# ExtendedXmlSerializer Documentation

*Release 4.0*

**ExtendedXmlSerializer Contributors**

**Dec 27, 2022**

# Contents


1 Welcome! 3

2 Information 5

3 The Basics 7

4 Xml Settings 9

5 Serialization 11

6 Deserialization 13

7 Classic Serialization/Deserialization 15

8 Serialization/Deserialization with Settings Overrides 17

9 Fluent API 19

10 Serialization of dictionary 21

11 Custom serialization 23

12 Deserialize old version of xml 25

13 Extensibility 29

14 Object reference and circular reference 31

15 Property Encryption 33

16 Custom Conversion 35

17 Optimized Namespaces 37

18 Implicit Namespaces/Typing 39

19 Auto-Formatting (Attributes) 41

20 Verbatim Content (CDATA) 43

21 Private Constructors 45

22 Parameterized Members and Content 47

23 Tuples 49

24 Experimental Xaml-ness: Attached Properties 51

25 Experimental Xaml-ness: Markup Extensions 53

26 How to Upgrade from v1.x to v2 55

27 History 57

28 Authors 59

29 Indices and tables 61

ExtendedXmlSerializer for .NET

# CHAPTER 1

## Welcome!

Welcome to ExtendedXmlSerializer's source repository, our little home on the webternets for ExtendedXmlSerializer. Please have a look around. If you have a question about our serializer, please do not hesitate to post a question in our issues: https://github.com/wojtpl2/ExtendedXmlSerializer/issues/new/

We will mark it as a question/discussion and talk it through with you using sample code. This process is currently being used to fill in the gaps with our documentation, which could use a little love.

Additionally, please make use of our documentation tag seen here:

https://github.com/wojtpl2/ExtendedXmlSerializer/issues?q=is%3Aissue+label%3ADocumentation

You can see the history of questions asked that we feel could eventually be added to our documentation. We'll eventually get to it, we swear.

# CHAPTER 2

# Information

Support platforms:

- .NET 4.5
- .NET Standard 2.0

Support features:

- Deserialization xml from standard *XMLSerializer* (mostly, see: https://github.com/wojtpl2/ExtendedXmlSerializer/issues/240)
- Serialization class, struct, generic class, primitive type, generic list and dictionary, array, enum
- Serialization class with property interface
- Serialization circular reference and reference Id
- Deserialization of old version of xml
- Property encryption
- Custom serializer
- Support *XmlElementAttribute*, *XmlRootAttribute*, and *XmlTypeAttribute* for identity resolution.
- Additional attribute support: *XmlIgnoreAttribute*, *XmlAttributeAttribute*, and *XmlEnumAttribute*.
- POCO - all configurations (migrations, custom serializer…) are outside the clas

Standard XML Serializer in .NET is very limited:

- Does not support serialization of class with circular reference or class with interface property.
- There is no mechanism for reading the old version of XML.
- Does not support properties that are defined with interface types.
- Does not support read-only collection properties (like Xaml does).
- Does not support parameterized constructors.
- Does not support private constructors.

- If you want create custom serializer, your class must inherit from *IXmlSerializable*. This means that your class will not be a POCO class.
- Does not support IoC

CHAPTER 3

# The Basics

Everything in *ExtendedXmlSerializer* begins with a configuration container, from which you can use to configure the serializer and ultimately create it:

```
IExtendedXmlSerializer serializer = new ConfigurationContainer()
                                        // Configure...
                                        .Create();
```

Using this simple subject class:

```
public sealed class Subject
{
    public string Message { get; set; }

    public int Count { get; set; }
}
```

The results of the default serializer will look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<Subject xmlns="clr-namespace:ExtendedXmlSerializer.Samples.Introduction;
↪assembly=ExtendedXmlSerializer.Samples">
  <Message>Hello World!</Message>
  <Count>6776</Count>
</Subject>
```

We can take this a step further by configuring the *Subject*'s Type and Member properties, which will effect how its Xml is emitted. Here is an example of configuring the *Subject*'s name to emit as *ModifiedSubject*:

```
IExtendedXmlSerializer serializer = new ConfigurationContainer().ConfigureType
↪<Subject>()
                                                                .Name("ModifiedSubject
↪")
                                                                .Create();
```

