# Middleware and Dependency Injection

By
Narasimha Rao T
***Microsoft.Net FSD Trainer***
Professional Development Trainer
tnrao.trainer@gmail.com

## 1. What is Middleware?

- **Definition:**
  Middleware is software that sits in the HTTP request/response pipeline in ASP.NET Core.
  Each middleware component processes the request and either:
  1. Passes it to the next middleware in the pipeline.
  2. Handles it directly and ends the pipeline.
- **Key Points:**
  - Runs sequentially in the order they are registered.
  - Can perform actions before and after calling the next component.
  - Examples: Authentication, Logging, Routing, Exception Handling.

## 2. Why Do We Use Middleware?

- **Common Use Cases:**
  - Authentication & Authorization
  - Logging and request monitoring
  - Exception handling
  - Response compression & caching
  - URL rewriting
- **Benefits:**
  - Modular and reusable
  - Centralized request handling
  - Configurable and easy to maintain

## 3. Middleware Pipeline – Deep Dive

- Configured inside `Program.cs` (or `Startup.cs` in older versions).
- Uses `app.UseXXX()`, `app.Run()`, and `app.Map()` methods.
- **Pipeline Flow:**
  - Request → Middleware 1 → Middleware 2 → Controller/Endpoint → Response
  - If one middleware doesn't call `next()`, the pipeline stops.

## Example:

```
app.Use(async (context, next) =>
{
    // Before next middleware
    await next();
    // After next middleware
});
```

- **Order matters:** Authentication should come before Authorization, Routing before Endpoints, etc.

## 4. How to Create Custom Middleware?

**Steps:**

1. Create a class with a constructor accepting `RequestDelegate`.
2. Implement an `Invoke` or `InvokeAsync` method.
3. Register middleware using `app.UseMiddleware<YourMiddleware>()`.

## Example:

```
public class CustomLoggingMiddleware
{
    private readonly RequestDelegate _next;
    public CustomLoggingMiddleware(RequestDelegate next) => _next = next;

    public async Task InvokeAsync(HttpContext context)
    {
        Console.WriteLine($"Request: {context.Request.Method} {context.Request.Path}");
        await _next(context);
    }
}

// Register in Program.cs
app.UseMiddleware<CustomLoggingMiddleware>();
```

## 5. Built-in vs Custom Middleware

- **Built-in Middleware Examples:**
    - `UseRouting`, `UseAuthentication`, `UseAuthorization`, `UseEndpoints`, `UseExceptionHandler`
- **Custom Middleware:**
    - Created for specific application needs (e.g., custom logging, custom response headers)

## 6. What is Dependency Injection (DI)?

- **Definition:**
  DI is a design pattern where dependencies are provided to a class rather than being created inside the class.
- **Benefits:**
  - Reduces coupling
  - Improves testability
  - Promotes clean code architecture

## 7. How to Implement Dependency Injection in ASP.NET Core?

- ASP.NET Core has a **built-in IoC container**.
- Register services inside `Program.cs`:

```
builder.Services.AddTransient<IMyService, MyService>();
```

- Inject into constructors:

```
public class MyController : ControllerBase
{
    private readonly IMyService _service;
    public MyController(IMyService service)
    {
        _service = service;
    }
}
```

## 8. DI Lifetimes

- **Transient:**
  - New instance created each time.
  - Use for lightweight, stateless services.

```
services.AddTransient<IMyService, MyService>();
```

- **Scoped:**
  - One instance per request.
  - Good for per-request operations like repositories.

