Clase 6 @September 29, 2022

Reflection

En .Net, Reflection es la **habilidad** de un programa de **autoexaminarse** con el objetivo de encontrar ensamblados (.dll), módulos, o información de tipos en **tiempo de ejecución**. En otras palabras, a nivel de código vamos a tener clases y objetos, que nos van a permitir referenciar a ensamblados, y a los tipos que se encuentran contenidos.

`Se dice que un programa se refleja en sí mismo (de ahí el termino "reflexión"), a partir de extraer metadata de sus assemblies y de usar esa metadata para ciertos fines. Ya sea para informarle al usuario o para modificar su comportamiento.`

```
public class ImporterFromXml:IImporter
{
}

public class ImporterFromJSon:IImporter
{
}
```

```
FileInfo[] files = directory.GetFiles("*.dll");
   foreach (var file in files){
       Assembly assembly = Assembly.LoadFile(file.FullName);
       Type type = assembly.GetTypes().FirstOrDefault(t => typeof(IImporter).IsAssignableFrom(t) && t.IsClass)!;
       IImporter importer = (Activator.CreateInstance(type) as IImporter)!;
       // ImporterFromXml importer = new ImporterFromJson();
}
```

Al usar Reflection en C#, estamos pudiendo obtener la información detallada de un objeto, sus métodos, e incluso crear objetos e invocar sus métodos en tiempo de ejecución, sin haber tenido que realizar una referencia al ensamblado que contiene la clase y a su namespace.

Específicamente lo que nos permite usar Reflection es el namespace `System.Reflecion`, que contiene clases e interfaces que nos permiten manejar todo lo mencionado anteriormente: ensamblados, tipos, métodos, campos, crear objetos, invocar métodos, etc.

¿Por qué usar reflection?

Permite escribir programas que no tienen que **"conocer todo"** en tiempo de compilación, lo que los hace más **"dinámicos"**, ya que pueden vincularse en **"tiempo de ejecución"**.

El código se puede escribir en interfaces conocidas, pero las clases reales que se utilizarán se pueden instanciar utilizando la reflexión de los archivos de configuración y localizando, en tiempo de ejecución, la implementación correcta.

Supongamos por ejemplo, que necesitamos que nuestra aplicación soporte diferentes tipos de loggers (mecanismos para registrar datos/eventos que van ocurriendo en el flujo del programa). Además, supongamos que hay desarrolladores terceros que nos brindan una .dll externa que escribe información de logger y la envía a un servidor. En ese caso, tenemos dos opciones:

1. Podemos referenciar al ensamblado directamente y llamar a sus métodos (como hemos hecho siempre)
2. Podemos usar Reflection para cargar el ensamblado y llamar a sus métodos a partir de sus interfaces.

En este caso, si quisiéramos que nuestra aplicación sea lo más desacoplada posible, de manera que otros loggers puedan ser agregados (o 'plugged in' -de ahí el nombre plugin-) de forma sencilla y SIN RECOMPILAR la aplicación, es necesario elegir la segunda opción.

Por ejemplo podríamos hacer que el usuario elija (a medida que está usando la aplicación), y descargue la .dll de logger para elegir usarla en la aplicación. La única forma de hacer esto es a partir de Reflection. De esta forma, podemos cargar ensamblados externos a nuestra aplicación, y cargar sus tipos en tiempo de ejecución.

Favoreciendo el desacoplamiento

Lo que es importante para lograr el desacoplamiento de tipos externos, es que nuestro código referencie a una Interfaz, que es la que toda .dll externa va a tener que cumplir. Tiene que existir entonces ese contrato previo, de lo contrario, no sería posible saber de antemano qué métodos llamar de las librerías externas que poseen clases para usar loggers.

