

## Exercise14: DI and Middleware

### Objective

In this exercise you get to do Dependency Injection and Configuration code. You will see the effect of the different DI scopes.

This exercise will take around 45 minutes.

### Dependency Injection and Scope

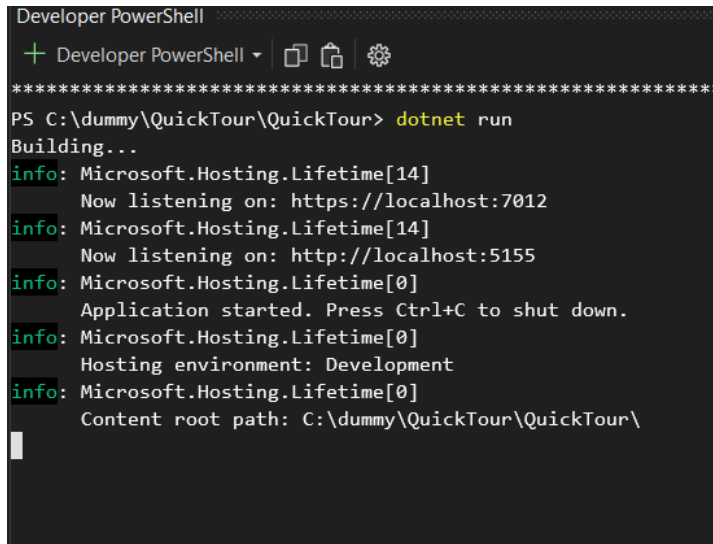
1	<p>The starter application is the “Quick Tour” that we did previously, with just a bit of extra stuff added (commented out for now, but we will un-comment it later).</p> <p>Open the solution in the ‘{installedFolder}\Labs\14_DlandConfig\Begin\QuickTour’ folder and compile the application. Check that it runs</p>
2	<p>At the moment, the ForumController is making its own MockForumContext object. We should always be wary of a class creating service classes itself. Let’s change it to use constructor injection, using the built-in dependency injection framework.</p> <p>Dependency injection works better if it’s based on interfaces, so add the following interface to the Models folder:</p> <pre>Public interface IForumContext {     Public IEnumerable&lt;Forum&gt; GetForums(); }</pre> <p>And modify the mock forum context to implement this interface:</p> <pre>Public class MockForumContext: IForumContext {     ... }</pre>
	<p>In the ForumController, remove the lines highlighted below.</p> <pre>public IActionResult Index() {     MockForumContext mockForum = new MockForumContext();     IEnumerable&lt;Forum&gt; forums = mockForum.GetForums(); }</pre> <p>Instead, use constructor injection by adding a constructor and the highlighted lines to the top of the ForumController class:</p> <pre>Private readonly IForumContext _context;  public ForumController(IForumContext context) {     _context = context; }</pre>

	<div data-bbox="308 197 1528 271" data-label="Text"> <pre>}</pre> </div> <p>And modify the Index action to use the injected database context:</p> <div data-bbox="308 344 1528 497" data-label="Text"> <pre>public IActionResult Index() {     IEnumerable&lt;Forum&gt; forums = _context.GetForums();     ... }</pre> </div>
	<p>In the Program.cs file, add the following line of code ABOVE the line of code that builds the app (<code>var app = builder.Build();</code>). This is where we identify all the services which can be provided by dependency injection and map the interface that the class requires to the concrete class that we are going to supply.</p> <div data-bbox="308 689 1528 875" data-label="Text"> <pre>... builder.Services.AddScoped&lt;IForumContext, MockForumContext&gt;(); var app = builder.Build();</pre> </div> <p>Run the application and check that it still works. Using dependency injection is as easy as that!</p>
	<div data-bbox="308 1137 1528 1223" data-label="Text"> <pre></pre> </div>
2	<p>In the next part of the lab, you are going to investigate the different lifetimes that are available to us when we use dependency injection.</p> <p>Drag the file Dependencies.cs from the Assets folder (in Explorer) onto the Models folder in your project. The contents are 3 classes following this pattern:</p> <div data-bbox="323 1615 866 1787" data-label="Text"> <pre>public interface ITransient {     void WriteGuidToConsole(); } public class TransientDependency { ...     .. }</pre> </div> <p>Which we will inject with the corresponding scope.</p> <p>Have a look at the TransientDependency. The logger is provide out-of-the-box by asp.net Core and is supplied itself via DI (we don't need to do anything ourselves other than state we want to use it in our constructor).</p>

