Unidad 23

Descargar estos apunte en pdf o html

Índice

- Índice
- Serialización de Objetos
  - Persistencia
    - ¿Cómo podemos conseguir la persistencia?
    - ¿A qué serializaremos una clase?
    - Concepto de Anotación o Atributo
  - Serialización a JSON
    - ¿Qué es JSON?
    - Tipos de datos en JSON
    - JSON en CSharp
    - Propiedades Opcionales

Serialización de Objetos

Persistencia

Se define por persistencia en el mundo de la POO, como la **capacidad que tienen los objetos de sobrevivir al proceso padre que los creo**. Esto decir, que su ciclo de vida excede de la del programa que lo instanció.

La persistencia permite al programador pues almacenar, transferir y recuperar fácilmente el estado de los objetos.

¿Cómo podemos conseguir la persistencia?

La forma más común de conseguirlo es mediante la **serialización**.

La serialización es el proceso de convertir el estado de un objeto a un formato que se pueda **almacenar** o **transportar**. Normalmente el proceso producirá una **secuencia** de bytes o texto de marcado.

El complemento de la serialización es la **deserialización**, que convierte una secuencia de bytes o texto de marcado a un objeto. Ambos procesos pues permiten almacenar y transferir fácilmente datos.

¿A qué serializaremos una clase?

.NET ofrece dos tecnologías de serialización:

1. La **serialización binaria** conserva la fidelidad de tipos, lo que resulta útil para conservar el estado de un objeto entre distintas llamadas a una aplicación. Puedes serializar un objeto en una secuencia de bytes que cómo hemos visto podemos pasar a disco, la memoria, a través de la red, etc. Ejemplos ...
   - Compartir un objeto entre distintas aplicaciones '*serializándolo*' en el Portapapeles.
   - Hacer **streaming** de vídeo serializando objetos con información de vídeo comprimido a través de la red.

> ⚠ **Aviso**
>
> En las últimas versiones de .NET **se recomienda usar la serialización binaria** sólo para casos muy específicos, como la comunicación entre aplicaciones .NET y ha sido marcada como **peligrosa** y **obsoleta**. Es por eso que en estos apuntes no se tratará la serialización binaria.

2. La **serialización a lenguajes de marcado** como:
    - La **serialización a XML** donde sólo se serializan las propiedades públicas y los campos, y no conserva la fidelidad de tipos. Esto resulta útil cuando desea proporcionar o consumir datos sin restringir la aplicación que utiliza los datos.
    - La **serialización a alguna notación de objetos** estándar cada vez más comunes y cuya función sería la misma que el XML pero menos '*verbosas*' y más fáciles de leer y/o modificar para humanos. Los más comunes son:
        - **JSON**: Definir configuraciones o consumo de datos a través de microservicios web o **bases de datos NoSQL**.
        - **YAML**: Muy usado para **definir archivos de configuración de sistemas**.

> **Importante**
>
> En la actualidad, debido al auge de JavaScript y las tecnologías web, la **serialización a JSON es la más usada** en el desarrollo de aplicaciones modernas convirtiéndose en un **estándar de facto para el intercambio de datos** entre aplicaciones.

La gran mayoría de librerías de serialización de objetos en .NET y en otros lenguajes utilizan el concepto de **anotaciones** o **atributos** para definir cómo se serializará un objeto. Es por eso, que antes de empezar a serializar un objeto, deberemos entender qué son las anotaciones o atributos.

Concepto de Anotación o Atributo

> **Importante**
>
> Las **anotaciones** son muy comunes en lenguajes como Java o Kotlin y básicamente son metadatos que se pueden añadir a clases, métodos, campos, etc. y **modificar el comportamiento justo del elemento al que antecede**. En C# se les denomina **atributos** aunque este término **puede ser confuso**, pues también es usado en la Programación Orientada a Objetos para hacer referencia a los que nosotros hemos llamado **campos** y **propiedades**.

