### Dependency Injection in ASP.NET vNext

Dependency Injection (DI) is a software design pattern, a particular case of the Inversion of Control pattern, in which one or more dependencies are injected into dependent objects. The pattern is used to create program designs that are loosely coupled and testable.

### DI in ASP.NET vNext

In ASP.NET vNext, dependency injection is a first class citizen. While in the previous versions of the framework, DI was partially supported, in ASP.NET vNext it is available throughout the entire stack. A minimalistic DI container is provided out of the box but we are leaving the door open to BYOC (Bring Your Own Container). The default container is useful in the cases when you don’t need any advanced injection capabilities (see the known limitations section at the end of the post).

BYOC is possible because of an abstraction over the actual DI container implementation. The abstraction is the IServiceProvider interface and it represents the least set of container behavior our components are limited to assuming are present. All the framework components (MVC, Routing, SignalR, Entity Framework, etc.) rely only on the capabilities of IServiceProvider, but your own application code can use any feature that your chosen DI container has. When you BYOC, you can replace the default implementation of IServiceProvider with a wrapper around your own container.  Once that happens, all the dependency resolution calls will be routed to your own container. In the case when you want to use your own container strictly for your own custom types, we support fallback to our default container.

Because all framework components use the same container to register services, we can now flow dependencies across the stack. This opens the door to new scenarios that were not possible before, like injecting a SignalR broadcaster into an MVC controller action. As we walk up the stack, there are different layers of dependency resolvers. All dependencies are added to a single container and everyone can see everybody else’s services. The single container also addresses the cross-cutting concern customization story. It is trivial in the new stack to change cross-cutting concerns (e.g. logging) via a single entry point.

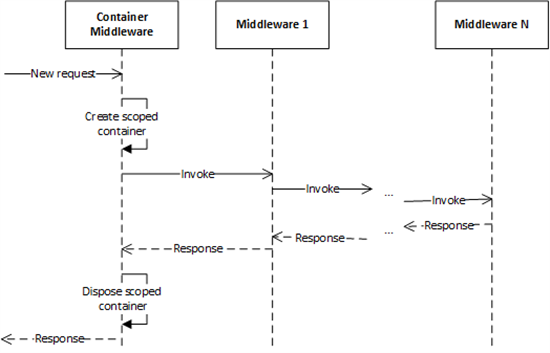
The out of the box container supports the following lifestyles:

|  |  |
| --- | --- |
| **Lifestyle** | **Description** |
| Instance | A specific instance is given all the time. You are responsible for its initial creation |
| Transient | A new instance is created every time |
| Singleton | A single instance is created and it acts like a singleton |
| Scoped | A single instance is created inside the current scope. It is equivalent to Singleton in the current scope |

### Per Request Scope

A popular feature for DI in web applications is to create objects that have a single instance per web request. This means that the objects acts as a singleton inside that request but two distinct requests will have different instances of the objects.

In ASP.NET vNext, the Per Request Scope is achieved using a middleware and a scoped lifestyle. The middleware, when invoked, will create a new scoped container which will replace the container for the current request. All the subsequent middleware in the pipeline will then utilize the scoped container. After the request flows through the pipeline and the container middleware is signaled to complete, the scope is destroyed and all the objects created inside it are disposed.

[](http://blogs.msdn.com/cfs-file.ashx/__key/communityserver-blogs-components-weblogfiles/00-00-00-63-56/4530.ContainerMiddleware.png)

The source code for the ContainerMiddleware, is [available on GitHub](https://github.com/aspnet/Hosting/blob/dev/src/Microsoft.AspNet.RequestContainer/ContainerMiddleware.cs).

In the rare case in which you need to create your own container scope, you can use the IServiceScopeFactory to do this. When the default implementation of the IServiceProvider is created, the IServiceScopeFactory is one of the services that are registered by default.

