🔷 SECTION 10: ASYNC, THREAD POOL & PERFORMANCE 🔷

1️⃣ What is Async / Await?

🔹 Imagine you order food 🍔. Instead of standing at the counter and blocking everyone until food is ready, you: Place the order → Sit down → Get notified when food is ready → That's async. The system is free to serve others while waiting.

🎯 **async and await** are used to write non-blocking asynchronous code.

In ASP.NET Core:

- Async frees the request thread
- Thread returns to Thread Pool
- When I/O completes, continuation runs on another thread

📌 Key point: **Async improves scalability, not CPU speed.**

🔁 Q: Does async create a new thread?

❌ No ; ✔️ It releases the current thread back to the Thread Pool

2️⃣ What is Thread Pool?

🔹 Thread Pool is like a shared worker team 👷.

Instead of hiring a new worker for each task:

- Reuse existing workers.
- Assign tasks when available

🎯 The .NET Thread Pool:

- Manages a pool of worker threads
- Handles request execution
- Threads are reused to avoid creation overhead

ASP.NET Core **does not create a thread per request**. It uses Thread Pool threads.

📌 **Blocking a thread = wasting resources.**

🔁 Q: Who manages Thread Pool?

✔️ .NET runtime

🔁 Q: Can Thread Pool grow?

✔️ Yes, dynamically (but slowly)

3️⃣ Async vs Sync in ASP.NET Core

🔹 Sync = wait and block

Async = wait without blocking

🎯 Synchronous code

- Blocks thread
- Reduces throughput
- Causes thread starvation

Asynchronous code

- Frees thread during I/O wait
- Handles more concurrent requests

📌 ASP.NET Core is optimized for async I/O.

🔁 Q: Should everything be async?

❌ No ; ✔️ Only I/O-bound operations

4️⃣ I/O-bound vs CPU-bound (VERY IMPORTANT)

I/O-bound	CPU-bound
• Database calls • HTTP calls • File access ✔️ Use async/await	• Encryption • Image processing • Large calculations ❖ ❌ Async does NOT help ❖ ✔️ Use background processing / Task.Run cautiously

🔁 Q: Does async improve CPU performance?

❌ No ; ✔️ Improves scalability

5️⃣ Thread Starvation (INTERVIEW FAVORITE)

🔹 Too many people waiting for service, but workers are stuck.

🎯 Thread starvation occurs when:

- Thread Pool threads are blocked
- New requests cannot get threads
- App becomes slow or unresponsive

Common causes: `.Result` `.Wait()` Blocking calls Sync-over-async

📌 This is a production killer.

🔁 Q: Symptoms of thread starvation?

✔ High latency, low CPU usage, request timeouts

6️⃣ Sync-over-Async (DANGEROUS)

🔹 Calling async code like sync code.

🎯 Examples:

```
var data = GetDataAsync().Result;
```

Problems:

- Blocks thread
- Causes deadlocks
- Kills scalability

📌 ASP.NET Core avoids classic deadlocks, but still causes starvation.

🔹 Interview Line: ***"Sync-over-async is one of the most common performance issues in ASP.NET Core applications."***

7️⃣ ConfigureAwait – Do We Need It?

🔹 It tells async code where to continue.

🎯 In ASP.NET Core:

- No SynchronizationContext
- `ConfigureAwait(false)` is optional
- Still useful in library code

📌 In ASP.NET Core, `ConfigureAwait` is not mandatory like classic ASP.NET.

8️⃣ Parallelism vs Async (COMMON CONFUSION)

Concept	Async	Parallel
Purpose	Non-blocking I/O	CPU utilization
Threads	Reused	Multiple threads
Use case	Web APIs	Heavy computation

Q: Should we use Parallel.For in APIs?

❌ No (can exhaust Thread Pool)

9️⃣ Performance Best Practices (INTERVIEW GOLD)

✅ Do

- Use async for all I/O
- Avoid blocking calls
- Use caching
- Use pagination
- Use streaming for large responses

❌ Don't

- Use `.Result`
- Use `Task.Run` in controllers
- Do heavy work in middleware

🔟 Measuring Performance

Tools:

- Application Insights
- dotnet-counters
- dotnet-trace
- Logs + metrics

Key metrics:

- Request latency
- Thread Pool usage
- CPU utilization

🧠 ONE-LINE INTERVIEW ANSWER (MEMORIZE)

> **"Async in ASP.NET Core improves scalability by freeing Thread Pool threads during I/O operations. Blocking threads or using sync-over-async can cause thread starvation and severe performance issues."**

✅ YOU NOW UNDERSTAND

✓ Async/await deeply
✓ Thread Pool internals
✓ Thread starvation
✓ I/O vs CPU bound
✓ Sync-over-async dangers
✓ Performance best practices

🔧 HANDS-ON 10: ASYNC, THREAD POOL & PERFORMANCE 🔧

🧱 STEP 0: Create Project

```
dotnet new webapi -n AsyncPerformanceDemo
cd AsyncPerformanceDemo
```

🧱 STEP 1: Create a Fake I/O Service (Simulate DB Call)

📁 Create folder: *Services*

```
public class FakeDatabaseService
{
    public async Task<string> GetDataAsync()
    {
        await Task.Delay(2000); // Simulate I/O delay
        return "Data from DB";
    }
}
```

Register it:

```
builder.Services.AddScoped<FakeDatabaseService>();
```

📌 Interview explanation: `Task.Delay` simulates non-blocking I/O, just like DB or HTTP calls.

