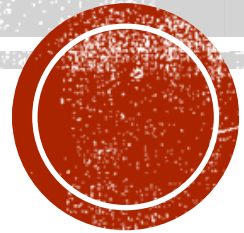
REGISTERING SERVICES WITH DEPENDENCY INJECTION

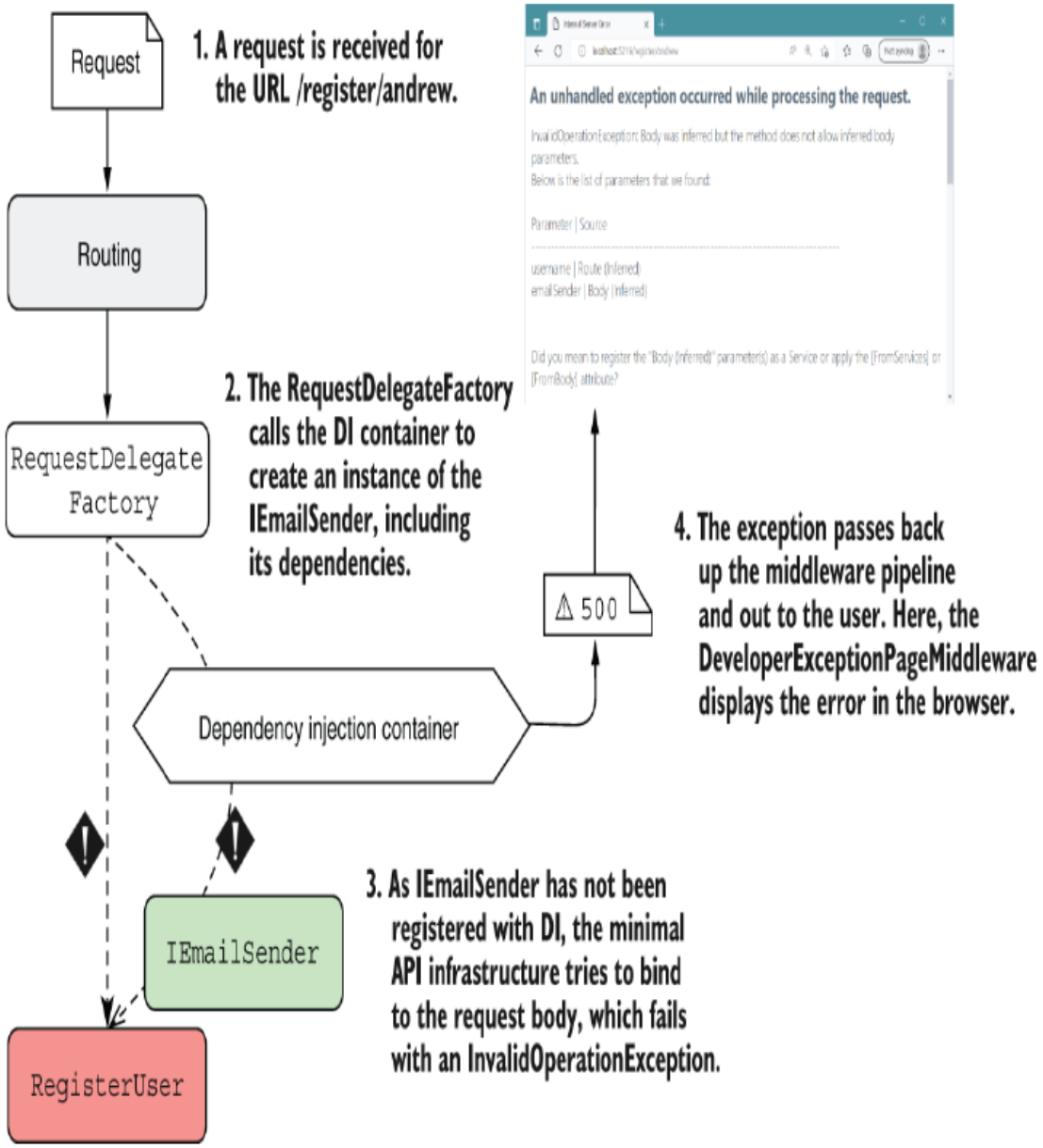


Kamal Beydoun
Lebanese University – Faculty of Sciences I
Kamal.beydoun@ul.edu.lb