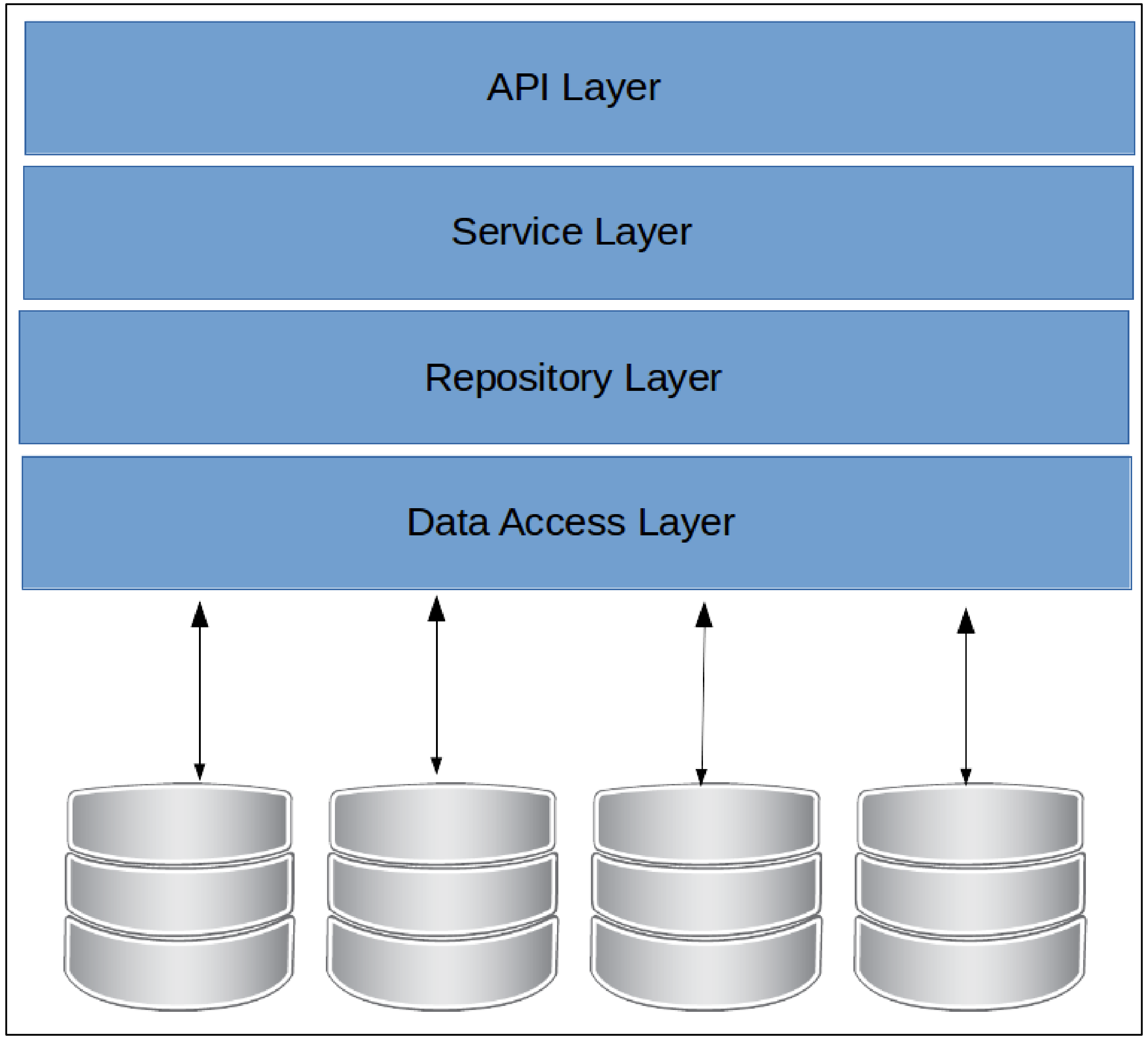
```
services.AddScoped<IMyService, MyService>();
```

- **Singleton:**
    - One instance for the entire application lifetime.
    - Use for heavy services like caching.

```
services.AddSingleton<IMyService, MyService>();
```

## 9. Repository and Service Pattern

- **Repository Pattern:**
  - Encapsulates data access logic.
  - Example: `IProductRepository` with methods like `GetAllProducts()`.
- **Service Pattern:**
  - Encapsulates business logic.
  - Uses repositories internally.


API Layer
Service Layer
Repository Layer
Data Access Layer

## Example:

```
public interface IProductRepository
{
    IEnumerable<Product> GetAll();
}

public class ProductRepository : IProductRepository
{
    // Inject DbContext
}

public interface IProductService
{
    IEnumerable<Product> GetProducts();
}

public class ProductService : IProductService
{
    private readonly IProductRepository _repo;
    public ProductService(IProductRepository repo) => _repo = repo;

    public IEnumerable<Product> GetProducts() => _repo.GetAll();
}
```

Register with DI:

```
services.AddScoped<IProductRepository, ProductRepository>();
services.AddScoped<IProductService, ProductService>();
```

## 10. Configuration via `appsettings.json` and `IOptions<T>`

- appsettings.json Example:

```json
{
  "MySettings": {
    "ApiKey": "12345",
    "Timeout": 30
  }
}
```

- Bind to a POCO:

```csharp
public class MySettings
{
    public string ApiKey { get; set; }
    public int Timeout { get; set; }
}
```

- **Register and Inject:**

```
builder.Services.Configure<MySettings>(builder.Configuration.GetSection("MySettings"));

public class MyController : ControllerBase
{
    private readonly MySettings _settings;
    public MyController(IOptions<MySettings> options)
    {
        _settings = options.Value;
    }
}
```

## Self-Checking Questions

1. What is middleware and how does it work in ASP.NET Core?
2. Difference between `Use`, `Run`, and `Map` in middleware?
3. Explain DI lifetimes and when to use each?
4. What are advantages of Repository pattern?
5. How does `IOptions<T>` differ from `IConfiguration`?
6. What happens if a middleware does not call `next()`?
7. Difference between `AddSingleton` and `AddScoped` in terms of threading?
8. How does ASP.NET Core handle configuration hierarchy (appsettings.json, env variables)?
9. How would you implement global exception handling middleware?
10. What is the order of middleware execution and why is it important?