	<p>You will see it writes out to the logger.</p> <ul style="list-style-type: none"> <li>a) Each time a new instance is instantiated</li> <li>b) Whenever the method WriteGuidToConsole() is called, it will show the unique Guid and the thread on which it is running</li> </ul>
3	<p>Again in the Program.cs file on the line where we just configured our MockForumContext (but ABOVE where we call the Build() method) register the class TransientDependency as the desired implementation of the interface ITransient and give it Transient scope.</p> <p>Do the same for Scoped and Singleton – like this:</p> <pre> services.AddTransient&lt;ITransient, TransientDependency&gt;(); services.AddScoped&lt;IScoped, ScopedDependency&gt;(); services.AddSingleton&lt;ISingleton, SingletonDependency&gt;(); </pre>
	<p>Modify the ForumController to require these dependencies, and also to add a bit more debug information:</p> <pre> Private readonly ITransient _tran; Private readonly IScoped _scoped; Private readonly ISingleton _single;  public ForumController(ILogger&lt;ForumController&gt; logger,                       IForumContext context,                       ITransient tran,                       IScoped scoped,                       ISingleton single) {     _logger = logger;     _context = context;     _tran = tran;     _scoped = scoped;     _single = single; }  public IActionResult Index() {     _logger.LogInformation("In the Forums Index() method &lt;=====");      _tran.WriteGuidToConsole();     _scoped.WriteGuidToConsole();     _single.WriteGuidToConsole();      _logger.LogDebug("About to get the data");      IEnumerable&lt;Forum&gt; forums = _context.GetForums();      _logger.LogDebug(\$"Number of forums: {forums.Count()}");     Return View(ForumViewModel.FromForums(forums)); } </pre>
	<p>Run the application (F5).</p> <p>As well as the browser appearing you should see a command window if you look in the Windows task bar. This is where the messages will be logged to.</p>

If you don't see a command window you can try to run the application using the Dotnet CLI in a Terminal Window

To do this: right-click on the project, select "Open in Terminal". In the Developer Powershell window that opens up, once the prompt is ready, type "dotnet run".

A screenshot of a Windows Developer PowerShell terminal window. The title bar reads "Developer PowerShell". Below the title bar is a toolbar with a plus icon, a dropdown menu showing "Developer PowerShell", and icons for copy, paste, and settings. The terminal content shows a command prompt at "PS C:\dummy\QuickTour\QuickTour>" where the command "dotnet run" has been entered. The output shows the application building and then starting, with logs indicating it is listening on https://localhost:7012 and http://localhost:5155. The logs also mention the hosting environment is "Development" and the content root path is "C:\dummy\QuickTour\QuickTour\".

```
Developer PowerShell
+ Developer PowerShell
*****
PS C:\dummy\QuickTour\QuickTour> dotnet run
Building...
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:7012
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5155
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:\dummy\QuickTour\QuickTour\
```

Open a web browser, go to <https://localhost:7012> (your port number may differ – see number in output above) and once the page has been displayed, press the refresh button to display it a second time.

The output is shown below with annotations which explain what has happened:

	<div data-bbox="292 194 598 1169"> <div data-bbox="292 194 542 365">Create the three dependencies</div> <div data-bbox="292 405 542 575">Each has their own unique GUID</div> <div data-bbox="327 638 539 712">Refresh page</div> <div data-bbox="292 730 542 900">New transient &amp; scoped – but not singleton</div> <div data-bbox="292 918 542 1169">Singleton keeps same GUID, despite different thread</div> </div> <div data-bbox="598 194 1549 1169">  <pre> C:\Windows\System32\cmd.exe - dotnet run info: QuickTour.Models.ITransient[0] transient constructed info: QuickTour.Models.IScoped[0] scoped constructed info: QuickTour.Models.ISingleton[0] singleton constructed info: QuickTour.Controllers.ForumController[0] In the Forums Index() method &lt;===== info: QuickTour.Models.ITransient[0] Transient : 4ead92bb-e4dc-4512-9bc8-1d2a5df4c403, thread=8 info: QuickTour.Models.IScoped[0] Scoped : 36a359dd-4b98-44b3-a587-37e2d227a0c8, thread=8 info: QuickTour.Models.ISingleton[0] Singleton : d512fd0e-7e56-4d36-b3a6-510d785ef30d, thread=8 dbug: QuickTour.Controllers.ForumController[0] About to get the data dbug: QuickTour.Controllers.ForumController[0] Number of forums: 2 ===== info: QuickTour.Models.ITransient[0] transient constructed info: QuickTour.Models.IScoped[0] scoped constructed info: QuickTour.Controllers.ForumController[0] In the Forums Index() method &lt;===== info: QuickTour.Models.ITransient[0] Transient : 97084832-349a-4ab8-84f0-2c8d009473b2, thread=9 info: QuickTour.Models.IScoped[0] Scoped : 2799c3cc-b301-45f6-88e7-a948630b56f1, thread=9 info: QuickTour.Models.ISingleton[0] Singleton : d512fd0e-7e56-4d36-b3a6-510d785ef30d, thread=9 dbug: QuickTour.Controllers.ForumController[0] About to get the data dbug: QuickTour.Controllers.ForumController[0] </pre> </div>
	<p>Take a closer look at the three dependency classes. Notice that each of them requires a constructor parameter – a logger. Where did the logger come from?</p> <p>When dependency injection is used to create the dependency object (or, indeed, any object), it will check whether that new object has any of its own dependencies, and will create those too! Dependency injection is used to create (for example) a ScopedDependency object, and as part of that process it also creates an ILogger&lt;IScoped&gt; that the ScopedDependency needs!</p> <p>This enables a complex series of dependencies to be built up very simply. As you write each class, you need to know what that class depends on, but you don't need to worry about any dependencies any deeper into the chain, because the dependency injection framework takes care of that for you.</p>
	<p>In the screenshot, above, the scoped dependency and the transient dependency appear to behave the same way – we get a new instance of the dependency each time we refresh the page.</p> <p>Let's modify our demonstration to show where these two types of lifecycle differ from each other.</p> <p>Add an instance of each dependency to the MockForumContext class, and use constructor injection to get instances of those dependencies. Then, call the WriteGuidToConsole() method on each dependency when creating the data:</p>

```

Public class MockForumContext : IForumContext
{
    Private readonly ITransient _tran;
    Private readonly IScoped _scoped;
    Private readonly ISingleton _single;

    Public MockForumContext(ITransient tran,
        IScoped scoped,
        ISingleton single)
    {
        _tran = tran;
        _scoped = scoped;
        _single = single;
    }

    Public IEnumerable<Forum>GetForums()
    {
        _tran.WriteGuidToConsole();
        _scoped.WriteGuidToConsole();
        _single.WriteGuidToConsole();
        ...
    }
}

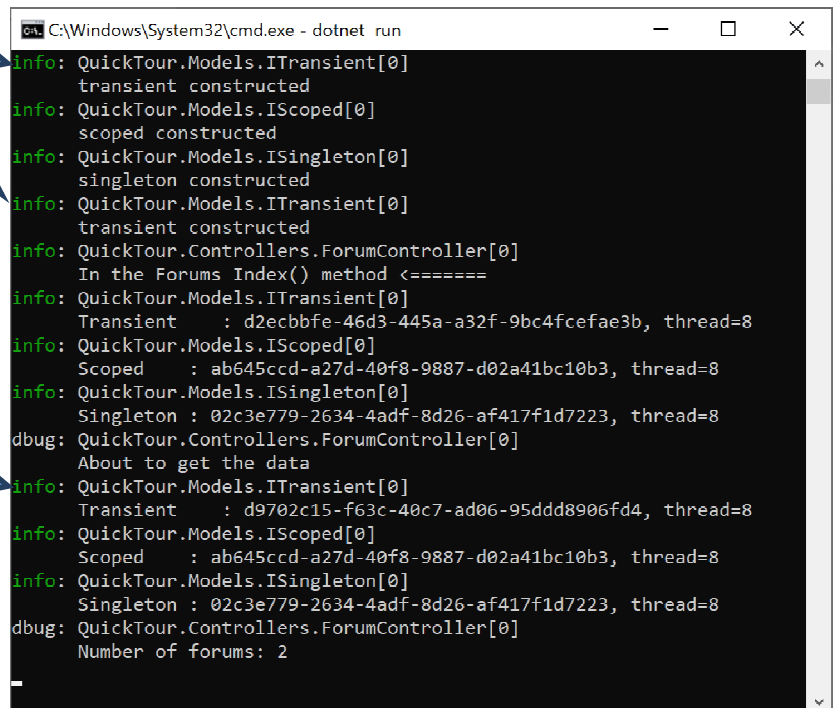
```

