Exam Manual

LearnSmart

Microsoft .NET 2.0 Framework (70-529)

Distributed Application Development

This LearnSmart exam manual breaks down the most important concepts you need to master in order to successfully complete the Microsoft .NET 2.0 Applications exam (70-529). By studying this guide, you will become familiar with an array of exam-related content, including:

- Creating and Accessing XML Web Service
- Configuring and Customizing a Web Service Application
- Implementing Asynchronous Calls and Remoting Events
- And more!

Give yourself the competitive edge necessary to further your career as an IT professional and purchase this exam manual today!

Microsoft .Net Framework 2.0 Distributed Application Development (70-529) LearnSmart Exam Manual

Product ID: 11094
Production Date: July 22, 2011

Warning and Disclaimer

Every effort has been made to make this document as complete and as accurate as possible, but no warranty or fitness is implied. The publisher and authors assume no responsibility for errors or omissions. The information provided is on an "as is" basis. The authors and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this document.

Volume, Corporate, and Educational Sales

PrepLogic offers favorable discounts on all products when ordered in quantity. For more information, please contact PrepLogic directly:

1-800-418-6789
solutions@learnsmartsystems.com

International Contact Information

International: +1 (813) 769-0920

United Kingdom: (0) 20 8816 8036

Table of Contents

Create and Configure an XML Web Service

Web services are a cross platform means of exposing data and functionality to applications in a distributed environment. Web services operate over the internet using the SOAP protocol, which is based on the XML format.

Web services, under .NET, can be thought of as a normal assembly that you can interact with. The .NET runtime shields the developer from the complexities of making calls "across the wire" and appear as though they're just a standard method call.

Create a Web Service

Visual Studio 2005 provides a project template, ASP.NET Web Service, which you can use to create an initial project; however, any ASP.NET Web project can be used to hold a Web Service.

If you select the Add New Item option for the Web project you'll see, as shown in Figure 1, that you can add a Web Service to the project.

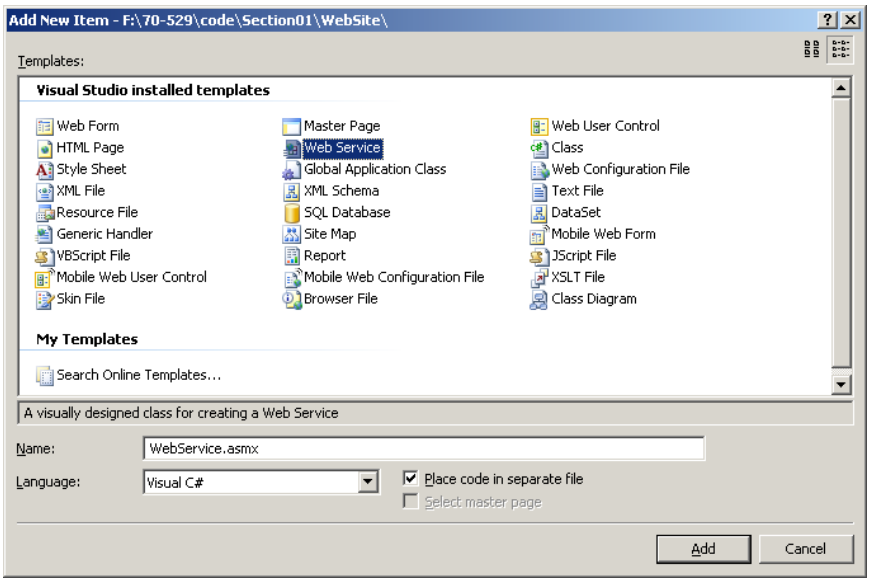

Figure 1 – Adding a Web Service

If you select the "Place code in separate file" option, referred to as the code-beside model, from the Add New Item dialog, you'll have created two files: an ASMX file, which is the public facing for the Web Service and is equivalent to the ASPX file for Web pages, and a CS file that is added to the `App_Code` folder. This is shown in Figure 2.

Figure 2 – The files created for a Web Service

If you've chosen not to place the code in a separate file, referred to as the code-inline model, you'll only have one file created; the ASMX file will contain everything necessary to run the Web Service.

The @WebService Directive

The first line in any ASMX file is a declaration that the file is actually as Web Service. As with the `@Page` directive for ASPX pages, there is a corresponding directive for Web services — `@WebService`. Depending on whether you've selected the code-beside or code-inline models, you'll have a slightly different syntax for the `@WebService` directive.

For a code-beside Web service, we specify the name of the class, the `Class` attribute, and the file that contains the class (the `CodeBehind` attribute), as follows:

```
<%@ WebService Language="C#" CodeBehind="~/App_Code/WebService.cs"
Class="WebService" %>
```

For a code-inline Web service, we simply specify the name of the class using the `Class` attribute:

```
<%@ WebService Language="C#" Class="WebService" %>
```

Creating the Web Service Class

In order for your Web service to be compiled correctly as a Web service, there are a couple of other things that must be done:

1. The class must inherit from `System.Web.Services.WebService`.
2. The class must have the `WebService` attribute applied.

This is shown in the example below:

```
[WebService]
public class WebService : System.Web.Services.WebService
{
  // code for the class
}
```

Browsing the Web Service

Web services in ASP.NET can be added quite easily to your application. ASP.NET also provides a handy method of checking that your Web service is available and previewing the exposed methods. If you navigate to your Web service in a browser, as shown in Figure 3, you'll get a handy view of your Web service and the methods that are exposed.

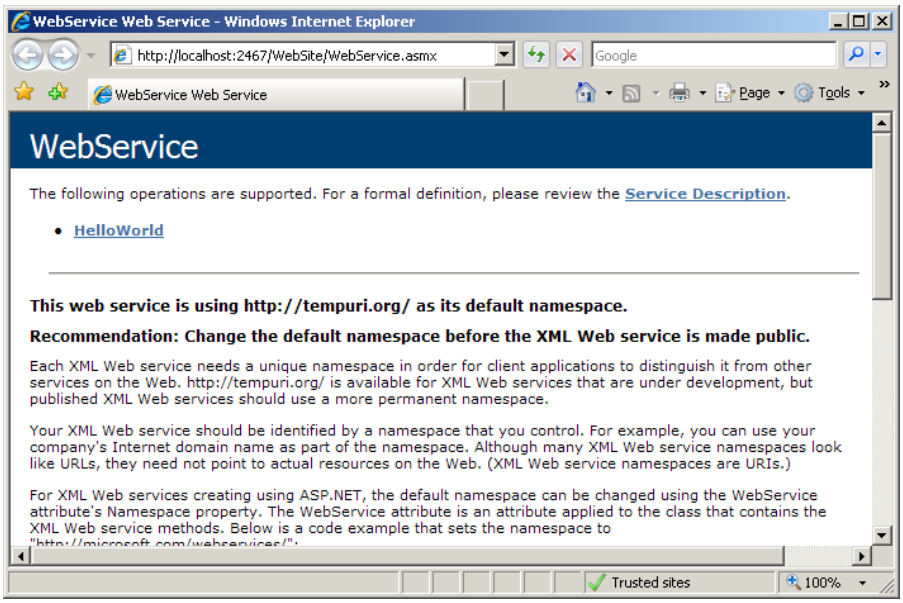

Figure 3 – Browsing a Web service

You'll see that we have a single exposed method called HelloWorld. This is added automatically to all Web services that are created in Visual Studio and the first thing that you'll normally do is delete the code for it.

Changing the Namespace

If you look again at Figure 3, you'll see that there is a recommendation to change the namespace for the Web service. All Web services created in Visual Studio are placed in this namespace. It can be changed quite easily by specifying the `Namespace` property of the `WebService` attribute:

```
[WebService(Namespace="http://preplogic.com")]
public class WebService : System.Web.Services.WebService
```

Using the Web Service

In order to use Web services in code, you need to add a reference to it to your code. In Visual Studio there is an "Add Web Reference" attribute available for the project. Selecting this allows you to browse, as shown in Figure 4, to the required Web Service.

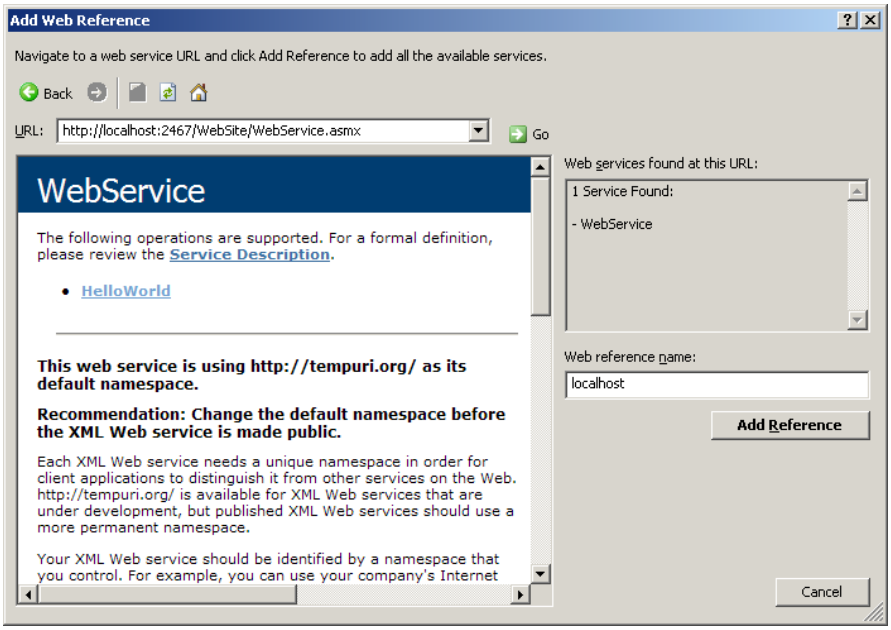

Figure 4 – Adding a reference to a Web service

The namespace box is the key; this determines the root namespace that is available within your code. In order to use the referenced Web service in code, you need to create an instance of the Web service proxy. If you leave it as localhost, as shown in Figure 4, you need to create this object as follows:

```
localhost.WebService myService = new localhost.WebService();
```

All of the exposed methods of the Web service can then be called directly on the Web service proxy. In this case we only have a `HelloWorld` method that returns a string, and we can call this as follows:

```
string myString = myService.HelloWorld();
```

Create Web Methods

We've already seen an example of a Web Method in the `HelloWorld` method that is added automatically to each Web service created in Visual Studio. There are only two requirements to create an exposed Web Method:

1. Create a public method in the Web service.
2. Add the `WebMethod` attribute to the method.

One of the simplest methods we can create is the aforementioned `HelloWorld` method:

```
[WebMethod]
public string HelloWorld()
{
  return "Hello World";
}
```

There are several properties that we can set within the `WebMethod` attribute. The two that you'll probably use most often are:

- `Description` – this allows you to add a description for the method. It is not available in code at the client but can provide a handy reference that is visible when viewing the Web service, such as the views we saw in Figure 3 and Figure 4.
- `MessasgeName` – by default, the name of the method is used as the name of the method in the proxy class. You can use the `MessageName` property to override this behavior. This is particularly handy when you have overloaded methods, as these are not supported by the SOAP protocol, and one of the overloaded methods will need to be given a different `MessageName`.

Create a OneWay Web Method

By default, when you make a call to a Web Method in your client application, the call blocks until a response is received. For the `HelloWorld` example we've seen, this is correct, as we need to return a `string`; however, for methods that don't return any values, we don't really need to wait for the method to return.

