

## Logging

Logging is een belangrijk aspect van elke applicatie. Als er een fout optreedt in een applicatie proberen we deze steeds na te bootsen om op deze manier uit te zoeken wat er is fout gelopen, maar dat is niet altijd evident. Logging kan ons daarbij helpen omdat het belangrijke informatie kan bevatten over wat er is gebeurd. Daarnaast kunnen we logging ook gebruiken om onze applicatie te monitoren.

## Configuratie

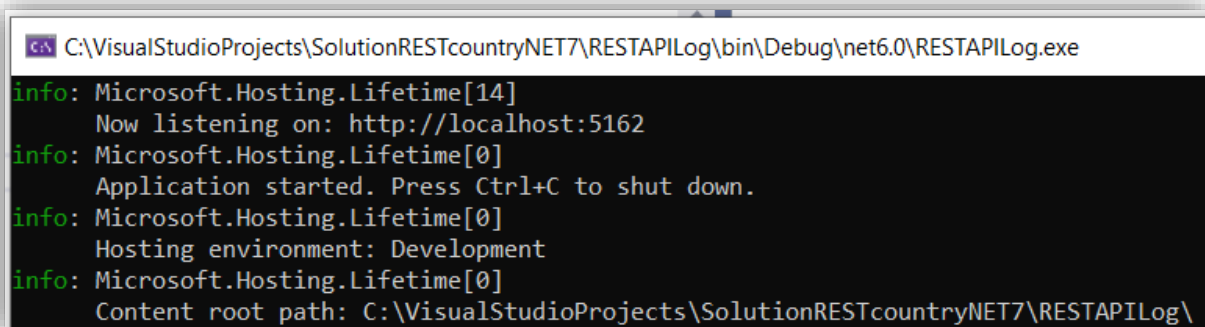
In een ASP.NET Core applicatie worden standaard een aantal logging providers toegevoegd.

- Call `WebApplication.CreateBuilder`, which adds the following logging providers:
  - `Console`
  - `Debug`
  - `EventSource`
  - `EventLog`: Windows only

Wensen we daar van af te wijken dan kunnen we de logging configureren in de Program klasse als volgt :

```
public static void Main(string[] args)
{
    var builder = WebApplication.CreateBuilder(args);
    builder.Logging.ClearProviders();
    builder.Logging.AddConsole();
    builder.Logging.AddDebug();
}
```

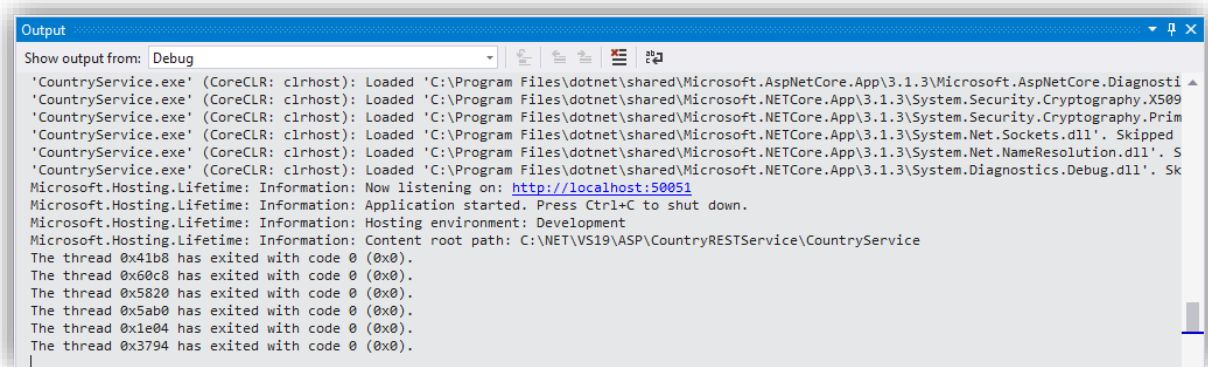
Console logging vinden we terug als we onze applicatie opstarten en ziet er bijvoorbeeld als volgt uit.



The screenshot shows a console window titled "C:\VisualStudioProjects\SolutionRESTcountryNET7\RESTAPILog\bin\Debug\net6.0\RESTAPILog.exe". The output consists of several log entries from Microsoft.Hosting.Lifetime, indicating the application is listening on http://localhost:5162, has started, and is running in the Development environment. The content root path is also displayed.

```
C:\VisualStudioProjects\SolutionRESTcountryNET7\RESTAPILog\bin\Debug\net6.0\RESTAPILog.exe
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5162
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\VisualStudioProjects\SolutionRESTcountryNET7\RESTAPILog\
```

Debug logging vinden we terug in Visual Studio in het Output venster.



We beperken ons enkel tot console en debug logging.

## Gebruik logger

Op welke manier kunnen we nu de logger gaan gebruiken ? We beginnen met het definiëren van een variabele logger van het type `ILogger` in onze `CountryController` klasse. Het initialiseren van de variabele vindt plaats in de constructor door middel van dependency injection (hierover later meer).

```
builder.Services.AddSingleton<ICountryRepository, CountryRepository>();
```

Aan de logger in de constructor geven we ook een type mee, in dit geval de `CountryController` klasse. Het gebruik van de logger is nu vrij eenvoudig, op de plaats in onze code waar we log info wensen weg te schrijven nemen we de variabele logger en selecteren één van de logmethoden (in dit voorbeeld is het `LogInformation`) en geven de te loggen boodschap mee.

```

public class CountryController : ControllerBase
{
    private readonly ICountryRepository repo;
    private readonly ILogger logger;

    1 reference
    public CountryController(ICountryRepository repo, ILogger<CountryController> logger)
    {
        this.repo = repo;
        this.logger = logger;
    }

    [HttpGet]
    [HttpHead]
    0 references
    public IEnumerable<Country> GetAll()
    {
        logger.LogInformation("GetAll called");
        return repo.GetAll();
    }
}

```

We starten onze applicatie en voeren via Swagger een GET request door waarmee we alle gegevens opvragen (<http://localhost:xxxx/api/country>) en bekijken het resultaat. In onze console zijn nu 5 logberichten te zien. Elk bericht geeft eerst het log level weer (zie verder), in dit geval 'info', daarna de categorie (vb CountryService.Controllers.CountryController in het laatste geval) en een eventid (vb [0]) en tenslotte de boodschap die we wensen te loggen (vb GetAll called).

