Create a .NET Core application with plugins

Article • 02/04/2022 • 8 minutes to read

This tutorial shows you how to create a custom AssemblyLoadContext to load plugins. An AssemblyDependencyResolver is used to resolve the dependencies of the plugin. The tutorial correctly isolates the plugin's dependencies from the hosting application. You'll learn how to:

- Structure a project to support plugins.
- Create a custom AssemblyLoadContext to load each plugin.
- Use the System.Runtime.Loader.AssemblyDependencyResolver type to allow plugins to have dependencies.
- Author plugins that can be easily deployed by just copying the build artifacts.

Prerequisites

- Install the .NET 5 SDK or a newer version.

> **Note**
>
> The sample code targets .NET 5, but all the features it uses were introduced in .NET Core 3.0 and are available in all .NET releases since then.

Create the application

The first step is to create the application:

1. Create a new folder, and in that folder run the following command:

   .NET CLI

   ```
   dotnet new console -o AppWithPlugin
   ```

2. To make building the project easier, create a Visual Studio solution file in the same folder. Run the following command:

   .NET CLI

```
dotnet new sln
```

3. Run the following command to add the app project to the solution:

.NET CLI

```
dotnet sln add AppWithPlugin/AppWithPlugin.csproj
```

Now we can fill in the skeleton of our application. Replace the code in the *AppWithPlugin/Program.cs* file with the following code:

C#

```
using PluginBase;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Reflection;

namespace AppWithPlugin
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                if (args.Length == 1 && args[0] == "/d")
                {
                    Console.WriteLine("Waiting for any key...");
                    Console.ReadLine();
                }

                // Load commands from plugins.

                if (args.Length == 0)
                {
                    Console.WriteLine("Commands: ");
                    // Output the loaded commands.
                }
                else
                {
                    foreach (string commandName in args)
                    {
                        Console.WriteLine($"-- {commandName} --");

                        // Execute the command with the name passed as an
argument.
```

```
                        Console.WriteLine();
                    }
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex);
            }
        }
    }
}
```

Create the plugin interfaces

The next step in building an app with plugins is defining the interface the plugins need to implement. We suggest that you make a class library that contains any types that you plan to use for communicating between your app and plugins. This division allows you to publish your plugin interface as a package without having to ship your full application.

In the root folder of the project, run `dotnet new classlib -o PluginBase`. Also, run `dotnet sln add PluginBase/PluginBase.csproj` to add the project to the solution file. Delete the `PluginBase/Class1.cs` file, and create a new file in the `PluginBase` folder named `ICommand.cs` with the following interface definition:

C#

```
namespace PluginBase
{
    public interface ICommand
    {
        string Name { get; }
        string Description { get; }

        int Execute();
    }
}
```

This `ICommand` interface is the interface that all of the plugins will implement.

Now that the `ICommand` interface is defined, the application project can be filled in a little more. Add a reference from the `AppWithPlugin` project to the `PluginBase` project with the `dotnet add AppWithPlugin/AppWithPlugin.csproj reference PluginBase/PluginBase.csproj` command from the root folder.

Replace the `// Load commands from plugins` comment with the following code snippet to enable it to load plugins from given file paths:

C#

```
string[] pluginPaths = new string[]
{
    // Paths to plugins to load.
};

IEnumerable<ICommand> commands = pluginPaths.SelectMany(pluginPath =>
{
    Assembly pluginAssembly = LoadPlugin(pluginPath);
    return CreateCommands(pluginAssembly);
}).ToList();
```

Then replace the `// Output the loaded commands` comment with the following code snippet:

C#

```
foreach (ICommand command in commands)
{
    Console.WriteLine($"{command.Name}\t - {command.Description}");
}
```

Replace the `// Execute the command with the name passed as an argument` comment with the following snippet:

C#

```
ICommand command = commands.FirstOrDefault(c => c.Name == commandName);
if (command == null)
{
    Console.WriteLine("No such command is known.");
    return;
}

command.Execute();
```

And finally, add static methods to the `Program` class named `LoadPlugin` and `CreateCommands`, as shown here:

C#

```
static Assembly LoadPlugin(string relativePath)
{
    throw new NotImplementedException();
```

```
}

static IEnumerable<ICommand> CreateCommands(Assembly assembly)
{
    int count = 0;

    foreach (Type type in assembly.GetTypes())
    {
        if (typeof(ICommand).IsAssignableFrom(type))
        {
            ICommand result = Activator.CreateInstance(type) as ICommand;
            if (result != null)
            {
                count++;
                yield return result;
            }
        }
    }

    if (count == 0)
    {
        string availableTypes = string.Join(",",
assembly.GetTypes().Select(t => t.FullName));
        throw new ApplicationException(
            $"Can't find any type which implements ICommand in {assembly}
from {assembly.Location}.\n" +
            $"Available types: {availableTypes}");
    }
}
```

Load plugins

Now the application can correctly load and instantiate commands from loaded plugin assemblies, but it's still unable to load the plugin assemblies. Create a file named *PluginLoadContext.cs* in the *AppWithPlugin* folder with the following contents:

C#

```
using System;
using System.Reflection;
using System.Runtime.Loader;

namespace AppWithPlugin
{
    class PluginLoadContext : AssemblyLoadContext
    {
        private AssemblyDependencyResolver _resolver;

        public PluginLoadContext(string pluginPath)
        {
            _resolver = new AssemblyDependencyResolver(pluginPath);
```

```
        }

        protected override Assembly Load(AssemblyName assemblyName)
        {
            string assemblyPath =
_resolver.ResolveAssemblyToPath(assemblyName);
            if (assemblyPath != null)
            {
                return LoadFromAssemblyPath(assemblyPath);
            }

            return null;
        }

        protected override IntPtr LoadUnmanagedDll(string unmanagedDll-
Name)
        {
            string libraryPath =
_resolver.ResolveUnmanagedDllToPath(unmanagedDllName);
            if (libraryPath != null)
            {
                return LoadUnmanagedDllFromPath(libraryPath);
            }

            return IntPtr.Zero;
        }
    }
}
```

The `PluginLoadContext` type derives from AssemblyLoadContext. The `AssemblyLoadContext` type is a special type in the runtime that allows developers to isolate loaded assemblies into different groups to ensure that assembly versions don't conflict. Additionally, a custom `AssemblyLoadContext` can choose different paths to load assemblies from and override the default behavior. The `PluginLoadContext` uses an instance of the `AssemblyDependencyResolver` type introduced in .NET Core 3.0 to resolve assembly names to paths. The `AssemblyDependencyResolver` object is constructed with the path to a .NET class library. It resolves assemblies and native libraries to their relative paths based on the *.deps.json* file for the class library whose path was passed to the `AssemblyDependencyResolver` constructor. The custom `AssemblyLoadContext` enables plugins to have their own dependencies, and the `AssemblyDependencyResolver` makes it easy to correctly load the dependencies.

Now that the `AppWithPlugin` project has the `PluginLoadContext` type, update the `Program.LoadPlugin` method with the following body:

C#

```
static Assembly LoadPlugin(string relativePath)
{
    // Navigate up to the solution root
    string root = Path.GetFullPath(Path.Combine(
        Path.GetDirectoryName(
            Path.GetDirectoryName(
                Path.GetDirectoryName(
                    Path.GetDirectoryName(

Path.GetDirectoryName(typeof(Program).Assembly.Location)))))));

    string pluginLocation = Path.GetFullPath(Path.Combine(root,
relativePath.Replace('\\', Path.DirectorySeparatorChar)));
    Console.WriteLine($"Loading commands from: {pluginLocation}");
    PluginLoadContext loadContext = new
PluginLoadContext(pluginLocation);
    return loadContext.LoadFromAssemblyName(new
AssemblyName(Path.GetFileNameWithoutExtension(pluginLocation)));
}
```

By using a different `PluginLoadContext` instance for each plugin, the plugins can have different or even conflicting dependencies without issue.

Simple plugin with no dependencies

Back in the root folder, do the following:

1. Run the following command to create a new class library project named `HelloPlugin`:

   .NET CLI

   ```
   dotnet new classlib -o HelloPlugin
   ```

2. Run the following command to add the project to the `AppWithPlugin` solution:

   .NET CLI

   ```
   dotnet sln add HelloPlugin/HelloPlugin.csproj
   ```

3. Replace the *HelloPlugin/Class1.cs* file with a file named *HelloCommand.cs* with the following contents:

C#

```
using PluginBase;
using System;

namespace HelloPlugin
{
    public class HelloCommand : ICommand
    {
        public string Name { get => "hello"; }
        public string Description { get => "Displays hello message."; }

        public int Execute()
        {
            Console.WriteLine("Hello !!!");
            return 0;
        }
    }
}
```

Now, open the *HelloPlugin.csproj* file. It should look similar to the following:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>

