Serialization of an abstract class

Asked 10 years ago Active 10 months ago Viewed 20k times

13

8

I'm trying to serialize, and I am facing a problem with an abstact class.

I googled for an answer, and I found this blogitem. I tried that and that work.

Ok, very nice. But check out the comment on the item:

> This methodology seems to be hiding the true problem and that is an inaccurate implementation of OO design patterns, namely the factory pattern.
>
> Having to change the base class to reference any new factory class is self-defeating.
>
> With a little after-thought, the code can be changed to where any derived type can be associated with the abstract class (through the miracle of interfaces) and no XmlInclude would be required.
>
> I suggest further research into factory patterns which seems to be what you are trying to implement here.

What is commenter talking about? He is kinda vague. Can someone explain it more in detail (with an example)? Or is he just talking nonsense?

Update (after reading the first answer)

Why does the commentor talk about

> factory pattern

and

> the code can be changed to where any derived type can be associated with the abstract class (through the miracle of interfaces)

?

```
public interface IWorkaround
{
    void Method();
}

public class SomeBase : IWorkaround
{
    public void Method()
    {
        // some logic here
    }
}

public class SomeConcrete : SomeBase, IWorkaround
{
    public new void Method()
    {
        base.Method();
    }
}
```

.net serialization abstract

edited Aug 26 '09 at 12:55

asked Aug 25 '09 at 9:14

Natrium

24.5k 15 51 70

2 Answers

40

He is both right and wrong at the same time.

With things like `BinaryFormatter`, this is a non-issue; the serialized stream contains full type metadata, so if you have:

```
[Serializable] abstract class SomeBase {}
[Serializable] class SomeConcrete : SomeBase {}
...
SomeBase obj = new SomeConcrete();
```

and serialize `obj`, then it includes "I'm a `SomeConcrete`" in the stream. This makes life simple, but is verbose, especially when repeated. It is also brittle, as it demands the same implementation when deserializing; bad for either different client/server implementations, or for long-term storage.

With `XmlSerializer` (which I guess the blog is talking about), there is no metadata - but the element names (or the `xsi:type` attributes) are used to help identify which is used. For this to work, the serializer **needs to know in advance** what names map to which types.

The simplest way to do this is to decorate the base-class with the subclasses we know about. The serializer can then inspect each of these (and any additional xml-specific attributes) to figure out that when it sees a `<someConcreteType>` element, that maps to a `SomeConcrete` instance (note that the names don't need to match, so it can't just look for it by name).

```
[XmlInclude(typeof(SomeConcrete))]
public abstract class SomeBase {}
public class SomeConcrete : SomeBase {}
...
SomeBase obj = new SomeConcrete();
XmlSerializer ser = new XmlSerializer(typeof(SomeBase));
ser.Serialize(Console.Out, obj);
```

However, if he is a purist (or the data isn't available), then there is an alternative; you can specify all this data *separately* via the overloaded constructor to `XmlSerializer`. For example, you might lookup the set of known subtypes from configuration (or maybe an IoC container), and setup the constructor manually. This isn't very tricky, but it is tricky enough that it isn't worth it unless you **actually need it**.

```
public abstract class SomeBase { } // no [XmlInclude]
public class SomeConcrete : SomeBase { }
...
SomeBase obj = new SomeConcrete();
Type[] extras = {typeof(SomeConcrete)}; // from config
XmlSerializer ser = new XmlSerializer(typeof(SomeBase), extras);
ser.Serialize(Console.Out, obj);
```

Additionally, with `XmlSerializer` if you go the custom ctor route, it is important to cache and re-use the `XmlSerializer` instance; otherwise a new dynamic assembly is loaded per usage - very expensive (they can't be unloaded). If you use the simple constructor it caches and re-uses the model, so only a single model is used.

YAGNI dictates that we should choose the simplest option; using `[XmlInclude]` removes the need for a complex constructor, and removes the need to worry about caching the serializer. The other option is there and is fully supported, though.

By "factory pattern", he's talking about the case where your code **doesn't know about** `SomeConcrete`, perhaps due to IoC/DI or similar frameworks; so you might have:

```
SomeBase obj = MyFactory.Create(typeof(SomeBase), someArgsMaybe);
```

Which figures out the appropriate `SomeBase` concrete implementation, instantiates it and hands it back. Obviously, if our code doesn't know about the concrete types (because they are only specified in a config file), then we can't use `XmlInclude`; but we *can* parse the config data and use the ctor approach (as above). In reality, most times `XmlSerializer` is used with POCO/DTO entities, so this is an artificial concern.

