

## **Exercise 4 : DI and Config**

### **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

	<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). Open the ‘Begin’ folder and compile the application. Check that it runs.</p>
	<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. 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 line highlighted below. (Remember, we’ve moved the controller into the Areas/Users/Controllers folder!):</p> <pre>public IActionResult Index() {     _logger.LogInformation("In the Forums Index() method &lt;=====");     MockForumContext mockForum = new MockForumContext(); }</pre> <p>Instead, use constructor injection by adding the highlighted lines to the top of the ForumController class:</p> <pre>private readonly ILogger&lt;ForumController&gt; _logger; private readonly IForumContext _context;  public ForumController(ILogger&lt;ForumController&gt; logger, IForumContext context) {     _logger = logger;     _context = context; }</pre> <p>And modify the Index action to use the injected database context:</p> <pre>public IActionResult Index() {     _logger.LogInformation("In the Forums Index() method &lt;=====");      IEnumerable&lt;Forum&gt; forums = _context.GetForums();     ... }</pre>
	<p>In the Startup.cs file, add the following line to the bottom of the ConfigureServices() method. This method 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> <pre>public void ConfigureServices(IServiceCollection services) {     ...     services.AddScoped&lt;IForumContext, MockForumContext&gt;(); }</pre> <p>Run the application and check that it still works. Using dependency injection is as easy as that!</p>

	<p>In the next part of the lab, we 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> <pre>public interface ITransient {     void WriteGuidToConsole(); } public class TransientDependency ...</pre> <p>Which we will inject with the corresponding scope.</p> <p>Have a look at the TransientDependency – you will see it writes out to the logger each time a new instance is instantiated</p> <p>Whenever the method WriteGuidToConsole() is called, it will show the unique Guid and the thread on which it is running</p>
	<p>At the beginning of Startup/ConfigureServices() 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"); }</pre>

	<pre> IEnumerable&lt;Forum&gt; forums = _context.GetForums();  _logger.LogDebug(\$"Number of forums: {forums.Count()}"); return View(ForumViewModel.FromForums(forums)); } </pre>
	<p>Run the application from the console.  (A reminder of how to do this: right-click on the project, select “Open project in file explorer”. Then, in file explorer, click into the address bar and type “cmd”. From the command window, type “dotnet run”).</p> <p>Open a web browser, go to <a href="https://localhost:5001">https://localhost:5001</a> and once the page has been displayed, press the refresh button to display it a second time.</p> <p>The console output is shown below, with annotations which explain what has happened:</p> <div data-bbox="304 645 1262 1617"> <p>Create the three dependencies</p> <p>Each has their own unique</p> <p>Refresh</p> <p>New transient &amp; scoped - but</p> <p>Singleton keeps same GUID, despite different</p> </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? 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>

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.

Let's modify our demonstration to show where these two types of lifecycle differ from each other.

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:

```
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 from the console, and once it's running, visit the web site *only once*. The console output now looks like this:

The screenshot shows a console window with the following output:

```

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    : 79-2634-4adf-8d26-af417f1d7223, thread=8
info: QuickTour.Controllers.ForumController[0]
      Singleton    : 15-f63c-40c7-ad06-95ddd8906fd4, thread=8
info: QuickTour.Controllers.ForumController[0]
      Scoped       : a27d-40f8-9887-d02a41bc10b3, thread=8
info: QuickTour.Controllers.ForumController[0]
      Singleton    : 79-2634-4adf-8d26-af417f1d7223, thread=8
debug: QuickTour.Controllers.ForumController[0]
      Number of forums: 2
  
```

Callout 1 (Transient): Transient constructor called twice – once for controller, once for transient

Callout 2 (Scope): Scope dependency lasts for whole request, keeps same GUID.

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.)

In Areas/Users/ForumController there is a commented-out method called Rare. Uncomment it. The idea is that this is a rarely used method, the resource it uses is expensive, so we don't want to create every time – just when this rare method is called.

You will find the interface and implementing class in Models/Dependencies.cs

Right-click the 'return View()' in the Rare method in ForumController and Add View. Accept all the default values.

Register in Startup/ConfiguresServices()