</Project>
```

In between the `<PropertyGroup>` tags, add the following element:

XML

```
  <EnableDynamicLoading>true</EnableDynamicLoading>
```

The `<EnableDynamicLoading>true</EnableDynamicLoading>` prepares the project so that it can be used as a plugin. Among other things, this will copy all of its dependencies to the output of the project. For more details see EnableDynamicLoading.

In between the `<Project>` tags, add the following elements:

XML

```
<ItemGroup>
    <ProjectReference Include="..\PluginBase\PluginBase.csproj">
        <Private>false</Private>
```

```
        <ExcludeAssets>runtime</ExcludeAssets>
    </ProjectReference>
</ItemGroup>
```

The `<Private>false</Private>` element is important. This tells MSBuild to not copy *PluginBase.dll* to the output directory for HelloPlugin. If the *PluginBase.dll* assembly is present in the output directory, `PluginLoadContext` will find the assembly there and load it when it loads the *HelloPlugin.dll* assembly. At this point, the `HelloPlugin.HelloCommand` type will implement the `ICommand` interface from the *PluginBase.dll* in the output directory of the `HelloPlugin` project, not the `ICommand` interface that is loaded into the default load context. Since the runtime sees these two types as different types from different assemblies, the `AppWithPlugin.Program.CreateCommands` method won't find the commands. As a result, the `<Private>false</Private>` metadata is required for the reference to the assembly containing the plugin interfaces.

Similarly, the `<ExcludeAssets>runtime</ExcludeAssets>` element is also important if the `PluginBase` references other packages. This setting has the same effect as `<Private>false</Private>` but works on package references that the `PluginBase` project or one of its dependencies may include.

Now that the `HelloPlugin` project is complete, you should update the `AppWithPlugin` project to know where the `HelloPlugin` plugin can be found. After the `// Paths to plugins to load` comment, add `@"HelloPlugin\bin\Debug\net5.0\HelloPlugin.dll"` (this path could be different based on the .NET Core version you use) as an element of the `pluginPaths` array.

Plugin with library dependencies

Almost all plugins are more complex than a simple "Hello World", and many plugins have dependencies on other libraries. The `JsonPlugin` and `OldJsonPlugin` projects in the sample show two examples of plugins with NuGet package dependencies on `Newtonsoft.Json`. Because of this, all plugin projects should add `<EnableDynamicLoading>true</EnableDynamicLoading>` to the project properties so that they copy all of their dependencies to the output of `dotnet build`. Publishing the class library with `dotnet publish` will also copy all of its dependencies to the publish output.

Other examples in the sample

The complete source code for this tutorial can be found in the dotnet/samples repository . The completed sample includes a few other examples of `AssemblyDependencyResolver` behavior. For example, the `AssemblyDependencyResolver` object can also resolve native libraries as well as localized satellite assemblies included in NuGet packages. The `UVPlugin` and `FrenchPlugin` in the samples repository demonstrate these scenarios.

Reference a plugin interface from a NuGet package

Let's say that there is an app A that has a plugin interface defined in the NuGet package named `A.PluginBase`. How do you reference the package correctly in your plugin project? For project references, using the `<Private>false</Private>` metadata on the `ProjectReference` element in the project file prevented the dll from being copied to the output.

To correctly reference the `A.PluginBase` package, you want to change the `<PackageReference>` element in the project file to the following:

XML

```
<PackageReference Include="A.PluginBase" Version="1.0.0">
    <ExcludeAssets>runtime</ExcludeAssets>
</PackageReference>
```

This prevents the `A.PluginBase` assemblies from being copied to the output directory of your plugin and ensures that your plugin will use A's version of `A.PluginBase`.

Plugin target framework recommendations

Because plugin dependency loading uses the *.deps.json* file, there is a gotcha related to the plugin's target framework. Specifically, your plugins should target a runtime, such as .NET 5, instead of a version of .NET Standard. The *.deps.json* file is generated based on which framework the project targets, and since many .NET Standard-compatible packages ship reference assemblies for building against .NET Standard and implementation assemblies for specific runtimes, the *.deps.json* may not correctly see implementation assemblies, or it may grab the .NET Standard version of an assembly instead of the .NET Core version you expect.

Plugin framework references

Currently, plugins can't introduce new frameworks into the process. For example, you can't load a plugin that uses the `Microsoft.AspNetCore.App` framework into an application that only uses the root `Microsoft.NETCore.App` framework. The host application must declare references to all frameworks needed by plugins.