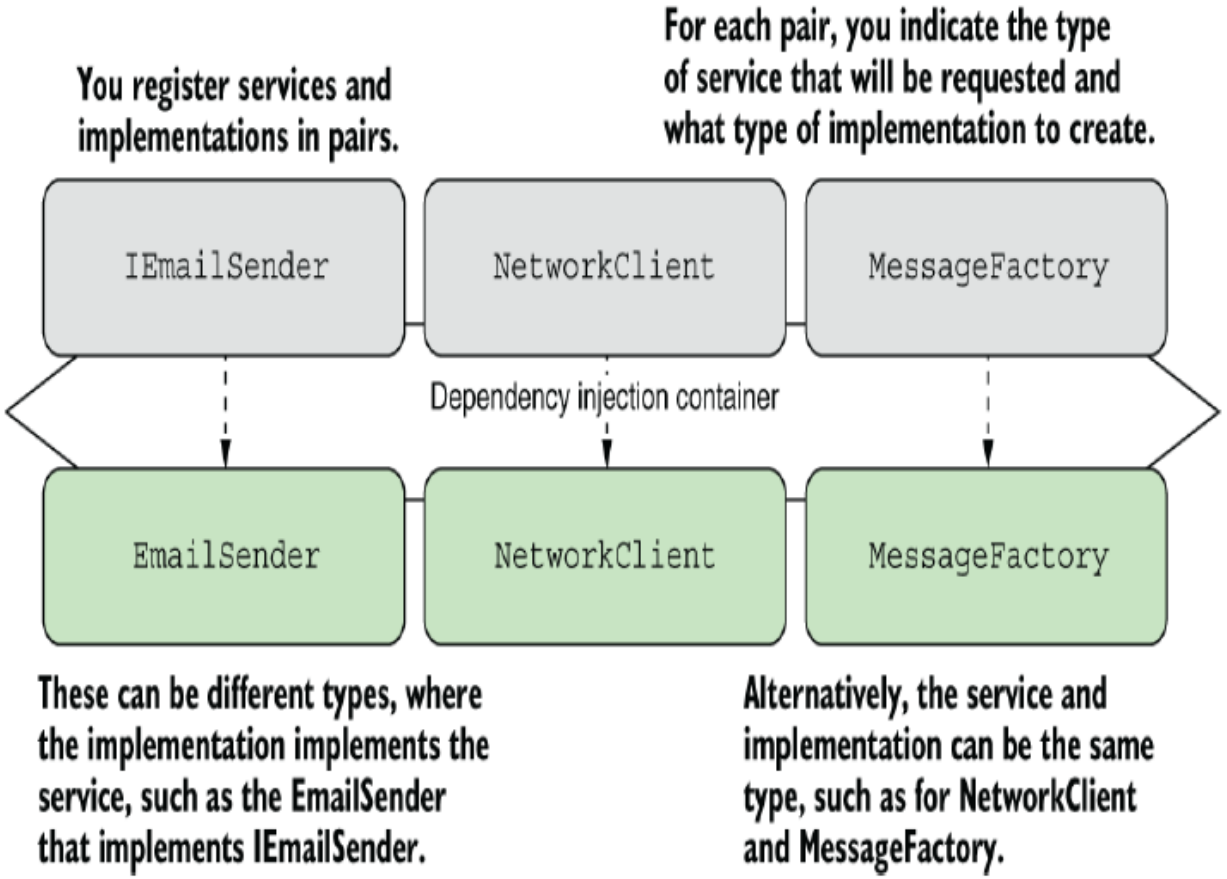
REGISTERING CUSTOM SERVICES WITH THE DI CONTAINER

```
app.MapGet("/register/{username}", RegisterUser);    ❶

app.Run();

string RegisterUser(string username)
{
    IEmailSender emailSender = new EmailSender(      ❷
        new MessageFactory(),                        ❸
        new NetworkClient(                           ❹
            new EmailServerSettings                  ❺
            (                                        ❺
                Host: "smtp.server.com",             ❺
                Port: 25                             ❺
            ))                                       ❺
        );
    emailSender.SendEmail(username);                 ❻
    return $"Email sent to {username}!";
}
```

```
app.MapGet("/register/{username}", RegisterUser);                ❶

app.Run();

string RegisterUser(string username, IEmailSender emailSender)   ❷
{
    emailSender.SendEmail(username);                             ❸
    return $"Email sent to {username}!";
}
```