And re interfaces; same thing, but more flexible (an interface doesn't demand a type hierarchy). But `XmlSerializer` doesn't support this model. Frankly, tough; that isn't its job. Its job is to allow you to store and transport data. Not implementation. Any xml-schema generated entities won't **have** methods. Data is concrete, not abstract. As long as you think "DTO", the interface debate is a non-issue. People who are vexed by not being able to use interfaces on their boundary haven't embraced separation of concerns, i.e. they are trying to do:

```
Client runtime entities <---transport---> Server runtime entities
```

rather than the less restrictive

```
Client runtime entities <---> Client DTO <--- transport--->
          Server DTO <---> Server runtime entities
```

Now, in many (most?) cases the DTO and entities **can** be the same; but if you are trying to do something that the transport doesn't like, introduce a DTO; don't fight the serializer. The same logic applies when people are struggling to write their object:

```
class Person {
    public string AddressLine1 {get;set;}
    public string AddressLine2 {get;set;}
}
```

as xml of the form:

```
<person>
    <address line1="..." line2="..."/>
```

```
// (in a different namespace for the DTO stuff)
[XmlType("person"), XmlRoot("person")]
public class Person {
    [XmlElement("address")]
    public Address Address {get;set;}
}
public class Address {
    [XmlAttribute("line1")] public string Line1 {get;set;}
    [XmlAttribute("line2")] public string Line2 {get;set;}
}
```

This also applies to all those other niggles like:

- why do I need a parameterless constructor?
- why do I need a setter for my collection properties?
- why can't I use an immutable type?
- why must my type be public?
- how do I handle complex versioning?
- how do I handle different clients with different data layouts?
- why can't I use interfaces?
- etc, etc

You don't always have these problems; but if you do - introduce a DTO (or several) and your problems go away. Taking this back to the question about interfaces; the DTO types might not be interface-based, but your runtime/business types can be.

edited Aug 26 '09 at 13:05

answered Aug 26 '09 at 10:53

Marc Gravell ♦
821k 212 2223 2621

thank you for the exellent info. – Natrium Aug 26 '09 at 13:14

```
--------------------------------------------------------------------------------**
```

```
@JsonSerialize(using = CatAbstractSerializer.class)
public enum CatTest implements Tes{

    TYPE1(1, "Type 1"), TYPE2(2, "Type 2");

    private int id;
    private String nome;

    private CatTest(int id, String nome) {
        // TODO Auto-generated constructor stub

        this.id = id;
        this.nome = nome;


    }
    @JsonValue
    public int getId() {
        return id;
    }
    @JsonSetter
    public void setId(int id) {
        this.id = id;
    }
    @JsonValue
    public String getNome() {
        return nome;
    }
    @JsonSetter
    public void setNome(String nome) {
        this.nome = nome;
    }
    @Override
     @JsonValue
     public String toString() {
            return nome;
     }
     @JsonCreator
        public static CatTest fromValueString(String nome) {
            if(nome == null) {
                throw new IllegalArgumentException();
            }
            for(CatTest nomeSalvo : values()) {
```

```
                }
                throw new IllegalArgumentException();
            }


    }


    public interface Tes {

        @JsonValue
        int getId();

        @JsonValue
        String getNome();

        @JsonSetter
        void setId(int id);

        @JsonSetter
        void setNome(String nome);


    }

    public class CatAbstractSerializer<T extends Tes> extends JsonSerializer<T> {

        @Override
        public void serialize(T value, JsonGenerator gen, SerializerProvider serializers)
                throws IOException, JsonProcessingException {
            // TODO Auto-generated method stub

            gen.writeStartObject();
            gen.writeFieldName("id");
            gen.writeNumber(value.getId());
            gen.writeFieldName("name");
            gen.writeString(value.getNome());
            gen.writeEndObject();

        }

    }
```

answered Nov 4 '18 at 4:22

1 Example of Enum Abstract Serializer Simple example of an abstract enum ...(Java)(Spring-Boot) – Joao Pedro Leite S Lisboa Nov 4 '18 at 4:27