De categorie wordt bepaald door het type dat we meegeven aan de ILogger, in dit voorbeeld hebben we in de constructor een variabele van het type ILogger<CountryController> meegegeven wat resulteert in een categorienaam 'CountryService.Controllers.CountryController' (naam van de klasse in combinatie met de namespace).

```

C:\VisualStudioProjects\SolutionRESTcountryNET7\RESTAPILog\bin\Debug\net6.0\RESTAPILog.exe
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5162
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\VisualStudioProjects\SolutionRESTcountryNET7\RESTAPILog\
info: RESTAPILog.Controllers.CountryController[1001]
      GetAll called

```

Naast de categorie is er ook de mogelijkheid om een EventId mee te geven bij het loggen. Dit kan op de volgende eenvoudige manier :

```

logger.LogInformation(1001, "GetAll called");

```

Het resultaat is dan :

```
info: RESTAPILog.Controllers.CountryController[1001]  
    GetAll called
```

EventIds kunnen nuttig zijn om in de logbestanden te zoeken naar overeenkomstige logs, dit kan dan door bijvoorbeeld alle logs te selecteren met een bepaald eventId.

## LogLevel

In de vorige paragraaf hebben we reeds het begrip log level aangehaald. In het voorbeeld hebben we in information log weggeschreven. Er zijn echter verschillende niveaus van logging mogelijk die telkens een bepaalde graad van belangrijkheid aanduiden. Een ophijsting vinden we terug in de volgende overzichtstabel.

LogLevel	Value	Method	Description
Trace	0	LogTrace	Contain the most detailed messages. These messages may contain sensitive app data. These messages are disabled by default and should <b>not</b> be enabled in production.
Debug	1	LogDebug	For debugging and development. Use with caution in production due to the high volume.
Information	2	LogInformation	Tracks the general flow of the app. May have long-term value.
Warning	3	LogWarning	For abnormal or unexpected events. Typically includes errors or conditions that don't cause the app to fail.
Error	4	LogError	For errors and exceptions that cannot be handled. These messages indicate a failure in the current operation or request, not an app-wide failure.
Critical	5	LogCritical	For failures that require immediate attention. Examples: data loss scenarios, out of disk space.
None	6		Specifies that a logging category should not write any messages.

Het gebruik van de verschillende log levels kan door middel van een bijhorende methode, voor elk level is er namelijk een specifieke methode voorzien. We passen nu de GetAll methode aan om alle methodes even uit te testen.

```

public IEnumerable<Country> GetAll()
{
    logger.LogInformation(1001, "GetAll called");
    logger.LogTrace("Trace log");
    logger.LogDebug("Debug log");
    logger.LogInformation("Informationlog");
    logger.LogError("Error log");
    logger.LogCritical("Critical log");
    return repo.GetAll();
}

```

In onze console verschijnen dan de volgende logs na het uitvoeren van een GET request.

```

C:\VisualStudioProjects\SolutionRESTcountryNET7\RESTAPILog\bin\Debug\net6.0\RESTAPILog.e...
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5162
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\VisualStudioProjects\SolutionRESTcountryNET7\RESTAPILog\
info: RESTAPILog.Controllers.CountryController[1001]
      GetAll called
trce: RESTAPILog.Controllers.CountryController[0]
      Trace log
dbug: RESTAPILog.Controllers.CountryController[0]
      Debug log
info: RESTAPILog.Controllers.CountryController[0]
      Informationlog
fail: RESTAPILog.Controllers.CountryController[0]
      Error log
crit: RESTAPILog.Controllers.CountryController[0]
      Critical log

```

Door middel van appsettings.json kunnen we bepalen wat we wel of niet wensen te zien verschijnen in onze logs. Bekijken we het Logging item even in detail dan merken we op dat we een "Default" setting hebben, met in dit geval de waarde "Information". Dit wil zeggen dat we standaard alle berichten loggen die van het type information (value 2 in de overzichtstabel) zijn of hoger (warning, error en critical). We kunnen echter ook specifiekere definiëren wat we wensen te loggen door aan een bepaalde klasse (namespace) een log level toe te kennen. In dit voorbeeld hebben we voor de CountryController het laagste niveau (trace) ingesteld waardoor dus alle berichten verschijnen in de console.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning",
      "RESTAPILog.Controllers.CountryController": "Trace"
    }
  },
  "AllowedHosts": "*"
}
```

Passen we de loglevels aan als volgt : waarbij we alle logs afkomstig van een Microsoft namespace (Microsoft\*) op het loglevel 'error' zetten (opmerking : gebruik "\*" om alle te selecteren) dan verdwijnen er een aantal logboodschappen.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft*": "Error",
      "RESTAPILog.Controllers.CountryController": "Trace"
    }
  },
  "AllowedHosts": "*"
}
```

```
C:\VisualStudioProjects\SolutionRESTcountryNET7\RESTAPILog\bin\Debug\net6.0\RESTAPILog.exe
info: RESTAPILog.Controllers.CountryController[1001]
      GetAll called
trce: RESTAPILog.Controllers.CountryController[0]
      Trace log
dbug: RESTAPILog.Controllers.CountryController[0]
      Debug log
info: RESTAPILog.Controllers.CountryController[0]
      Informationlog
fail: RESTAPILog.Controllers.CountryController[0]
      Error log
crit: RESTAPILog.Controllers.CountryController[0]
      Critical log
```