### New vs Old

For the purpose of showing the differences between DI in the old and new stack, we are going to use an MVC controller that writes a string provided through an injected dependency:

**Code Snippet**

1. public interface IMessageGenerator
2. {
3. string GenerateMessage();
4. }
6. public class HelloMessageGenerator : IMessageGenerator
7. {
8. public string GenerateMessage()
9. {
10. return"Hello DI!";
11. }
12. }
14. public class MessageController : Controller
15. {
16. private readonly IMessageGenerator messageGenerator;
18. public MessageController(IMessageGenerator generator)
19. {
20. if (generator == null)
21. {
22. throw new ArgumentNullException("generator", "The generator dependecy is mandatory");
23. }
25. this.messageGenerator = generator;
26. }
28. public string GetMessage()
29. {
30. return this.messageGenerator.GenerateMessage();
31. }
32. }

None of the code above is changed between the old and the new stack. The only difference is where and how the dependency are registered and resolved.

In the old MVC stack, controller dependencies are resolved through a custom controller factory. For the purpose of this demo, we are going to implement the Poor Man’s DI and manually compose the dependencies:

**Code Snippet**

1. public class DIControllerFactory : DefaultControllerFactory
2. {
3. public override IController CreateController(RequestContext requestContext, string controllerName)
4. {
5. // If a message controller is requested...
6. if (controllerName == "Message")
7. {
8. // ... then create a new controller and set up the dependency
9. return new MessageController(new HelloMessageGenerator());
10. }
12. // Otherwise, fallback to the default implementation
13. return base.CreateController(requestContext, controllerName);
14. }
15. }
17. public class MvcApplication : HttpApplication
18. {
19. protected void Application\_Start()
20. {
21. ...
23. // Register the controller factor
24. ControllerBuilder.Current.SetControllerFactory(new DIControllerFactory());
26. ...
27. }
28. }

The controller factory will inject a concrete implementation (HelloMessageGenerator) of the interface (IMessageGenerator) in the Message controller.

The same code, ported to ASP.NET vNext, doesn’t need a Poor Man’s DI implementation. As mentioned before, in this new stack, a DI container is available out of the box. Thus, all we need to do is tell it what is the mapping between the interface and the concrete implementation.

**Code Snippet**

1. public class Startup
2. {
3. public void Configure(IBuilder app)
4. {
5. ...
6. app.UseServices(services =>
7. {
8. ...
9. // Set up the dependencies
10. services.AddTransient<IMessageGenerator, HelloMessageGenerator>();
11. ...
12. });
13. ...
14. }
15. }

The default implementation of the controller factory in ASP.NET vNext uses the IServiceProvider to resolve the dependencies. If you BYOC, none of the code above will change. You would only have to tell the application to use a different implementation of IServiceProvider.

### Replacing the default DI container

The code below shows how the previous sample can be rewritten to use Autofac instead of the out of the box container.

The code uses compiler directives to use Autofac code only for the full .NET 4.5 Framework because Autofac is not available for the .NET Core Framework 4.5. In Visual Studio “14” CTP you can change the target framework of a project by right clicking on it and going to Properties -> Active Target Framework.

**Code Snippet**

1. using System;
2. using Microsoft.AspNet.Builder;
3. using Microsoft.AspNet.Routing;
4. using Microsoft.Framework.DependencyInjection;
5. using Microsoft.Framework.DependencyInjection.Fallback;
6. using Microsoft.AspNet.Mvc;
7. using Microsoft.AspNet.RequestContainer;
8. using DISample.Models;
10. #if NET45
11. using Autofac;
12. using Microsoft.Framework.DependencyInjection.Autofac;
13. using Microsoft.Framework.OptionsModel;
14. #endif
16. namespace DISample
17. {
18. public class Startup
19. {
20. public void Configure(IBuilder app)
21. {
22. // Add the MVC services
23. ServiceCollection services = new ServiceCollection();
24. services.AddMvc();
25. services.Add(OptionsServices.GetDefaultServices());
27. // The NET45 symbol is defined when the project targets .NET Framework 4.5
28. #if NET45
29. // Create the autofac container
30. ContainerBuilder builder = new ContainerBuilder();
32. // Register the message generator through autofac
33. builder.RegisterType<HelloMessageGenerator>().As<IMessageGenerator>();
35. // Create the container and use the default application services as a fallback
36. AutofacRegistration.Populate(
37. builder,
38. services,
39. fallbackServiceProvider: app.ApplicationServices);
41. IContainer container = builder.Build();
43. // Replace the default container
44. app.ApplicationServices = container.Resolve<IServiceProvider>();
45. #else
46. // Here we are running on .NET Core Framework 4.5 so we cannot use Autofac
48. services.AddTransient<IMessageGenerator, HelloMessageGenerator>();
49. app.ApplicationServices = services.BuildServiceProvider(app.ApplicationServices);
50. #endif
51. // MVC requires the container middleware
52. app.UseMiddleware(typeof(ContainerMiddleware));
54. app.UseMvc(routes =>
55. {
56. routes.MapRoute(
57. name: "default",
58. template: "{controller}/{action}/{id?}",
59. defaults: new { controller = "Message", action = "GetMessage" });
60. });
61. }
62. }
63. }