In order to mark a Web Method as *one way*, you need to use the `OneWay` property of either the `SoapDocumentMethod` or `SoapRpcMethod` attributes in the `System.Web.Services.Protocols` namespace. These two attributes change the formatting of the SOAP messages that are passed between your client and the Web service. We'll look at these two attributes in more detail later.

To add the `OneWay` attribute, you can either use the `SoapDocumentMethod` attribute:

```
[WebMethod(Namespace="http://preplogic.com")]
[SoapDocumentMethod(OneWay = true)]
public void DoWork()
```

Or the `SoapRpcMethod` attribute:

```
[WebMethod(Namespace="http://preplogic.com")]
[SoapRpcMethod(OneWay = true)]
public void DoWork()
```

Use Discovery Files to Publish a List of Web Services that are Installed on a Web Server

All ASP.NET Web services can be queried for a discovery file by adding the DISCO parameter to the query string. For example:

```
http://localhost:2467/WebSite/WebService.asmx?DISCO
```

This will return a dynamically generated discovery file for that particular Web service, as in the following:

```
<?xml version="1.0" encoding="utf-8" ?>
<discovery xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns="http://schemas.xmlsoap.org/disco/">
  <contractRef ref="http://localhost:2467/WebSite/WebService.asmx?wsdl"
   docRef="http://localhost:2467/WebSite/WebService.asmx"
   xmlns="http://schemas.xmlsoap.org/disco/scl/" />
  <soap address="http://localhost:2467/WebSite/WebService.asmx"
   xmlns:q1="http://tempuri.org/" binding="q1:WebServiceSoap"
   xmlns="http://schemas.xmlsoap.org/disco/soap/" />
  <soap address="http://localhost:2467/WebSite/WebService.asmx"
   xmlns:q2="http://tempuri.org/" binding="q2:WebServiceSoap12"
   xmlns="http://schemas.xmlsoap.org/disco/soap/" />
</discovery>
```

It is also possible to manually create this file and make it available to the consumers of your Web service; however, it is a lot easier to let the Web service create this file automatically.

Dynamically Discovering Web Services

In order to retrieve the discovery file for a Web service, you still need to know the address of the Web service. ASP.NET allows dynamic discovery that will return references to all of the Web services available under the requested URL.

To enable dynamic discovery, you need to add a mapping for `.vsdisco` to the `httpHandlers` section of `Web.config` (or `Machine.config`, if you want to enable it for the entire server):

```
<add verb="*" path=".vsdisco"
type="System.Web.Services.Discovery.DiscoveryRequestHandler "/>
```

Configuring and Customizing a Web Service Application

Configure SOAP Messages

Specify the Basic Information for a Web Service Application

As we saw earlier, there are two attributes that are used to configure Web Services. The `WebService` attribute is used to configure the overall Web service and the `WebMethod` attribute is used to configure the methods of the Web Service. Both of these attributes are from the `System.Web.Services` namespace.

We've already looked at a few of the properties that can be used; we'll now take a closer look.
The `WebService` attribute configures the basic details for the entire Web Service. The three properties that you'll most likely work with are as follows:

- `Description` – this allows you to add a description for the service. It is not available in code at the client but can provide a handy reference that is visible when viewing the Web service, such as the views we saw in Figure 3 and Figure 4 earlier.
- `Name` – by default, the name of the class is used as the name of the Web Service in the proxy class. You can use the `Name` property to override this behavior.
- `Namespace` – this property is used to specify a namespace, rather than the default `http://tempuri.org`, for the Web Service.

The `WebMethod` attribute configures the details for a specific method of the Web Service. We've already looked at two properties, `Description` and `MessageName`, of the `WebMethod` attribute. There are several other properties that you may work with:

- `BufferResponse` – used to determine whether the response to the client is buffered in memory before it is sent to the client. The default value is true.
- `CacheDuration` – setting to a value other than the default of zero caches the response for the specified number of seconds and returns the cached response to the client.
- `EnableSession` – by default, Web Services are non-sessional. Setting `EnableSession` to true changes this behavior and allows the specified method to store session state as required.
- `TransactionOption` – specifies whether the method will be transactional, using a value from the `TransactionOption` enumeration. By default, transactions are disabled.

Configure the Formatting of SOAP Messages for a Web Service Method

By using the classes in the `System.Web.Services.Protocols` namespace, you can configure the formatting of Web Service messages.

Configuring the Parameter Formatting and Style for the Web Service

The SOAP specification supports two methods of formatting parameters, RPC and Document, and you can specify which of these you want to use for your Web Service. By default, Web Services use Document formatting.

You can change the formatting of the entire Web Service by applying the `SoapDocumentService` and `SoapRpcService` attributes to the Web Service class. So, to change to RPC formatting, you'd declare your Web Service as follows:

```
[WebService(Namespace="http://preplogic.com")]
[SoapRpcService()]
public class WebService : System.Web.Services.WebService
```

In addition to the overall formatting of the Web Service messages, you can also change the way that parameters are encoded by setting the `Use` property of the `SoapDocumentService` and `SoapRpc-Service` attributes. This property can take two values — `Literal` or `Encoded` — that specify how the parameters are formatted within the message.

If you're using Document formatting, it is also possible to specify how the parameters are encapsulated within the body of the messages. By setting the `ParameterStyle` property of the `SoapDocument-Service` attribute to either `Bare` or `Wrapped`, you can change how the parameters are encapsulated within the message.

Configuring the Formatting for Methods of the Web Service

It is also possible to configure the formatting of individual methods of the Web Service using the `Soap-DocumentMethod` and `SoapRpcMethod` attributes. Any properties that are set on the method override the values that are set on the overall Web Service.

We've already seen one property of these two attributes earlier — `OneWay` — but there are also several more that you may use:

- `Action` – specifies the name of the `SOAPAction` header of the SOAP request.
- `Binding` – specifies the name of the binding for the method. By default, this is the name of the method suffixed with `Soap`.
- `RequestElementName` – specifies the XML element in the message for the request. The default value is the name of the Web Service method.
- `RequestNamespace` – the namespace for the request. By default, this is the same namespace as specified for the overall Web Service.
- `ResponseElementName` – specifies the XML element in the message for the response. The default value is the name of the Web Service method suffixed with `Response`.
- `ResponseNamespace` – the namespace for the response. By default, this is the same namespace as specified for the overall Web Service.
- `Use` – as with the same property for the overall Web Service, we can specify the parameter encoding for individual methods.

If you're using Document formatting, you may also specify how the parameters are encapsulated within the body of the message. By setting the `ParameterStyle` property of the `SoapDocumente-Method` attribute to either `Bare` or `Wrapped`, you can change how the parameters are encapsulated within the message.

Specify the Bindings of a Web Service Application by Using the WebServiceBinding Attribute

The SOAP specification allows for several different methods of formatting Web Service messages. The WS-I Basic Profile was specified by the Web Services Interoperability Organization with the goal of standardizing the functionality and formatting that is used across the different platforms.

You can use the `WebServiceBinding` attribute from the `System.Web.Services` namespace to configure the binding information for the Web Service. There are several properties that you may be interested in:

- `ConformTo` – specifies the WS-I standard to which the Web Service claims to conform. The default value is `None`, but this can also be set to `BasicProfile1_1` to claim conformance to version 1.1 of the WS-I Basic Profile.
- `EmitConformanceClaims` – Set to `true` to specify that the binding outputs its conformance claims.
- `Location` – specifies the location where the binding is defined. By default, this is the URL of the Web Service.
- `Name` – species the name of the binding. By default, this is the name of the Web Service suffixed with `Soap`.
- `Namespace` – specifies the namespace of the binding. By default, this is the same namespace as specified for the overall Web Service.

Configure a Web Service Application by Using a Configuration File

The overall functionality of Web Services can be configured using configuration files — either `Machine.config` or `Web.config`. The `<webServices>` element of `<system.web>` allows you to specify several configuration settings for Web Services using child elements:

- `protocols` – allows you to specify the protocols that are supported by the Web Service:
  - `HttpGet` – the Web Service will accept parameters passed in the query string. The return value is the body of the response and a simple XML document (it is not a SOAP message).
  - `HttpPost` – the Web Service will accept parameters passed in the body of the HTTP request. The return value is the body of the response and a simple XML document (it is not a SOAP message).
  - `HttpPost` – the Web Service will accept parameters passed in the body of the HTTP request but only from the localhost. The return value is the body of the response and a simple XML document (it is not a SOAP message). This value is ideal for testing purposes.
  - `HttpSoap` – the Web Service supports the SOAP protocol. A SOAP message is sent in the request to the Web Service and the response is also a SOAP message.
  - `Documentation` – this is a special value that turns on the documentation page, shown earlier in Figure 3 and Figure 4, for the Web Service. The documentation page is returned when the Web Service is requested directly.

- `serviceDescriptionFormatExtensionTypes` – used to control the service description format extension classes that are used to extend the WSDL that is automatically generated for Web Services.
- `soapExtensionImporterTypes` – specifies (for client applications only) any extension classes that are used to extend the proxy generation process.
- `soapExtensionImporterTypes` – specifies (for Web Services only) any extension classes that are used to extend the WSDL generation process.
- `soapExtensionTypes` – specifies any SOAP extensions that are used to inspect or modify the SOAP message during processing. This element applies at both the client application and the Web Service.

Manage Session State in Web Services

Web Services can handle session state in the same was as any ASP.NET application. There are three methods that we can use and each of those methods has different performance implications. The most scalable Web Services are those that don't use session state.

In order for a method in a Web Service to use session state, you must set the `EnableSession` property of the `WebMethod` attribute to true. It is set to false, by default, which means that your method will not store any session information and accessing the `Session` object will cause a runtime error. The `Application` object is available to all methods within the Web Service and does not need the method to be marked as requiring session state.

Implement Session State by using the Application Object

Web Services have access to all of the functionality of ASP.NET; the `Application` object is no different. All values set on the `Application` object in the method are visible to every other method that is using session state.

Implement Session State by using the Session Object

As with the `Application` object, a Web Service can access the `Session` object as they would for any other ASP.NET application. Only the current session can see values set in the `Session` object.

Implement Session State by using Cookies

In order for session state to work with Web Service methods, you must also manually store the client-side cookie value that indentifies the session. When browsing an ASP.NET application, the browser is responsible for managing cookies any will automatically pass the cookie that identifies the session to any calls to the ASP.NET application. When calling a Web Service method in code, you don't have this functionality, and you must manually manage the cookies that are passed to the method call.

You need to store an instance of the `CookieContainer` class from the `System.Net` namespace and attach this to every call to the Web Service.

Whatever the client application, you need to use the same `CookieContainer` instance for all calls in the same session. For a Windows Forms client application, you may store the `CookieContainer` in a global variable, whereas an ASP.NET client application may store the `CookieContainer` in the session.

The `CookieContainer` then needs to be added to the `CookieContainer` property of the proxy class that is generated:

```
// get the cookie collection

System.Net.CookieContainer myCookies = null;
if (Session["cookies"] == null)
{
  myCookies = new System.Net.CookieContainer();
}
else
{
  myCookies = (System.Net.CookieContainer)Session["cookies"];
}

// create the web service proxy
localhost.WebService myService = new localhost.WebService();

// add the cookie container to the proxy
myService.CookieContainer = myCookies;

// call the required methods
string myString = myService.HelloWorld();

// store the returned cookie collection
Session["cookies"] = myService.CookieContainer;
```

Implement SOAP Headers

A SOAP Header is an optional element of the SOAP Envelope that we saw earlier. SOAP Headers are defined within the Web Service.

A SOAP Header can be added to individual method calls within the Web Service. It is also possible to add multiple SOAP Headers to a method.

