Introduction to Serialization

Last update:
Lesya Klakovych
August 2016
Reviewed by Nazar Ivchenko

Agenda

- What is Serialization?
- Serialization in .NET
- Binary serialization
- Custom serialization
- XML Serialization in C#
- Using DataContract
- Serialization in JSON format

What is Serialization?

- **Serialization** is the process of transforming an object or object graph that you have in-memory into a stream of bytes or text.
- **Deserialization** is the opposite. You take some bytes or text and transform them into an object.

```
[Serializable]
public class Person
{
…
}
```

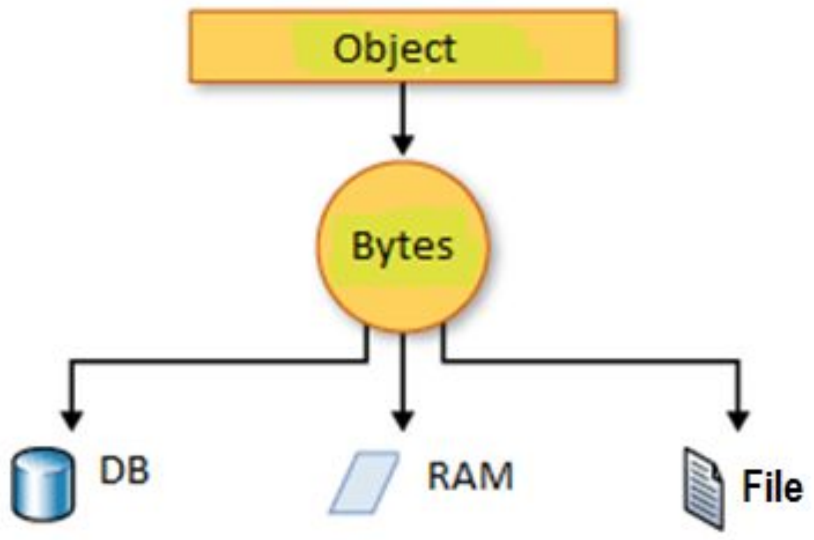
```
Person st1 = new Person();
st1.FirstName = "Iryna";
st1.LastName = "Koval";
st1.BirthDate = new DateTime(1981, 8, 17);
```

Deserialization

Serialization

AC ED 00 05 73 72 00 0A 53 65 72 69 61 6C 54 65
73 74 A0 0C 34 00 FE B1 DD F9 02 00 02 42 00 05
63 6F 75 6E 74 42 00 07 76 65 72 73 69 6F 6E 78
70 00 64

Serialization in .NET



Serialization in .NET

- .NET Framework has classes (in the ***System.Runtime.Serialization*** and ***System.Xml.Serialization*** namespaces) that support:
  - **binary,**
  - **XML,**
  - **JSON**,
  - **own custom serialization**.

- The .NET Framework offers three serialization mechanisms that you can use by default:
  - ***BinaryFormatter***
  - ***XmlSerializer***
  - ***DataContractSerializer***

Binary serialization

- In **binary** serialization all items are serialized, **even private field and read-only**, increasing productivity.
- In **binary** serialization, there is used a binary encoding to provide a compact object serialization for storage or transmission in a network flows based on sockets.
- It is not suitable for data transmission through the firewall, but provides better performance while saving data.
- namespace **System.Runtime.Serialization.Formatters.Binary**
- classes **BinaryFormatter** and **SoapFormatter** .

BinaryFormatter

```
[Serializable]
class Person {
   private int _id;
   public string FirstName;
   public string LastName;
   public void SetId(int id)
  {
     _id = id;
  }
}
```

```
Person person = new Person();
person.SetId(1);
person.FirstName = "Joe";
person.LastName = "Smith";

IFormatter formatter = new BinaryFormatter();
Stream stream = new FileStream("Person.bin",
 FileMode.Create, FileAccess.Write, FileShare.None);

formatter.Serialize(stream, person);

stream.Close();
```

```
stream = new FileStream("Person.bin",
         FileMode.Open,FileAccess.Read,FileShare.Read);

Person person2 =(Person)formatter.Deserialize(stream);

stream.Close();
```

BinaryFormatter

- To indicate that instances of this type can be serialized, mark it with the **[Serializable] attribute**. When you try to serialize the type that has no such attribute, a **SerializationException** occurs.
- If you do not want to serialize the fields within a class, apply the **[NonSerialized] attribute**.
- If a serializable class contains references to objects of other classes that are marked with a **[Serializable]** attribute, those objects are also serializable.
- the **[OptionalField]** attribute is used to make sure that the binary serializer knows that a field is added in a later version and that earlier serialized objects won’t contain this field

Custom serialization. ISerializable

- With a **custom serialization**, you can specify exactly which objects will be serialized, and how they will be serialized.
- This class must be marked with the **SerializableAttribute** attribute and implement the **Iserializable interface** .
- **ISerializable** interface: the **Formatter** calls the **GetObjectData()** at serialization time and populates the supplied **SerializationInfo** with all the data required to represent the object . For the custom deserialization, you should **use a custom constructor**.