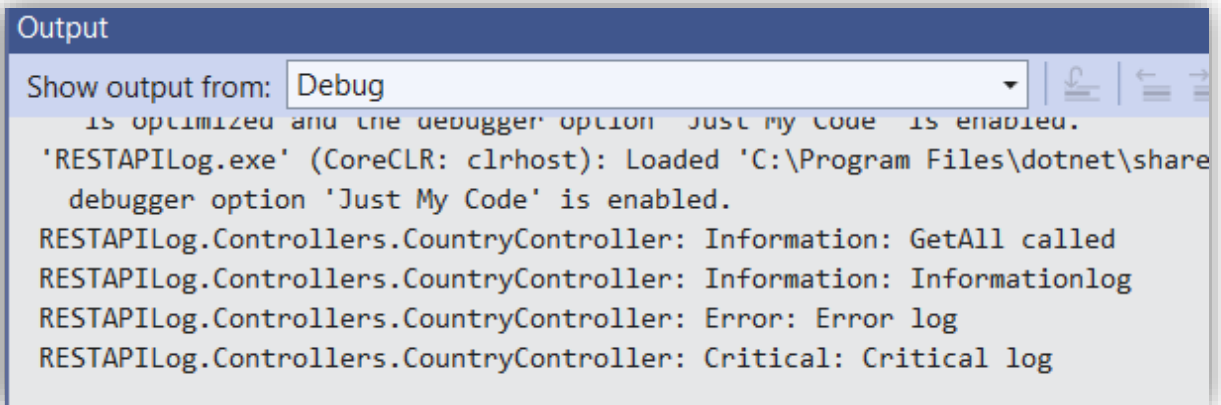
Daarnaast is het ook mogelijk om voor elke logger (console, debug, ...) de nodige instellingen te voorzien. Dit kan door in de logging sectie de logger op te lijsten samen met zijn instellingen. In het volgende voorbeeld hebben we voor de console logger ingesteld dat het standaard niveau 'Information' (of hoger) is en dat voor alle 'Microsoft\*' namespaces we enkel de berichten vanaf het niveau 'Error' wensen te loggen. Voor onze controller (countrycontroller) hebben we het loglevel op 'Trace' gezet in tegenstelling tot het loglevel 'information' dat we gebruiken als standaard instelling.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning",
      "RESTAPILog.Controllers.CountryController": "Information"
    },
    "Console": {
      "LogLevel": {
        "Default": "Information",
        "Microsoft*": "Error",
        "RESTAPILog.Controllers.CountryController": "Trace"
      }
    }
  },
  "AllowedHosts": "*"
}
```

Deze nieuwe instellingen toegepast op onze applicatie na uitvoeren van een GET request geeft dan de volgende output in de console :

```
C:\VisualStudioProjects\SolutionRESTcountryNET7\RESTAPILog\bin\Debug\net6.0\RESTAPILog.exe
info: RESTAPILog.Controllers.CountryController[1001]
      GetAll called
trce: RESTAPILog.Controllers.CountryController[0]
      Trace log
dbug: RESTAPILog.Controllers.CountryController[0]
      Debug log
info: RESTAPILog.Controllers.CountryController[0]
      Informationlog
fail: RESTAPILog.Controllers.CountryController[0]
      Error log
crit: RESTAPILog.Controllers.CountryController[0]
      Critical log
```

In het debug-output venster verschijnen er echter minder berichten omdat het standaardloglevel daar anders is ingesteld.



## LoggerFactory

We hebben gezien dat we in de constructor van de controller een ILogger van een bepaald type konden meegeven waardoor de categorie werd bepaald. Er is echter ook een mogelijkheid om met een ILoggerFactory te werken waarbij we zelf een naam voor de categorie kunnen meegeven. Dit kan als volgt worden geïmplementeerd, we geven nu aan de constructor een ILoggerfactory mee als parameter en de logger variabele (van het type ILogger) wordt nu aangemaakt door de functie CreateLogger op te roepen van de factory. Als parameter geven we de naam van de categorie mee.

```
public CountryController(ICountryRepository repo, ILoggerFactory loggerFactory)
{
    this.repo = repo;
    this.logger = loggerFactory.CreateLogger("Country");
}
```

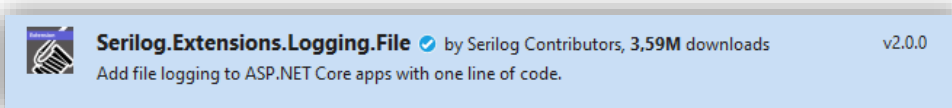
Het resultaat wordt dan :



```
C:\VisualStudioProjects\SolutionRESTcountryNET7\RESTAPILog\bin\Debug\net6.0\RESTAPILog.exe
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5162
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\VisualStudioProjects\SolutionRESTcountryNET7\RESTAPILog\
info: Country[1001]
      GetAll called
info: Country[1001]
      GetAll called
info: Country[0]
      Informationlog
fail: Country[0]
      Error log
crit: Country[0]
      Critical log
```

## Logging naar bestanden

Nog een belangrijk aspect bij loggen is het wegschrijven van de loginfo zodat deze achteraf kunnen worden geanalyseerd. Tot nu toe hebben we steeds gebruik gemaakt van de door Microsoft ingebouwde logging mogelijkheden. Voor het wegschrijven van de loginfo naar een bestand moeten we gebruik maken van externe tools. Hier kiezen we voor het NuGet package `Serilog.Extensions.Logging.File`. Dit packet zorgt ervoor dat de extension method `AddFile` nu beschikbaar is bij de `ILoggerFactory` variabele.



Het toevoegen van een file logging kan nu eenvoudig door aan de `AddFile` methode de naam van het log bestand mee te geven.

```
public CountryController(ICountryRepository repo, ILoggerFactory loggerFactory)
{
    this.repo = repo;
    this.logger = loggerFactory.AddFile("ControllerLogs.txt").CreateLogger("Country");
}
```

Voeren we onze applicatie weer uit met het GET request, dan krijgen we de volgende uitvoer in de console :

```
C:\NET\VS19\ASP\CountryRESTService\CountryService\bin\Debug\netcoreapp3.1\CountryService.exe
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://localhost:50051
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\NET\VS19\ASP\CountryRESTService\CountryService
info: Country[1001]
      GetAll called
info: Country[0]
      Informationlog
fail: Country[0]
      Error log
crit: Country[0]
      Critical log
```

En verschijnt er een ControllerLogs-20200909.txt bestand met als inhoud :

