

1

Technology Primer

Microsoft .NET software development relies on a wide range of technologies and “tricks of the trade.” Some of these you might not be familiar with and others you may already use on a day-to-day basis. We’ll cover a number of areas of particular relevance and value to BizTalk Server development projects to ensure we’re all on the same page as we progress through the book.

Specifically, this chapter introduces the following concepts that apply to software development in general rather than to BizTalk in particular. In my experience, understanding these concepts is key to development and debugging.

- XML Schema
- XML namespaces
- XPath
- Serializable classes

This chapter is meant simply to be an introduction to these topics and does not cover them in depth or discuss all of their aspects; there are many books and online resources that will cover them in depth. If, however, you are familiar with these topics then feel free to move on to the next chapter.

XML Schema

XML Schema is a World Wide Web Consortium (W3C) standard for defining the structure and content of Extensible Markup Language (XML) documents. In short, this means that you can define how an XML document should look and what types any data in it should conform to; it is a form of data contract, if you like. BizTalk is driven by messages, which, in most cases, are based on an XML schema that must be deployed to BizTalk.

Chapter 1: Technology Primer

The following is an example of a simple schema used to describe a person:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Person">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Name" type="xs:string"/>
        <xs:element name="Age" type="xs:int"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

This schema defines an element called `Person` that contains a complex type, which, in its simplest form, is a collection of subelements (like a class in object-oriented programming). Two further elements with differing data types have been defined within this complex type.

The following is the XML document that conforms to this schema:

```
<Person>
  <Name>Lucy</Name>
  <Age>5</Age>
</Person>
```

Defining the schema in this way enables you to enforce how you expect data to be encoded or represented. This schema can be given to any consumers of your application to ensure that they respect this contract and it enables them to communicate with you easily.

You can also use the schema to validate any incoming XML documents, which can eliminate the need for extensive data checking inside your application and improve the overall reliability of your solution.

BizTalk Server 2000 and 2002 used an earlier “schema” standard called Document Type Definition (DTD). DTD has subsequently been superseded by XML Schema, hence the native support for schemas in BizTalk Server 2004 and 2006.

If you have DTD files that you wish to leverage, you can migrate them to XML Schema by using the Add Generated Items feature, which is accessible by right-clicking your BizTalk project in the Visual Studio Solution Explorer.

XML Namespaces

Namespaces in XML documents are conceptually the same as namespaces in .NET. Any types you define are “enclosed” within the context of the specified namespace. This ensures against name clashes and allows content from multiple sources to be stored within the same document but still be isolated, as required.

Chapter 1: Technology Primer

Namespaces are used within XML documents and, therefore, BizTalk messages. Namespaces can often be the cause for problems when integrating different systems, when a message doesn't use the correct namespace and, therefore, doesn't comply with the appropriate schema.

The following XML document specifies that the `Person` element and its contents are within the "http://www.wiley.com" namespace. (This is called a *default namespace*.)

```
<Person xmlns="http://www.wiley.com/person">
  <Name>Lucy</Name>
  <Age>5</Age>
</Person>
```

The following XML document shows how you can explicitly set the namespaces, in contrast to the default namespace example shown before. The `Person` element is contained within the "http://www.wiley.com/person" namespace, and the `Address` element is defined within the "http://www.wiley.com/address" namespace.

Instead of having to prefix the nodes with the full namespace name, you can use aliases. In this instance, we've used `ps` and `ad`, which allows easier reading and, of course, a smaller payload on the wire, as less text will need to be transmitted.

```
<ps:Person xmlns:ps="http://www.wiley.com/person"
  xmlns:ad="http://www.wiley.com/address">
  <ps:Name>Lucy</ps:Name>
  <ps:Age>5</ps:Age>
  <ad:Address>
    <ad:Town>Chippenham</ad:Town>
  </ad:Address>
</ps:Person>
```

As you can see, namespace names are typically specified as URLs, which can be confusing because people tend to assume that they must point to something on the Internet. They don't have to but often do. A namespace name can be any form of URL.

This holds true for .NET namespaces as well. Microsoft, for example, tends to prefix its code in the `Microsoft` or `System` namespaces.

XPath

XPath is a navigation language for finding information held in an XML document. Navigation is described using "expressions," which in their basic form are analogous to the paths you use to navigate the filesystem on Windows.