In order to compile the code above, you must add the dependency injection dependencies to project.json. Since Autofac is not available for .NET Core Framework 4.5, the Autofac dependency is only defined for in the net45 section:

**Code Snippet**

1. {
2. "dependencies": {
3. "Helios": "0.1-alpha-build-\*",
4. "Microsoft.AspNet.Mvc": "0.1-alpha-build-\*",
5. "Microsoft.AspNet.Identity.Entity": "0.1-alpha-build-\*",
6. "Microsoft.AspNet.Identity.Security": "0.1-alpha-build-\*",
7. "Microsoft.AspNet.Security.Cookies": "0.1-alpha-build-\*",
8. "Microsoft.AspNet.Server.WebListener": "0.1-alpha-build-\*",
9. "Microsoft.AspNet.StaticFiles": "0.1-alpha-build-\*",
10. "Microsoft.Data.Entity": "0.1-alpha-build-\*",
11. "Microsoft.Data.Entity.SqlServer": "0.1-alpha-build-\*",
12. "Microsoft.Framework.ConfigurationModel.Json": "0.1-alpha-build-\*",
13. "Microsoft.VisualStudio.Web.BrowserLink.Loader": "14.0-alpha",
14. "Microsoft.Framework.DependencyInjection": "0.1-alpha-build-\*"
15. },
16. "commands": {
17. "web": "Microsoft.AspNet.Hosting --server Microsoft.AspNet.Server.WebListener --server.urls http://localhost:5000"
18. },
19. "configurations": {
20. "net45": {
21. "dependencies": {
22. "System.Data": "",
23. "System.ComponentModel.DataAnnotations": "",
24. "Microsoft.Framework.DependencyInjection.Autofac": "0.1-alpha-build-\*",
25. "Autofac": "3.3.0"
26. }
27. },
28. "k10": {
29. }
30. }
31. }

### Best Practices for DI in ASP.NET vNext

When using DI, we recommend the following practices:

1. Register all dependencies in the application startup method, before doing anything else.
2. Avoid consuming the IServiceProvider interface directly. Instead, add explicit dependencies as constructor parameters and let the callers resolve them. Use abstract factories when the number of dependencies becomes hard to manage.
3. Code against contracts rather than actual implementations.

### Known Limitations

The out of the box container for ASP.NET vNext Alpha has a few known limitations:

* It only supports constructor injection
* It can only resolve types with one and only one public constructor
* It doesn’t support advanced features (like per thread scope or auto discovery)

### Summary

ASP.NET vNext is DI-friendly and it uses dependency injection throughout the stack. While there is a default minimalistic DI container out of the box, you can Bring Your Own Container and use advanced DI capabilities. The source code for Dependency Injection [is available on GitHub](https://github.com/aspnet/DependencyInjection).

We’d love to hear your feedback. Please provide it in [Github](http://github.com/aspnet), comments on this blog, or the [ASP.NET vNext forum](http://forums.asp.net/1255.aspx/1?ASP+NET+vNext). If you ask a question in Stack Overflow, use [asp.net-vnext](http://stackoverflow.com/questions/tagged/asp.net-vnext) tag.

ASP.NET vNext is an open source project released under Apache License Version 2.0 by [Microsoft Open Technologies](http://msopentech.com/), Inc. You can follow its progress and find instructions on how to contribute on <https://github.com/aspnet>.