```
services.AddTransient<IActionInjection, ActionInjectionDependency>();
```

Run again from the command line.

Do a few browser refreshes and note that the ActionInjection object is not created. Now append '/Users/Forum/Rare' to the Url and note that the action dependency is now created. (We've used the LogWarning method here, in contrast to LogInformation in other places, so you can clearly see the difference by the different colour.)

## Configuring the Pipeline

	<p>Add a trivial module into the pipeline – drag the folder Middleware from the Assets folder onto the project. Have a look at the code in the classes in here.</p> <p>Inject this module into the pipeline by adding it in Start/Configure() as shown:</p> <pre>app.UseAuthentication(); app.UseAuthorization();  app.UseMiddleware&lt;CustomMiddleware1&gt;(); app.UseMiddleware&lt;CustomMiddleware2&gt;();</pre>
	<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>Start using dotnet run and open a browser at port 5001</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 modules to the pipeline),so it's definitely worth understanding what we mean when we talk about middleware, although writing your own middleware is not something you will need to do too often.</p> <p>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>

## Configuration – The Options Pattern



	<p>Drag the folder Configuration from the Assets folder onto your project. This contains a class with 2 configuration options</p> <pre>public class FeaturesConfiguration {     public bool EnableMyOption1 { get; set; }     public bool EnableMyOption2 { get; set; } }</pre>
	<p>Add this section to the end of appsettings.json, just above the final }</p> <pre>"Features": {   "EnableMyOption1": true,   "EnableMyOption2": false }</pre> <p>Visual Studio will automatically add the trailing comma to the preceding entry</p>
	<p>Now bind the json section to the strongly-typed FeaturesConfiguration in Startup/ConfigureServices()</p> <pre>services.Configure&lt;FeaturesConfiguration&gt;(Configuration.GetSection("Features"));</pre>
	<p>Go to ForumController. Modify the constructor:</p> <pre>private readonly FeaturesConfiguration _features; public ForumController(..., IOptions&lt;FeaturesConfiguration&gt; features) {     ...     _features = features.Value; }</pre> <p>And add this line to beginning of the Index() method</p> <pre>_logger.LogInformation(\$"MyOption1 = {_features.EnableMyOption1}, MyOption2 = {_features.EnableMyOption2}");</pre>