XPath is used by BizTalk messaging and orchestrations to retrieve interesting pieces of data from a message, which can then be used to route the message or perform a conditional business process. As with namespaces, a little understanding of XPath can go a long way toward helping you diagnose problems during development.

Chapter 1: Technology Primer

An XPath expression returns a node or collection of nodes called a *node set* and in some cases the result of a function — a integer representing a count for example. The XPath expression `/ps:Person/ps:Name` will return the Name node of the following XML document.

```
<ps:Person xmlns:ps="http://www.wiley.com/person"
xmlns:ad="http://www.wiley.com/address">
  <ps:Name>Toby</ps:Name>
  <ps:Age>3</ps:Age>
  <ad:Address>
    <ad:Town>Chippenham</ad:Town>
  </ad:Address>
</ps:Person>
```

As you may have noticed, the namespace prefixes have been used in the XPath expression. Because XPath is “namespace aware,” if namespaces are omitted, it will be unable to find the node. Omitting the namespaces in this way will cause XPath to use only the default namespace. So if `/Person/Name` were to be used, it would not exist outside of the correct namespace.

The .NET code to retrieve the Name element from the preceding document is shown below. Note that the namespace alias used in the XPath query is different from the alias used in the XML document above.

Aliases are not part of the document structure; they just refer to the namespace name. Under the covers, the namespace alias is expanded to the full namespace. In the following code, you can see that you have to register a namespace prefix so that it will point at the namespace name. This alias can be anything, as long as it points to the underlying namespace definition.

Developers are often confused by namespace aliases. When faced with XML documents that BizTalk has received or generated, you might find that the namespace aliases differ from a sample XML document you are working against and assume they are wrong, when in fact they are actually the same.

I have seen issues with Web services deployed on other platforms, where the developers have assumed that the namespace alias will be a fixed name and subsequently the webservice fails to work when a perfectly valid XML document is transmitted that uses different alias names. This is incorrect behavior on the service end and should be corrected wherever possible

This alias is then used in the XPath expression to retrieve the Name element. The following code uses the .NET `XPathDocument` and `XPathNavigator` classes, which are part of the `System.Xml` namespace and provide the ability to use XPath expressions efficiently against XML documents:

```
XPathDocument doc = new XPathDocument("person.xml");
XPathNavigator nav = doc.CreateNavigator();

XmlNamespaceManager nsmgr = new XmlNamespaceManager(nav.NameTable);
nsmgr.AddNamespace("p", "http://www.wiley.com/person");

foreach (XPathNavigator node in nav.Select("/p:Person/p:Name", nsmgr))
{
    Console.WriteLine(node.Value);
}
```

Chapter 1: Technology Primer

The XPath language is comprehensive and offers a variety of mechanisms with which to retrieve data. We won't cover them all here, but we will pull out a couple of useful techniques for BizTalk Server development.

You may have noticed the use of `XPathDocument` and `XPathNavigator` in these examples rather than `XmlDocument`, which is often used. The `XPathDocument` uses a read-only, in-memory tree that is optimized for XPath expressions and the Extensible Style Language Transformation (XSLT) language.

The local-name Function

As you've probably already noticed, BizTalk uses XPath in a number of areas — pipeline component configuration and property promotions are two examples. In both of these areas, BizTalk builds the XPath statement for you but uses an XPath function called `local-name`.

As the name implies, the `local-name` function returns the name of the XML element in the local context. Therefore, if an XML element has a namespace prefix, it's considered to be within that namespace. Using `local-name` removes the namespace prefix and returns the raw element name, thus eliminating the need to jump through the hoops you had to when using the `XmlNamespaceManager` previously.

The following code returns exactly the same information as the previous code sample and is far simpler as a result of using the `local-name` function. You should also bear in mind that use of the `local-name` function incurs slightly more overhead because it has to perform extra processing.

```
XPathDocument doc = new XPathDocument("person.xml");
XPathNavigator nav = doc.CreateNavigator();

foreach (XPathNavigator node in nav.Select("/*[local-name()='Person']/*
[local-name()='Name']"))
{
    Console.WriteLine(node.Value);
}
```

About 90 percent of all the XPath “problems” I’ve ever seen have been rooted in namespace problems. A quick way to see if you’re hitting such problems is to try using `local-name` in your XPath expressions. Although I wouldn’t recommend it by default for everything, as it may cause problems in a project where there could be conflicting types. Namespaces are there for a reason!