Request
1. A request is received for the URL /register/andrew.
Routing
RequestDelegate Factory
2. The RequestDelegateFactory calls the DI container to create an instance of the IEmailSender, including its dependencies.
An unhandled exception occurred while processing the request.
InvalidOperationException: Body was inferred but the method does not allow inferred body parameters.
Below is the list of parameters that we found:
Parameter | Source
username | Route (Inferred)
emailSender | Body (Inferred)
Did you mean to register the "Body (Inferred)" parameter(s) as a Service or apply the [FromServices] or [FromBody] attribute?
500
4. The exception passes back up the middleware pipeline and out to the user. Here, the DeveloperExceptionPageMiddleware displays the error in the browser.
Dependency injection container
IEmailSender
3. As IEmailSender has not been registered with DI, the minimal API infrastructure tries to bind to the request body, which fails with an InvalidOperationException.
RegisterUser

REGISTERING CUSTOM SERVICES WITH THE DI CONTAINER



you need to register an IEmailSender implementation and all its dependencies with the DI container

REGISTERING CUSTOM SERVICES WITH THE DI CONTAINER

Configuring DI consists of making a series of statements made by calling various Add* methods on the IServiceCollection.

Each **Add*** method provides three pieces of information to the DI container:

- **Service type—TService**. This class or interface will be requested as a dependency. It's often an interface, such as IEmailSender, but sometimes a concrete type, such as NetworkClient or MessageFactory.
- **Implementation type—TService or TImplementation**. The container should create this class to fulfill the dependency. It must be a concrete type, such as EmailSender. It **may** be the same as the service type, as for NetworkClient and MessageFactory.
- **Lifetime—transient, singleton, or scoped**. The lifetime defines how long an instance of the service should be used by the DI container.

REGISTERING CUSTOM SERVICES WITH THE DI CONTAINER

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

builder.Services.AddScoped<IEmailSender, EmailSender>();    ❶
builder.Services.AddScoped<NetworkClient>();                ❷
builder.Services.AddSingleton<MessageFactory>();            ❸

WebApplication app = builder.Build();

app.MapGet("/register/{username}", RegisterUser);

app.Run();

string RegisterUser(string username, IEmailSender emailSender)
{
    emailSender.SendEmail(username);
    return $"Email sent to {username}!";
}
```

REGISTERING SERVICES USING OBJECTS AND LAMBDAS

- The preceding Add* methods use generics to specify the Type of the class to register, but they don't give any indication of how to construct an instance of that type. Instead, the container makes several assumptions that you have to adhere to:
  - The class must be a **concrete type**.
  - The class must have only a **single relevant constructor** that the container can use.
  - For a constructor to be relevant, all constructor **arguments** must **be registered with the container** or must be arguments with a **default value**.

REGISTERING SERVICES USING OBJECTS AND LAMBDAS

- The EmailServerSettings record doesn't meet these requirements, as it requires you to provide a Host and Port in the constructor, which are a string and int, respectively, without default values:

```
public record EmailServerSettings(string Host, int Port);
```

REGISTERING SERVICES USING OBJECTS AND LAMBDAS

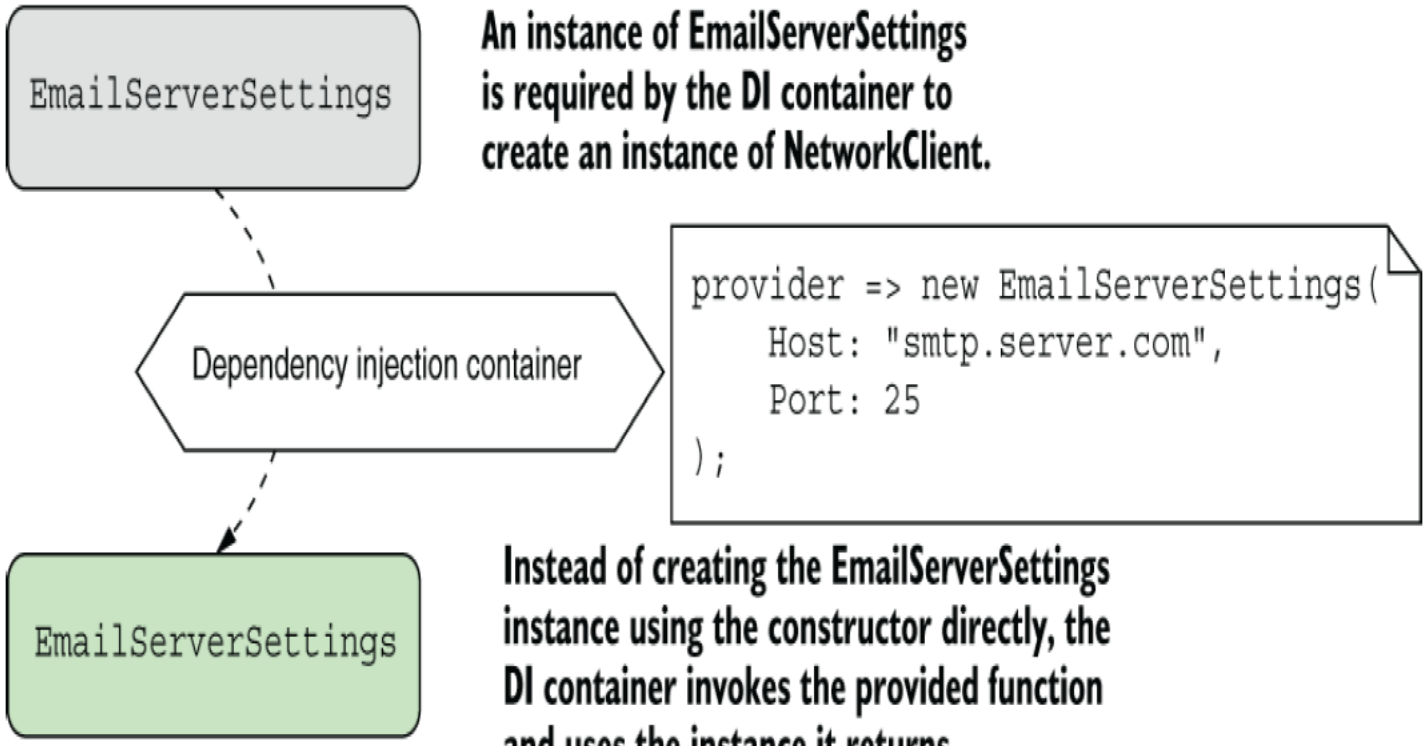
```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

builder.Services.AddScoped<IEmailSender, EmailSender>();
builder.Services.AddScoped<NetworkClient>();
builder.Services.AddSingleton<MessageFactory>();
builder.Services.AddSingleton(
        new EmailServerSettings          ❶
        (                                ❶
            Host: "smtp.server.com",     ❶
            Port: 25                     ❶
        ));                              ❶

WebApplication app = builder.Build();
```

```
app.MapGet("/register/{username}", RegisterUser);

app.Run();
```

❶ This instance of EmailServerSettings will be used whenever an instance is required.

This code works fine if you want to have only a single instance of EmailServerSettings in your application;
The same object will be shared everywhere. But what if you want to create a new object each time one is requested?

REGISTERING SERVICES USING OBJECTS AND LAMBDAS



REGISTERING SERVICES USING OBJECTS AND LAMBDAS

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

builder.Services.AddScoped<IEmailSender, EmailSender>();
builder.Services.AddScoped<NetworkClient>();
builder.Services.AddSingleton<MessageFactory>();
builder.Services.AddScoped(                    ❶
        provider =>                            ❷
            new EmailServerSettings            ❸
            (                                  ❸
                Host: "smtp.server.com",       ❸
                Port: 25                       ❸
            ));                                ❸


WebApplication app = builder.Build();

app.MapGet("/register/{username}", RegisterUser);

app.Run();
```

EXTENSION METHOD

```
public static class EmailSenderServiceCollectionExtensions
{
    public static IServiceCollection AddEmailSender(
        this IServiceCollection services)                    ❶
    {
        services.AddScoped<IEmailSender, EmailSender>();     ❷
        services.AddSingleton<NetworkClient>();              ❷
        services.AddScoped<MessageFactory>();                ❷
        services.AddSingleton(                               ❷
            new EmailServerSettings                          ❷
            (                                                ❷
                host: "smtp.server.com",                     ❷
                port: 25                                     ❷
            ));                                              ❷
        return services;                                     ❸
    }
}
```