Un **atributo** en .NET es una etiqueta de la sintaxis `[nombre]` que podremos aplicar a un ensamblado, clase, constructor, enumeración, campo, interfaz, método, etc… y que **genera información en el ensamblado** en forma de metadatos heredando de la clase `Attribute`.

Por ejemplo, el atributo `[Obsolete]` es un atributo que se puede aplicar a un método o a una clase y que indica que el método o la clase están obsoletos y que no deberían usarse.

```
[Obsolete("Este método está obsoleto. Utilice en su lugar el método X")]
public void MetodoObsoleto() // Código del método }
```

Otro ejemplo sería el atributo `[NotNull]` que se puede aplicar a un parámetro de un método y que indica que el parámetro no puede ser nulo incluso si el tipo de dato del parámetro es un tipo de dato que puede ser nulo.

```
public void Metodo([NotNull] string? parametro) { // Código del método }
```

En el ejemplo anterior, el atributo `[NotNull]` indica que el parámetro `parametro` no puede ser nulo.

Serialización a JSON

La serialización a JSON es un proceso que convierte un objeto en una cadena JSON y viceversa. JSON es un formato de texto que es fácil de leer y escribir para los humanos y fácil de analizar y generar para las máquinas.

¿Qué es JSON?

Aunque este tema se tratará con más profundidad en el módulo de LM y en segundo curso. Vamos ha realizar un resumen rápido sobre dicho formato para entenderlo por por encima.

JSON (**Javascript Object Notation**) es un formato ligero de intercambio de datos entre clientes y servidores, basado en la sintaxis de Javascript para representar estructuras en forma organizada. Es un formato en texto plano independiente de todo lenguaje de programación, es más, soporta el intercambio de datos en gran variedad de lenguajes. El estándar de JSON se especifica en el **RFC 7159** y se ha estandarizado en la **ECMA-404**. Así pues, puedes consultar la especificación del RFC para profundizar más en él.

Tipos de datos en JSON

Similar a la estructuración de datos primitivos y complejos en los lenguajes de programación, JSON establece varios tipos de datos: `cadenas`, `números`, `booleanos`, `arrays` y `objetos`. El propósito es crear objetos que contengan varios atributos compuestos como pares clave valor. Donde la clave es un nombre que identifique el uso del valor que lo acompaña. Veamos un ejemplo:

El JSON de ejemplo estña compuesto por los datos de un **estudiante**. Los objetos JSON contienen sus atributos entre llaves `{}`, al igual que un bloque de código en Javascript, donde cada atributo debe ir separado por coma `,` para diferenciar cada par. La sintaxis de los pares debe contener dos puntos `:` para dividir la clave del valor. El nombre del par debe tratarse como cadena y añadirle **comillas dobles**.

```
{
   "id": 101,
   "nombre": "Carlos",
   "estaActivo": true,
   "notas": [2.3, 4.3, 5.0]
}
```

Si te fijas en nuestro ejemplo, este trae un ejemplo de cada tipo de dato:

- `id` es de tipo entero, ya que contiene un número que representa el código del estudiante.
- `nombre` es un string. Usa comillas dobles para definirlas.
- `estaActivo` es un tipo booleano que representa si el estudiante se encuentra en la institución educativa o no. Usa las palabras reservadas `true` y `false` para declarar el valor.
- `notas` es un arreglo de números reales. El conjunto de sus elementos debes incluirlos dentro de corchetes `[ ]` y separarlos por coma.