```
<?xml version="1.0" encoding="utf-8"?>
<ModifiedSubject xmlns="clr-namespace:ExtendedXmlSerializer.Samples.Introduction;
↪assembly=ExtendedXmlSerializer.Samples">
  <Message>Hello World!</Message>
  <Count>6776</Count>
</ModifiedSubject>
```

Diving a bit further, we can also configure the type's member information. For example, configuring *Subject.Message* to emit as *Text* instead:

```
IExtendedXmlSerializer serializer = new ConfigurationContainer().ConfigureType
↪<Subject>()
                                                                .Member(x => x.
↪Message)
                                                                .Name("Text")
                                                                .Create();
```

```
<?xml version="1.0" encoding="utf-8"?>
<Subject xmlns="clr-namespace:ExtendedXmlSerializer.Samples.Introduction;
↪assembly=ExtendedXmlSerializer.Samples">
  <Text>Hello World!</Text>
  <Count>6776</Count>
</Subject>
```

# CHAPTER 4

## Xml Settings

In case you want to configure the XML write and read settings via *XmlWriterSettings* and *XmlReaderSettings* respectively, you can do that via extension methods created for you to do so:

```
Subject subject = new Subject{ Count = 6776, Message = "Hello World!" };
IExtendedXmlSerializer serializer = new ConfigurationContainer().Create();
string contents = serializer.Serialize(new XmlWriterSettings {Indent = true},␣
↪subject);
// ...
```

And for reading:

```
Subject instance = serializer.Deserialize<Subject>(new XmlReaderSettings
↪{IgnoreWhitespace = false}, contents);
// ...
```

# CHAPTER 5

## Serialization

Now that your configuration container has been configured and your serializer has been created, it's time to get to the serialization.

```
IExtendedXmlSerializer serializer = new ConfigurationContainer().Create();
TestClass obj = new TestClass();
string xml = serializer.Serialize(obj);
```

CHAPTER 6

# Deserialization

```
TestClass obj2 = serializer.Deserialize<TestClass>(xml);
```

CHAPTER 7

# Classic Serialization/Deserialization

Most of the code examples that you see in this documentation make use of useful extension methods that make serialization and deserialization a snap with ‘ExtendedXmlSerializer. However, if you would like to break down into the basic, classical pattern of serialization, and deserialization, this is supported too, as seen by the following examples. Here’s serialization:

```
IExtendedXmlSerializer serializer = new ConfigurationContainer().Create();
TestClass              instance   = new TestClass();
MemoryStream           stream     = new MemoryStream();
using (XmlWriter writer = XmlWriter.Create(stream))
{
    serializer.Serialize(writer, instance);
    writer.Flush();
}

stream.Seek(0, SeekOrigin.Begin);
string contents = new StreamReader(stream).ReadToEnd();
```

And here’s how to deserialize:

```
TestClass deserialized;
MemoryStream contentStream = new MemoryStream(Encoding.UTF8.GetBytes(contents));
using (XmlReader reader = XmlReader.Create(contentStream))
{
    deserialized = (TestClass)serializer.Deserialize(reader);
}
```

CHAPTER 8

# Serialization/Deserialization with Settings Overrides

Additionally, *ExtendedXmlSerializer* also supports overrides for serialization and deserialization that allow you to pass in *XmlWriterSettings* and *XmlReaderSettings* respectively. Here's an example of this for serialization:

```
IExtendedXmlSerializer serializer = new ConfigurationContainer().Create();
TestClass              instance   = new TestClass();
MemoryStream           stream     = new MemoryStream();

string contents = serializer.Serialize(new XmlWriterSettings { /* ... */ }, stream,␣
↪instance);
```

And for deserialization:

```
MemoryStream contentStream = new MemoryStream(Encoding.UTF8.GetBytes(contents));
TestClass deserialized = serializer.Deserialize<TestClass>(new XmlReaderSettings{ /* .
↪.. */ }, contentStream);
```

CHAPTER 9

# Fluent API

ExtendedXmlSerializer use fluent API to configuration. Example:

```
IExtendedXmlSerializer serializer = new ConfigurationContainer()
    .UseEncryptionAlgorithm(new CustomEncryption())
    .Type<Person>() // Configuration of Person class
        .Member(p => p.Password) // First member
            .Name("P")
            .Encrypt()
        .Member(p => p.Name) // Second member
            .Name("T")
    .Type<TestClass>() // Configuration of another class
        .CustomSerializer(new TestClassSerializer())
    .Create();
```

# CHAPTER 10

## Serialization of dictionary

You can serialize generic dictionary, that can store any type.