🧱 STEP 2: Create BAD Controller (Sync-over-Async ❌)

📁 *Controllers/BadController.cs*

```
[ApiController]
[Route("api/bad")]
public class BadController : ControllerBase
{
    private readonly FakeDatabaseService _db;
    public BadController(FakeDatabaseService db)
    {
        _db = db;
    }
    [HttpGet]
    public IActionResult Get()
    {
```

```
        // ❌ BAD PRACTICE
        var data = _db.GetDataAsync().Result;
        return Ok(data);
    }
}
```

🔍 What's happening internally?

- Request thread is blocked
- Thread Pool thread is stuck
- Cannot serve other requests
- Under load → thread starvation

📌 "This is sync-over-async and causes Thread Pool exhaustion."

🧱 STEP 3: Create GOOD Controller (Async Correctly ✅)

📁 *Controllers/GoodController.cs*

```
[ApiController]
[Route("api/good")]
public class GoodController : ControllerBase
{
    private readonly FakeDatabaseService _db;
    public GoodController(FakeDatabaseService db)
    {
        _db = db;
    }
    [HttpGet]
    public async Task<IActionResult> Get()
    {
        var data = await _db.GetDataAsync();
        return Ok(data);
    }
}
```

🔍 Why this is good?

- Thread released during await
- Thread Pool reused
- High concurrency supported

📌 "Async frees the request thread during I/O waits."

🧱 STEP 4: Simulate Load (Conceptually)

Send multiple requests (10–50 concurrent):

Endpoint : Behavior

`/api/bad` : Slow, hangs, timeouts

`/api/good` : Handles load smoothly

📌 Bad endpoint blocks threads; good endpoint scales.

🧱 STEP 5: Thread Pool Starvation DEMO

Create Blocking Endpoint

```
[HttpGet("block")]
public IActionResult Block()
{
    Thread.Sleep(5000); // ❌ BLOCKING
    return Ok("Blocked");
}
```

📌 Problem:

- Thread.Sleep blocks Thread Pool thread
- App becomes unresponsive under load

Fix with Async:

```
[HttpGet("noblock")]
public async Task<IActionResult> NoBlock()
{
    await Task.Delay(5000);
    return Ok("Non-blocking");
}
```

📌 `Task.Delay` is async; `Thread.Sleep` is blocking.

🧱 STEP 6: CPU-Bound Task Mistake

❌ Wrong Use of Task.Run

```
[HttpGet("cpu")]
public async Task<IActionResult> Cpu()
```

```
{
    await Task.Run(() => HeavyCalculation());
    return Ok("Done");
}
```

📌 Why bad?

- Steals Thread Pool threads
- Reduces request throughput

✔ Correct Interview Answer: **“CPU-bound work should be offloaded to background workers or separate services, not run inside API requests.”**

🧱 STEP 7: Measuring Thread Pool Health (Interview Knowledge)

You can say:

“I monitor thread pool starvation using dotnet-counters and Application Insights by observing request latency and thread usage.”

Metrics to watch:

- Thread Pool Queue Length
- Request Duration
- CPU vs Latency mismatch

🧱 STEP 8: ConfigureAwait Demo (Library Code)

```
public async Task<string> LibraryMethod()
{
    await Task.Delay(1000).ConfigureAwait(false);
    return "Done";
}
```

📌 **In ASP.NET Core, `ConfigureAwait` is optional, but recommended in reusable libraries.**

🧱 STEP 9: Common Interview Traps (WITH ANSWERS)

❓ Does async make code faster?

❌ No, ✔ Makes app scalable

❓ Does async create new threads?

❌ No, ✔ Reuses Thread Pool threads

❓ Can async cause memory leaks?

✔ Yes, if tasks are not awaited properly

🧠 FINAL INTERVIEW STORY (VERY IMPORTANT)

You can confidently say:

“We had performance issues due to sync-over-async and `Thread.Sleep` blocking Thread Pool threads. We fixed it by making all I/O calls async, removing blocking code, and monitoring Thread Pool metrics, which significantly improved throughput.”

🔥 This sounds like real production experience.

✅ YOU NOW MASTER (HANDS-ON)

✓ Async vs Sync ✓ Thread Pool behaviour ✓ Thread starvation

✓ Sync-over-async ✓ CPU vs I/O tasks ✓ Real performance fixes

---------- Notes----------