Add a Custom SOAP Header Class

A SOAP Header is created like any other class and must derive from the `SoapHeader` class in the `System.Web.Services.Protocols` namespace. By deriving from the `SoapHeader` class, the SOAP Header gains access to the following properties:

- `DidUnderstand` – whether the Web Service method understood and processed the header that was passed.
- `MustUnderstand` – set to true to indicate that the Web Service method must understand the header that is passed. Failure to do so will cause ASP.NET to throw a `SoapHeaderException` which will be returned to the client application.

Within the SOAP Header, you are free to define whatever properties you wish. A very simple SOAP Header would be as follows:

```
public class MessageHeader : System.Web.Services.Protocols.SoapHeader
{
  public string Message;
}
```

Create a Public Instance of the Custom SOAP Header Class in a Web Service Class

SOAP Headers should be defined within the Web Service and exposed as a public property of the Web Service. For our simple SOAP Header, above, we would expose the property as follows:

```
public MessageHeader theMessage;
```

This exposes a property, `theMessage`, which can be set to a `MessageHeader` instance.

Apply a SoapHeader Attribute to a Web Method

To set a Web Service method to accept a SOAP Header, we use the SoapHeader attribute from the `System.Web.Services.Protocols` namespace. So, to define a Web Service method that accepts a `MessageHeader` SOAP Header we can add the `SoapHeader` attribute as follows:

```
[WebMethod]
[SoapHeader("theMessage")]
public string ReturnMessage()
{
  return String.Empty;
}
```

The `SoapHeader` attribute requires one parameter – the name of the public property (not the type of the property) that is used for the header.

Add SOAP Headers to Web Service Calls

The classes and properties that are defined in the Web Service for any SOAP Headers are automatically generated at the proxy when adding a reference to the Web Service to your application. There is no need to have the code that is required for the SOAP Header referenced at both the Web Service and the client application. All of the required classes and properties will be defined within the proxy class for the Web Service.

To add a SOAP Header to a Web Service method call, you need to create an instance of the required header and pass this to the public property for that header:

```
// create the web service proxy
localhost.WebService myService = new localhost.WebService();

// create the required header
localhost.MessageHeader myHeader = new localhost.MessageHeader();
myHeader.Message = "I was passed as a SOAP Header";

// attach the header to the Web Service
myService.theMessage = myHeader;
```

Any calls to the Web Service that are made after the SOAP header is attached will have the SOAP Header automatically added.

Access and Process a SOAP Header in a Web Method

Within the Web Service method, the SOAP Headers that have been added can be accessed through the public property that was defined for the SOAP Header. The SOAP Header is not passed to the Web Service method as a parameter.

We can modify our `ReturnMessage` method from earlier to handle the `MessageHeader` SOAP Header:

```
[WebMethod]
[SoapHeader("theMessage")]
public string ReturnMessage()
{
  if (theMessage == null)
    return "You did not pass a message";
  else
    return "Your message was: " + theMessage;
}
```

Set the Direction of a SOAP Header

By default, all SOAP Headers that you create are defined as only being passed from the client application to the Web Service. It is possible to change this behaviour using the `Direction` property of the `SoapHeader` attribute. The `Direction` property can accept four values:

- `In` – the default value, indicating that the header is only passed from the client application to the Web Service method. Any changes to the header within the Web Service method are not passed back to the client application.
- `Out` – used to indicate that the header only applies to the client application. Any value set for the header by the client application is ignored and the value set by the Web Service method is available to the client once the Web Service method call is complete.
- `InOut` – combines both the `In` and `Out` functionality. Any value set by the client application is available to the Web Service and any changes made by the Web Service method are passed back to the client application.
- `Fault` – specifies that the header is only available to the client if an exception is thrown by the Web Service method.

To allow changes to the `MessageHeader` SOAP Header by the Web Service method to be returned to the client application, we would modify the `SoapHeader` attribute as follows:

```
[WebMethod]
[SoapHeader("theMessage", Direction=InOut)]
public string ReturnMessage()
{
  if (theMessage == null)
    theMessage = "NO MESSAGE";
    return "You did not pass a message";
  else
    return "Your message was: " + theMessage;
}
```

When returning from the Web Service method call, `theMessage` will contain "NO MESSAGE" if we didn't pass a SOAP Header to the Web Service method call.

Handle Unknown SOAP Headers

As we saw earlier, there are a couple of properties of the `SoapHeader` base class that allow us to specify how headers are handled.

To force a Web Service method to understand a particular SOAP Header, you set the `MustUnderstand` property to `true`:

```
myHeader.MustUnderstand = true;
```

The Web Service must then set the `DidUnderstand` property for all SOAP Headers that it doesn't understand to false. ASP.NET assumes that all SOAP Headers are understood and sets the `DidUnderstand` property to true – you must manually set it to `false`.

An unknown header cannot be instantiated at the Web Service (after all, it is unknown) so ASP.NET provides the `SoapUnknownHeader` class in the `System.Web.Services.Protocols` namespace. By adding a public instance of this class to your Web Service, and specifying it for Web Service methods using the `SoapHeader` attribute, any unknown SOAP Headers will be available within the Web Service method.

We may have multiple unknown SOAP headers, so we need to create an array of `SoapUnknownHeader` objects as the public instance:

```
public SoapUnknownHeader[] unknownHeaders;
```

We can then add handling of unknown SOAP Headers to Web Service methods by adding a `SoapHeader` attribute:

```
[WebMethod]
[SoapHeader("theMessage", Direction=InOut)]
[SoapHeader("unknownHeaders")]
public string ReturnMessage()
```

And then, within the method itself, we can set the `DidUnderstand` property to `false`:

- `foreach (SoapUnknownHeader unknownHeader in unknownHeaders)`

```
{
  unknownHeader.DidUnderstand = false;
}
```

Implement SOAP Extensions

A SOAP Extension is a means of modifying the XML that is passed between the client application and the Web Service before it is mapped to .NET objects. SOAP Extensions are used quite extensively by Web Service Enhancements (WSE) to provide the required extra functionality.

Create a Custom SOAP Extension

A SOAP Extension is created like any other class, and it must derive from the `SoapExtension` class in the `System.Web.Services.Protocols` namespace. There are several methods and properties of this class, but the ones that you'll be most interested in are `ChainStream` and `ProcessMessage`.

The `ChainStream` method is used to provide access to the `Stream` that contains the message. Within your SOAP Extension, you should store the `Stream` that is passed into the `ChainStream` method, as this contains the stream that the other SOAP Extensions have modified. You must also provide a `Stream` that the current SOAP Extension can modify:

```
Stream oldStream;
Stream newStream;

public override Stream ChainStream(Stream stream)
{
    oldStream = stream;
    newStream = new MemoryStream();
    return newStream;
}
```

The `ProcessMessage` method is abstract and must be implemented in a SOAP Extension. It accepts one parameter, a `SoapMessage` instance, which contains the data at a specific stage in the serialization and deserialization process. It is here that you must modify the `Stream` passed to `ChainStream` and modify this as required to create the contents of the new `Stream` that is returned from the `ChainStream` method.

At the Web Service, there are four stages to handling an incoming SOAP Message from the client application:

- `BeforeDeserialize` – the request has been received in the request stream, but the message has not been deserialized into .NET objects.
- `AfterDeserialize` – the message has been deserialized from the request stream, but the Web Service method has not been called.
- `BeforeSerialize` – the Web Service method has been called, but the return values have not been serialized into the response stream.
- `AfterSerialize` – the return values have been serialized into the response stream.

A similar process occurs at the client when a request is made to the Web Service, except that the stages occur in a slightly different order:

- `BeforeSerialize` – the Web Service method has been called, but the request has not been serialized into the request stream.
- `AfterSerialize` – the request has been serialized into the request stream, but has not been sent to the Web Service.

- `BeforeDeserialize` – the response has been received from the Web Service in the response stream, but the message has not been deserialized into .NET objects.
- `AfterDeserialize` – the message has been deserialized from the response stream, but the client application has been given the results yet.

You can determine which stage of the process that the `ProcessMessage` method is being called by checking the `Stage` property of the `SoapMessage` instance:

```
public void ProcessMessage (SoapMessage message)
{
  switch (message.Stage)
  {
    case SoapMessageStage.BeforeSerialize:
      // do something
      break;
    case SoapMessageStage.AfterSerialize:
      // do something
      break;
    case SoapMessageStage.BeforeDeserialize:
      // do something
      break;
    case SoapMessageStage.AfterDeserialize:
      // do something
      break;
    default:
      // unknown message stage
      break;
  }
}
```

You can also determine whether the message being passed through `ProcessMessage` is a message at the client application or a message at the Web Service by checking the type of the `SoapMessage` instance. `SoapMessage` is an abstract class and has two concrete derived classes:

- `SoapClientMessage` – a message at the client application: either the request being passed to the Web Service or the response from the Web Service.
- `SoapServerMessage` – a message at the Web Service: either the request received from the client application or the response to the client application.

We can use this to determine the action to perform. For instance, we can change the `BeforeSerialize` processing as follows:

```
case SoapMessageStage.BeforeSerialize:
  if (message is SoapClientMessage)
  {
    // do something at client
  }
  else if (message is SoapServerMessage)
  {
    // do something at server
  }
  else
  {
    // unknown message type
  }

  break;
```

Configure a SOAP Extension

SOAP Extensions are added to the `<soapExtensionTypes>` element of `<webServices>` as follows:

```
<configuration>
  <webServices>
    <soapExtensionTypes />
  </webServices>
</configuration>
```

Within the `<soapExtensionTypes>` element you can use the `<add>`, `<clear>`, and `<remove>` elements to change the SOAP Extensions that are currently enabled.

Adding SOAP Extensions is accomplished using the `<add>` element and specifying the following attributes, all of which are required:

Attribute	Description
Type	Specifies the fully qualified type of the SOAP Extension to add. If the SOAP Extension is in the GAC it must include the version, culture and public key of the assembly containing the SOAP Extension.
Group	Used with `Priority` to specify the ordering that the SOAP Extensions are applied. This can be either `0` or `1` with a value of zero having the highest priority.
Priority	Used with `Group` to specify the ordering that the SOAP Extensions are applied. Any integer value is allowed with lower values having a higher priority.

Creating, Configuring, and Deploying Remoting Applications

Remoting is a way to call objects located in different processes on different machines as if they were objects within the same application. Remoting is the successor to DCOM and hides all the complexities that usually accompany these types of calls.

It is possible to host remote objects in four different places:

- **Console application** – must be manually started and provides a very limited user-interface to the client. Can support TCP, HTTP and IPC as the communications channel.
- **Windows application** – must be manually started, but provides a more functional user-interface to the client. Can support TCP, HTTP and IPC as the communications channel.
- **ASP.NET application** – hosted within IIS and able to take advantage of the full functionality of IIS (session state, caching, security features such as SSL). Can only be used with HTTP as the communication channel.
- **Windows service** – runs automatically and can easily be monitored; however, debugging is limited. Can support TCP, HTTP and IPC as the communications channel.

To connect to a remote object you use a *channel*. All communications with the remote object are performed across the same channel. To use a remote object, a channel must be registered and the same channel can only be registered once on a machine. There are three possible channels:

- **TCP** – communication is formatted in binary and transmitted across the network using sockets. This is the fastest channel that you can use when the remote object is on a different machine.
- **HTTP** – communication is SOAP formatted using the XML serializer to handle messages to the remote object. Can be configured to use the binary formatter.
- **IPC** – an inter-process communication (IPC) channel that can be used when communication is on the same machine. Security is controlled using access control lists.

