# Activity 8

NeDL Transition Academy

- Programming Activity 8: Create .NET Core console application that supports configuration, dependency inversion, and Entity Framework.

.NET Core has changed things quite a bit from the old .NET Framework days. While it is really easy to look at a .NET Core console application and think it is pretty much a .NET Framework application, there are potentially many differences.

Now in both .NET Framework and .NET Core you can quickly just create a console application. But where it gets interesting is if you want to access many of the new features available in .NET Core. For example, if you want access to the .NET Core's dependency inversion (Service Collection) features, you are going to have to do a little more work, and your simple console application won't be so simple. Another good example of one of these changes would be accessing a database. If you want to access a database you may want to make some changes to the simple set.

In this activity we are going to configure a .NET Core Console application to run with dependency inversion, configuration, and entity framework. Things that would be pretty likely to want in a console application. (Note, we won’t use entity framework in this example)

Step 1. Create a console application. In Visual Studio you can do this with File | New. Or you can use the dotnet CLI command "dotnet new console".

Step 2. Add some nuget package references. This can be done with the CLI command "dotnet add package <package name>" or using the Add Reference dialog in Visual Studio.

```
Microsoft.EntityFrameworkCore.InMemory
Microsoft.EntityFrameworkCore.Sqlite
Microsoft.EntityFrameworkCore.SqlServer
Microsoft.Extensions.Hosting
Microsoft.Extensions.Configuration
Microsoft.Extensions.Configuration.EnvironmentVariables
Microsoft.Extensions.Configuration.Json
Microsoft.Extensions.Configuration.CommandLine
```

Step 3. Update our Program class.

An out of the box Program class would look like

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello...")
    }
}
```

But we need to make some pretty big changes here. First we must make the Main method an async method. Second, we need to create a new property to initialize the host "CreateHostBuilder". A lot of work occurs here. We are initializing how we interact with configuration, and we are configuring some of our services.

```
static class Program
{
    public static IConfigurationRoot _configuration { get;
private set; }

    static async Task Main(string[] args)
    {
        using IHost host = CreateHostBuilder(args).Build();
        await host.RunAsync();
    }


    static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
        .ConfigureAppConfiguration((hostingContext,
configuration) =>
        {
            configuration.Sources.Clear();
            configuration
                    .AddJsonFile("appsettings.json", optional:
true, reloadOnChange: true);
```

```
            IConfigurationRoot configurationRoot =
configuration.Build();
            _configuration = configurationRoot;
        }).ConfigureServices((services) =>
        {
            services.AddHostedService<ConsoleService>();
        });
}
```

Step 4. We need to put our code into the ConsoleService class. The ConsoleService class will be where we put our code that would have previously lived in our Main method.

```
using System;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.Extensions.Hosting;

public class ConsoleService : IHostedService
{
    private readonly IHostApplicationLifetime _appLifetime;
    private readonly CRMContext _dbContext;

    public ConsoleService(
        CRMContext dbContext,
        IHostApplicationLifetime appLifetime)
    {
        _dbContext = dbContext;
        _appLifetime = appLifetime;
    }

    public Task StartAsync(CancellationToken cancellationToken)
    {

        _appLifetime.ApplicationStarted.Register(() =>
        {
            Task.Run(async () =>
            {
                try
                {
                    Console.WriteLine("hello...");
                }
                catch (Exception ex)
                {
                    Console.WriteLine(ex.ToString());
                }
                finally
                {

                    _appLifetime.StopApplication();
                }
            });
        });

        return Task.CompletedTask;
    }

    public Task StopAsync(CancellationToken cancellationToken)
    {
        return Task.CompletedTask;
    }
}
```

Why would we go through all of this effort? Feels like a lot of work, right? Very true. Lots of code to for a console application. But we now have dependency inversion setup. We can read from configuration files. While a lot of code to achieve this, it is a good starting point for writing a console application. In the next blog post we are going to parse a list of contacts and inject them into a database.

## Part 2

Add parsing a CSV from activity 4 into the Console Service.

```
using System;
using CsvHelper.Configuration.Attributes;

public class Contact
{
    [Name("id")]
    public int Id { get; set; }
    [Name("first_name")]
    public string FirstName { get; set; }
    [Name("last_name")]
    public string LastName { get; set; }
    [Name("email")]
    public string Email { get; set; }
    [Name("gender")]
    public string Gender { get; set; }
    [Name("skill")]
    public string Skill { get; set; }
    [Name("ip_address")]
    public string IpAddress { get; set; }

    [Name("GUID")]
    public Guid GUID { get; set; }
}
```

Method to read contacts.

```
        static Contact[] ReadContacts()
        {
            using var reader = new StreamReader("Contacts.csv");
            using var csv = new CsvReader(reader, CultureInfo.InvariantCulture);
            var contacts = csv.GetRecords<Contact>().ToArray();
            return contacts;
        }
```