```
public class TestClass
{
    public Dictionary<int, string> Dictionary { get; set; }
}
```

```
TestClass obj = new TestClass
{
    Dictionary = new Dictionary<int, string>
    {
        {1, "First"},
        {2, "Second"},
        {3, "Other"},
    }
};
```

Output XML will look like:

```
<?xml version="1.0" encoding="utf-8"?>
<TestClass xmlns="clr-namespace:ExtendedXmlSerializer.Samples.Dictianary;
↪assembly=ExtendedXmlSerializer.Samples">
  <Dictionary>
    <Item xmlns="https://extendedxmlserializer.github.io/system">
      <Key>1</Key>
      <Value>First</Value>
    </Item>
    <Item xmlns="https://extendedxmlserializer.github.io/system">
      <Key>2</Key>
      <Value>Second</Value>
    </Item>
    <Item xmlns="https://extendedxmlserializer.github.io/system">
      <Key>3</Key>
      <Value>Other</Value>
```

(continues on next page)

(continued from previous page)

```
    </Item>
  </Dictionary>
</TestClass>
```

If you use UseOptimizedNamespaces function xml will look like:

```
<?xml version="1.0" encoding="utf-8"?>
<TestClass xmlns:sys="https://extendedxmlserializer.github.io/system" xmlns:exs=
↪"https://extendedxmlserializer.github.io/v2" xmlns="clr-
↪namespace:ExtendedXmlSerializer.Samples.Dictianary;assembly=ExtendedXmlSerializer.
↪Samples">
  <Dictionary>
    <sys:Item>
      <Key>1</Key>
      <Value>First</Value>
    </sys:Item>
    <sys:Item>
      <Key>2</Key>
      <Value>Second</Value>
    </sys:Item>
    <sys:Item>
      <Key>3</Key>
      <Value>Other</Value>
    </sys:Item>
  </Dictionary>
</TestClass>
```

CHAPTER 11

# Custom serialization

If your class has to be serialized in a non-standard way:

```
public class TestClass
{
    public TestClass(string paramStr, int paramInt)
    {
        PropStr = paramStr;
        PropInt = paramInt;
    }

    public string PropStr { get; private set; }
    public int PropInt { get; private set; }
}
```

You must create custom serializer:

```
public class TestClassSerializer : IExtendedXmlCustomSerializer<TestClass>
{
    public TestClass Deserialize(XElement element)
    {
        XElement xElement = element.Member("String");
        XElement xElement1 = element.Member("Int");
        if (xElement != null && xElement1 != null)
        {
            string strValue = xElement.Value;

            int intValue = Convert.ToInt32(xElement1.Value);
            return new TestClass(strValue, intValue);
        }
        throw new InvalidOperationException("Invalid xml for class␣
↪TestClassWithSerializer");
    }

    public void Serializer(XmlWriter writer, TestClass obj)
```

(continues on next page)

(continued from previous page)

```
    {
        writer.WriteElementString("String", obj.PropStr);
        writer.WriteElementString("Int", obj.PropInt.ToString(CultureInfo.
↪InvariantCulture));
    }
}
```

Then, you have to add custom serializer to configuration of TestClass:

```
IExtendedXmlSerializer serializer = new ConfigurationContainer().Type<TestClass>()
                                                                 .CustomSerializer(new␣
↪TestClassSerializer())
                                                                 .Create();
```

CHAPTER 12

# Deserialize old version of xml

In standard *XMLSerializer* you can't deserialize XML in case you change model. In *ExtendedXMLSerializer* you can create migrator for each class separately. E.g.: If you have big class, that uses small class and this small class will be changed you can create migrator only for this small class. You don't have to modify whole big XML. Now I will show you a simple example: If you had a class:

```
public class TestClass
{
    public int Id { get; set; }
    public string Type { get; set; }
}
```

and generated XML look like:

```
<?xml version="1.0" encoding="utf-8"?>
<TestClass xmlns="clr-namespace:ExtendedXmlSerialization.Samples.MigrationMap;
↪assembly=ExtendedXmlSerializer.Samples">
  <Id>1</Id>
  <Type>Type</Type>
</TestClass>
```

Then you renamed property:

```
public class TestClass
{
    public int Id { get; set; }
    public string Name { get; set; }
}
```

and generated XML look like:

```
<?xml version="1.0" encoding="utf-8"?>
<TestClass xmlns:exs="https://extendedxmlserializer.github.io/v2" exs:version="1"␣
↪xmlns="clr-namespace:ExtendedXmlSerialization.Samples.MigrationMap;
↪assembly=ExtendedXmlSerializer.Samples">
```

(continues on next page)

(continued from previous page)

```
  <Id>1</Id>
  <Name>Type</Name>
</TestClass>
```

Then, you added new property and you wanted to calculate a new value during deserialization.

```
public class TestClass
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Value { get; set; }
}
```

and new XML should look like:

```
<?xml version="1.0" encoding="utf-8"?>
<TestClass xmlns:exs="https://extendedxmlserializer.github.io/v2" exs:version="2"
↪xmlns="clr-namespace:ExtendedXmlSerializer.Samples.MigrationMap;
↪assembly=ExtendedXmlSerializer.Samples">
  <Id>1</Id>
  <Name>Type</Name>
  <Value>Calculated</Value>
</TestClass>
```

You can migrate (read) old version of XML using migrations:

```
public class TestClassMigrations : IEnumerable<Action<XElement>>
{
    public static void MigrationV0(XElement node)
    {
        XElement typeElement = node.Member("Type");
        // Add new node
        node.Add(new XElement("Name", typeElement.Value));
        // Remove old node
        typeElement.Remove();
    }

    public static void MigrationV1(XElement node)
    {
        // Add new node
        node.Add(new XElement("Value", "Calculated"));
    }

    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();

    public IEnumerator<Action<XElement>> GetEnumerator()
    {
        yield return MigrationV0;
        yield return MigrationV1;
    }
}
```

Then, you must register your *TestClassMigrations* class in configuration

```
IExtendedXmlSerializer serializer = new ConfigurationContainer().ConfigureType
↪<TestClass>()
                                                                .AddMigration(new
↪TestClassMigrations())
```

(continues on next page)

(continued from previous page)

```
                                                                    .Create();
```

# CHAPTER 13

# Extensibility

With type and member configuration out of the way, we can turn our attention to what really makes ExtendedXmlSeralizer tick: extensibility. As its name suggests, ExtendedXmlSeralizer offers a very flexible (but albeit new) extension model from which you can build your own extensions. Pretty much all if not all features you encounter with ExtendedXmlSeralizer are through extensions. There are quite a few in our latest version here that showcase this extensibility. The remainder of this document will showcase the top features of ExtendedXmlSerializer that are accomplished through its extension system.

CHAPTER 14

# Object reference and circular reference

If you have a class:

```
public class Person
{
    public int Id { get; set; }
    public string Name { get; set; }

    public Person Boss { get; set; }
}

public class Company
{
    public List<Person> Employees { get; set; }
}
```

then you create object with circular reference, like this:

```
Person boss = new Person {Id = 1, Name = "John"};
boss.Boss = boss; //himself boss
Person worker = new Person {Id = 2, Name = "Oliver"};
worker.Boss = boss;
Company obj = new Company
{
    Employees = new List<Person>
    {
        worker,
        boss
    }
};
```

You must configure Person class as reference object:

```
IExtendedXmlSerializer serializer = new ConfigurationContainer().ConfigureType<Person>
↪()
```

(continues on next page)

(continued from previous page)

```
                                                                    .EnableReferences(p =>
↪ p.Id)
                                                                    .Create();
```

Output XML will look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<Company xmlns="clr-namespace:ExtendedXmlSerializer.Samples.ObjectReference;
↪assembly=ExtendedXmlSerializer.Samples">
  <Employees>
    <Capacity>4</Capacity>
    <Person Id="2">
      <Name>Oliver</Name>
      <Boss Id="1">
        <Name>John</Name>
        <Boss xmlns:exs="https://extendedxmlserializer.github.io/v2" exs:entity="1" />
      </Boss>
    </Person>
    <Person xmlns:exs="https://extendedxmlserializer.github.io/v2" exs:entity="1" />
  </Employees>
</Company>
```