There are three types of remote objects that can be created:

- **Single-call** – these objects are managed on the server and are only used in a single method call and then disposed of. They don't maintain state between calls.
- **Singleton** – these objects are managed on the server and one object is used for all requests. State is maintained between method calls. Singleton remote objects can offer a performance advantage over single-call, as a new object doesn't need to be created on every call to the remote object.
- **Client-activated** – only activated when requested by a client with the client receiving a dedicated object, even though the remote object exists on the server.

Create and Configure a Server Application

Create a Server Application Domain

An object can be instantiated as a remote object if it inherits from *MarshalByRefObject* rather than the default *Object*. All remotable objects must inherit from *MarshalByRefObject*.

All remotable objects, being derived from *MarshalByRefObject*, are passed by reference. It is also possible to create a pass-by-value remote object by marking the object serializable using the *Serializable* attribute.

Once you have a remotable object, you need to configure the hosting application. You can do this either programmatically or by using configuration files.

Configure a Server Application Programmatically

Configuring Channels

To configure a hosting application for remoting you need to decide on the channel you wish to use and then call the static *RegisterChannel* method of *System.Runtime. Remoting.Channels.ChannelServices*.

Each of the supported channels has their own class, implementing the *IChannel* interface, which you need to create and then pass as the constructor to the *RegisterChannel* method:

- **TCP** – *System.Runtime.Remoting.Channels.Tcp.TcpChannel* – pass the required port number to the class constructor.
- **HTTP** – *System.Runtime.Remoting.Channels.Http.HttpChannel* – pass the required port number to the class constructor.
- **IPC** – *System.Runtime.Remoting.Channels.Ipc.IpcChannel* – pass the name of the channel to the class constructor.

You can also unregister a channel and stop listening for requests for remote objects by calling the *UnregisterChannel* method — again passing in a class that implements the *IChannel* interface.

Configuring Remote Objects

Once you call the *RegisterChannel* method, and provided an error is not returned, the hosting application then needs to be configured for the specific remote objects. You need to call a static method of the *System.Runtime.Remoting.RemotingConfiguration* class.

To enable a server-activated remote object, you need to call the static *RegisterWellKnowServiceType* method passing the type of the object, a unique URI for the object, and the activation mode. For a single-call object you'd call:

```
RemotingConfiguration.RegisterWellKnownServiceType(
  typeof(MyRemoteClass), "MyRemoteObject",
  WellKnownObjectMode.SingleCall);
```

And for a singleton object you'd call:

```
RemotingConfiguration.RegisterWellKnownServiceType(
  typeof(MyRemoteClass), "MyRemoteObject",
  WellKnownObjectMode.Singleton);
```

For client-activated remote objects you need to call the *RegisterActivatedServiceType* method:

```
RemotingConfiguration.RegisterActivatedServiceType(
  typeof(MyRemoteClass));
```

Versioning

It is possible to have different versions of the same assembly used by a hosting application. By default, the runtime uses the latest version of the type. It is possible to override this behavior by specifying the version of the type to use in the call to the *RegisterWellKnownServiceType* method. For example:

```
RemotingConfiguration.RegisterWellKnownServiceType(
  typeof(MyRemoteClass, Version=1.0.0.0), "MyRemoteObject",
  WellKnownObjectMode.SingleCall);
```

Changing the Channel Formatting

By default, a *TcpChannel* uses binary formatting, an *HttpChannel* uses SOAP formatting and an *IpcChannel* uses binary formatting. It is possible to change this behavior by passing a different channel to the *RegisterChannel* method.

- **TCP** – *System.Runtime.Remoting.Channels.Tcp.TcpServerChannel* – pass the required port number and the required formatter to the class constructor.
- **HTTP** – *System.Runtime.Remoting.Channels.Http.HttpServerChannel* – pass the required port number and the required formatter to the class constructor.
- **IPC** – *System.Runtime.Remoting.Channels.Ipc.IpcServerChannel* – pass the name of the channel and the required formatter to the class constructor.

The required formatter is an instance of a class that implements the *IServerChannelSinkProvider* interface — either the *BinaryServerFormatterSinkProvider* or *SoapServerFormatterSinkProvider*.

Configure a Server Application using Configuration Files

Programmatically configuring the hosting application removes a lot of flexibility from the application by hard coding all the channel properties. To remove this problem you use configuration files to specify all of the properties for the hosting application and the remote types that are hosted.

Remoting settings for an application are stored within the <*system.runtime.remoting*> element of a configuration file. All configuration settings are contained within an <*application*> element.
For ASP.NET applications, the configuration settings will be stored in the standard Web.config file and will automatically be picked up by the application.

For all other hosting applications, you need to call the *Configure* method of the *RemotingConfiguration* class, specifying the name of the configuration file.

Configuring Channels

Channels are configured using the <*channels*> element of the <*system.runtime.remoting*> element. For each channel, there will be a separate <*channel*> element. This element has several attributes, the most important of which are shown in the following table:

Attribute	Description
ref	Specifies the channel type to use - *tcp*, *http* or *ipc*. This is a shortcut way to specify the *type*.
type	The full type name of the channel. Can be used instead of *ref*.
port	Used with TCP and HTTP to specify the port number to use.
portName	Used with IPC to specify the name of the IPC channel.
machineName	The name of the machine hosting the remote object.
useIPAddress	Specifies whether machineName is an IP address or a URL.

Configuring Remote Objects

Once channels are configured, you then need to register the remote objects by specifying the objects in the <*service*> element of <*system.runtime.remoting*>.

For server-activated objects, you specify each object using a <*wellKnown*> element, specifying the following attributes:

Attribute	Description
mode	Specifies whether the object is *SingleCall* or *Singleton*.
Type	The full type name of the remote object.
objectUri	Used to specify a unique URI for the remote object. This is not the same as the URL that is used to access the object from the client.

For client-activated objects, you specify each object using an <*activated*> element, specifying the following attributes:

Attribute	Description
type	The full type name of the remote object.

Versioning

By default, the latest version of an assembly is used if two versions of the same assembly are used by the hosting application. You can specify the version required by adding the version number to the *type* attribute of the <*wellKnown*> and <*activated*> elements.

Change the Channel Formatting

By default, a *TcpChannel* uses binary formatting, an *HttpChannel* uses SOAP formatting and an *IpcChannel* uses binary formatting. It is possible to change this behavior by adding a <*serviceProvider*><*formatter*> element as a child of the required <*channel*> element.

The <*formatter*> element is configured using, among others, the following attributes:

Attribute	Description
ref	Specifies the formatter to use — *binary* or *soap*. This is a shortcut way to specify the *type*.
type	The full type name of the formatter. Can be used instead of *ref*.

Create a Client Application to Access a Remote Object

Create a Remote Object

When using a remote object, your application is actually using a proxy object, which is just a pointer to the remote object. This proxy makes it appear as though the remote object is no different than a normal object.

The proxy is created whenever the client requests a remote object. When the remote object is instantiated depends on whether the object is a server or client activated object. A server-activated object (single-call or singleton) is not instantiated until a request is made to one of the methods of the object. With a client-activated object, the object is activated as soon as it is created.

For both server-activated and client-activated objects, the client application needs a reference to the assembly containing the remote object being activated.

Configure a Client Application Programmatically

Configuring Channels

To configure a client application to access a remote object, you need to know which channel you're using and then call the static *RegisterChannel* method of *System.Runtime. Remoting.Channels.ChannelServices*.

Each of the supported channels has their own class implementing the *ICientChannel* interface, which you need to create and then pass as the constructor to the *RegisterChannel* method:

- **TCP** – *System.Runtime.Remoting.Channels.Tcp.TcpClientChannel* – pass the required port number to the class constructor.
- **HTTP** – *System.Runtime.Remoting.Chan.nels.Http.HttpClientChannel* – pass the required port number to the class constructor.
- **IPC** – *System.Runtime.Remoting.Channels.Ipc.IpcClientChannel* – pass the name of the channel to the class constructor.

Configuring Remote Objects

To configure a client to access a server-activated remote object, you need to call the static *Register-WellKnownClientType* method passing the type of the object and its URL:

For both single-call and singleton objects, you'd call:

```
RemotingConfiguration.RegisterWellKnownClientType(
  typeof(MyRemoteClass), "http://remoteServer/object.rem");
```

For client-activated remote objects, you need to call the *RegisterActivatedClientType* method passing the type of the object and its URL:

```
RemotingConfiguration.RegisterActivatedClientType(
  typeof(MyRemoteClass) , "http://remoteServer/object.rem");
```

Configure a Client Application using Configuration Files

Remoting settings are stored within the <*system.runtime.remoting*> element of a configuration file. All configuration settings are contained within an <*application*> element.

For ASP.NET applications, the configuration settings will be stored in the standard Web.config file and will automatically be picked up by the application.

For all other hosting applications, you need to call the *Configure* method of the *RemotingConfiguration* class, specifying the name of the configuration file.

Configuring Channels

Channels are configured using the <*channels*> element of the <*system.runtime.remoting*> element. For each channel, there will be a separate <*channel*> element. This element has several attributes, the most important of which are shown in the following table:

Attribute	Description
ref	Specifies the channel type to use — *tcp*, *http* or *ipc*. This is a shortcut way to specify the *type*.
type	The full type name of the channel. Can be used instead of *ref*.
priority	Higher priority channels will be used first.

Configuring Remote Objects

Once the channels are configured, you will then need to register the remote objects by specifying the objects in the <*client*> element of <*system.runtime.remoting*>.

For server-activated objects, you specify each object using a <*wellKnown*> element, specifying the following attributes:

Attribute	Description
type	The full type name of the remote object.
url	The URL used to access the remote object.

For client-activated objects, you specify each object using an <*activated*> element, specifying the following attributes:

Attribute	Description
type	The full type name of the remote object.

Access the Remoting Service by Calling a Remote Method

There are two ways to create and access a remote object. If the remote object is already configured at the client (as we've seen in the last two sections), creating an instance of the remote object using *new* will create the proxy and connect to the remote object correctly.

It is also possible to use the *Activator.GetObject* method to create an instance of a remote object passing in the type and URL of the remote object. The *GetObject* method returns an *Object*, so you need to cast it to the correct type, as follows:

```
MyRemoteObject objRemote = (MyRemoteClass)Activator.GetObject(
  typeof(MyRemoteClass), "http://remoteServer/object.rem");
```

Once the object has been created (via *Activator.GetObject* or by using the *new* keyword), accessing the remote object synchronously is the same as accessing a local object. The proxy for the remote object makes all requests to the remote object no differently than it does for a local object.

Debug and Deploy a Remoting Application

Use Performance Counters to Monitor a Remoting Application

There are several Remoting-specific performance counters available. When adding performance counters in the Performance Monitor utility, there is a specific category of counters, *.NET CLR Remoting*, that contain all of the counters specific to remoting, as shown in the chart below:

Counter	Description
Channels	Shows the total number of channels registered since the application started.
Context Proxies	Shows the total number of proxy objects created since the application started.
Context-Bound Classes Loaded	Shows the current number of context-bound classes loaded.
Context-Bound Objects Alloc/sec	Shows the current number of context-bound objects allocated per second.
Contexts	Shows the current number of contexts in the application.
Remote Calls / sec	Shows the number of calls to remote objects per second.
Total Remote Calls	Shows the total number of calls to remote objects since the application started.

Debug a Remoting Application

To debug remote objects in Visual Studio, you need to attach the debugger to the hosting application before you execute the client application. You can do this by selecting Attach to Process from the Debug menu in Visual Studio (if the hosting application is on another machine, you will need to configure that machine for remote debugging). Once you've attached to the hosting application, you can start debugging of the client application.