❶ Creates an extension method on IServiceCollection by using the “this” keyword

❷ Cuts and pastes your registration code from Program.cs

❸ By convention, returns the IServiceCollection to allow method chaining

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

builder.Services.AddEmailSender();    ❶

WebApplication app = builder.Build();

app.MapGet("/register/{username}", RegisterUser);

app.Run();
```

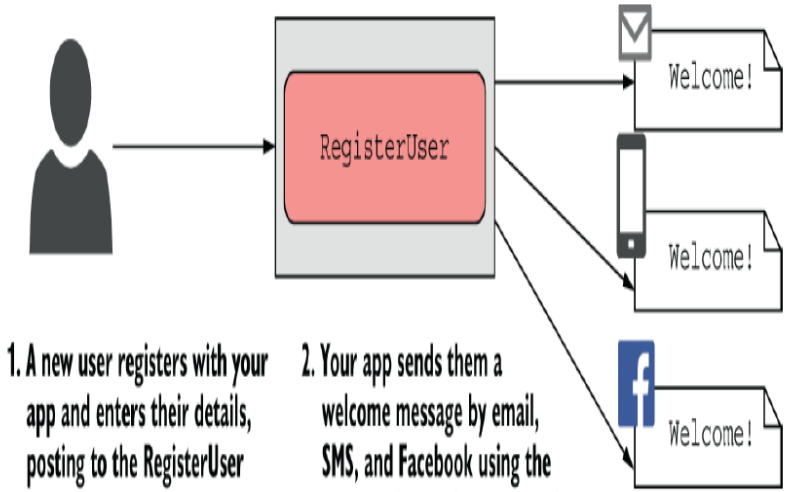
❶ The extension method registers all the services associated with the EmailSender.

REGISTERING A SERVICE IN THE CONTAINER MULTIPLE TIMES

- One advantage of coding to interfaces is that **you can create multiple implementations** of a service.
- IEmailSender so that you can send messages via Short Message Service (SMS) or Facebook, as well as by email.

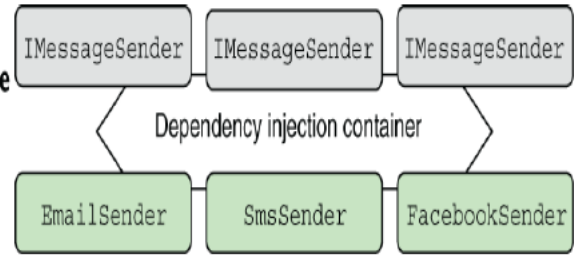
```
public interface IMessageSender
{
    public void SendMessage(string message);
}
```

INJECTING MULTIPLE IMPLEMENTATIONS OF AN INTERFACE

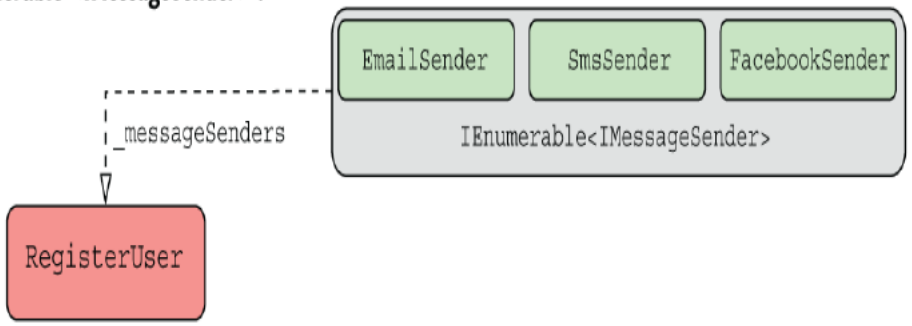


1. A new user registers with your app and enters their details, posting to the RegisterUser handler.
2. Your app sends them a welcome message by email, SMS, and Facebook using the IMessageSender implementations.

1. Multiple implementations of IMessageSender are registered with the DI container using the normal Add* methods.



2. The DI container creates one of each IMessageSender implementation and injects them into the RegisterUser as an Enumerable<IMessageSender>.