Básicamente esta sería la sintaxis, pero en los documentos se pueden dar casos más complejos como objetos anidados, arrays de objetos, etc. Por ejemplo si modificamos el ejemplo anterior para que el estudiante en lugar de una array de notas numérico tenga un array de objetos con las asignaturas y sus notas, el documento JSON quedaría así:

Fíjate que el documento tiene un **objeto raíz** que contiene el resto de objetos. En nuestro caso el objeto raíz es un estudiante. Pero también podríamos tener un array de estudiantes directamente en el documento JSON.

```
{
   "id": 101,
   "nombre": "Carlos",
   "estaActivo": true,
   "asignaturas": [
      {
         "nombre": "Matemáticas",
         "nota": 4.5
      },
      {
         "nombre": "Lengua",
         "nota": 5.0
      }
   ]
}
```

JSON en CSharp

C# dentro de las BCL y dentro del espacio de nombres `System.Text.Json` define una serie de clases que nos permiten trabajar con JSON. EN concreto nos centraremos en la clase `JsonSerializer` que nos permitirá serializar y deserializar objetos a JSON y se define en el espacio de nombres `System.Text.Json.Serialization`.

Supongamos el siguiente `record` con datos de un estudiante:

```
record Estudiante(string Nombre, string Apellido, int Edad);
```

Si la serializamos tal cual a JSON, obtendremos un JSON como el del ejemplo usando como claves los nombres de las propiedades del record en PascalCase y el convenio en JSON es usar camelCase. Para cambiar el nombre de la clave asociada a una propiedad, podemos usar la anotación/atributo `[JsonPropertyName]` de la siguiente manera ...

```
{
  "Nombre": "Carlos",
  "Apellido": "García",
  "Edad": 18
}
```

```
record Estudiante(
    [property: JsonPropertyName("nombre")]
    string Nombre,
    [property: JsonPropertyName("apellido")]
    string Apellido,
    [property: JsonPropertyName("edad")]
    int Edad);
```

```
{
  "nombre": "Carlos",
  "apellido": "García",
  "edad": 18
}
```

Si quisiéramos serializar los datos de varios estudiantes en un documento JSON con un único objeto raíz, podríamos hacerlo a través de un `record` que contenga una colección de estudiantes como puede ser una clase completa con un nombre de clase, un tutor y una colección de estudiantes.

```
record Clase(
    [property: JsonPropertyName("nombre")]
    string Nombre,
    [property: JsonPropertyName("tutor")]
    string Tutor,
    [property: JsonPropertyName("alumnos")]
    IEnumerable<Estudiante> Estudiantes)
{
    public override string ToString() =>
        $"{Nombre} ({Tutor})\n\n" +
        string.Join("\n", Estudiantes.Select(a => $"{a}"));
}
```

Para aplicar algunos de los conceptos que hemos visto en otros temas, como son el principio OCP (SOLID) y los métodos de extensión. Vamos a definir los métodos de serialización y deserialización en una clase estática que llamaremos `ClaseJson`.

```
static class ClaseJson
{
    public static Clase? Recupera(string path)
    {
        using FileStream s = new(path, FileMode.Open, FileAccess.Read);
        // Deserializamos el objeto JSON en el documento a un objeto de la clase o tipo Clase
        return JsonSerializer.Deserialize<Clase>(s);
    }

    // Método de extensión que serializa un objeto de la clase Clase a un documento JSON
    public static void Guarda(this Clase clase, string path)
    {
        using FileStream s = new(path, FileMode.Create, FileAccess.Write);
        JsonSerializerOptions options = new()
        {
            // Indentará la cadena con el JSON para hacerla más legible
            WriteIndented = true,
            // No permitirá comas al final de los arrays
            AllowTrailingCommas = false,
            // Codificará los caracteres Unicode básicos y los de la página de códigos ISO-8859-1 (Latin-1).
            // lo normal es hacer Encoder = JavaScriptEncoder.Create(UnicodeRanges.All),
            Encoder = JavaScriptEncoder.Create(
                UnicodeRanges.BasicLatin,
                UnicodeRanges.Latin1Supplement),
            // Ignorará una propiedad si su valor es null
            DefaultIgnoreCondition = JsonIgnoreCondition.WhenWritingNull
        };
        JsonSerializer.Serialize(s, clase, options);
    }
}
```