Handling Exceptions

Errors that occur in the remote object are handled by the runtime and passed back to the client application as a *RemotingExcepction*. You should catch this particular type of exception and handle it correctly.

Tracking Remoting

In addition to performance counters, it is possible to provide more granular reporting by making use of Tracking Services provided by *System.Runtime.Remoting*.

To make use of tracking you need to create a class that implements the *ITrackingHandler* interface (in *System.Runtime.Remoting*), implementing the following methods:

Method	Description
DisconnectedObject	Called whenever an object is disconnected from the proxy.
MarshalledObject	Called when an object is marshaled.
UnmarshalledObject	Called when an object is unmarshaled.

You then use the *TrackingServices* class (in *System.Runtime.Remoting*) to register (*RegisterTrackingHandler*) and unregister (*UnregisterTrackingHandler*) the class that implements the *ITrackingHandler* interface.

Deploy a Remoting Application

Remote objects must be deployed with both the hosting application and each client application that calls the remote object.

Deploying a Hosting Application

The most common method of deploying a hosting application is to create a setup project for the application. The type of setup project is determined by the hosting application:

- **Windows Setup Project** – used to deploy Console Applications, Windows Applications and Windows Services.
- **Web Setup Project** – used to deploy ASP.NET Applications.

Deploy a Client Application

You can deploy a client application in three different ways:

- **Deploy the remote assembly** – simply deploy the remote assembly and reference it directly within the client application.
- **Deploy an interface** – define an interface that is implemented by the remote object and build it to its own assembly. As the interface specifies the functionality of the remote object, you only need to deploy the interface to the client application for it to reference.
- **Use soapsuds.exe** – the soapsuds.exe application can be used from the command-line to create an assembly that can be referenced by the client. The developer only needs to know the URL to access the hosting application and create the necessary assembly.

Manage the Lifetime of Remote Objects

Due to remoting operating over process boundaries (i.e., from the client application to the hosting application) the garbage collector is not able to manage objects that are involved in remoting correctly. The client application holds the reference to a remote object in the hosting application but the hosting application has no knowledge of the client application; therefore, the remote object instantiated has, as far as the garbage collector is concerned, no references and can be cleaned up.

This problem is overcome by the use of lease objects in the hosting application. A lease object is similar to a proxy object in the client application. The lease object references the remote object and stops the garbage collector from automatically cleaning up the remote object.

Initialize the Lifetime of a Remote Object

Lease objects at the hosting application are created automatically when a remote object is requested. Lease objects are only maintained for a specified period of time (5 minutes by default) and, once the lease expires, the remote object is garbage collected. It is possible to modify the lease settings by overriding the *InitializeLifetimeService* method of the *MarshalByReference* base class.

Within the overridden *InitializeLifetimeService* method, you first need to get a reference to the lease itself (as an *ILease* interface from the *System.Runtime.Remoting.Lifetime* namespace), check that it's not an active lease, set the properties of the lease as required and then return the lease from the method:

```
public override object InitializeLifetimeService()
{
  ILease myLease = (ILease) base.InitializeLifetimeService()

  if (myLease.CurrentState == LeaseState.Initial)
  {
    // set the properties we want as required
  }

  return(myLease);
}
```

You can specify how the lease is handled using, amongst others, the following methods:

Property	Description
InitialLeaseTime	Gets or sets the initial time to keep the lease alive for. The default value is five minutes.
RenewOnCallTime	Used to renew the lease every time the object is used. The default value is 2 minutes. If the object is called with less than RenewOnCallTime minutes until it expires it is renewed for a further period of RenewOnCallTime.

It is also possible to configure the lease of a remote object in the configuration files for the hosting application. The <*application*> element of the <*system.runtime.remoting*> element has a <*lifetime*> element that can be used to specify the lease time for all remote objects in the application (which can then be overridden by the remote objects themselves in the *InitializeLifetimeService* method). There are several attributes of the <*lifetime*> element the most useful of which are as follows:

Property	Description
leaseTime	Sets the default value for InitialLeaseTime.
renewOnCallTime	Sets the default value for RenewOnCallTime.

Renew the Lifetime of a Remote Object

Once a lease has been created (i.e., the *InitializeLifetimeService* method has been called), changing the properties of the lease has no effect. Although the lease will be renewed automatically, it is also possible to manually renew the lease, for a specific period of time, by calling the *Renew* method of the *ILease* interface on the client.

To do this you must have an instance of the remote object and call the static *GetLifetimeService* method of the *RemotingServices* class to return the *ILease* object. You can then call the *Renew* method specifying the amount of time to keep the lease active. For example, to renew a lease for 30 minutes, you would call the following:

```
MyRemoteObject objRemote = new MyRemoteObject();
ILease myLease = (ILease) RemotingServices.GetLifetimeService(objRemote);
myLease.Renew(TimeSpan.FromMinutes(30));
```

Implementing Asynchronous Calls and Remoting Events

Call Web Methods Asynchronously

You've already seen that calling a Web Method synchronously is no different than calling a method on a local class. The proxy object created at the client makes the methods of the Web Service appear as though they're local method calls.

So if we have a Web Method called `CalculateCost`, exposed by the `CostingService` Web Service, then we have a synchronous method created as follows:

```
public int CalculateCost(int intProductID)
```

We can call this method of the proxy class as we would any other method and the call to the Web Service will be made synchronously.

It is also possible to call the methods of a Web Service asynchronously.

Call a Web Method

When the proxy object for the Web Service is created, a corresponding method is created for each method exposed by the Web Service using the `WebMethod` attribute. There are also several other methods created to enable asynchronous access.

There will also be methods created that allow asynchronous calling of the method. In particular, there will be a method created with `Async` appended to the method name that allows the method to be called asynchronously. This method will have the following signature:

```
public void CalculateCostAsync(int intProductID, int intQuantity)
```

You'll notice that it doesn't return the same type as `CalculateCost`. In fact, it returns nothing, and you will need to make use of another auto-generated construct — the completed event handler for the asynchronous method — for it to work correctly.

Within the proxy, there will be a `CalculateCostCompletedEventArgs` class that inherits from the `System.ComponentModel.AsyncCompletedEventArgs` class. This class exposes a `Result` property that is typed as the return from the synchronous method call — in this case, an `int`. This event arguments class is used in the event handler for the asynchronous event handler completion:

```
public event CalculateCostCompletedEventHandler CalculateCostCompleted;

public delegate void CalculateCostCompletedEventHandler(
  object sender, CalculateCostCompletedEventArgs e)
```

By making use of the asynchronous method and the event handler, we can call the method asynchronously. We first need to create an instance of the proxy class and attach an event handler before invoking the asynchronous call, as follows:

```
// create an instance of the Web Service proxy
CostingService myService = new CostingService();

// add the completed event handler
myService.CalculateCostCompleted +=
  new CalcualateCostCompletedEventHandler(CostingCompleted);

// call the asynchronous method
myService.CalculateCostAsync(5, 10);
```

And, we then need to define the method that handles the completed event:

```
private void CostingCompleted(object sender,
  CalculateCostCompletedEventArgs e)
{
  // do what we need to do with "e"
  // it has a property called Result
  // that is typed correctly
}
```

Once you make an asynchronous call to a Web Service, it is a case of waiting until the attached event handler is fired. You can, however, cancel the asynchronous call by calling the `CancelAsync` method of the proxy instance:

```
myService.CancelAsync();
```

This will fire the completed event handler for any outstanding asynchronous calls; you can handle this by checking the Boolean `Completed` property of the event arguments class passed to the event handler.

Poll for the Completion of a Web Method

It is also possible to call the methods of a Web Service using the *standard* asynchronous methodology provided by the `IAsyncResult` interface.

Each method exposed by the Web Service also has *Begin<method>* and *End<method>* methods that allow the Web Service to be called asynchronously and polled to check if the method has completed.

So, for our `CalculateCost` method, we also have `BeginCalculateCost` and `EndCalculateCost` methods that can be used to call the method asynchronously, as follows:

```
// create an instance of the Web Service proxy
CostingService myService = new CostingService();

// call the asynchronous method
IAsyncResult myResult = myService.BeginCalculateCost(5, 10, null, null);

// loop until the method is completed
while (myResult.IsCompleted == false)
{
  // wait
}

// return the results of the method call
int intCost = EndCalculateCost(myResult);

// deal with the result
```

Note, however, that the `while` loop is blocking execution of the thread until the method returns — effectively no better than a synchronous call. A better solution is to use a callback to handle completion of the method call.

Implement Callback

Instead of polling for completion of the asynchronous call, we can also use the *standard* callback mechanism to retrieve the results of the method call. We still make use of *Begin<method>* and *End<method>* methods but, instead, we add a callback handler as follows:

```
// create an instance of the Web Service proxy
CostingService myService = new CostingService();


// create the callback handler
AsyncCallback myCallback = new AsyncCallback(CostReturned);


// call the asynchronous method
IAsyncResult myResult = myService.BeginCalculateCost(5, 10,
myCallback, null);
```

We then need to implement the callback handler:

```
private void CostReturned(IAsyncResult myResult)
{
  // return the results of the method call
  int intCost = EndCalculateCost(myResult);


  // deal with the result
}
```

Call a One-Way Web Method

Web methods that are marked as one-way methods (by setting the `OneWay` property of the `SoapDocumentMethod` or the `SoapRpcMethod` attributes) are designed to be called and then forgotten about — a *fire-and-forget* scenario. When calling an OneWay method, you use the synchronous method of the proxy and the code within your application continues immediately — there is no delay in waiting for the method call to complete.

Call Remoting Methods Asynchronously

When we looked at remoting earlier, all calls to the remote object were synchronous. Irrespective of how the remote object is activated (either on the server or on the client), the call to the remote method is the same as calling a method of an object created locally.

It is also possible to call the methods of remote object asynchronously — i.e., we can poll for completion of the method call or we can use a callback mechanism to indicate that the method call has completed. First, however, we'll look at calling the methods of a remote object in a *fire-and-forget* scenario.

Implement One-Way Methods by Using the OneWay Attribute

In addition to calling the methods of remote objects and waiting for a response (which is the way that we saw when we looked at remoting earlier), it is also possible to call methods of a remote object and then forget about them — the aforementioned *fire-and-forget* scenario.

Within the remote object, we need to mark the method of the class (or, if we're sharing an interface rather than the full class, both the interface and the class) that we want to make one-way with the `OneWay` attribute, as follows:

```
[OneWay()]
public string DoSomeWork()
{
  // do some work
}
```

Now when we call this method within our client application, irrespective of what happens within the remote object, the call to the `DoSomeWork` method will have no effect on the client application. Any results returned by the `DoSomeWork` method are ignored (we've forgotten about it after all) and any exceptions raised during the call (even if the remote object is not available) will not be propagated to the client application.

Call a Remote Method Asynchronously

Polling for completion of a method call to a remote object is fundamentally the same as polling to check the completion of a call to a Web Service. We return an instance of an `IAsyncResult` object and check the status of the `IsCompleted` property. However, calling a method of a remoting object is slightly more complex, as the asynchronous methods aren't created automatically for you. You need to create a delegate for each method that you want to call asynchronously.

If we take the `DoSomeWork()` method that we've just seen — assuming that we no longer have it marked as a one-way method call — we can make it asynchronous relatively easily.

We must first create a delegate that matches the signature of the method:

```
private delegate string BuildDoSomeWorkDelegate();
```

The naming of the delegate is arbitrary, and we've chosen to prefix the name of the method with `Build` and suffix it with `Delegate`. We can then use this delegate to call the method asynchronously.