```
foreach (var messageSender in _messageSenders)
{
  messageSender.SendMessage(username);
}
```

3. The RegisterUser handler loops over the IMessageSender instances and calls SendMessage on each.

INJECTING MULTIPLE IMPLEMENTATIONS OF AN INTERFACE

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Services.AddScoped<IMessageSender, EmailSender>();
builder.Services.AddScoped<IMessageSender, SmsSender>();
builder.Services.AddScoped<IMessageSender, FacebookSender>();
```

```
string RegisterUser(
    string username,
    IEnumerable<IMessageSender> senders)            ❶
{
    foreach(var sender in senders)                  ❷
    {                                               ❷
        Sender.SendMessage($"Hello {username}!");   ❷
    }                                               ❷

    return $"Welcome message sent to {username}";
}
```

❶ Requests an IEnumerable injects an array of IMessageSender

❷ Each IMessageSender in the IEnumerable is a different implementation.

INJECTING A SINGLE IMPLEMENTATION WHEN MULTIPLE SERVICES ARE REGISTERED

```
public class SingleMessageSender
{
    private readonly IMessageSender _messageSender;
    public SingleMessageSender(IMessageSender messageSender)
    {
        _messageSender = messageSender;
    }
}
```

The DI container will use the **last registered** implementation of a service when resolving a single instance of the service.

CONDITIONALLY REGISTERING SERVICES USING TRYADD

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

builder.Services.AddScoped<IMessageSender, EmailSender>();      ❶
builder.Services.TryAddScoped<IMessageSender, SmsSender>();     ❷
```

❶ EmailSender is registered with the container.

❷ There's already an IMessageSender implementation, so SmsSender isn't registered.

UNDERSTANDING LIFETIMES: WHEN ARE SERVICES CREATED?

- Whenever the DI container is asked for a particular registered service, such as an instance of IMessageSender, it can do either of two things to fulfill the request:
  - Create and return a **new instance** of the service.
  - Return an **existing instance** of the service.
- The **lifetime** of a service controls the behavior of the DI container with respect to these two options.

UNDERSTANDING LIFETIMES: WHEN ARE SERVICES CREATED?

- In ASP.NET Core, you can specify one of three lifetimes when **registering a service with the built-in container**:
- **Transient**—Every time a service is requested, a new instance is created. Potentially, you can have different instances of the same class within the same dependency graph.
- **Scoped**—Within a scope, all requests for a service give you the same object. For different scopes, you get different objects. In ASP.NET Core, each web request gets its own scope.
- **Singleton**—You always get the same instance of the service, regardless of scope.

EXAMPLE

```
class DataContext
{
    public int RowCount { get; }                ❶
        = Random.Shared.Next(1, 1_000_000_000); ❷
}
```

❶ The property is read-only, so it always returns the same value.

❷ Generates a random number between 1 and 1,000,000,000

```
public class Repository
{
    private readonly DataContext _dataContext;      ❶
    public Repository(DataContext dataContext)      ❶
    {                                               ❶
        _dataContext = dataContext;                 ❶
    }                                               ❶
    public int RowCount => _dataContext.RowCount;   ❷
}
```

❶ An instance of DataContext is provided using DI.

❷ RowCount returns the same value as the current instance of DataContext.

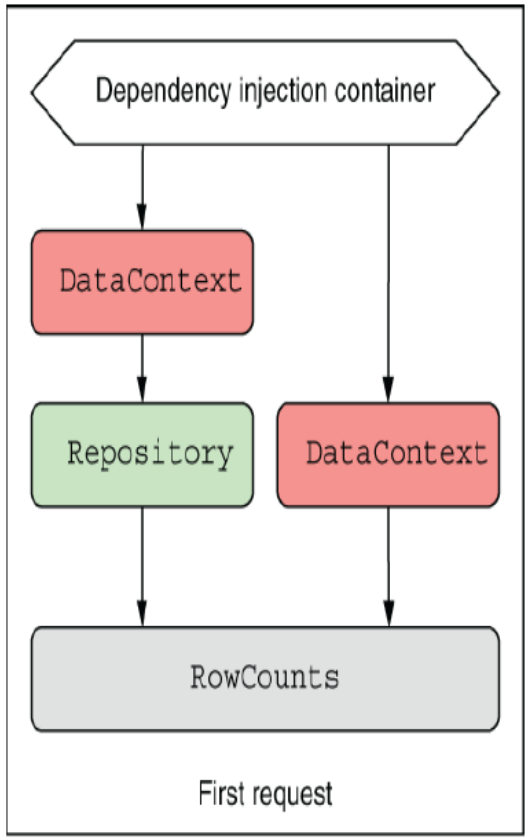
```
static string RowCounts(                                                    ❶
    DataContext db,                                                         ❶
    Repository repository)                                                  ❶
{
    int dbCount = db.RowCount;                                              ❷
    int repositoryCount = repository.RowCount;                              ❷