```
ControllerLogs-20200909.txt - Kladblok
Bestand  Bewerken  Opmaak  Beeld  Help
2020-09-09T09:36:40.1236757+02:00 [INF] GetAll called (2ecccc5e)
2020-09-09T09:36:40.2215264+02:00 [INF] Informationlog (02e6b7ef)
2020-09-09T09:36:40.2510073+02:00 [ERR] Error log (d1918764)
2020-09-09T09:36:40.2616275+02:00 [FTL] Critical log (f958a3f4)
```

Er werd automatisch een datum toegevoegd aan de bestandsnaam.

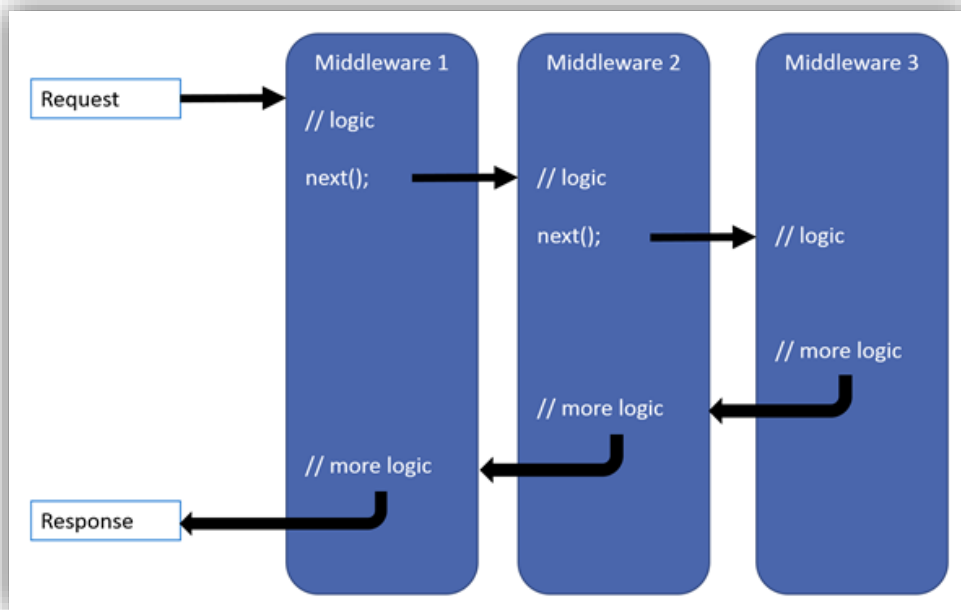
**Opgave : verklaar de verschillen tussen de output in de console en het logbestand, waarom zijn niet alle items terug te vinden in het logbestand ?**

## Middleware

Middleware speelt een belangrijker rol in een ASP.NET Core webapplicatie. Het is software die een pijplijn vormt voor de afhandeling van requests en responses. Elke component bepaald of het verzoek wordt doorgegeven naar de volgende component en kan ook een bepaalde taak uitvoeren voor of na een aansluitende component. Om de pijplijn te bouwen wordt er gebruik gemaakt van request delegates.

Hoe werkt het ?

Aan de hand van het volgende schema wordt het principe van de pijplijn uitgelegd.



Een verzoek komt binnen bij de eerste middleware component, hier kan nu een bepaald (business) logica worden toegepast waarna het verzoek wordt doorgestuurd naar de volgende component (middleware 2). In deze tweede component kan nu een andere verwerking (logica) worden toegekend vooraleer door te sturen naar de derde (en in dit voorbeeld laatste) component. In deze laatste component wordt het verzoek verder behandelt en start ook het antwoord. Dit antwoord wordt nu naar de vorige (middleware 2) component gestuurd waar alweer een eventuele bewerking kan worden uitgevoerd. De laatste stap is nu het doorsturen van de response naar de eerste middleware uit onze pijplijn waar nog een eventuele bewerking kan worden gedaan alvorens de pijplijn te verlaten.

We tonen aan de hand van een eenvoudig voorbeeld hoe dit precies in zijn werk gaat. Het definiëren van de pijplijn vindt plaats in de Program klasse. Het toevoegen van een middleware component gebeurt steeds door middel van de Use methode van app variabele.

We registreren hier 2 middleware componenten die enkel een boodschap loggen wanneer een request binnen komt en wanneer een response de middleware verlaat.

```

var app = builder.Build();
app.Logger.LogInformation("start app");
var logFactory = LoggerFactory.Create(builder => builder.AddConsole());
ILogger logger=logFactory.CreateLogger("middleware");

app.Use(async (context, next) =>
{
    logger.LogInformation("M1 start");
    await next();
    logger.LogInformation("M1 end");
});
app.Logger.LogInformation("use 2");
app.Use(async (context, next) =>
{
    logger.LogInformation("M2 start");
    await next();
    logger.LogInformation("M2 end");
});

app.MapControllers();
app.Logger.LogInformation("run");
app.Run();

```

De middleware wordt eerst geregistreerd, maar nog niet uitgevoerd. App.Run() is de laatste component die wordt uitgevoerd in request pipeline.

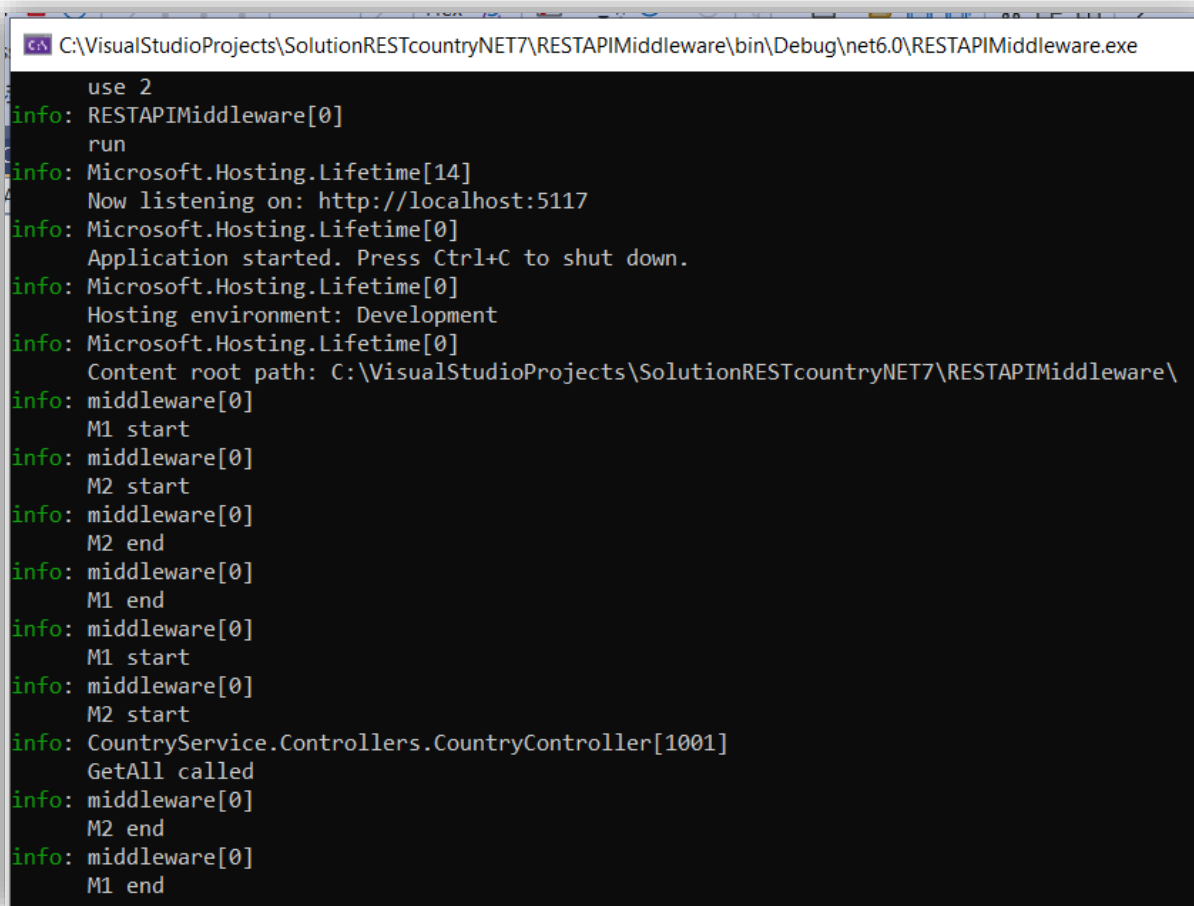
```