We first need to create an instance of the remote class (or an instance of the shared interface) and activate it. Assuming that we've already configured remoting to operate across an HTTP channel, we can create an instance of our remoting object as follows:

```
// create an instance of the remote object
MyRemoteClass objRemote = (MyRemoteClass)Activator.GetObject(
  typeof(MyRemoteClass), "http://remoteServer/object.rem");
```

We can then use this class to create our delegate:

```
// create the necessary delegate
BuildDoSomeWorkDelegate myDelegate = new
  BuildDoSomeWorkDelegate(objRemote.DoSomeWork);
```

We then call the `BeginInvoke()` method of the delegate passing `null` for both parameters (if the method had parameters, these would be passed before the two `null` values that we pass here):

```
// call the method asynchronously
IAsyncResult myResults = myDelegate.BeginInvoke(null, null);
```

We can then poll the `IsCompleted` property to check that the method hasn't completed, before calling the `EndInvoke()` method to retrieve the result of the method call:

```
// loop until the method is completed
while (myResult.IsCompleted == false)
{
  // wait
}

// return the results of the method call
string strReturn = EndInvoke(myResult);

// deal with the result
```

The `while` loop blocks execution and this obviously isn't ideal. We can make use of the callback mechanism to provide a more elegant solution.

Implement Callback

Once we've created a delegate we can create a callback mechanism by making use of the `AsyncCallback` object to handle the return from the method call.

We need to create the remote object and the necessary delegate. This is the same code as we saw when we polled for completion of the method call:

```
private delegate string BuildDoSomeWorkDelegate();
// create an instance of the remote object
MyRemoteClass objRemote = (MyRemoteClass)Activator.GetObject(
  typeof(MyRemoteClass), "http://remoteServer/object.rem");

// create the necessary delegate
BuildDoSomeWorkDelegate myDelegate = new
  BuildDoSomeWorkDelegate(objRemote.DoSomeWork);
```

We then need to create the callback handler before:

```
// create the callback handler
AsyncCallback myCallback = new AsyncCallback(DoSomeWorkReturned);
```

We then pass the `AsyncCallback` instance as the first parameter to the `BeginInvoke()` method:

```
// call the asynchronous method
IAsyncResult myResult = myDelegate.BeginInvoke(myCallback, null);
```

We then need to implement the callback handler:

```
private void DoSomeWorkReturned (IAsyncResult myResult)
{
  // return the results of the method call
  string strReturn = EndInvoke(myResult);

  // deal with the result
}
```

Implement Events in Remoting Applications

As we discussed earlier, handling method calls to a remote object is, in many ways, no different than calling the methods of a normal object. As a developer, you're shielded from most of the complexities of calling the remote methods.

The same is also true of events. However, there is a little more configuration required at the client.

Method calls are one way events (i.e. the client calls the server), but when an event is raised, it is the server calling the client. The client needs to accept the incoming event and must have channels that correspond to the channels that are used to access the remote object.

So if we have a remote object that is configured to accept HTTP connections using the following configuration:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <service>
        <wellknown mode="Singleton" type="MyRemoteClass"
          objectUri="MyRemoteClass" />
      </service>
      <channels>
        <channel ref="http" port="8080" />
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

We also need to define a corresponding channel at the client. As we must accept incoming connections on any port (we talk to the server on port 8080, but it may talk back to us on any port) we need to define a similar client configuration:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <client>
        <wellknown type="MyRemoteClass"
          url="http://remoteServer:8080/object.rem" />
      </client>
      <channels>
        <channel ref="http" port="0" />
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

The client accepts connections across the same channel, but instead of listening on a single port (as the server does), it listens on all the ports that are available.

As it stands, we've still not quite configured the channels correctly. If you try to pass an event from the remote object to the client using this configuration you'll get a runtime error — by default, events cannot be passed across remoting boundaries. We need to configure the formatter for the channel to pass delegates by setting the `typeFilterLevel` property of the formatter element. For both the client and server configuration, we need to set the formatter as a child of the `<channel>` element, as follows:

```
<serverProviders>
  <formatter ref="soap" typeFilterLevel="full">
</serverProviders>
```

We're configuring the SOAP formatter for the HTTP channel (as you'll recall, this is the channel's default formatter) to pass the delegate. We must set this attribute on every channel that we want to accept events.

Create and Fire Events

Events in a remote object are no different from events in a local object. The shared object (or the shared interface) will define the event delegate and the implementation of the object will raise the specific event.

As an example, we can modify our `DoSomeWork()` method from earlier to raise a `WorkDone` event in the same way as we'd raise any other event.

We need to create a delegate for the event that we're going to raise. We're going to assume the simplest delegate we can:

```
public delegate void WorkDoneEvent(object sender, EventArgs e);
```

Within the `MyRemoteClass` class, we can now create an instance of the `WorkDoneEvent` delegate and the `WorkDone` event, and raise it within the `DoSomeWork()` method:

```
[Serializable()]
public class MyRemoteClass : MarshalByRefObject
{
  // event that we can attach to
  public event WorkDoneEvent WorkDone;

  public string DoSomeWork()
  {
    // raise the event
    WorkDone(this, EventArgs.Empty);

    // do some work
  }
}
```

This is the same process for creating and raising the event, irrespective of whether we're raising the event in a remote object or a local object. The one change that we would make to the process is that we'd probably create a class derived from `EventArgs` so that we can pass some meaningful information within the event, rather than passing nothing, as we do in this example.

Passing the Event from the Remote Object to the Client

However, we can't simply attach the client to the events exposed by the remote object. The remote object needs a reference to the local object to call the event — there is no way that we can provide this. The client has a reference to the server, but the server does not have a reference to the client. It would therefore be impossible to add a reference to every possible client to the remote object.

In order to get around this problem, we need to create a helper class that we can share between the client and server. This helper class will perform the "heavy lifting" for us. We'll create this helper class at the client and it will subscribe to the server events. When the server raises its events, the helper class will simply pass the same event to the client.

We first need to create a matching event in the helper, just as we have for the remote object:

```
[Serializable()]
public class MyRemoteClassHelper : MarshalByRefObject
{
  // matching event from MyRemoteClass class
  public event WorkDoneEvent WorkDone;
```

We then need to create a constructor that accepts an instance of the remote object and attaches to its `WorkDone` event:

```
  // variable to hold the remote object
  private MyRemoteClass m_objSource;

  public MyRemoteClassHelper(ref MyRemoteClass objSource)
  {
    // store the server object
    m_objSource = objSource;

    // add the event handler
    m_objSource.WorkDone += new WorkDoneEvent(WorkDoneHandler);
  }
```

And then we need to implement the event handler that simply passes the incoming event to any subscribed event handlers:

```
  private void WorkDoneHandler(object sender, EventArgs e)
  {
    // pass the event out
    WorkDone(sender, e);
  }
```

We also need to ensure that the helper object is never garbage collected by overriding the `InitializeLifetimeService()` method, returning a `null` value:

```
  // override to stop any garbage collection issues
  public override object InitializeLifetimeService()
  {
    return (null);
  }
}
```

We can now use an instance of the helper class to subscribe to the events of the remote object. By making the helper object available to the server, the event can be passed from the server to the helper object. Additionally, because the helper object is also available to the client, the event can then be passed from the helper object to the client application.

Implement Event Handlers for the Events of Remote Objects

The final piece of the puzzle is subscribing the client to the necessary events. As we've seen, we can't do this directly, and we need to make use of the helper object.

Assuming that we've already configured remoting, we can create an instance of the remote object as we would with any other remoting object:

```
// create an instance of the remote object
MyRemoteClass objRemote = (MyRemoteClass)Activator.GetObject(
  typeof(MyRemoteClass), "http://remoteServer/object.rem");
```

We then need an instance of our helper object. We create this by passing in a reference to the remote object that we've created:

```
// create an instance of the helper object
MyRemoteClassHelper objHelper = new MyRemoteClassHelper(objHelper);
```

At this point, the helper would receive the events raised by the server, but wouldn't pass these to the client application. We need to add the event handler for the `WorkDone` event of the helper object:

```
// add the event handler
objHelper.WorkDone += new WorkDoneEvent(WorkDoneHandler);
```

We then need the method, `WorkDoneHandler()`, which handles the event from the helper object:

```
private void WorkDoneHandler(object sender, EventArgs e)
{
  // handle the event
}
```

Whenever the `DoSomeWork()` method of the remote object will raise the `WorkDone` event, that will be propagated from the remote object to the client via the helper object.

Implementing Web Service Enhancements (WSE) 3.0

Web Service Enhancements (WSE) 3.0 is the third iteration of Microsoft's implementation of the WS-* suite of Web Service standards. A complete overview of WSE 3.0 can be found at http://msdn2.microsoft.com/webservices/aa740663.aspx.

As we saw when we looked at both Web Services and Remoting earlier, the developer is shielded from most of the underlying complexities. The same is also true of WSE 3.0.

Enable WSE in Client and Server Applications

Before you can use the features of WSE 3.0, you will need to install it. You'll find the download for WSE 3.0 linked from the WSE 3.0 homepage at http://msdn2.microsoft.com/webservices/aa740663.aspx. Make sure that you install WSE 3.0, as the previous two versions of WSE contain enough differences to be considered completely different implementations.

You need to install WSE 3.0 on both the client and server machines and, by default, it is installed in the `Microsoft WSE\v3.0\` folder in `C:\Program Files\`. The main assembly that you will make use of is `Microsoft.Web.Services3` and this is, along with being in the installation folder, also installed in the Global Assembly Cache (GAC).

Add References to the WSE Assemblies

In order to use WSE 3.0, you need to add a reference to the Microsoft.Web.Services3 assembly. This needs to be done in both the client and server applications and is as simple as selecting the Add Reference option for a project and selecting the `Microsoft.Web.Services3` assembly, as shown in Figure 5-1.

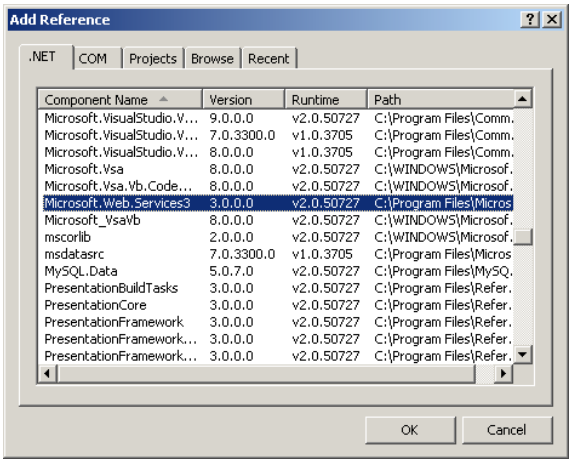

Figure 5-1 - Adding a reference to WSE 3.0

Clicking OK will add the reference to the application. For a Web Site or Web Service project, you'll see that a reference has been added to the `<assemblies>` section of `web.config`. For all other project types, you'll see the `Microsoft.Web.Services3` assembly listed under the References node in Project Explorer.

WSE 3.0 Configuration under Visual Studio 2005

When using WSE 3.0 in conjunction with Visual Studio 2005, you can use a handy utility that allows you to perform the configuration necessary to enable WSE 3.0.

The context menu for projects in Project Explorer has a new entry (WSE Settings 3.0) added that allows you to configure WSE 3.0 very easily. The first page of the dialog is shown in Figure 5-2.