The count Function

The `count` XPath function, as the name implies, enables you to count the number of elements matching an XPath expression, which is useful if you want to calculate the number of line items in an order, for example.

The following XML document and code counts the number of `Person` elements present in the XML document.

```
<People xmlns="http://www.wiley.com/people">
```

Chapter 1: Technology Primer

```

<Person>
  <Name>Lucy</Name>
  <Age>5</Age>
</Person>
<Person>
  <Toby>
    <Age>3</Age>
  </Person>
</People>

XPathDocument doc = new XPathDocument("person.xml");
XPathNavigator nav = doc.CreateNavigator();

int count = Convert.ToInt32(nav.Evaluate("count(/*[local-name()='People']/*[local-name()='Person'])"));

```

This technique is handy for times when you need to calculate the number of items in an XML document and then initiate a Loop shape in a BizTalk orchestration.

A number of other XPath functions are supported. Table 1-1 lists the functions supported at the time of this writing for reference. Check the MSDN documentation for the latest list.

Table 1-1: XPath Functions

Node-Set	String	Boolean	Number	Microsoft Extensions
count	concat	boolean	ceiling	ms:type-is
id	contains	false	floor	ms:type-local-name
last	normalize-space	long	number	ms:type-namespace-uri
local-name	starts-with	not	round	ms:schema-info-available
name	string	true	sum	ms:string-compare
namespace-uri	string-length			ms:utc
position	substring			ms:namespace-uri
	substring-after			ms:local-name
	substring-before			ms:number
	translate			ms:format-date
				ms:format-time

Serializable Classes

Serializable classes are one of the more powerful features of the .NET Framework and, sadly, one of the most overlooked, largely because there is no native IDE support to generate them and, therefore, their existence is not obvious to developers.

Despite this, you do end up using them whenever you return a class type from an ASP.NET Web service method. Under the covers a serializable class is generated for you to handle the class being transmitted across the wire in a Simple Object Access Protocol (SOAP) envelope.

By their very definition, serializable classes are classes that can have their “content” dehydrated for later use. The .NET implementation allows serializable classes to be dehydrated to any form of *Stream*.

Many API's in the .NET Framework use the concept of streams, which represent an abstract view of data by presenting it in a contiguous sequence of bytes that you can read or write. Thus, they isolate you from any underlying data store, such as a file, serial port, or network connection. This is useful because it enables you to write to and read from a variety of “stores” using the same application programming interface (API).

The following code uses a *StreamReader* to write to a *Memory* and *File* stream demonstrating the ability to use a generic “writer” against different stores.

```
using (MemoryStream ms = new MemoryStream())
{
    StreamWriter writer = new StreamWriter(ms);
    writer.WriteLine("Hello World!");
}

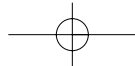
using (FileStream fs = new FileStream("output.txt", FileMode.Create))
{
    StreamWriter writer = new StreamWriter(fs);
    writer.WriteLine("Hello World");
}
```

If you want to serialize a .NET class to a stream, you need to decorate the class with a `[Serializable]` attribute, which tells the .NET runtime that your class is allowed to be serialized.

The following code demonstrates how to serialize a class to a file:

```
[Serializable]
public class Person
{
    public string Name;
    public string Town;
    public int Age;
}

class Program
{
    static void Main(string[] args)
    {
        using (FileStream fs = new FileStream("output.txt", FileMode.Create))
```



Chapter 1: Technology Primer

```

        {
            Person p = new Person();
            p.Name = "Lucy";
            p.Age = 5;
            p.Town = "Chippenham";

            XmlSerializer xs = new XmlSerializer(typeof(Person));
            xs.Serialize(fs,p);
        }
    }
}

```

The resulting `output.txt` file contains the following XML document that contains all the data of the `Person` class:

```

<?xml version="1.0"?>
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Name>Lucy</Name>
  <Town>Chippenham</Town>
  <Age>5</Age>
</Person>

```

As you can see, all the component parts of the class have been written out automatically by the `XmlSerializer` — a process often called *dehydration*. To rehydrate the class, use the same technique, but use the `Deserialize` method, instead.

```

using (FileStream fs = new FileStream("output.txt", FileMode.Open))
{
    XmlSerializer xs = new XmlSerializer(typeof(Person));
    Person p = xs.Deserialize(fs) as Person;
}

```

The resulting XML document can be customized in myriad ways. You can specify what namespaces elements are members of, whether variables are dehydrated as attributes or elements, or even to just omit certain elements from dehydration.