C:\VisualStudioProjects\SolutionRESTcountryNET7\RESTAPIMiddleware\bin\Debug\net6.0\RESTAPIMiddleware.exe
info: RESTAPIMiddleware[0]
      start app
info: RESTAPIMiddleware[0]
      use 2
info: RESTAPIMiddleware[0]
      run
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5117
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\VisualStudioProjects\SolutionRESTcountryNET7\RESTAPIMiddleware\
info: middleware[0]
      M1 start
info: middleware[0]
      M2 start
info: middleware[0]
      M2 end
info: middleware[0]
      M1 end

```

Via de `app.Use` methode is het mogelijk om rechtstreeks bepaalde functionaliteit toe te voegen. In voorgaand voorbeeld beperken we ons tot het loggen van een bericht zowel als de middleware start als wanneer de component wordt beëindigd. Om de controle door te geven naar de volgende middleware component gebruiken we het `await next()` statement.

Voeren we deze code uit samen met een GET request dan ziet de console log er als volgt uit :



```
C:\VisualStudioProjects\SolutionRESTcountryNET7\RESTAPIMiddleware\bin\Debug\net6.0\RESTAPIMiddleware.exe
use 2
info: RESTAPIMiddleware[0]
  run
info: Microsoft.Hosting.Lifetime[14]
  Now listening on: http://localhost:5117
info: Microsoft.Hosting.Lifetime[0]
  Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
  Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
  Content root path: C:\VisualStudioProjects\SolutionRESTcountryNET7\RESTAPIMiddleware\
info: middleware[0]
  M1 start
info: middleware[0]
  M2 start
info: middleware[0]
  M2 end
info: middleware[0]
  M1 end
info: middleware[0]
  M1 start
info: middleware[0]
  M2 start
info: CountryService.Controllers.CountryController[1001]
  GetAll called
info: middleware[0]
  M2 end
info: middleware[0]
  M1 end
```

De start logs worden getoond, daarna de boodschap dat de eerste middleware component start, vervolgens de start van de tweede middleware component. Het GET request wordt nu verwerkt en de logs die daarbij horen worden getoond, waarna de controle weer overgaat naar de tweede middleware component en vervolgens naar de eerste component.

### Eigen middleware - Logging voorbeeld

Als voorbeeld zullen we nu zelf een klasse implementeren die we als middleware zullen toevoegen aan de pijplijn. We noemen onze klasse `LogURLMiddleware` en deze heeft als doel om voor elk binnenkomend verzoek de URL uit het request te loggen. We hebben daarbij dus een `ILogger` nodig, die we via een `ILoggerFactory` zullen aanmaken. Daarnaast is er ook nog een variabele van het type `RequestDelegate` die er moet voor zorgen dat de controle kan worden doorgegeven naar de volgende component in de pijplijn. Naast de constructor hebben we ook een methode `Invoke` nodig die zal worden opgeroepen door de pijplijn. In deze methode plaatsen we onze logica en zorgen we er tevens

voor dat de volgende component wordt aangeroepen, dit laatste gebeurt door het statement `await _next(context)`.

```
public class LogURLMiddleware
{
    private readonly RequestDelegate _next;
    private readonly ILogger _logger;

    0 references
    public LogURLMiddleware(RequestDelegate next, ILoggerFactory loggerFactory)
    {
        _next = next;
        _logger = loggerFactory.CreateLogger<LogURLMiddleware>();
    }

    0 references
    public async Task Invoke(HttpContext context)
    {
        try
        {
            await _next(context);
        }
        finally
        {
            _logger.LogInformation(
                "Request {method} {url} => {statusCode}",
                context.Request?.Method,
                context.Request?.Path.Value,
                context.Response?.StatusCode);
        }
    }
}
```

De middleware component kan nu worden geregistreerd door middel van het volgende statement in de Configure methode :

```
app.UseMiddleware<LogURLMiddleware>();
```

Maar we kunnen ook nog een stapje verder gaan en in plaats van het gebruik van de generieke methode `UseMiddleware` met onze klasse als type, kunnen we ook een extension method maken zodat we een specifieke methode kunnen gebruiken om onze middleware te registreren. De extension method ziet er dan als volgt uit.

```

public static class LogURLMiddlewareExtension
{
    1 reference
    public static IApplicationBuilder UseLogURLMiddleware(this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<LogURLMiddleware>();
    }
}

```

En op deze manier kunnen we de middleware dan via de methode UseLogURLMiddleware toevoegen.