ISerializable. Example of implementation

```
[Serializable]
public class Person : ISerializable
{
    private int _id;
    public string FirstName;
    public string LastName;
    public void SetId(int id)
    {
      _id = id;
    }
    public Person() { }
    public Person(SerializationInfo info, StreamingContext context)
    {
            FirstName = info.GetString("custom field 1");
            LastName = info.GetString("custom field 2");
     }

    public void GetObjectData(SerializationInfo info, StreamingContext context)
   {
            info.AddValue("custom field 1", FirstName);
            info.AddValue("custom field 2", LastName);
    }
}
```

Custom serialization. Using attributes

- Add attribute before a custom method that manipulates the object's data during and upon completion of serialization and deserialization.
- **OnDeserializedAttribute, OnDeserializingAttribute, OnSerializedAttribute, and OnSerializingAttribute.**

```
[OnSerializing()]
internal void OnSerializingMethod(StreamingContext context)
  { FirstName = "Bob";}

[OnSerialized()]
internal void OnSerializedMethod(StreamingContext context)
   {  FirstName = "Serialize Complete"; }

[OnDeserializing()]
internal void OnDeserializingMethod(StreamingContext context)
   {    FirstName = "John"; }

[OnDeserialized()]
internal void OnDeserializedMethod(StreamingContext context)
   { FirstName = "Deserialize Complete"; }
```

XMLSerializer

- The ***XmlSerializer*** (namespace **System.Xml.Serialization)** was created with the idea of Simple Object Access Protocol (SOAP) messaging in mind. SOAP is a protocol for exchanging information with web services. It uses XML as the format for messages. XML is readable by both humans and machines, and it is independent of the environment it is used in.
- To **serialize** an object:
- Create the object and set its public fields and properties.
- Construct a **XmlSerializer** using the type of the object.
- Call the **Serialize** method to generate either an XML stream or a file representation of the object's public properties and fields.

```
Person2 st1 = new Person2();
// Insert code to set properties and fields of the object.


XmlSerializer xmlser = new XmlSerializer(typeof(Person2));
Stream serialStream = new FileStream("person.xml", FileMode.Create);

xmlser.Serialize(serialStream, st1);
```

SoftServe

XMLSerializer

```
// Displays
//<?xml version="1.0" encoding="utf-16"?>
//<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
// xmlns:xsd="http://www.w3.org/2001/XMLSchema">
// <FirstName>John</FirstName>
// <LastName>Doe</LastName>
//</Person>
```

- To **deserialize** an object:
  - Construct a **XmlSerializer** using the type of the object to deserialize.
  - Call the **Deserialize** method to produce a replica of the object. After deserialization you must cast the returned object to the type of the original

```
serialStream = new FileStream("person.xml", FileMode.Open);

Person2 st2 = xmlser.Deserialize(serialStream) as Person2;

Console.WriteLine(st2);
```

XMLSerializer

- You can configure how the XmlSerializer serializes your type by using attributes. These attributes are defined in the *System.Xml.Serialization* namespace :
  - ***XmlIgnore -*** can be used to make sure that an element is not serialized
  - ***XmlAttribute -*** you can map a member to an attribute on its parent node.
  - ***XmlElement –*** *by default*
  - ***XmlArray -*** *is* used when serializing collections.
  - ***XmlArrayItem -*** is used when serializing collections.

SoftServe Complex and derived types serialization

```
[Serializable]
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Age { get; set; }
}
```

```
[Serializable]
public class Order
{
     [XmlAttribute]
     public int ID { get; set; }

    [XmlIgnore]
     public bool IsDirty { get; set; }

     [XmlArray("Lines")]
     [XmlArrayItem("OrderLine")]
     public List<OrderLine> OrderLines { get; set; }
}
```

```
[Serializable]
public class VIPOrder : Order
{
    public string Description { get; set;}
}
[Serializable]
public class OrderLine
{
    [XmlAttribute]
     public int ID { get; set; }

     [XmlAttribute]
     public int Amount { get; set; }

      [XmlElement("OrderedProduct")]
      public Product Product { get; set; }
}
```

```
[Serializable]
public class Product
{
    [XmlAttribute]
    public int ID { get; set; }

    public decimal Price { get; set; }
    public string Description { get; set; }}
```

SoftServe Complex and derived types serialization

```
private static Order CreateOrder()
{
    Product p1 = new Product { ID = 1, Description = "p2", Price = 9 };
    Product p2 = new Product { ID = 2, Description = "p3", Price = 6 };
    Order order = new VIPOrder { ID = 4, Description = "Order for John Doe. Use the nice giftwrap",
                                 OrderLines = new List<OrderLine> {
                                                  new OrderLine { ID = 5, Amount = 1, Product = p1},
                                                  new OrderLine { ID = 6 ,Amount = 10, Product = p2},
                                                              }
                                };
     return order;
}

XmlSerializer serializer = new XmlSerializer(typeof(Order), new Type[] { typeof(VIPOrder) });
string xml;
using (StringWriter stringWriter = new StringWriter())
{
     Order order = CreateOrder();
     serializer.Serialize(stringWriter, order);
      xml = stringWriter.ToString();
}
using (StringReader stringReader = new StringReader(xml))
    { Order o = (Order)serializer.Deserialize(stringReader);
        // Use the order}
```

Using *DataContract*

- DataContract is used when you use WCF.
- The ***DataContractSerializer*** is used by WCF to serialize your objects to **XML or JSON.**
- You should use ***DataContractAttribute*** instead of *SerializableAttribute*.
- The class members **are not serialized by default**. You have to explicitly mark them with the ***DataMember*** attribute.
- As with binary serialization, you can use ***OnDeserializedAttribute*, *OnDeserializingAttribute*, *OnSerializedAttribute*, and *OnSerializingAttribute*** to configure the four phases of the serialization and deserialization process.