	<p>Dotnet run and launch a browser. Note that the appsettings.json settings have been set into a FeaturesConfiguration object:</p> <pre>MyOption1 = True, MyOption2 = False -----</pre> <p>In appsettings.json, set MyOption2 to be true and save the file (without re-compiling). Refresh the browser and note that the new value has not been read in. Close and restart the app: it is only read in on app startup.</p>
16	<p>Make these changes to ForumController</p> <pre>private readonly IConfiguration _config; public ForumViewModel(... IConfiguration config) {     _config = config; }</pre> <p>And add this line just after your current features output:</p> <pre>_logger.LogInformation(\$"MyOption1 = {_config["Features:EnableMyOption1"]}, MyOption2 = {_config["Features:EnableMyOption2"]}");</pre> <p>So now we have a type-safe way of reading configuration data (IOptions) and a type-unsafe way (config["key"]).</p>
17	<p>Refresh the page and check that the type-unsafe options are the same as the type-safe options.</p>

**If you have time**

**Reading environment-specific variants of appsettings.json**

	<p>Add this to appsettings.json, just before the final }</p> <pre>"Message": "Hello from appsettings.json"</pre>
	<p>Add this to appsettings.Development.json, just before the final }</p> <p>(Note that you may need to click the arrow next to appsettings.json to see the Development file)</p> <pre>"Message": "Hello from appsettings.Development.json"</pre>
	<p>Add to program.cs a line that declares a Microsoft.Extensions.Configuration.ConfigurationManager called config and make it equal to builder.Configuration:</p> <pre>Microsoft.Extensions.Configuration.ConfigurationManager config = builder.Configuration;</pre> <p>In Program.cs insert the following code just after the app.UseMiddleware lines you added earlier</p> <pre>Console.ForegroundColor = ConsoleColor.Magenta; Console.WriteLine(config["Message"]); Console.ForegroundColor = ConsoleColor.White;</pre>

Open the Project > Properties and go to the Debug tab and select the "Open debug launch profiles UI" link.

