You Try It! Hands-On Implementation of .NET Libraries

Objective: by the end of this activity, you will be able to integrate a commonly used .NET library into a C# project using NuGet and demonstrate its basic functionality.

Step 1: Create a Basic C# Console Application in Visual Studio Code

Before working with any .NET libraries, you need to create a basic C# console application in Visual Studio Code. This step will guide you through setting up the project.

Steps:

1. Open Visual Studio Code.
2. Open the terminal by selecting View > Terminal.
3. In the terminal, create a new directory for your project and navigate into it using the cd command:
    - Create a new folder for your project with a command like mkdir MyFirstConsoleApp.
    - Navigate into the folder with cd MyFirstConsoleApp.
4. Initialize a new C# console application by running the command:
    - dotnet new console -n JsonExample
    - This command creates a new folder named JsonExample with the necessary files for a basic console app.
5. Change directories into your project folder with:
    - cd JsonExample
6. Open the project in Visual Studio Code by running:
    - code .
7. Confirm that your project has been created by checking the presence of the Program.cs file. It should contain a basic "Hello World" program.

Step 2: Run the Basic Console Application

Now that you have created the project, you should run it to make sure everything is set up correctly.

Steps:

1. In Visual Studio Code, ensure you are in the root directory of the project (where Program.cs and JsonExample.csproj are located).
2. Open the terminal and run the following command to build and execute the project using dotnet run
3. Confirm that the output displays "Hello World!" to ensure your project is working.

Step 3: Install a .NET Library Using NuGet

In this step, you will add the Newtonsoft.Json library to a .NET project using NuGet. This library is widely used for handling JSON data in web applications and APIs.

Steps:

1. Open your console app project in Visual Studio Code.
2. In Visual Studio Code, open the Terminal and install the Newtonsoft.Json package using the appropriate command.
3. Verify that the package was installed by checking the csproj file, which should list Newtonsoft.Json as a dependency.

Step 4: Use the .NET Library to Parse JSON Data

Now that you have installed the Newtonsoft.Json library, you will use it to parse a JSON string into a C# object.

Steps:

1. In the Program.cs file, define a simple class named Person.
2. Create a JSON string inside the Main method.
3. Use Newtonsoft.Json to convert the JSON string into a Person object. Write code to output the parsed data.
4. Run the program using the terminal and check the output to confirm that the JSON string was successfully parsed.

Step 5: Serialize an Object to JSON Format

In this step, you will take a C# object and convert it back into a JSON string using the Newtonsoft.Json library.

Steps:

1. Add additional code to the Main method to create a new Person object.
2. Use Newtonsoft.Json to serialize the Person object to a JSON string.
3. Run the program again using the terminal.
4. Verify that the output displays the JSON string representation of the Person object.

Program.cs:

```
using System.Text.Json;
using Newtonsoft.Json;
using Serilog;

class Person
{
    public required string Name { get; set; }
    public int Age { get; set; }
    public required string Country { get; set; }
}

class Program
{
    static void Main(string[] args)
    {
        Log.Logger = new
LoggerConfiguration().WriteTo.Console().CreateLogger();

        Console.WriteLine("=== Variant A: System.Text.Json ===");
        string jsonA1 = @"{ ""Name"": ""Alice"", ""Age"": 30,
""Country"": ""USA"" }";
        using var doc = JsonDocument.Parse(jsonA1);
        var root = doc.RootElement;
        Console.WriteLine("Dynamic-like name: " +
root.GetProperty("Name").GetString());

        string jsonA2 = @"{ ""Name"": ""Bob"", ""Age"": 25, ""Country"":
""UK"" }";
        var typedA =
System.Text.Json.JsonSerializer.Deserialize<Person>(jsonA2);
        Console.WriteLine("Typed name: " + (typedA != null ? typedA.Name
: "null"));

        var newPersonA = new Person { Name = "Diana", Age = 28, Country
= "Germany" };
        string serializedA =
System.Text.Json.JsonSerializer.Serialize(newPersonA, new
JsonSerializerOptions { WriteIndented = true });
        Console.WriteLine("Serialized JSON (A):");
        Console.WriteLine(serializedA);

        Console.WriteLine("\nA notes:");
        Console.WriteLine("- Built-in, no extra dependency.");
        Console.WriteLine("- Dynamic-like access uses JsonDocument and
explicit property APIs.");
```

```
        Console.WriteLine("- Strong typing via Person works similarly to
B.");
        Console.WriteLine("- Fast and modern defaults; fewer convenience
features for dynamic work.");

        Log.Information("Completed Variant A");

        Console.WriteLine("\n=== Variant B: Newtonsoft.Json ===");
        string jsonB1 = @"{ ""Name"": ""Eve"", ""Age"": 32, ""Country"":
""Canada"" }";
        var dynB =
Newtonsoft.Json.JsonConvert.DeserializeObject<dynamic>(jsonB1);
        Console.WriteLine("Dynamic name: " + (dynB != null ?
dynB.Name.ToString() : "null"));

        string jsonB2 = @"{ ""Name"": ""Frank"", ""Age"": 27,
""Country"": ""France"" }";
        var typedB =
Newtonsoft.Json.JsonConvert.DeserializeObject<Person>(jsonB2);
        Console.WriteLine("Typed name: " + (typedB != null ? typedB.Name
: "null"));

        var newPersonB = new Person { Name = "Grace", Age = 29, Country
= "Italy" };
        string serializedB =
Newtonsoft.Json.JsonConvert.SerializeObject(newPersonB,
Formatting.Indented);
        Console.WriteLine("Serialized JSON (B):");
        Console.WriteLine(serializedB);

        Console.WriteLine("\nB notes:");
        Console.WriteLine("- External package via NuGet.");
        Console.WriteLine("- Dynamic parsing is straightforward with
dynamic/JObject.");
        Console.WriteLine("- Rich feature set and ecosystem; great for
complex scenarios.");
        Console.WriteLine("- Adds a dependency but very popular and
flexible.");

        Console.WriteLine("\nKey differences:");
        Console.WriteLine("- A is dependency-free and uses structured
JsonDocument for dynamic-like access.");
        Console.WriteLine("- B uses a NuGet package and offers easier
dynamic handling and more features.");
        Console.WriteLine("- Both support strongly-typed models; outputs
are similar for basic cases.");

        Log.Information("Completed Variant B");
    }
}
```