CHAPTER 15

# Property Encryption

If you have a class with a property that needs to be encrypted:

```
public class Person
{
    public string Name { get; set; }
    public string Password { get; set; }
}
```

You must implement interface IEncryption. For example, it will show the Base64 encoding, but in the real world better to use something safer, eg. RSA.:

```
public class CustomEncryption : IEncryption
{
    public string Parse(string data)
        => Encoding.UTF8.GetString(Convert.FromBase64String(data));

    public string Format(string instance)
        => Convert.ToBase64String(Encoding.UTF8.GetBytes(instance));
}
```

Then, you have to specify which properties are to be encrypted and register your IEncryption implementation.

```
IExtendedXmlSerializer serializer = new ConfigurationContainer().
↪UseEncryptionAlgorithm(new CustomEncryption())
                                                                .ConfigureType<Person>
↪()
                                                                .Member(p => p.
↪Password)
                                                                .Encrypt()
                                                                .Create();
```

CHAPTER 16

# Custom Conversion

ExtendedXmlSerializer does a pretty decent job (if we do say so ourselves) of composing and decomposing objects, but if you happen to have a type that you want serialized in a certain way, and this type can be destructured into a *string*, then you can register a custom converter for it.

Using the following:

```
public sealed class CustomStructConverter : IConverter<CustomStruct>
{
    public static CustomStructConverter Default { get; } = new␣
↪CustomStructConverter();
    CustomStructConverter() {}

    public bool IsSatisfiedBy(TypeInfo parameter) => typeof(CustomStruct).
↪GetTypeInfo()
                                                                          .
↪IsAssignableFrom(parameter);

    public CustomStruct Parse(string data) =>
        int.TryParse(data, out int number) ? new CustomStruct(number) : CustomStruct.
↪Default;

    public string Format(CustomStruct instance) => instance.Number.ToString();
}

public struct CustomStruct
{
    public static CustomStruct Default { get; } = new CustomStruct(6776);

    public CustomStruct(int number)
    {
        Number = number;
    }
    public int Number { get; }
}
```

Register the converter:

```
IExtendedXmlSerializer serializer = new ConfigurationContainer().
↪Register(CustomStructConverter.Default).Create();
CustomStruct subject = new CustomStruct(123);
string contents = serializer.Serialize(subject);
// ...
```

```
<?xml version="1.0" encoding="utf-8"?>
<CustomStruct xmlns="clr-namespace:ExtendedXmlSerializer.Samples.Extensibility;
↪assembly=ExtendedXmlSerializer.Samples">123</CustomStruct>
```

CHAPTER 17

# Optimized Namespaces

By default Xml namespaces are emitted on an "as needed" basis:

```
<?xml version="1.0" encoding="utf-8"?>
<List xmlns:exs="https://extendedxmlserializer.github.io/v2" exs:arguments="Object"␣
↪xmlns="https://extendedxmlserializer.github.io/system">
  <Capacity>4</Capacity>
  <Subject xmlns="clr-namespace:ExtendedXmlSerializer.Samples.Extensibility;
↪assembly=ExtendedXmlSerializer.Samples">
    <Message>First</Message>
  </Subject>
  <Subject xmlns="clr-namespace:ExtendedXmlSerializer.Samples.Extensibility;
↪assembly=ExtendedXmlSerializer.Samples">
    <Message>Second</Message>
  </Subject>
  <Subject xmlns="clr-namespace:ExtendedXmlSerializer.Samples.Extensibility;
↪assembly=ExtendedXmlSerializer.Samples">
    <Message>Third</Message>
  </Subject>
</List>
```

But with one call to the *UseOptimizedNamespaces* call, namespaces get placed at the root of the document, thereby reducing document footprint:

```
IExtendedXmlSerializer serializer = new ConfigurationContainer().
↪UseOptimizedNamespaces()
                                                                .Create();
List<object> subject = new List<object>
                {
                    new Subject {Message = "First"},
                    new Subject {Message = "Second"},
                    new Subject {Message = "Third"}
                };
string contents = serializer.Serialize(subject);
// ...
```

```
<?xml version="1.0" encoding="utf-8"?>
<List xmlns:ns1="clr-namespace:ExtendedXmlSerializer.Samples.Extensibility;
↪assembly=ExtendedXmlSerializer.Samples" xmlns:exs="https://extendedxmlserializer.
↪github.io/v2" exs:arguments="Object" xmlns="https://extendedxmlserializer.github.io/
↪system">
  <Capacity>4</Capacity>
  <ns1:Subject>
    <Message>First</Message>
  </ns1:Subject>
  <ns1:Subject>
    <Message>Second</Message>
  </ns1:Subject>
  <ns1:Subject>
    <Message>Third</Message>
  </ns1:Subject>
</List>
```