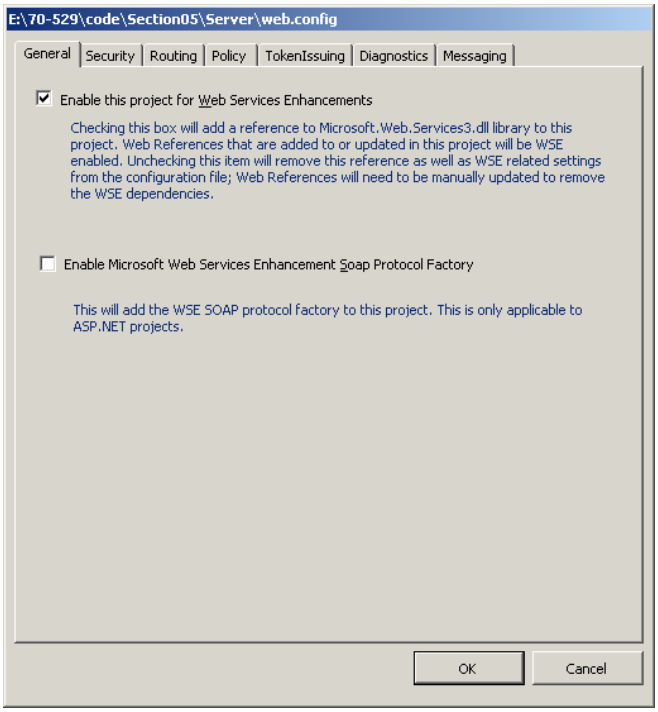

Figure 5-2 – Enabling WSE 3.0 in a Visual Studio project

Selecting the first option — Enable the project for Web Service Enhancements — will add the reference to the `Microsoft.Web.Extensions3` assembly automatically and also configure the project configuration file (either `app.config` or `web.config`) by adding the `<configSections>` element.

The second option — Enable Microsoft Web Services Enhancement Soap Protocol Factory — is only available if the project is an ASP.NET Web Site or ASP.NET Web Service project. This option adds the configuration section necessary to enable the exposed Web Services to access the WSE 3.0 functionality.

When WSE 3.0 is enabled using the WSE 3.0 Settings tool, any proxy classes created will actually be added as two proxies. The standard proxy that inherits from `System.Web.Services.Protocols.SoapHttpClientProtocol` and a WSE 3.0 enabled proxy, its name suffixed with Wse, which inherits from `Microsoft.Web.Services3.WebServicesClientProtocol`.

Manual WSE 3.0 Configuration

If you're not using Visual Studio 2005, you'll need to make the configuration and code changes that are no longer automatically made.

Edit the Web Service Proxy Class to Derive From the WebServiceClientProtocol Class

When using Visual Studio 2005 in conjunction with WSE 3.0, as we have so far, the configuration required to use WSE 3.0 is handled automatically. In addition, you also saw how Visual Studio 2005 also builds a WSE enabled proxy in the client application in addition to the standard proxy.

To enable a proxy to use WSE 3.0, you need to change the proxy class for the Web Service to use the standard proxy that inherits from `Microsoft.Web.Services3.WebServicesClientProtocol` instead of `System.Web.Services.Protocols.SoapHttpClientProtocol`.

You'll need to make this change every time you update the reference to the Web Service.

Add a <configSections> Element to add the <microsoft.web.services3> Section to a Configuration File

All configuration options for WSE 3.0 are stored in either the `web.config` or `app.config` files in a `<microsoft.web.services3>` custom section. You need to add this element to the configuration file by adding an entry to the `<configSections>` element:

```
<configuration>
  <configSections>
    <section name="microsoft.web.services3"
      type="Microsoft.Web.Services3.Configuration.WebServicesConfiguration,
      Microsoft.Web.Services3, Version=3.0.0.0, Culture=neutral,
      PublicKeyToken=31bf3856ad364e35" />
  </configSections>
</configuration>
```

You can now make use of the `<microsoft.web.services3>` section to add configuration settings specific to the various WSE 3.0 features.

Add a <soapExtensionTypes> Element under the <webService> Element in a Configuration File

The power of WSE 3.0 lies in the use of SOAP extensions. In addition to adding SOAP Extensions programmatically using a class derived from the `SoapExtensionAttribute` class, we can also add SOAP Extensions declaratively in the configuration file.

SOAP Extensions are added to the `<soapExtensionTypes>` element of `<webServices>` as follows:

```
<configuration>
  <webServices>
    <soapExtensionTypes />
  </webServices>
</configuration>
```

Within the `<soapExtensionTypes>` element you can use the `<add>`, `<clear>`, and `<remove>` elements to change the SOAP Extensions that are currently enabled.

Adding SOAP Extensions is accomplished using the `<add>` element and specifying the following attributes, all of which are required:

Attribute	Description
Type	Specifies the fully qualified type of the SOAP Extension to add. If the SOAP Extension is in the GAC, it must include the version, culture and public key of the assembly containing the SOAP Extension.
Group	Used with `Priority` to specify the ordering that the SOAP Extensions are applied. This can be either `0` or `1` with a value of zero having the highest priority.
Priority	Used with `Group` to specify the ordering that the SOAP Extensions are applied. Any integer value is allowed with lower values having a higher priority.

Accessing the WSE 3.0 Facilities

When accessing the facilities of WSE 3.0, you make use of an instance of the `Microsoft.Web.Services3.SoapContext` object. This object contains various properties such as, `Addressing`, `Envelope` and `Security`, which allow access to the individual WSE 3.0 facilities.

On the client, there is an instance of this object for both the request and response to the Web Service method. These are available from the proxy class using the `RequestSoapContext` and `ResponseSoapContext` properties.

As there are no changes made to the base class of the Web Service, there aren't any properties directly defined allowing access to the SOAP Context. Instead, you need to make use of the static `Current` property of the `RequestSoapContext` and `ResponseSoapContext` classes. This will return the correct SOAP Context instance.

The WSE 3.0 Message Pipeline

You saw earlier that you can use SOAP Extensions to alter the `Stream` containing the message to and from the Web Service method. This is quite cumbersome and dealing with streams is never the most pleasant experience.

WSE 3.0 adds a further means of altering the message but this time working with a `Microsoft.Web.Services3.SoapEnvelope` class. This is accomplished by adding an input and output pipeline that the message is passed through.

The pipeline contains a series of filters derived from the `SoapFilter` class in the `Microsoft.Web.Services3` namespace. Both the client calling the Web Service and the Web Service itself make use of the input and output pipelines, as shown in Figure 5-3.

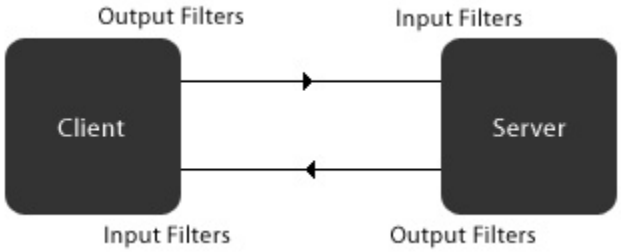

Figure 5-3 – The WSE 3.0 Message Pipeline

On making a request to a Web Service method, the outgoing message is passed through the defined output filters and then passed through any defined SOAP Extensions (the `BeforeSerialize` and `AfterSerialize` stages). The message is then transmitted to the Web Service. On arriving at the Web Service, the message is passed through any defined SOAP Extensions (the `BeforeDeserialize` and `AfterDeserialize` stages) before it is passed to the input pipeline and any defined input filters.

On execution of the Web Service method, the process is repeated when the message to is returned — the message is passed through any defined output filters before any SOAP Extensions handle the message (the `BeforeSerialize` and `AfterSerialize` stages). The message is then returned to the client where it is passed through any defined SOAP Extensions (the `BeforeDeserialize` and `AfterDeserialize` stages) before being passed to the input pipeline and any defined input filters.

In previous versions of WSE, you had manual access to the input and output pipelines and you were free to add filters manually to the pipelines. WSE 3.0 changed this and filters can only be added by the use of policy assertions.

Implement a Policy for a Web Service Application

WSE 3.0 implements the WS-Policy specification and allows its various features to be configured in code or declaratively in configuration files. Policy is mainly concerned with the security requirements of your Web Service and Microsoft has several "turnkey security assertions" already defined. However, you can use policy assertions to configure any custom requirements of your Web Service that are outside of the scope of WSE 3.0.

A WSE 3.0 policy describes the requirements for calling your Web Service. The policy is declared in a policy file and configured and compiled before any communication takes place and effectively sits as a SOAP Filter in the WSE 3.0 Message Pipeline. The policy requirements are then applied in the client application and enforced at the Web Service.

You should always use a policy to configure the requirements of your Web Service. Not only does this separate any configuration requirements from the business logic of your Web Service, it also allows the configuration to be changed without re-compiling the Web Service.

Create a Policy File Manually

A WSE policy file is simply an XML file that contains the details of the policy assertions that are to be added to the WSE pipeline. The simplest policy file contains no assertions at all:

```
<policies xmlns="http://schemas.microsoft.com/wse/2005/06/policy">
</policies>
```

Within the `<policies>` element, you then define the policy assertions that you want to use with the `<extensions>` element. Finally, you configure those assertions in the `<policy>` element.

Each policy assertion that you wish to use is defined as an `<extension>` element (as a child of the `<extensions>` element). There are quite a few turnkey security assertions already defined. You may also define a custom policy assertion by deriving from either the `PolicyAssertion` or `SecurityPolicyAssertion` classes (both in the `Microsoft.Web.Services3.Design` namespace). We'll look at the turnkey security assertions and creating custom policy assertions shortly.

Policy assertions are then grouped into a policy defined as a `<policy>` element, identified by the `name` attribute. A policy file can contain multiple policies, each defined in their own `<policy>` element. The `<policy>` element configures the policy assertions to be used by making use of the policy extensions defined in the `<extensions>` element and providing the configuration for each policy extension.

Creating a policy file by hand is very tedious and it can be accomplished much easier using the WseConfigEditor3 tool. However the WseConfigEditor3 is geared towards the turnkey security assertions that are defined as part of WSE 3.0. In other words, if you want to make use of custom policy assertions, you'll need to manually configure the policy file.

Create a Policy File Using the WseConfigEditor3 Tool

The WseConfigEditor3 tool is a graphical tool, which can be executed in Visual Studio 2005 (VS 2005) or externally, for creating a policy file automatically. Within VS2005, it can be accessed from the context menu for a WSE3.0 enabled project by selecting the WSE Settings 3.0 option or by running WseConfigEditor3.exe from the `tools` folder of the WSE 3.0 installation folder.

Within WseConfigEditor3, clicking the "Policy" tab and selecting the "Enable Policy" option will allow you to add policies to the application. You can either browse to an existing policy file (by clicking the "Browse" button) or you can add a new policy file by clicking the "Add" button.

The WseConfigEditor3 tool only allows you to configure policies using the turnkey security assertions that are defined by WSE 3.0. This is immediately obvious from the first screen of the wizard that is loaded when clicking the "Add" button. From the wizard, you can configure the type of authentication (Anonymous, Username, Certificate or Windows) as well as any authorization rules that you wish to apply. You also have the option of configuring the integrity and confidentiality requirements for the communication.

The simplest direction you can take through the wizard is to elect to secure a "service application" using Username authentication or no authentication and to disable the WS-Security 1.1 Extensions. This will give you a very basic policy file as follows:

```
<policies xmlns="http://schemas.microsoft.com/wse/2005/06/policy">
  <extensions>
    <extension name="usernameOverTransportSecurity"
      type="Microsoft.Web.Services3.Design.UsernameOverTransportAssertion,
      Microsoft.Web.Services3, Version=3.0.0.0, Culture=neutral,
      PublicKeyToken=31bf3856ad364e35" />
    <extension name="requireActionHeader"
      type="Microsoft.Web.Services3.Design.RequireActionHeaderAssertion,
      Microsoft.Web.Services3, Version=3.0.0.0, Culture=neutral,
      PublicKeyToken=31bf3856ad364e35" />
  </extensions>
  <policy name="examplePolicy">
    <usernameOverTransportSecurity />
    <requireActionHeader />
  </policy>
</policies>
```

You can see that we've added two policy assertions in the `<extensions>` element (`UsernameOver-TransportAssertion` and `RequireActionHeaderAssertion`). The fully qualified type of the assertion is specified in the `type` attribute and the `name` specifies the name of the policy assertion's configuration under the `<policy>` element.