```
app.UseLogURLMiddleware();
```

We testen dit nog even uit en sturen een GET request na het opstarten van onze applicatie. Het resultaat is dan :

```

C:\VisualStudioProjects\SolutionRESTcountryNET7\RESTAPIMiddleware\bin\Debug\net6.0\RESTAPIMiddleware.exe
info: RESTAPIMiddleware[0]
      start app
info: RESTAPIMiddleware[0]
      run
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5117
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\VisualStudioProjects\SolutionRESTcountryNET7\RESTAPIMiddleware\
info: RESTAPIMiddleware.Middleware.LogURLMiddleware[0]
      Request GET /swagger/index.html => 200
info: RESTAPIMiddleware.Middleware.LogURLMiddleware[0]
      Request GET /swagger/v1/swagger.json => 200
info: CountryService.Controllers.CountryController[1001]
      GetAll called
info: RESTAPIMiddleware.Middleware.LogURLMiddleware[0]
      Request GET /api/Country => 200

```

De request URL wordt nu in de console gelogd na het uitvoeren van het GET request.

## Dependency injection

### Configuratie

ASP.NET Core beschikt over een ingebouwde DI container. Deze wordt zowel gebruikt om interne componenten te configureren als op maat gemaakte services. Om de verschillende

componenten/klassen at runtime te kunnen gebruiken is het noodzakelijk om deze te registreren. Dit kan in de Program klasse.

Om een service te registreren bij de DI container zijn er drie stukken informatie nodig :

- Het type van de service (welke interface of klasse wordt er gevraagd)
- Welke implementatie moet er gebruikt worden
- Hoe lang moet de instantie van de service worden gebruikt (lifetime)

Voorbeelden zijn :

```
services.AddSingleton<ICountryRepository, CountryRepository>();  
services.AddSingleton<DIDatabaseContext>();
```

In het eerste statement geven we aan dat het type een ICountryRepository moet zijn en dat de implementatie de klasse CountryRepository is. De lifetime (zie verder) is singleton. In het tweede statement is het type een effectieve klasse en stemt deze meteen ook overeen met de implementatie.

## Lifetimes

Wanneer aan de DI container wordt gevraagd om een bepaalde instantie van een klasse (service) te leveren, zijn er twee mogelijkheden :

- Er wordt een nieuwe instantie aangemaakt
- Een bestaande instantie wordt geleverd

De keuze hangt af van de 'lifetime' instelling van de service bij het registreren aan de DI container. Er zijn drie verschillende mogelijkheden :

- Transient – telkens deze service wordt gevraagd, wordt er een nieuwe instantie aangemaakt. Er kunnen dus verschillende instanties naast elkaar bestaan.
- Scoped – binnen een bepaalde 'scope' wordt telkens hetzelfde object aangeleverd. Binnen ASP.NET Core heeft elk web request zijn eigen scope.
- Singleton – ongeacht de scope wordt telkens hetzelfde object geleverd.

Voor elk van deze 'lifetimes' wordt een specifieke methode voorzien om de service met deze 'lifetime' te registreren (AddTransient, AddScoped en AddSingleton).

Het gebruik van Transient lifetimes heeft het meest zin wanneer het gaat om objecten die weinig middelen vragen en weinig of geen status info, aangezien we wellicht veel van deze objecten zullen aanmaken. Door de eigenschappen van een web request is de scoped lifetime een veelgemaakte keuze. Telkens we nood hebben aan een object dat binnen één verzoek moet worden gedeeld, maar tussen verschillende requests moet kunnen worden aangepast (we denken bijvoorbeeld aan het gebruik van databanken) kunnen we hiervan gebruik maken. Singletons worden vooral gebruikt wanneer de creatie van het object veel middelen vraagt of wanneer er geen status moet worden bijgehouden.



We zullen aan de hand van een voorbeeld de werking van de verschillende lifetimes demonstreren. Dit voorbeeld is gebaseerd op [ASP.NET Core in action, Andrew Lock, Manning].

We starten met het aanmaken van een `DIDatabaseContext` klasse. Met deze klasse simuleren we de werking van een databank, telkens we een nieuwe instantie van deze klasse aanmaken komt dit overeen met de nieuwe toestand van de databank. De toestand van de databank stellen we voor door de property `RowCount` die we bij aanmaak telkens een nieuwe waarde geven (hier nagebootst door een random waarde).

```
public class DIDatabaseContext
{
    static readonly Random _rand = new Random();
    3 references
    public int RowCount { get; }

    0 references
    public DIDatabaseContext()
    {
        RowCount = _rand.Next(1, 1000);
    }
}
```

We voorzien ook een klasse die een repository simuleert, deze klasse maakt gebruik van de `DIDatabaseContext` klasse die we via DI aan onze constructor meegeven. We voorzien hier ook een methode die eveneens de `RowCount` kan teruggeven.

```
public class DIRepository
{
    private readonly DIDatabaseContext _context;
    0 references
    public int RowCount => _context.RowCount;

    0 references
    public DIRepository(DIDatabaseContext context)
    {
        _context = context;
    }
}
```

Tenslotte hebben we nog een controller klasse die zowel een `DIDatabaseContext` als een `DIRepository` variabele bevat die via de constructor worden geïnitieerd. We voorzien een eenvoudige GET methode die de waarde van de `RowCount` van zowel de repository als van de databasecontext niet alleen weergeeft, maar ook logt.

```

[Route("api/[controller]")]
[ApiController]
2 references
public class DIRowCountController : ControllerBase
{
    private readonly DIDatabaseContext _context;
    private readonly DIRepository _repository;
    private readonly ILogger _logger;
    0 references
    public DIRowCountController(DIDatabaseContext context, DIRepository repository, ILogger<DIRowCountController> logger)
    {
        _context = context;
        _repository = repository;
        _logger = logger;
    }
    [HttpGet]
    0 references
    public IActionResult Get()
    {
        string msg = $"context : {_context.RowCount}, repo : {_repository.RowCount}";
        _logger.LogInformation(msg);
        return Ok(msg);
    }
}

```

De verschillende lifetime settings worden nu getest in drie verschillende scenario's.

### Scenario 1

In het eerste scenario registreren we zowel de DIDatabaseContext klasse als de DIRepository klasse als Transient bij de DI container. We doen dit door de volgende statements toe te voegen aan de Program klasse.