# CHAPTER 18

## Implicit Namespaces/Typing

If you don't like namespaces at all, you can register types so that they do not emit namespaces when they are rendered into a document:

```
IExtendedXmlSerializer serializer = new ConfigurationContainer().
↪EnableImplicitTyping(typeof(Subject))
                                                                .Create();
Subject subject = new Subject{ Message = "Hello World!  No namespaces, yay!" };
string contents = serializer.Serialize(subject);
// ...
```

```
<?xml version="1.0" encoding="utf-8"?>
<Subject>
  <Message>Hello World!  No namespaces, yay!</Message>
</Subject>
```

CHAPTER 19

# Auto-Formatting (Attributes)

The default behavior for emitting data in an Xml document is to use elements, which can be a little chatty and verbose:

```
IExtendedXmlSerializer serializer = new ConfigurationContainer().
↪UseOptimizedNamespaces()
                                                                .Create();
List<object> subject = new List<object>
                {
                    new Subject {Message = "First"},
                    new Subject {Message = "Second"},
                    new Subject {Message = "Third"}
                };
string contents = serializer.Serialize(subject);
// ...
```

```
<?xml version="1.0" encoding="utf-8"?>
<SubjectWithThreeProperties xmlns="clr-namespace:ExtendedXmlSerializer.Samples.
↪Extensibility;assembly=ExtendedXmlSerializer.Samples">
  <Number>123</Number>
  <Message>Hello World!</Message>
  <Time>2018-05-26T11:52:19.4981212-04:00</Time>
</SubjectWithThreeProperties>
```

Making use of the *UseAutoFormatting* call will enable all types that have a registered *IConverter* (convert to string and back) to emit as attributes:

```
<?xml version="1.0" encoding="utf-8"?>
<SubjectWithThreeProperties Number="123" Message="Hello World!" Time="2018-05-
↪26T11:52:19.4981212-04:00" xmlns="clr-namespace:ExtendedXmlSerializer.Samples.
↪Extensibility;assembly=ExtendedXmlSerializer.Samples" />
```

CHAPTER 20

# Verbatim Content (CDATA)

If you have an element with a member that can hold lots of data, or data that has illegal characters, you configure it to be a verbatim field and it will emit a CDATA section around it:

```
IExtendedXmlSerializer serializer = new ConfigurationContainer().Type<Subject>()
                                                                 .Member(x => x.
↪Message)
                                                                 .Verbatim()
                                                                 .Create();
Subject subject = new Subject {Message = @"<{""Ilegal characters and such""}>"};
string contents = serializer.Serialize(subject);
// ...
```

```
<?xml version="1.0" encoding="utf-8"?>
<Subject xmlns="clr-namespace:ExtendedXmlSerializer.Samples.Extensibility;
↪assembly=ExtendedXmlSerializer.Samples">
  <Message><![CDATA[<{"Ilegal characters and such"}>]]></Message>
</Subject>
```

You can also denote these fields with an attribute and get the same functionality:

```
public sealed class VerbatimSubject
{
    [Verbatim]
    public string Message { get; set; }
}
```

CHAPTER 21

# Private Constructors

One of the limitations of the classic *XmlSerializer* is that it does not support private constructors, but *ExtendedXmlSerializer* does via its *EnableAllConstructors* call:

```
public sealed class SubjectByFactory
{
    public static SubjectByFactory Create(string message) => new␣
↪SubjectByFactory(message);

    SubjectByFactory() : this(null) {} // Used by serializer.

    SubjectByFactory(string message) => Message = message;

    public string Message { get; set; }
}
```

```
IExtendedXmlSerializer serializer = new ConfigurationContainer().
↪EnableAllConstructors()
                                                                .Create();
SubjectByFactory subject = SubjectByFactory.Create("Hello World!");
string contents = serializer.Serialize(subject);
// ...
```

```
<?xml version="1.0" encoding="utf-8"?>
<SubjectByFactory xmlns="clr-namespace:ExtendedXmlSerializer.Samples.Extensibility;
↪assembly=ExtendedXmlSerializer.Samples">
  <Message>Hello World!</Message>
</SubjectByFactory>
```