Using *DataContract*

```
[DataContract]
public class PersonDataContract
{
     [DataMember]
     public int Id { get; set; }

      [DataMember]
      public string Name { get; set; }

       private bool isDirty = false;}
```

- You can use the *DataContractSerializer* from the *System.Runtime.Serialization* namespace in the same way you used the *XmlSerializer* and *BinarySerializer*.
- You need to specify a *Stream* object that has the input or output when serializing or deserializing an object.

Using *DataContract*

```
PersonDataContract p = new PersonDataContract{ Id = 1, Name = “John Doe”};

using (Stream stream = new FileStream(“data.xml”, FileMode.Create))
{
    DataContractSerializer ser = new DataContractSerializer(typeof(PersonDataContract));
    ser.WriteObject(stream, p);
}

using (Stream stream = new FileStream(“data.xml”, FileMode.Open))
{
    DataContractSerializer ser = new DataContractSerializer(typeof(PersonDataContract));
    PersonDataContract result = (PersonDataContract)ser.ReadObject(stream);
}
```

NetDataContractSerializer

Diffrence between DataContractSerializer and NetDataContractSerializer is that the NetDataContractSerializer passes type information in the XML which allows you to create a tighter .NET to .NET implementation.

Here is the DataContractSerializer version of the Person data

```
<Customer xmlns="http://www.contoso.com"
xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
  <FirstName>Zighetti</FirstName>
  <ID>101</ID>
  <LastName>Barbara</LastName>
</Customer>
```

And here is the version from the NetDataContractSerializer

```
<Customer z:Id="1" z:Type="NetDCS.Person" z:Assembly="NetDCS, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=null" xmlns="http://www.contoso.com"
xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/">
  <FirstName z:Id="2">Zighetti</FirstName>
  <ID>101</ID>
  <LastName z:Id="3">Barbara</LastName>
</Customer>
```

As you can see the NetDataContractSerializer included information about the type and the assembly which can be used on the client side.

JSON Serialization

- We can use **DataContractJsonSerializer** to serialize type instance to JSON string and deserialize JSON string to type instance
- **DataContractJsonSerializer** is under **System.Runtime.Serialization.Json** namespace.
- It is included in ***System.ServiceModel.Web.dll*** in .NET Framework 3.5 and **System.Runtime.Serialization** in .NET Framework 4.0. We need to add it as reference
- **http://www.codeproject.com/Articles/272335/JSON-Serialization-and-Deserialization-in-ASP-NET#**

Demonstration 4. Json Serialization. class Person

```
[DataContract]
    internal class Person
    {
        [DataMember]
        internal string name;

        [DataMember]
        internal int age;
    }
```

```
Person p = new Person();
p.name = "John";
p.age = 42;
MemoryStream stream1 = new MemoryStream();
DataContractJsonSerializer ser = new
DataContractJsonSerializer(typeof(Person));
ser.WriteObject(stream1, p);
stream1.Position = 0;
StreamReader sr = new StreamReader(stream1);
Console.Write("JSON form of Person object: ");
Console.WriteLine(sr.ReadToEnd());
```

Demonstration 4. Json Deserialization. class Person

```
stream1.Position = 0;

Person p2 = (Person)ser.ReadObject(stream1);

Console.Write("Deserialized back, got name=");
Console.Write(p2.name);
Console.Write(", age=");
Console.WriteLine(p2.age);
```

JSON Brief Introduction

- JSON (JavaScript Object Notation) is one lightweight data exchange format.
- JSON is "name/value" assembly. Its structure is made up with {}, [], comma, colon and double quotation marks. And it includes the following data types: Object, Number, Boolean, String, Array, NULL.
- JSON has three styles:
- **1. Object:** An unordered "name/value" assembly. An object begins with "{" and ends with "}". Behind each "name", there is a colon. And comma is used to separate much "name/value". For example:

```
var user={"name":"Tom","gender":"Male","birthday":"1983-8-8"}
```

- **2. Array:** Value order set. An array begins with "[" and end with "]". And values are separated with comma. For example:

```
var userlist=[{"user":{"name":"Tom","gender":"Male","birthday":"1983-8-8"}},
 {"user":{"name":"Lucy","gender":"Female","birthday":"1984-7-7"}}]
```

- **3. String:** Any quantity unicode character assembly which is enclosed with quotation marks. It uses backslash to escape.

Questions?