Ensure the Environment variables for http, https and IIS Express each have the following entry: ASPNETCORE\_ENVIRONMENT=Development:

Launch Profiles

http  
https  
IIS Express

**Command line arguments**  
Command line arguments to pass to the executable. You may break arguments into multiple lines.

**Working directory**  
Path to the working directory where the process will be started.

**Environment variables**  
The environment variables to set prior to running the process.

Name	Value
ASPNETCORE_ENVIRONMENT	Development

**Enable Hot Reload**  
☒ Apply code changes to the running application.

Launch Profiles

http  
https  
IIS Express

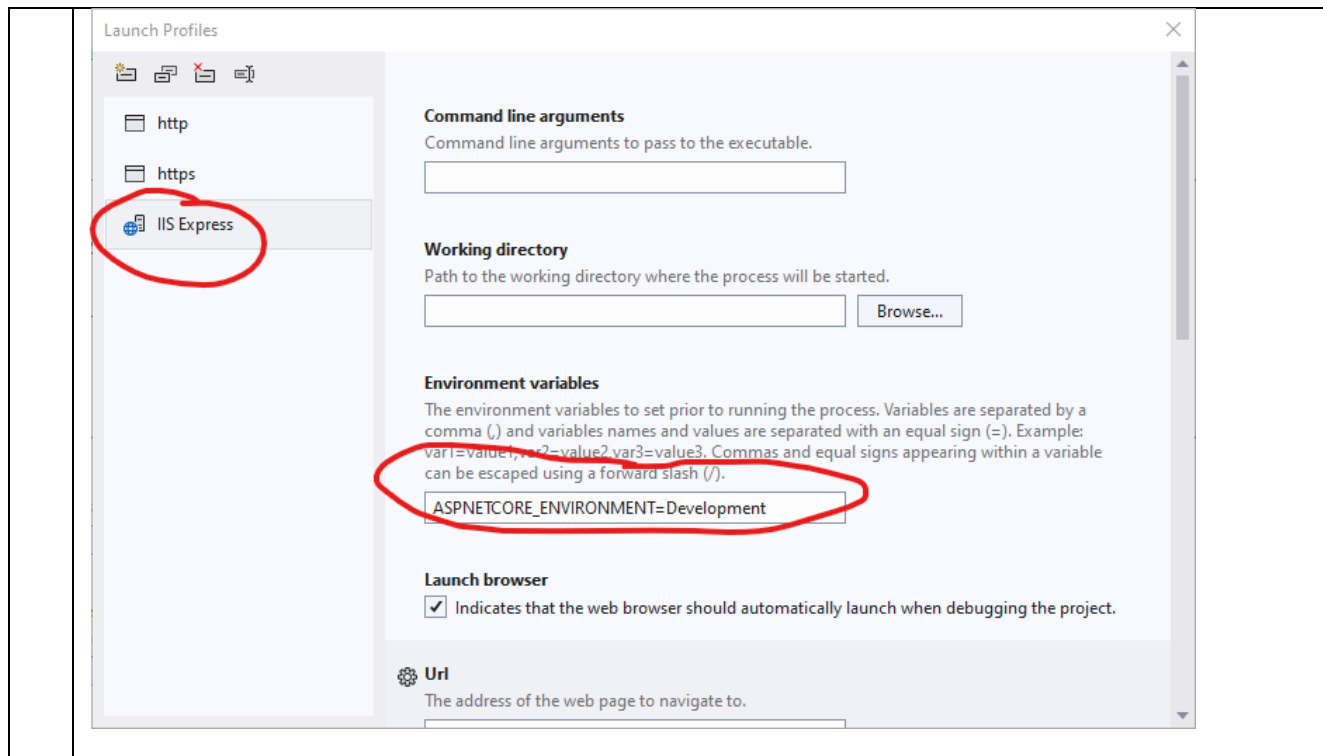
**Command line arguments**  
Command line arguments to pass to the executable. You may break arguments into multiple lines.

**Working directory**  
Path to the working directory where the process will be started.

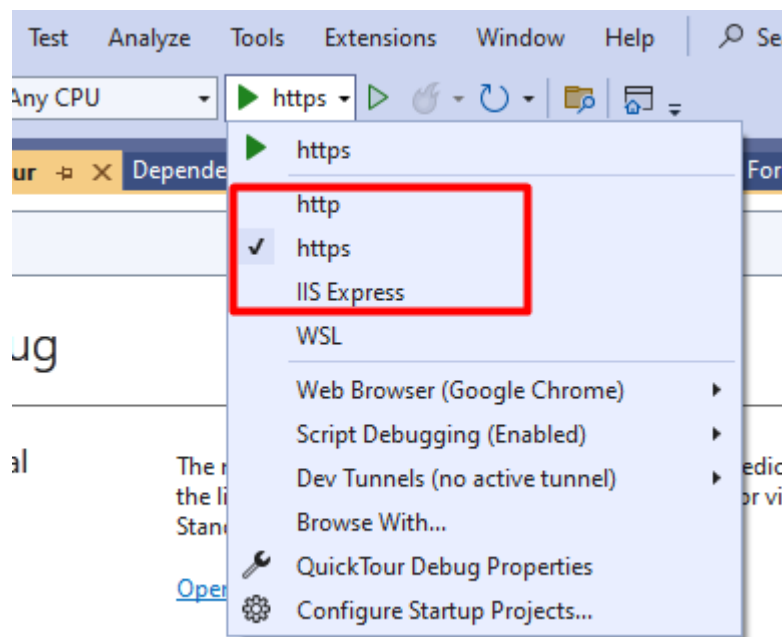
**Environment variables**  
The environment variables to set prior to running the process.

Name	Value
ASPNETCORE_ENVIRONMENT	Development

**Enable Hot Reload**  
☒ Apply code changes to the running application.



Note you can select a number of different ways of communicating with the browser from Visual Studio's drop down start debug menu:



Launch the app in debug mode using any of the three options and note the cyan message written to the console is from appsettings.Development.json i.e. "Hello from appsettings.Development.json"

Close the app, reopen the "Open debug launch profile UI" window and delete 'Development' for each of the three protocols:

The screenshot shows the 'Launch Profiles' window. In the left sidebar, the 'http' profile is selected and circled in red. The main area shows the configuration for this profile. Under 'Environment variables', the 'Name' field contains 'ASPNETCORE\_ENVIRONMENT' and the 'Value' field is empty, with a red 'X' icon to its right. The 'Enable Hot Reload' checkbox is checked.

**Launch Profiles**

Command line arguments  
Command line arguments to pass to the executable. You may break arguments into multiple lines.

Working directory  
Path to the working directory where the process will be started.

Environment variables  
The environment variables to set prior to running the process.