CHAPTER 22

# Parameterized Members and Content

Taking this concept bit further leads to a favorite feature of ours in *ExtendedXmlSerlializer*. The classic serializer only supports parameterless public constructors. With *ExtendedXmlSerializer*, you can use the *EnableParameterizedContent* call to enable parameterized parameters in the constructor that by convention have the same name as the property for which they are meant to assign:

```
public sealed class ParameterizedSubject
{
    public ParameterizedSubject(string message, int number, DateTime time)
    {
        Message = message;
        Number = number;
        Time = time;
    }

    public string Message { get; }
    public int Number { get; }
    public DateTime Time { get; }
}
```

```
IExtendedXmlSerializer serializer = new ConfigurationContainer().
↪EnableParameterizedContent()
                                                                .Create();
ParameterizedSubject subject = new ParameterizedSubject("Hello World!", 123, DateTime.
↪Now);
string contents = serializer.Serialize(subject);
// ...
```

```
<?xml version="1.0" encoding="utf-8"?>
<ParameterizedSubject xmlns="clr-namespace:ExtendedXmlSerializer.Samples.
↪Extensibility;assembly=ExtendedXmlSerializer.Samples">
  <Message>Hello World!</Message>
  <Number>123</Number>
  <Time>2018-05-26T11:52:19.7551187-04:00</Time>
</ParameterizedSubject>
```

CHAPTER 23

# Tuples

By enabling parameterized content, it opens up a lot of possibilities, like being able to serialize Tuples. Of course, serializable Tuples were introduced recently with the latest version of C#. Here, however, you can couple this with our member-naming funtionality and provide better naming for your tuple properties:

```
IExtendedXmlSerializer serializer = new ConfigurationContainer().
↪EnableParameterizedContent()
                                                                .Type<Tuple<string>>()
                                                                .Member(x => x.Item1)
                                                                .Name("Message")
                                                                .Create();
Tuple<string> subject = Tuple.Create("Hello World!");
string contents = serializer.Serialize(subject);
// ...
```

```
<?xml version="1.0" encoding="utf-8"?>
<Tuple xmlns:exs="https://extendedxmlserializer.github.io/v2" exs:arguments="string"␣
↪xmlns="https://extendedxmlserializer.github.io/system">
  <Message>Hello World!</Message>
</Tuple>
```

CHAPTER 24

# Experimental Xaml-ness: Attached Properties

We went ahead and got a little cute with v2 of *ExtendedXmlSerializer*, adding support for Attached Properties on objects in your serialized object graph. But instead of constraining it to objects that inherit from *DependencyObject*, *every* object can benefit from it. Check it out:

```
sealed class NameProperty : ReferenceProperty<Subject, string>
{
    public const string DefaultMessage = "The Name Has Not Been Set";

    public static NameProperty Default { get; } = new NameProperty();
    NameProperty() : base(() => Default, x => DefaultMessage) {}
}

sealed class NumberProperty : StructureProperty<Subject, int>
{
    public const int DefaultValue = 123;

    public static NumberProperty Default { get; } = new NumberProperty();
    NumberProperty() : base(() => Default, x => DefaultValue) {}
}
```

```
IExtendedXmlSerializer serializer = new ConfigurationContainer()
                                    .EnableAttachedProperties(NameProperty.Default,␣
↪NumberProperty.Default)
                                    .Create();
Subject subject = new Subject {Message = "Hello World!"};
subject.Set(NameProperty.Default, "Hello World from Attached Properties!");
subject.Set(NumberProperty.Default, 123);

string contents = serializer.Serialize(subject);
// ...
```

```
<?xml version="1.0" encoding="utf-8"?>
<Subject xmlns="clr-namespace:ExtendedXmlSerializer.Samples.Extensibility;
↪assembly=ExtendedXmlSerializer.Samples">
```

(continues on next page)

(continued from previous page)

```
  <Message>Hello World!</Message>
  <NameProperty.Default>Hello World from Attached Properties!</NameProperty.Default>
  <NumberProperty.Default>123</NumberProperty.Default>
</Subject>
```

(Please note that this feature is experimental, but please try it out and let us know what you think!)

