there is only one thread of execution. Without concurrent access to shared resources by multiple threads, there is no opportunity for conflicting operations or non-deterministic behavior.

Deadlocks:

- Definition: A deadlock occurs when two or more threads are blocked indefinitely because each thread is waiting for a resource held by another thread, creating a circular dependency. As a result, none of the threads can proceed, and the program becomes stuck in a deadlock state.
- Example: Thread A holds Resource 1 and waits for Resource 2, while Thread B holds Resource 2 and waits for Resource 1. Neither thread can release its resource to allow the other thread to proceed, leading to a deadlock.
- Possibility in Single-Threaded Applications: Deadlocks do not occur in single-threaded applications because there is only one thread, and circular dependencies cannot arise. Since there is no concurrent access to resources, there are no situations where threads can be blocked waiting for each other's resources.

Summary:

- Race Conditions: Occur in multi-threaded applications when multiple threads access shared resources concurrently. Not possible in single-threaded applications.
- Deadlocks: Occur in multi-threaded applications when threads are blocked indefinitely waiting for resources held by each other. Not possible in single-threaded applications.

Why is it not safe to use static constructors/fields when your code is running in a multithreaded application?

Using static constructors or fields in a multi-threaded application can lead to various concurrency issues, including race conditions and inconsistent state, due to the following reasons:

- Non-deterministic Initialization: In a multi-threaded environment, multiple threads may attempt to access or initialize static constructors or fields concurrently. This can lead to non-deterministic behavior where the

order of initialization is unpredictable, potentially causing inconsistent or unexpected results.

- Race Conditions: If multiple threads access static constructors or fields simultaneously, there is a risk of race conditions. For example, if one thread modifies a static field while another thread is reading or writing to it concurrently, the resulting state may be inconsistent or corrupted.
- Deadlocks: Static constructors or fields may involve initialization operations that require synchronization or resource acquisition. If multiple threads attempt to initialize static members that are dependent on each other, it can lead to deadlocks where threads are indefinitely blocked waiting for resources held by other threads.
- Thread Safety: Static constructors or fields are not inherently thread-safe unless explicitly designed and synchronized for concurrent access. Without proper synchronization mechanisms such as locks or synchronization primitives, concurrent access to static members can result in data corruption or race conditions.

To mitigate these issues, it's essential to ensure proper synchronization and thread safety when using static constructors or fields in multi-threaded applications. This may involve using synchronization primitives such as locks, mutexes, or concurrent data structures to coordinate access and ensure consistency across multiple threads. Additionally, avoiding or minimizing the use of static state in favor of instance-level state can help reduce the risk of concurrency issues in multi-threaded environments.

What objects and features .NET proposes to solve race conditions and deadlocks?

.NET provides several objects and features to help solve race conditions and deadlocks in multi-threaded applications:

- Locking Mechanisms: Monitor-based locks (lock keyword), Mutex, Semaphore are synchronization primitives that help control access to shared resources by allowing only one thread to enter a critical section at a time.
- Thread-Safe Collections: NET offers thread-safe collections such as ConcurrentDictionary, ConcurrentQueue, ConcurrentStack, and ConcurrentBag, which allow safe manipulation of data in multi-threaded scenarios without requiring external synchronization.
- Asynchronous Programming: Async/Await, Asynchronous programming with async/await keywords enables writing concurrent code that doesn't block threads unnecessarily. This improves responsiveness and resource utilization in applications with high concurrency.
- Task Parallel Library (TPL): TPL provides the Task class for representing asynchronous operations and the Parallel class for parallelizing loops and computations. TPL also offers cancellation support and task scheduling.
- Interlocked Operations: Interlocked Class - .NET offers atomic operations such as increment, decrement, compare-and-swap, and exchange through the Interlocked class. These operations provide thread-safe manipulation of shared variables without the need for locks.
- Thread Pool: .NET's thread pool manages a pool of worker threads for executing asynchronous tasks. It efficiently schedules and manages threads, reducing the overhead of thread creation and management.