Ejemplo

1. Crear solución App con interfaz ILogger y luego implementamos esa interfaz en otro proyecto que va a ser el que genera nuestra dll.

```
using System;

namespace Contract
{
    public interface ILogger
    {
        void Log(string  toSave, string config);
    }
}
```

```
using System;
using Contract;

namespace implementation
{
    public class LogTxt : ILogger
    {
        public string Name { get; set; }
        public LogTxt() { }
        public LogTxt(string aName)
        {
```

```
            Name = aName;
        }

        public void Log(string toSave, string route)
        {
            string text = "Campo 1: " + toSave.ToString();
            System.IO.File.WriteAllText(@route, text);
        }
    }
}
```

2. Generar dll corriendo el proyecto y dejarla en carpeta común

3. Ahora vamos a hacer nuestro método en Program.cs que lea de la consola

```
static void Main(string[] args)
        {
            string input = "";
            Console.WriteLine("Bienvienido a la app de reflection");
            while (input != "S")
            {
                try
                {
                    Console.WriteLine("Ingrese la ruta del assembly");
                    input = Console.ReadLine();
                    //recorro el contenido del assembly, es a modo de ejemplo
                    //para poder entender como esta estructurado un assembly y los comandos para navegarlo
                    InspectAssembly(input);
                    Console.WriteLine("Ingrese el texto que desea guardar");
                    string toSave= Console.ReadLine();
                    Console.WriteLine("Ingrese la ruta donde desea guardar");
                    string route = Console.ReadLine();
                    InstanceAndExecuteLog(input, toSave, route);

                }
                catch (Exception e)
                {
                    Console.WriteLine("Algo salío mal, vuelve a intentarlo.." + e.ToString());
                }
            }

        }
```

4. Creamos el método que inspecciona el assembly

```
private static void InspectAssembly(string input)
        {
            // Cargamos el assembly en memoria
            Assembly myAssembly = Assembly.LoadFile(input);
            // Mostraremos toda la info del assembly
```

```
            foreach (Type tipo in myAssembly.GetTypes())
            {
                Console.WriteLine(string.Format("Clase: {0}", tipo.Name));

                Console.WriteLine("Fields");
                foreach (FieldInfo prop in tipo.GetFields())
                {
                    Console.WriteLine(string.Format("\t{0} : {1}", prop.Name, prop.FieldType.Name));
                }

                Console.WriteLine("Propiedades");
                foreach (PropertyInfo prop in tipo.GetProperties())
                {
                    Console.WriteLine(string.Format("\t{0} : {1}", prop.Name, prop.PropertyType.Name));
                }

                Console.WriteLine("Constructores");
                foreach (ConstructorInfo con in tipo.GetConstructors())
                {
                    Console.Write("\tConstructor: ");
                    foreach (ParameterInfo param in con.GetParameters())
                    {
                        Console.Write(string.Format("{0} : {1} ", param.Name, param.ParameterType.Name));
                    }
                    Console.WriteLine();
                }
                Console.WriteLine();
                Console.WriteLine("Metodos");
                foreach (MethodInfo met in tipo.GetMethods())
                {
                    Console.Write(string.Format("\t{0} ", met.Name));
                    foreach (ParameterInfo param in met.GetParameters())
                    {
                        Console.Write(string.Format("{0} : {1} ", param.Name, param.ParameterType.Name));
                    }
                    Console.WriteLine();
                }
                Console.WriteLine();
            }
        }
```

5. Creamos los métodos que ejecutan y crean una nueva instancia

```
private static void InstanceAndExecuteLog(string input, string toSave, string route)
{
            Assembly myAssembly = Assembly.LoadFile(input);

            //Cargo todas las clases que implementen Ilogger
            IEnumerable<Type> implementations = GetTypesInAssembly<ILogger>(myAssembly);
            //Instancio la primera de la lista(en este ejemplo la única)
            ILogger logInstance = (ILogger)Activator.CreateInstance(implementations.First());
            logInstance.Log(toSave, route);
}
```

```
private static IEnumerable<Type> GetTypesInAssembly<Interface>(Assembly myAssembly)
{
            List<Type> types = new List<Type>();
            foreach(var type in myAssembly.GetTypes())
            {
                if(typeof(Interface).IsAssignableFrom(type))
                        types.Add(type);
            }
            return types;
}
```

5. Dar pasada a como seguramente lo van a usar en el obligatorio en ejemplo avanzado

6. Pasar por IImporterInterface y los importers que lo implementan
7. Pasar por ImporterManager
    a. GetAllImporters hace reflection
    b. ImportOrders usa el GetImporterImplementations y ejecuta el ImportOrders de la interfaz
8. Mostrar en WebApi como ponemos las dlls en la carpeta Importers
9. Mostrar los nuevos endpoints y ejecución en Postman