So, you've seen how to serialize classes and data into a variety of stores, but how does this technique help with BizTalk Server development?

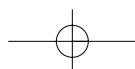
Generating Serializable Classes from Schema

Consider the following schema, which represents the evolved XML `Person` type that we've used previously:

```

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Person">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Name" type="xs:string"/>
        <xs:element name="Age" type="xs:int"/>
        <xs:element name="Address">

```



Chapter 1: Technology Primer

```
<xs:complexType>
  <xs:sequence>
    <xs:element name="Town" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

You need to be able to generate an XML document that conforms to this schema. As mentioned previously, you can create a *Person* class and decorate it with the appropriate serialization attributes. However, this requires you to manually transcribe all the schema “settings” into a serializable class, when in fact all you really want to do is have a class generated from the provided schema that will generate an XML document conforming to the schema automatically.

This is where *XSD.EXE* comes in. Visual Studio 2005 and the previous .NET versions ship with a command-line tool called *XSD.EXE* that can generate serializable classes from a schema definition, and vice versa. Using *XSD.EXE /C* to pass the schema name as a parameter results in a serializable class definition, as follows:

```
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
[System.Xml.Serialization.XmlTypeAttribute(AnonymousType=true)]
[System.Xml.Serialization.XmlRootAttribute(Namespace="", IsNullable=false)]
public partial class Person {

    private string nameField;

    private int ageField;

    private PersonAddress addressField;

    /// <remarks/>
    [System.Xml.Serialization.XmlElementAttribute(Form=System.Xml.schema
.XmlschemaForm.Unqualified)]
    public string Name {
        get {
            return this.nameField;
        }
        set {
            this.nameField = value;
        }
    }

    /// <remarks/>
    [System.Xml.Serialization.XmlElementAttribute(Form=System.Xml.schema
.XmlschemaForm.Unqualified)]
    public int Age {
        get {
```

Chapter 1: Technology Primer

```
        return this.ageField;
    }
    set {
        this.ageField = value;
    }
}

/// <remarks/>
[System.Xml.Serialization.XmlElementAttribute(Form=System.Xml.schema
.XmlschemaForm.Unqualified)]
public PersonAddress Address {
    get {
        return this.addressField;
    }
    set {
        this.addressField = value;
    }
}

/// <remarks/>
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
[System.Xml.Serialization.XmlTypeAttribute(AnonymousType=true)]
public partial class PersonAddress {

    private string townField;

    /// <remarks/>
    [System.Xml.Serialization.XmlElementAttribute(Form=System.Xml.schema
.XmlschemaForm.Unqualified)]
    public string Town {
        get {
            return this.townField;
        }
        set {
            this.townField = value;
        }
    }
}
}
```

As you can see, there are a number of extra attributes present on the class definition, which are there to control the XML serialization to ensure that the XML document conforms to the schema definition and that some property “setters” and “getters” have been provided. This strong-typing approach removes any problems associated with handling XML documents manually.

Serializing this class across the wire will result in an XML document that strictly adheres to the XML schema. You haven’t had to write any `XmlDocument` code to manually build an XML document and ensure it conforms to the schema — it’s all been done for you! This is a very powerful technique.

Using the `XmlSerializer` to construct XML documents that conform to a given XSD schema is one approach that works well for some scenarios. Other common approaches include loading a “template” XML document and replacing tokens with the required data, or using the `XmlWriter` class to build the instance document.

The `XmlSerializer` approach has the advantage that a valid XML document will be automatically constructed for you thus removing the need to write such code along with compile-time checks to ensure that, for example, you are using the correct data-types for XML elements.

Other approaches typically require details of the schema such as the structure to be encoded within a template XML document or within code that creates a valid document, if the schema is changed you will need to ensure that these are changed. However, a thorough set of validation tests as detailed in Chapter 8 could identify such problems automatically.

There are other aspects that you should consider when selecting an approach, such as the required performance, rate of schema change, and ease of programming model and size and complexity of the XML documents that you are creating.

You can now use the `XmlSerializer`, as shown before, to serialize this class. The following is an example of using this generated class:

```
Person p = new Person();
p.Name = "Lucy";
p.Age = 5;
p.Address = new PersonAddress();
p.Address.Town = "Chippenham";

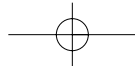
using ( MemoryStream ms = new MemoryStream() )
{
    XmlSerializer xs = new XmlSerializer( typeof(Person) );
    xs.Serialize(ms, p);
}
```

Serializable Classes and Performance

When you first use the `XmlSerializer` in your application against a given type, a dynamic class is created and compiled on the fly to represent the serialized class. This has an obvious performance overhead, but it will be cached for subsequent requests.

This cache is maintained per the common language runtime (CLR) `AppDomain`, which is fine for applications such as BizTalk, as there is by default one `AppDomain` per BizTalk host. This means that any orchestrations and .NET code will be run within the same `AppDomain` for all subsequent requests. For other applications, where the `AppDomain` is torn down each time, this may be an issue.

The .NET Framework 2.0 release introduced a new tool, `sgen.exe`, which enables you to pregenerate a serialization assembly and deploy this to your server to avoid the first-time compilation hit (which I would recommend that you measure to gauge the impact).



Chapter 1: Technology Primer

Serializing a .NET class does have an obvious CPU penalty, so bear in mind the size of your class and the frequency with which you serialize classes. It should be noted that for large complex schemas the programming model provided by serializable classes can be complex and may therefore not be appropriate. Another consideration is around message size, as we will discuss later in detail in Chapter 4. BizTalk processes messages in a forward-only streaming fashion, for large messages this approach will cause the entire message to be loaded into memory, which is not desirable.

BizTalk Development and Serializable Classes

Typically, all messages passing into and out of BizTalk are based on an underlying XML schema.

A typical BizTalk solution has systems on either side of BizTalk: systems that provide data into BizTalk and systems that BizTalk queries or updates during message processing. These systems, therefore, have to use the same message formats, along with any custom .NET components written by you and used by your BizTalk solution.

By leveraging serializable classes, you can leverage these same schemas everywhere to produce strongly typed classes that enable developers to easily work the messages and not have to fiddle with XML documents and schemas. Adopting such an approach enables the seamless interchange of data between all the key parts of your solution and removes the usually painful process of integrating the components of your solution.

Within BizTalk orchestrations you often need to pass a BizTalk message into C# class libraries for processing. If you declare the interface to use serializable classes generated from the XML schema underlying the BizTalk message, you can pass this message directly into your class library and have a strong-typed representation of your class. Thus, you eliminate any need to use XPath to retrieve data or to use the `XmlDocument` API.

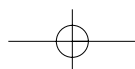
You can also use these schemas in “schema-aware” products such as Microsoft InfoPath to produce a rich interactive form with little or no development effort, to allow users to fill in data and submit it to a back-end processing system, such as BizTalk, via a Web service.

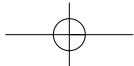
InfoPath will validate all data inputted against the underlying XML schema, thus providing a data validation layer driven by the XML schema. This validation, of course, will then occur at all points in your solution where the schema is used.

Summary

This chapter has covered four technologies and approaches that are key to effective BizTalk Server development and debugging. Although this chapter is simply an introduction, it should give you the bare minimum information required to understand what’s going on under the covers and why certain technologies behave the way they do.

XML schemas are at the center of BizTalk development. They define how messages flowing into and out of BizTalk should be represented and what they must include.





Chapter 1: Technology Primer

Namespaces are used as part of XML documents to protect against elements with the same name causing conflicts and also to convey ownership. However, they often cause problems during development because developers fail to indicate which namespace an element belongs to (for example, when querying for elements using XPath).

XPath enables you to interrogate an XML document and retrieve information to use during processing. This technique is invaluable; when combined with an XPathDocument, it provides a fast, streaming approach to retrieving data from your XML document without, for example, loading the entire document into memory.

Serializable classes are a key concept to understand, as they can greatly simplify the complexity of your solution and enable any applications using your BizTalk solution or being called by BizTalk to leverage the same schemas and classes. They also eliminate the need to write any custom code to handle and validate messages.

The following chapter discusses the BizTalk architecture at a high level to formalize its concepts and explain the value BizTalk brings to your solution.