```

builder.Services.AddTransient<DIDatabaseContext>();
builder.Services.AddTransient<DIRepository>();

```

Een eenvoudig GET request via Swagger geeft ons dan :

Curl

```
curl -X 'GET' \
'http://localhost:5095/api/DIRowCount' \
-H 'accept: */*'

```

Request URL

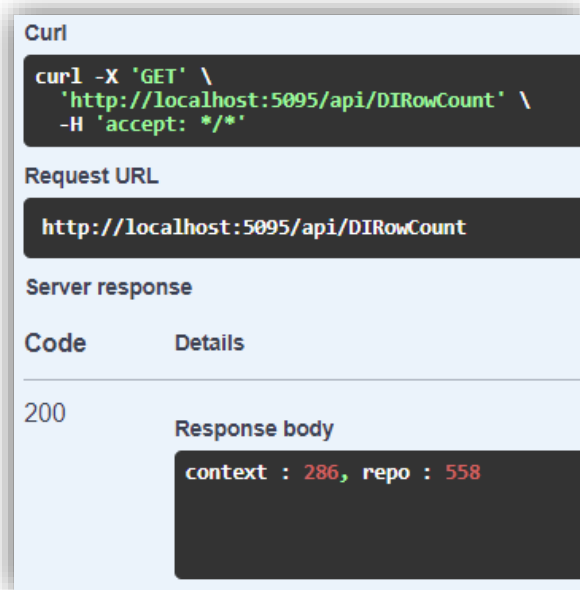
```
http://localhost:5095/api/DIRowCount

```

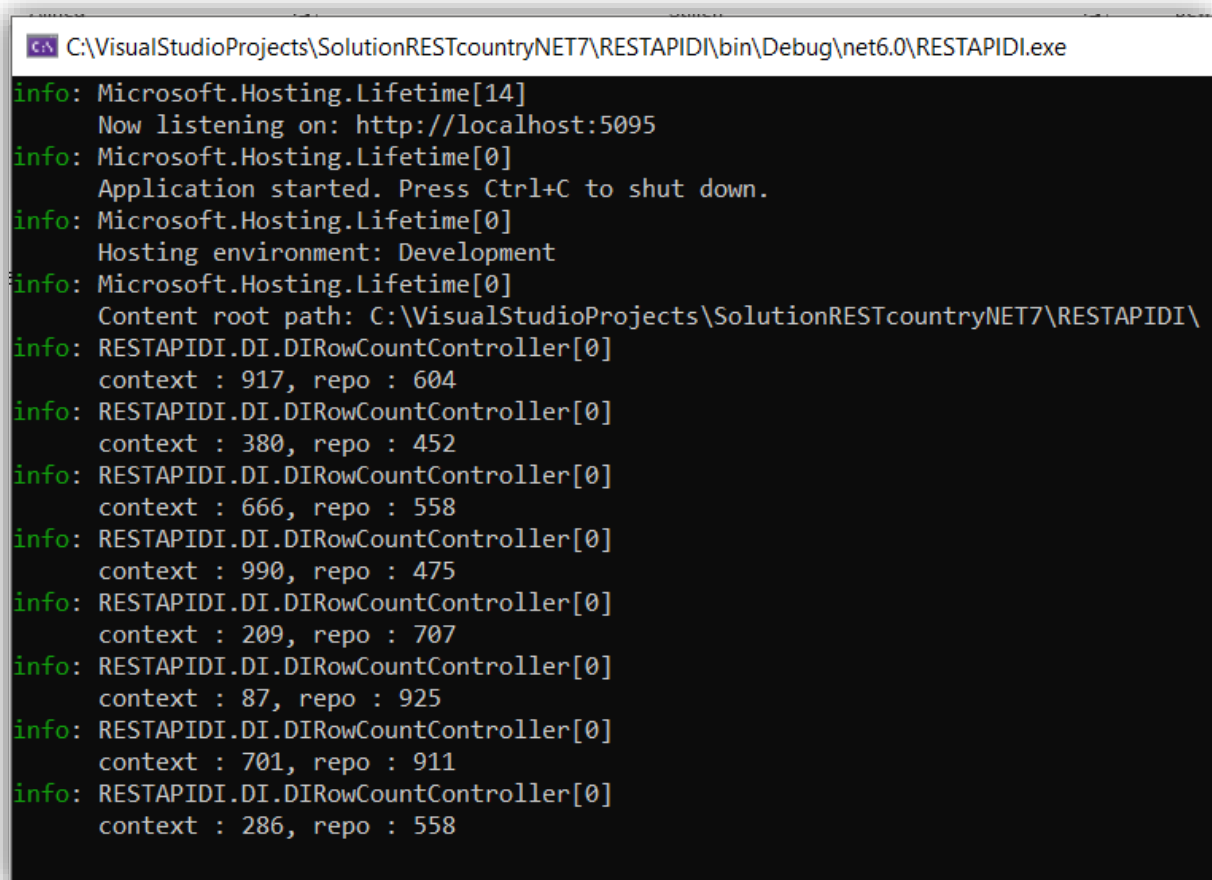
Server response

Code	Details
200	Response body <pre>context : 87, repo : 925 </pre>

Voeren we een tweede GET request uit bekomen we een ander resultaat.

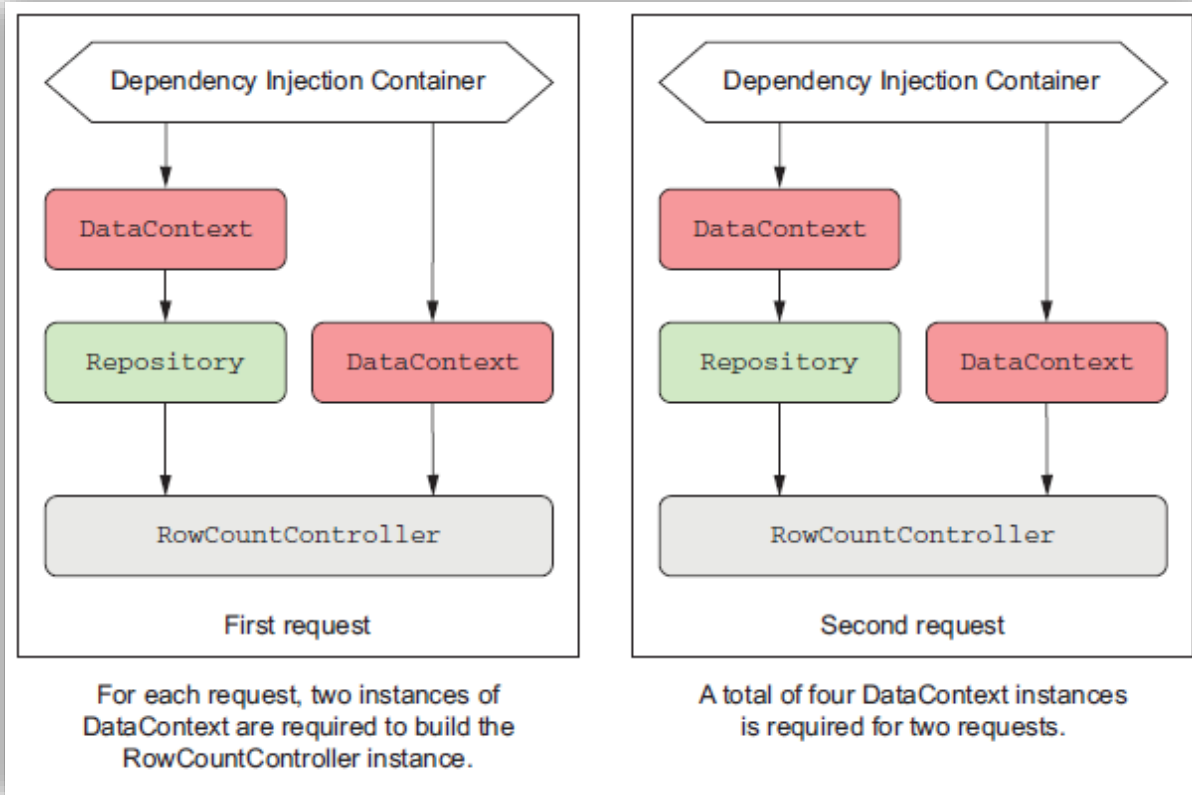


Samenvattend kunnen we de resultaten nog eens bekijken in de console log.



Het resultaat is dat we vier verschillende waarden krijgen voor de variabele RowCount. Hoe kunnen we dit verklaren ? Bekijken we de volgende figuur. Bij het eerste request worden er twee Context

instanties aangemaakt, één voor de repository en één voor de controller (door keuze transient). Bij het tweede request gebeurt net hetzelfde, waardoor we vier verschillende context instanties hebben en dus ook vier verschillende waarden voor de variabele RowCount.



(Figuur uit [ASP.NET Core in action, Andrew Lock, Manning])

## Scenario 2

In een tweede scenario geven we voor de Context een lifetime Scoped mee.