Name	Value
ASPNETCORE_ENVIRONMENT	

Enable Hot Reload  
☒ Apply code changes to the running application.

This screenshot is identical to the one above, but the 'https' profile is selected in the left sidebar and circled in red. The configuration details in the main area remain the same.

**Launch Profiles**

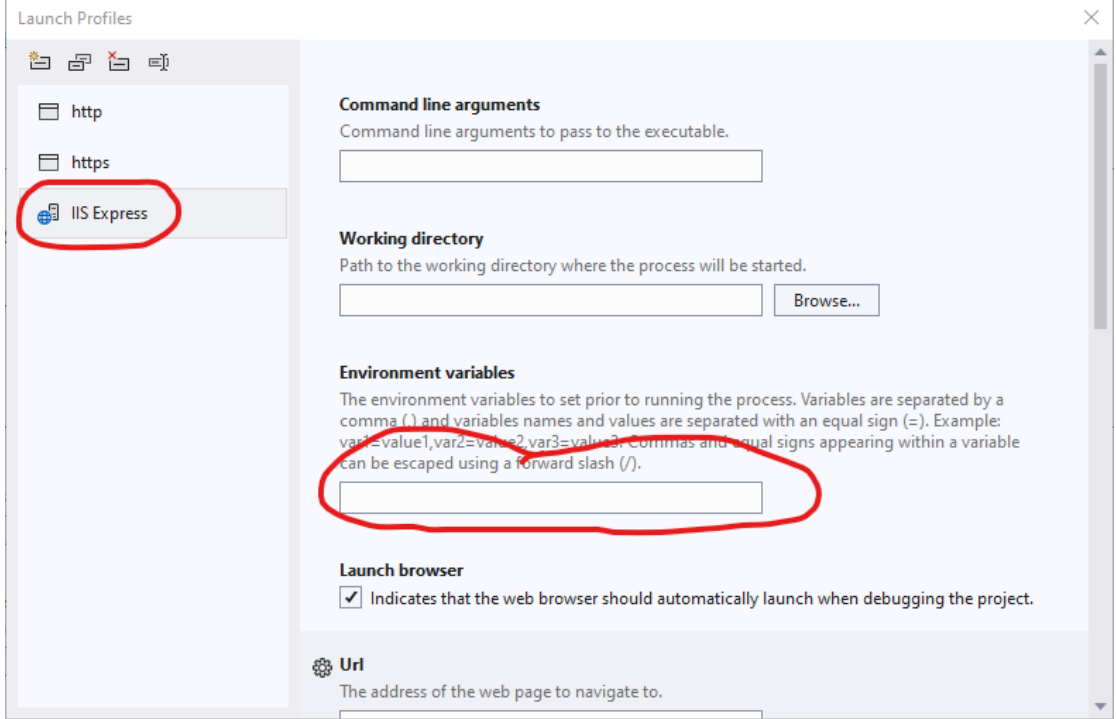
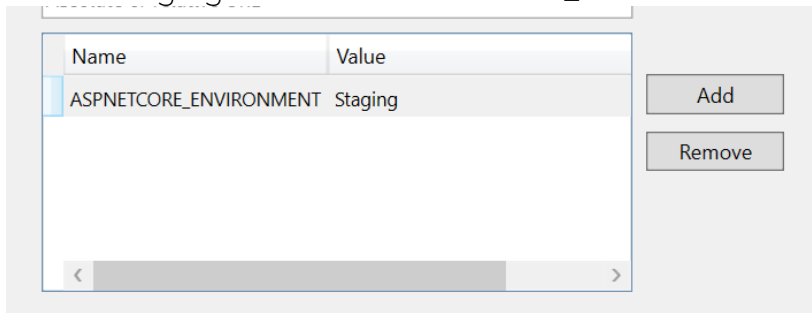
Command line arguments  
Command line arguments to pass to the executable. You may break arguments into multiple lines.

Working directory  
Path to the working directory where the process will be started.

Environment variables  
The environment variables to set prior to running the process.

Name	Value
ASPNETCORE_ENVIRONMENT	

Enable Hot Reload  
☒ Apply code changes to the running application.

	
	Run the app in debug mode again – and note it is now showing the message defined in appsetting.json i.e. " Hello from appsettings.json"
	Copy appsetting.Development.json to appsettings.Staging.json and modify the message to show it from Staging. If the file doesn't exist then add one to the project.
	Edit in 'Staging' into the ASPNETCORE_ENVIRONMENT for all three launch profiles: 
	Ctrl+F5 and you will see it now reads the Staging file

Lastly, just to confirm what has happened.  
When you set 'Staging' in the environment, it modified the file 'launchsettings.json' (expand under Properties and you'll see it).

```
"profiles": {  
  "http": {  
    "commandName": "Project",  
    "launchBrowser": true,  
    "environmentVariables": {  
      "ASPNETCORE_ENVIRONMENT": "Staging"  
    },  
    "dotnetRunMessages": true,  
    "applicationUrl": "http://localhost:5126"  
  },  
  "https": {  
    "commandName": "Project",  
    "launchBrowser": true,  
    "environmentVariables": {  
      "ASPNETCORE_ENVIRONMENT": "Staging"  
    },  
    "dotnetRunMessages": true,  
    "applicationUrl": "https://localhost:7236;http://localhost:5126"  
  },  
  "IIS Express": {  
    "commandName": "IISExpress",  
    "launchBrowser": true,  
    "environmentVariables": {  
      "ASPNETCORE_ENVIRONMENT": "Staging"  
    }  
  }  
}  
}...
```

Depending on the chosen launch protocol, the project that gets run is the first one in the list with

```
"commandName": "Project",
```