Ahora ya podríamos serializar y deserializar objetos de la clase `Clase` a JSON de la siguiente manera (Puedes descargar el código completo desde **este enlace**):

```
static class Program
{
    static void Main()
    {
        Clase clase = new(
            Nombre: "1º DAM",
            Tutor: "Juan",
            Estudiantes: [
                new("Pepa", "Pérez", 25),
                new("María", "Peláez", 22),
                new("Rosa", "López", 26)
            ]
        );

        clase.Guarda("1DAM.json");
        Clase? c = ClaseJson.Recupera("1DAM.json");
        Console.WriteLine(
            c?.ToString()
            ?? "No se ha podido recuperar la clase")
    }
}
```

```
{
    "nombre": "1º DAM",
    "tutor": "Juan",
    "alumnos": [
        {
            "nombre": "Pepa",
            "apellido": "Pérez",
            "edad": 25
        },
        {
            "nombre": "María",
            "apellido": "Peláez",
            "edad": 22
        },
        {
            "nombre": "Rosa",
            "apellido": "López",
            "edad": 26
        }
    ]
}
```

Podremos comprobar que al recuperar el objeto de la clase `Clase` de un documento JSON, se ha recuperado correctamente.

Propiedades Opcionales

Pude darse el caso de que queramos que una propiedad de un objeto sea opcional. Para ello, fíjate que en las opciones de serialización hemos añadido la propiedad `DefaultIgnoreCondition` con el valor `JsonIgnoreCondition.WhenWritingNull`. Esto hará que si una propiedad de un objeto es `null`, no se serialice en el documento JSON. Además, al deserializar el objeto, si esta propiedad no existe en el documento JSON, se inicializará a `null`.

Por ejemplo, supongamos que modificamos el record `Estudiante` para que tenga una propiedad que será un objeto de tipo `Direccion` que **sera opcional y por tanto anulable con valor por defecto a null**.

```
record Estudiante(
    [property: JsonPropertyName("nombre")]
    string Nombre,
    [property: JsonPropertyName("apellido")]
    string Apellido,
    [property: JsonPropertyName("edad")]
    int Edad,
    [property: JsonPropertyName("direccion")]
    Direccion? Direccion = null);
```

```
record Direccion(
    [property: JsonPropertyName("calle")]
    string Calle,
    [property: JsonPropertyName("numero")]
    int Numero,
    [property: JsonPropertyName("ciudad")]
    string Ciudad,
    [property: JsonPropertyName("pais")]
    string Pais);
```

Básicamente ya podríamos serializar y deserializar objetos de la clase `Estudiante` igual que antes. Teniendo en cuenta eso sí que la propiedad `Direccion` es opcional y por tanto podría ser `null` al leerse del documento JSON.

```
Clase clase = new(
    Nombre: "1º DAM",
    Tutor: "Juan",
    Estudiantes: [
        new("Rosa", "López", 26),
        new("Juan", "Gómez", 24, new("Calle Falsa", 123, "Elche", "España"))
    ]
);
```

Se añadirán un objeto JSON anidado para la propiedad `direccion` en aquellos estudiantes que la tengan distinta de `null` y a la hora de deserializar el objeto, si la propiedad `direccion` no existe en el documento JSON, se inicializará a `null` y es por eso que sea anulable.

Por ejemplo si la edad fuera opcional, como es un campo numérico tendremos dos opciones, o bien lo inicializamos a `null` y el campo pasaría a ser de tipo `int?`.

```
[property: JsonPropertyName("edad")]
int? Edad = null,
```

```
{
    "nombre": "1º DAM",
    "tutor": "Juan",
    "alumnos": [
        {
            "nombre": "Rosa",
            "apellido": "López",
            "edad": 26
        },
        {
            "nombre": "Juan",
            "apellido": "Gómez",
            "edad": 24,
            "direccion": {
                "calle": "Calle Falsa",
                "numero": 123,
                "ciudad": "Elche",
                "pais": "España"
            }
        }
    ]
}
```