CHAPTER 25

# Experimental Xaml-ness: Markup Extensions

Saving the best feaure for last, we have experimental support for one of Xaml's greatest features, Markup Extensions:

```
sealed class Extension : IMarkupExtension
{
    const string Message = "Hello World from Markup Extension! Your message is: ",␣
↪None = "N/A";

    readonly string _message;

    public Extension() : this(None) {}

    public Extension(string message)
    {
        _message = message;
    }

    public object ProvideValue(IServiceProvider serviceProvider) => string.
↪Concat(Message, _message);
}
```

```
string contents =
    @"<?xml version=""1.0"" encoding=""utf-8""?>
        <Subject xmlns=""clr-namespace:ExtendedXmlSerializer.Samples.Extensibility;
↪assembly=ExtendedXmlSerializer.Samples""
        Message=""{Extension 'PRETTY COOL HUH!!!'}"" />";
IExtendedXmlSerializer serializer = new ConfigurationContainer().
↪EnableMarkupExtensions()
                                                                    .Create();
Subject subject = serializer.Deserialize<Subject>(contents);
Console.WriteLine(subject.Message); // "Hello World from Markup Extension! Your␣
↪message is: PRETTY COOL HUH!!!"
```

(Please note that this feature is experimental, but please try it out and let us know what you think!)

CHAPTER 26

# How to Upgrade from v1.x to v2

Finally, if you have documents from v1, you will need to upgrade them to v2 to work. This involves reading the document in an instance of v1 serializer, and then writing it in an instance of v2 serializer. We have provided the *ExtendedXmlSerializer.Legacy* nuget package to assist in this goal.

```
<?xml version="1.0" encoding="utf-8"?><ArrayOfSubject><Subject type=
↪"ExtendedXmlSerializer.Samples.Introduction.Subject"><Message>First</Message><Count>
↪0</Count></Subject><Subject type="ExtendedXmlSerializer.Samples.Introduction.Subject
↪"><Message>Second</Message><Count>0</Count></Subject><Subject type=
↪"ExtendedXmlSerializer.Samples.Introduction.Subject"><Message>Third</Message><Count>
↪0</Count></Subject></ArrayOfSubject>
```

```
ExtendedXmlSerialization.ExtendedXmlSerializer legacySerializer = new␣
↪ExtendedXmlSerialization.ExtendedXmlSerializer();
string content = File.ReadAllText(@"bin\Upgrade.Example.v1.xml"); // Path to your␣
↪legacy xml file.
List<Subject> subject = legacySerializer.Deserialize<List<Subject>>(content);

// Upgrade:
IExtendedXmlSerializer serializer = new ConfigurationContainer().Create();
string contents = serializer.Serialize(new XmlWriterSettings {Indent = true},␣
↪subject);
File.WriteAllText(@"bin\Upgrade.Example.v2.xml", contents);
// ...
```

```
<?xml version="1.0" encoding="utf-8"?>
<List xmlns:ns1="clr-namespace:ExtendedXmlSerializer.Samples.Introduction;
↪assembly=ExtendedXmlSerializer.Samples" xmlns:exs="https://extendedxmlserializer.
↪github.io/v2" exs:arguments="ns1:Subject" xmlns="https://extendedxmlserializer.
↪github.io/system">
  <Capacity>4</Capacity>
  <ns1:Subject>
    <Message>First</Message>
    <Count>0</Count>
```

(continues on next page)

(continued from previous page)

```
  </ns1:Subject>
  <ns1:Subject>
    <Message>Second</Message>
    <Count>0</Count>
  </ns1:Subject>
  <ns1:Subject>
    <Message>Third</Message>
    <Count>0</Count>
  </ns1:Subject>
</List>
```

CHAPTER 27

# History

- 2017-11-14 - v2.0.0 - Rewritten version

CHAPTER 28

# Authors

- Wojciech Nagórski
- Mike-EEE

CHAPTER 29

# Indices and tables

- genindex
- modindex
- search