    return: $"DataContext: {dbCount}, Repository: {repositoryCount}";       ❸
}
```

❶ DataContext and Repository are created using DI.

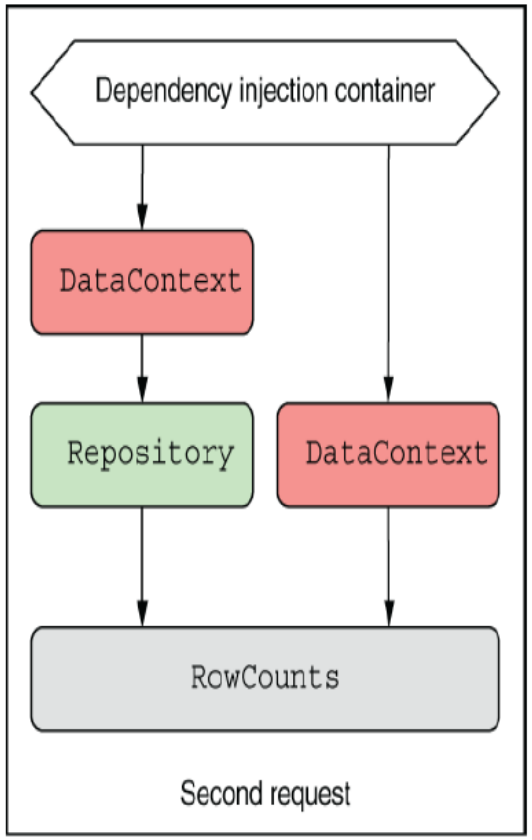
❷ When invoked, the page handler retrieves and records RowCount from both dependencies.

❸ The counts are returned in the response.

EXAMPLE



For each request, two instances of DataContext are required to call the RowCounts handler.



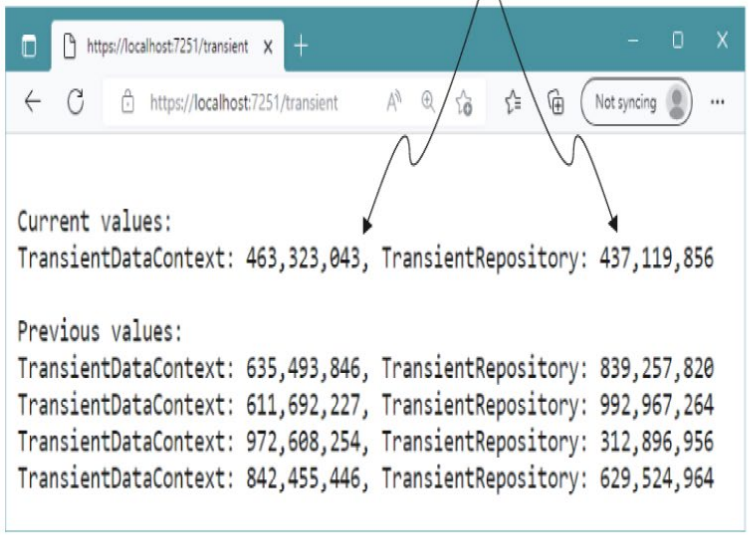
A total of four DataContext instances are required for two requests.

TRANSIENT: EVERYONE IS UNIQUE

- When you register services this way, every time a dependency is required, the container creates a new one.