Since we already use dependency injection to create the MockForumContext, and the dependencies we need are already registered, we don't need to do anything else

Run the application (F5). The console output now looks like this:

Transient constructor called twice – once for controller, once for context

Scope dependency lasts for whole request, keeps same GUID.



```

C:\Windows\System32\cmd.exe - dotnet run
info: QuickTour.Models.ITransient[0]
      transient constructed
info: QuickTour.Models.IScoped[0]
      scoped constructed
info: QuickTour.Models.ISingleton[0]
      singleton constructed
info: QuickTour.Models.ITransient[0]
      transient constructed
info: QuickTour.Controllers.ForumController[0]
      In the Forums Index() method <=====
info: QuickTour.Models.ITransient[0]
      Transient      : d2ecbbfe-46d3-445a-a32f-9bc4fcefae3b, thread=8
info: QuickTour.Models.IScoped[0]
      Scoped       : ab645ccd-a27d-40f8-9887-d02a41bc10b3, thread=8
info: QuickTour.Models.ISingleton[0]
      Singleton    : 02c3e779-2634-4adf-8d26-af417f1d7223, thread=8
dbug: QuickTour.Controllers.ForumController[0]
      About to get the data
info: QuickTour.Models.ITransient[0]
      Transient      : d9702c15-f63c-40c7-ad06-95ddd8906fd4, thread=8
info: QuickTour.Models.IScoped[0]
      Scoped       : ab645ccd-a27d-40f8-9887-d02a41bc10b3, thread=8
info: QuickTour.Models.ISingleton[0]
      Singleton    : 02c3e779-2634-4adf-8d26-af417f1d7223, thread=8
dbug: QuickTour.Controllers.ForumController[0]
      Number of forums: 2

```

Here, you can see that the scoped dependency is shared between the two classes that use it, whereas the transient dependency is not (and therefore the dependency injection framework needs to create two separate instances of the transient dependency.)


## Configuring the Pipeline

4	<p>Add some trivial middleware into the pipeline – drag the folder Middleware from the Assets folder onto the project. Have a look at the code in the 2 classes in here.</p> <p>Inject the middleware into the pipeline by adding it in the Program.cs file this time AFTER the call to the Build() method. This is where middleware is plugged in.</p> <pre>app.UseAuthentication(); app.UseAuthorization();  app.UseMiddleware&lt;CustomMiddleware1&gt;(); app.UseMiddleware&lt;CustomMiddleware2&gt;();</pre>
5	<p>Suppress most of the trace information – in appsettings.Development.json</p> <pre>"Logging": {   "LogLevel": {     "Default": "None",     "Microsoft": "None",     "QuickTour": "None",     "QuickTour.Controllers": "Information",     "QuickTour.Middleware": "Debug"   } }</pre>
6	<p>Run the application.</p> <p>Refresh the browser a couple of times. Again, we've used LogWarning() to make the middleware messages stand out. Note that the middleware objects are created once at application startup, and then invoked for every web request.</p> <p>Adding middleware is the process by which a standard MVC application is configured (for example, adding authentication and authorisation middleware to the pipeline). It's definitely worth understanding what we mean when we talk about middleware. Writing your own middleware is not something you will need to do too often. Each middleware component can check details of the request, and either return a response to the web browser, or pass the request on to the next middleware component, modifying either the request or the response as appropriate. Our very basic example simply passes the request along to the next component without altering it in any way.</p>