.NET Mentoring Program Intermediate+ Practice_Q2_2024

ASYNCHRONOUS ASP.NET CORE APIS

What are the benefits and drawbacks of async programming?

Benefits:

- Improved responsiveness: Asynchronous programming allows the application to remain responsive while performing long-running tasks, such as I/O operations or network requests. This can lead to a better user experience.
- Utilization of resources: Asynchronous programming allows the efficient use of system resources by enabling other tasks to continue execution while waiting for I/O-bound operations to complete.
- Scalability: Async programming enables applications to handle more concurrent requests without blocking threads, making them more scalable.
- Simplified code: Asynchronous programming in C# provides a straightforward way to write non-blocking code without resorting to complex thread management or callback mechanisms.

Drawbacks:

- Complexity: Asynchronous programming can introduce complexity, especially when dealing with error handling, synchronization, and cancellation. Understanding and debugging async code may require additional effort.
- Learning curve: Developers who are new to asynchronous programming may face a learning curve when understanding concepts such as async/await, Tasks, and asynchronous patterns.
- Potential for bugs: Asynchronous programming introduces the potential for concurrency-related bugs, such as race conditions or deadlocks, if not handled correctly.
- Overhead: Async/await introduces overhead in terms of context switching and managing asynchronous tasks. While this overhead is usually minimal, it can impact performance in certain scenarios.

How to make APS.NET controller action support async flow?

Need to change the return type of the action method to Task<IActionResult> or Task<SomeModel> if the action is expected to return data.
Use the async keyword before the method definition to mark it as an asynchronous method.
Within the method body, use asynchronous versions of I/O-bound operations, such as database queries, network requests, or file I/O operations. You can use await keyword to asynchronously wait for the completion of these operations.
Below is example of an async controller action:

```
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;

namespace YourNamespace.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class SampleController : ControllerBase
    {
        private readonly YourService _service;

        public SampleController(YourService service)
        {
            _service = service;
        }

        [HttpGet("{id}")]
```

```
        public async Task<IActionResult> GetAsync(int id)
        {
            var result = await _service.GetAsync(id);
            if (result != null)
            {
                return Ok(result);
            }
            else
            {
                return NotFound();
            }
        }
    }
}
```

In the above example, GetAsync method is an asynchronous action method that returns a Task<IActionResult>. Inside the method, _service.GetAsync(id) is awaited to asynchronously retrieve data from a service layer.

How does async flow influences APS.NET request executions (life cycle)?

In ASP.NET, the lifecycle of a request is affected by asynchronous programming in the following ways:

- Start of Request Handling: When a request is received by the ASP.NET application, it enters the request pipeline. If the application contains asynchronous controller actions or asynchronous middleware components, these components can be invoked at the beginning of the request processing pipeline.
- Asynchronous Operations: As the request progresses through the pipeline, asynchronous operations such as database queries, network requests, or file I/O may be initiated. When an asynchronous operation is encountered, the executing thread is returned to the thread pool, allowing it to be used to handle other requests while waiting for the asynchronous operation to complete.
- Non-blocking Execution: Asynchronous programming enables non-blocking execution of code. This means that while waiting for asynchronous operations to complete, the thread handling the request is not blocked. Instead, it can be used to handle other requests or perform other tasks within the application.
- Continuation after Await: When an asynchronous operation is awaited, the execution of the request is paused at that point. Once the awaited operation completes, the continuation of the request execution resumes at the point immediately after the await keyword.
- Completion of Request Handling: After all asynchronous operations have completed and all middleware components and controller actions have been executed, the request handling process is completed. The response is then generated and sent back to the client.

Overall, asynchronous programming in ASP.NET allows the application to handle multiple requests concurrently without blocking threads, resulting in improved scalability, responsiveness, and performance.

List at least 5 tips on ASP.NET API performance best practices.

- Use Asynchronous Programming: Utilize asynchronous programming techniques (async/await) to perform I/O-bound operations asynchronously. This allows the server to handle more concurrent requests without blocking threads, leading to better scalability and responsiveness.
- Implement Caching: Cache frequently accessed data or responses to reduce the need for expensive database queries or computations. Use caching mechanisms like in-memory caching (e.g., MemoryCache) or distributed caching (e.g., Redis) to store and retrieve cached data efficiently.
- Optimize Database Queries: Optimize database queries to minimize response times. Use techniques like indexing, query optimization, and database profiling to identify and resolve performance bottlenecks in database interactions.
- Reduce Payload Size: Minimize the size of data transferred over the network by using efficient serialization

formats (e.g., JSON) and avoiding unnecessary data in API responses. Consider implementing pagination, filtering, and projection techniques to limit the amount of data returned in each API call.

- Optimize Authentication and Authorization: Streamline authentication and authorization processes to minimize overhead and. improve request processing times. Use lightweight authentication schemes like JWT (JSON Web Tokens) and efficient authorization mechanisms to ensure secure access control without sacrificing performance. Additionally, consider implementing caching or token-based authentication strategies to reduce the need for repeated authentication checks on every request.

Vertical vs Horizontal scalability. Where to use each?

Vertical Scalability:

- Use Cases: Vertical scalability is suitable when the application's workload can be handled by a single server or a small number of servers. It's often used for applications with predictable or moderate loads, where adding resources to existing servers can efficiently handle increased demand.
- Advantages: Simplified management, as you only need to scale up resources on existing servers. It can be cost-effective for smaller workloads compared to horizontal scaling.
- Disadvantages: Limited scalability compared to horizontal scaling. Eventually, you may reach the physical limits of a single server, leading to bottlenecks and diminishing returns.

Horizontal Scalability:

- Use Cases: Horizontal scalability is suitable for applications with unpredictable or rapidly growing workloads. It's commonly used for web applications, APIs, and services that need to handle a large number of concurrent users or requests. By adding more servers or nodes, horizontal scaling distributes the load and improves fault tolerance.
- Advantages: Virtually unlimited scalability, as you can add more servers or nodes as needed to accommodate increasing demand. Improved fault tolerance and reliability due to redundancy across multiple servers.
- Disadvantages: Increased complexity in managing a distributed system. Requires careful design to ensure statelessness and scalability across multiple nodes. Costlier than vertical scaling, especially if each node requires similar resources.

In summary, vertical scalability is suitable for applications with moderate workloads and simpler infrastructure needs, while horizontal scalability is ideal for applications with unpredictable or rapidly growing workloads that require high scalability.

Explain why the PUT method was suggested for the book action on the order?

The PUT method is suggested for the book action on the order because it aligns well with the HTTP method semantics and RESTful principles:

- Idempotent: The PUT method is idempotent, meaning that performing the same request multiple times will have the same effect as performing it once. In the context of booking an order, if a client sends a PUT request to book a specific order with a specific ID, it will either create or update that order to reflect the new booking details.
- Update or Create: The PUT method is commonly used for both updating existing resources and creating new ones. If the order with the specified ID already exists, the PUT request will update its details with the new booking information. If the order does not exist, the server can create a new order with the provided ID and booking details.
- Clear Intent: Using PUT for booking operations communicates a clear intent to either update or create an order. It adheres to the principle of method uniformity in RESTful APIs, where each HTTP method has a specific and consistent meaning.