```
builder.Services.AddTransient<DataContext>();
builder.Services.AddTransient<Repository>();
```

The RowCount is different within a single request, indicating that two different DataContexts are created.



The RowCount changes with each request, indicating that new DataContexts are created for every request.

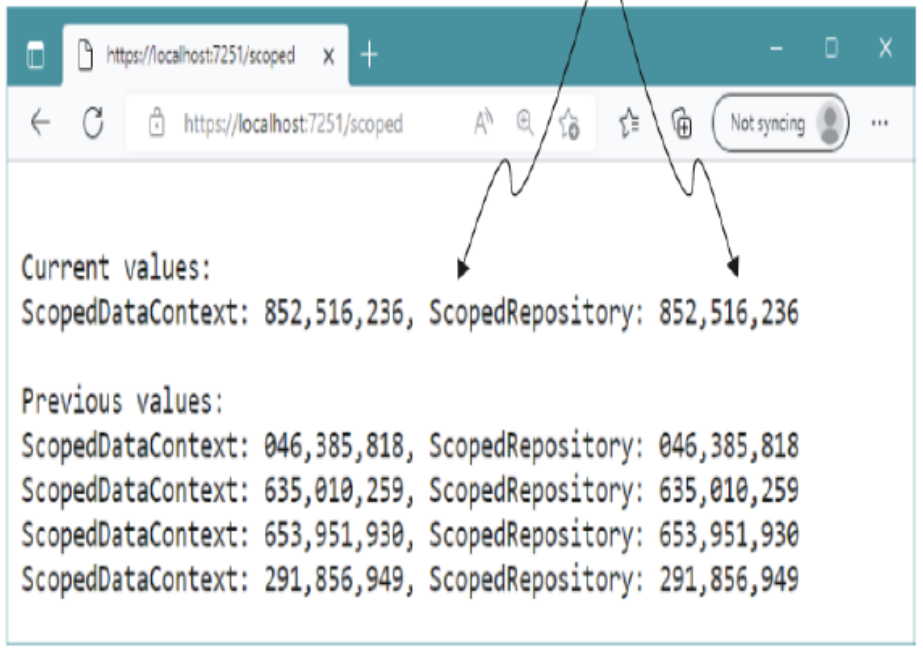
SCOPED: LET'S STICK TOGETHER

- The scoped lifetime states that a **single instance of an object will be used within a given scope, but a different instance will be used between different scopes**.
- You can register dependencies as scoped by using the AddScoped extension methods.

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

SCOPED: LET'S STICK TOGETHER

The RowCount is the same within a single request, indicating that a single DataContext is used in the dependency graph.



The RowCount changes with each request, indicating that a new DataContext is created for every request.

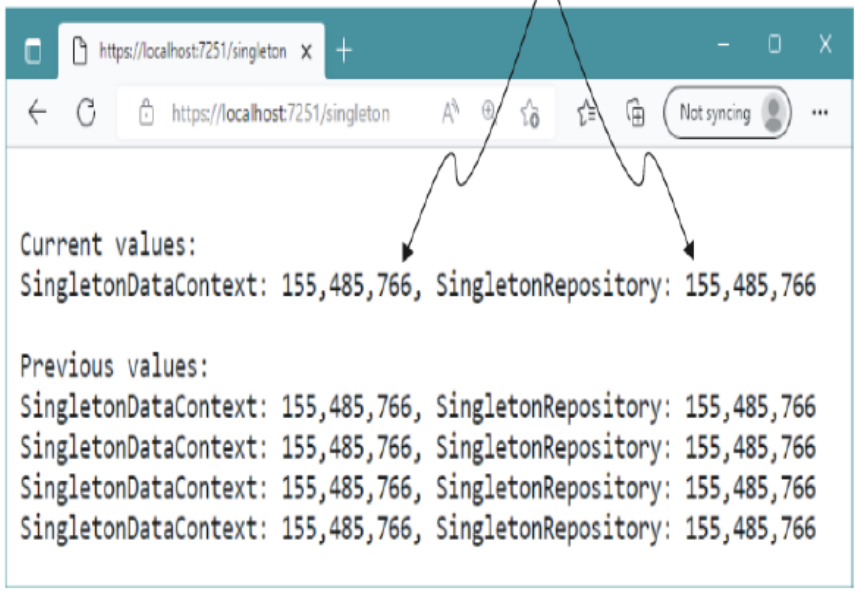
SINGLETON: THERE CAN BE ONLY ONE

- The singleton is conceptually simple: an instance of the service is created when it's first needed.
- You'll always get the same instance injected into your services.
- Singleton services must be thread-safe in a web application, as they'll typically be used by multiple threads during concurrent requests.

```
builder.Services.AddSingleton<DataContext>();
```

SINGLETON: THERE CAN BE ONLY ONE

The RowCount is the same within a single request, indicating that a single DataContext is used in the dependency graph.



The RowCount is the same for all requests, indicating that the same DataContext is used for every request.

KEEPING AN EYE OUT FOR CAPTIVE DEPENDENCIES

Suppose that you're configuring the lifetime for the DataContext and Repository examples.

- DataContext—**Scoped**, as it should be shared for a single request
- Repository—**Singleton**, as it has no state of its own and is thread-safe, so why not?