```
builder.Services.AddScoped<DIDatabaseContext>();  
builder.Services.AddTransient<DIRepository>();
```

Het resultaat voor het eerste GET request is :

Curl

```
curl -X 'GET' \
'http://localhost:5095/api/DIRowCount' \
-H 'accept: */*'
```

Request URL

```
http://localhost:5095/api/DIRowCount
```

Server response

Code	Details
200	<p>Response body</p> <pre>context : 538, repo : 538</pre>

Het tweede GET request geeft ons :

Curl

```
curl -X 'GET' \
'http://localhost:5095/api/DIRowCount' \
-H 'accept: */*'
```

Request URL

```
http://localhost:5095/api/DIRowCount
```

Server response

Code	Details
200	<p>Response body</p> <pre>context : 336, repo : 336</pre>

```
C:\VisualStudioProjects\SolutionRESTcountryNET7\RESTAPIDI\bin\Debug\net6.0\RESTAPIDI.exe
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5095
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\VisualStudioProjects\SolutionRESTcountryNET7\RESTAPIDI\
info: RESTAPIDI.DI.DIRowCountController[0]
      context : 538, repo : 538
info: RESTAPIDI.DI.DIRowCountController[0]
      context : 336, repo : 336
info: RESTAPIDI.DI.DIRowCountController[0]
      context : 294, repo : 294
info: RESTAPIDI.DI.DIRowCountController[0]
      context : 337, repo : 337
```

In tegenstelling tot scenario 1 zien we nu dat de RowCount variabelen binnen hetzelfde request zowel voor de context als de repository dezelfde waarde hebben. Aangezien we de DI container hebben meegegeven dat de context een lifetime 'scoped' heeft, wordt er binnen een request maar één instantie aangemaakt. Dit resulteert in een hergebruik en dus zijn de context in de repository en de context in de controller dezelfde.

### Scenario 3

In het derde scenario maken we de lifetime van de context van het type singleton.

```
builder.Services.AddSingleton<DIDatabaseContext>();
builder.Services.AddTransient<DIRepository>();
```

De resultaten zijn nu :

#### Curl

```
curl -X 'GET' \
'http://localhost:5095/api/DIRowCount' \
-H 'accept: */*'
```

#### Request URL

`http://localhost:5095/api/DIRowCount`

#### Server response

##### Code

##### Details

200

##### Response body

```
context : 592, repo : 592
```

##### Response headers

```
content-type: text/plain; charset=utf-8
date: Thu,02 Nov 2023 11:40:42 GMT
server: Kestrel
transfer-encoding: chunked
```

#### Curl

```
curl -X 'GET' \
'http://localhost:5095/api/DIRowCount' \
-H 'accept: */*'
```

#### Request URL

`http://localhost:5095/api/DIRowCount`

#### Server response

##### Code

##### Details

200

##### Response body

```
context : 592, repo : 592
```

##### Response headers

```
content-type: text/plain; charset=utf-8
date: Thu,02 Nov 2023 11:41:14 GMT
server: Kestrel
transfer-encoding: chunked
```

```
C:\VisualStudioProjects\SolutionRESTcountryNET7\RESTAPIDI\bin\Debug\net6.0\RESTAPIDI.exe
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5095
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\VisualStudioProjects\SolutionRESTcountryNET7\RESTAPIDI\
info: RESTAPIDI.DI.DIRowCountController[0]
      context : 592, repo : 592
info: RESTAPIDI.DI.DIRowCountController[0]
      context : 592, repo : 592
info: RESTAPIDI.DI.DIRowCountController[0]
      context : 592, repo : 592
```

Hier merken we op dat in alle gevallen de waarde van RowCount overal hetzelfde is. Dit stemt overeen met onze verwachtingen, er is namelijk maar één context instantie aangemaakt en deze wordt dus telkens opnieuw hergebruikt.

## Referenties

<https://www.youtube.com/watch?v=HCxAERjO4C4> [Middleware]

<https://www.youtube.com/watch?v=A1ZmMoiBELc> [Middleware]

<https://www.youtube.com/watch?v=6NAqDi0Gy5Y> [Middleware]

<https://www.youtube.com/watch?v=oXNslgIXIbQ> [Logging]

<https://www.c-sharpcorner.com/article/add-file-logging-to-an-asp-net-core-mvc-application/>

<https://docs.microsoft.com/en-us/aspnet/core/fundamentals/logging/?view=aspnetcore-3.1>

<https://blog.elmah.io/asp-net-core-request-logging-middleware/>

<https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-3.1>

<https://elanderson.net/2019/12/log-requests-and-responses-in-asp-net-core-3/>

<https://www.c-sharpcorner.com/article/overview-of-middleware-in-asp-net-core/>

ASP.NET Core in action, Andrew Lock, Manning