The policy assertion, which is named using the `name` attribute of the `<policy>` element, then specifies the configuration of the two policy assertions. Each extension (identified by its `name` attribute) has its own XML configuration specified under the `<policy>` element.

Configure a Policy File in a Configuration File

Once the policy file has been created, it needs to be added to `web.config` or `app.config` in order for it to be used. This is accomplished by adding the configuration file details to the `<policy>` element of the WSE 3.0 configuration section, as follows:

```
<microsoft.web.services3>
  <policy filename=" wse3policyCache.config"/>
</microsoft.web.services3>
```

Applying a Policy to a Web Service

You can configure a Web Service to use a policy in two ways — declaratively or programmatically.

Declaratively Apply a Policy to a Web Service

To apply a defined policy to a Web Service, you make use of the `Policy` attribute to decorate the Web Service definition with the name of the policy to use. If we use the policy that we declared earlier, `examplePolicy`, we can apply this to our Web Service as follows:

```
<Policy("examplePolicy")>
public class ExampleService : System.Web.Services.WebService
```

You can then change the policy by modifying the policy file; any changes will be automatically applied to your Web Service, without requiring any changes to the code.

Programmatically Apply a Policy to a Web Service

It is also possible to apply a policy to a Web Service without using a policy file, by manually creating a policy in code.

To apply a policy, you need to create a class derived from the `Policy` class in the `Microsoft.Web.Services3.Design` namespace. Then, add a constructor that creates the necessary policy assertions and adds them to the `Assertions` collection:

```
public class ExamplePolicy : Microsoft.Web.Services3.Policy
{
  public ExamplePolicy()
  {
    // create any assertions you need

    // add the assertions to the policy
    this.Assertions.Add(myPolicyAssertion);
  }
}
```

We can then apply the coded policy to the Web Service by making using of the `Policy` attribute:

```
<Policy(typeof(ExamplePolicy))>
public class ExampleService : System.Web.Services.WebService
```

Add a Policy to a Client Application

As with Web Services, you can also configure a Client Application declaratively or programmatically.

Declaratively Apply a Policy to a Client Application

To add a policy to calls to a Web Service, you need to call the `SetPolicy()` method of the Web Service proxy, specifying the name of the policy to add:

```
// create the proxy object
ExampleService myProxy = new ExampleService();


// apply the correct policy
myProxy.SetPolicy("examplePolicy");


// call the proxy methods
```

Programmatically Apply a Policy to a Client Application

Adding a policy to be used by a client application is very similar to adding a policy to the Web Service. You will need to create a class derived from the `Policy` class in the `Microsoft.Web.Services3.Design` namespace. Then, add a constructor that creates the necessary policy assertions and adds them to the `Assertions` collection:

```
public class ExamplePolicy : Microsoft.Web.Services3.Policy
{
  public ExamplePolicy()
  {
    // create any assertions you need


    // add the assertions to the policy
    this.Assertions.Add(myPolicyAssertion);
  }
}
```

You can then add the policy to the calls to the Web Service by creating an instance of the proxy and adding any necessary security tokens (by calling the `SetClientCredential` and `SetServiceCredential` methods) before adding the policy to the proxy:

```
// create the proxy object
ExampleService myProxy = new ExampleService();

// add the required security credentials
myProxy.SetClientCredential(myClientCredential);
myProxy.SetServiceCredential(myServiceCredential);

// apply the correct policy
ExamplePolicy myPolicy = new ExamplePolicy();
myProxy.SetPolicy(myPolicy);
// call the proxy methods
```

The one extra piece of the process is the addition of the required security credentials to the message. All security credentials are derived from the `SecurityToken` class in the `Microsoft.Web.Services3.Security.Tokens` namespace.

Security Tokens

There are three objectives when securing web services:

- **Authentication** – ensuring that the sender of the message is who they say they are.
- **Integrity** – ensuring that a message has not been tampered with during its transmission.
- **Confidentiality** – ensuring that the message can only be viewed by authorized parties.

In order to authenticate the sender of a message, you need to add security credentials corresponding to the sender. These security credentials can also be used to ensure the integrity and confidentiality of the message.

WSE 3.0 supports all of the security credentials defined in the WS-Security 1.1 specification. These security credentials are represented by security tokens, derived from the `SecurityToken` class in the `Microsoft.Web.Services3.Security.Tokens` namespace. Some of the more common security tokens are as follows:

- `UsernameToken`
- `KerberosToken`
- `X509SecurityToken`

In addition, there is also an abstract `BinarySecurityToken` class that you can inherit from to implement custom security tokens.

The Turnkey Security Assertions

As mentioned previously, policy is mainly concerned with security and, in particular, the details of the WS-Security 1.1 specification. WS-Security 1.1 is an OASIS standard and more details can be found at http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss.

Rather than having to create your own policy assertions when using WSE 3.0, there are several turnkey security assertions as can be seen at http://msdn2.microsoft.com/en-us/library/aa528756.aspx. These six security assertions are summarized below:

- `AnonymousForCertificateAssertion` – the client is not authenticated and the service is authenticated using an X509 Certificate (an `X509SecurityToken`). The server's X509 Certificate can be used to ensure the integrity and confidentiality of the message.
- `KerberosAssertion` – the client and server are authenticated using Kerberos tickets (a `KerberosToken`). The integrity and confidentiality of the message can be ensured using Kerberos tokens.
- `MutualCertificate10Assertion` – the client and server are authenticated using X509 Certificates. The integrity and confidentiality of the message can be ensured using the X509 Certificates. This assertion is compatible with WS-Security 1.0.
- `MutualCertificate11Assertion` – the client and server are authenticated using X509 Certificates. The integrity and confidentiality of the message can be ensured using the X509 Certificates. This assertion is compatible with WS-Security 1.1.
- `UsernameOverTransportSecurity` – the client is authenticated using a username and password (a `UsernameToken`). The integrity and confidentiality of the message is not covered in the assertion and is assumed to be provided by the underlying transport mechanism (e.g. HTTPS).
- `UsernameForCertificateAssertion` – the client is authenticated using a username and password and the server is authenticated using an X509 Certificate. The integrity and confidentiality of the message can be ensured using the server's X509 Certificate.

Create a Custom Policy Assertion

As we've already seen, there are several policy assertions already specified by the default WSE 3.0 installation and these should cover most of the situations that you will encounter. There may, however, be cases where you need to create your own security assertions. The means to do this depends upon whether you're creating a security or non-security based policy assertion.

In either case, you need to define `SoapFilter` derived classes that you, using a class derived from `PolicyAssertion`, return for each of the possible states of the WSE 3.0 Message Pipeline, as shown in Figure 5-3. You return the correct filter by overriding methods of the `PolicyAssertion` class:

- `CreateClientOutputFilter` – return the `SoapFilter` to be used for a request to a Web Service method at the client.
- `CreateServiceInputFilter` – return the `SoapFilter` to be used when a request to a Web Service method is received at the server
- `CreateServiceOutputFilter` – return the `SoapFilter` to be used at the server when a Web Service method is complete.
- `CreateClientInputFilter` – return the `SoapFilter` to be used at the client before the response from the Web Service method is returned.

If the policy assertion doesn't add a filter for a particular place in the WSE 3.0 message pipeline, then you don't need to override that particular method; by default, a policy assertion doesn't add any filters to the WSE 3.0 pipeline.

The custom policy assertion can then be added to the policy file using the `<extension>` element. It is also possible to add configuration of the policy assertion and you will need to override the `GetExtensions` and `ReadXml` methods of the `PolicyAssertion` class to read the configuration correctly.

Custom Non-Security Policy Assertions

To build a non-security policy assertion, you can follow the process outlined above directly. You need to create a `SoapFilter` derived class for the required stages of the WSE 3.0 Message Pipeline, overriding the `ProcessMessage` method to apply or enforce the policy assertion.

You then create the policy assertion itself by creating a class derived from `PolicyAssertion`, overriding `CreateClientOutputFilter`, `CreateServiceInputFilter`, `CreateServiceOutputFilter` and `CreateClientInputFilter` to return the correct filter.

The policy assertion can then be included in a policy file using the `<extension>` element and any configuration defined underneath the `<policy>` element.

A very thorough example of creating a non-security policy assertion can be found on MSDN at http://msdn2.microsoft.com/en-us/library/aa529313.aspx.

Custom Security Policy Assertions

Custom security policy assertions follow a similar principal to the non-security policy assertions, except that the classes from which they are derived are different. Rather than deriving from the `SoapFilter` and `PolicyAssertion` classes, you derive from slightly different classes — these classes already provide some of the building blocks required for security.

Instead of deriving from `SoapFilter` to create the filters to be added to the WSE 3.0 Message Pipeline, you derive from a different class depending upon whether you're creating the input or output filter:

- For output filters, you derive from the `SendSecurityFilter` class, overriding the `SecureMessage` method.
- For input filters, you derive from the `RecieveSecurityFilter` class and override the `ValidateMessageSecurity` method.

The policy assertion itself is also derived from a slightly different class. Instead of deriving from the `PolicyAssertion` class, you derive from `SecurityPolicyAssertion`, overriding the necessary methods to return the correct filter for the various stages in the WSE 3.0 Message Pipeline.

A very thorough example of creating a custom security policy assertion can be found on MSDN at http://msdn2.microsoft.com/en-us/library/aa528788.aspx.

Using the Custom Policy Assertion

You've already seen how to apply policy assertions to both the Web Service and a client application. A custom policy assertion is no different than a turnkey security assertion and can be added in a policy file (defined using an `<extension>` element) or in a custom policy by creating an instance of the policy assertion in the `Policy` derived class.

Implement WSE SOAP Messaging

One of the key tenets of Service Orientated Architecture (SOA) is messaging; and messaging in WSE 3.0 has several major advantages over previous versions. WSE Messaging allows you to add additional capabilities to your Web Services, such as changing the protocol used (such as to TCP from HTTP), specifying one-way or bi-directional message and handling attachments.

When we look at WSE SOAP Messaging, we start to move away from Web Services and into the realms of Service Oriented Architectures (SOA) and the Windows Communication Foundation. We're moving from Web Services to the more generic Services field.

To TCP or HTTP?

When using the HTTP protocol to communicate with a Web Service, you're constrained to using IIS to host your Web Service. There are instances when you may not want to use IIS and may want to host your Web Service in an application (such as a console application or a Windows service). In these cases you can change the protocol to TCP and, with a little bit of configuration, your application can receive those messages. WSE 3.0 allows the easy use of TCP instead of HTTP by specifying the address as a soap.tcp:// address rather than an http:// address.

For example, to connect to the local machine using TCP you'd use the following address:

```
soap.tcp://localhost/
```

By default, this uses port 8081, which may not be available, and you can specify the correct port number, in this case 1974, as follows:

```
soap.tcp://localhost:1974/
```

Implement One-way SOAP Messaging

One-way messaging is implemented in WSE 3.0 using the `SoapSender` and `SoapReceiver` classes in the `Microsoft.Web.Services3.Messaging` namespace.

Send Messages

To send a one-way message, you need to create a `SoapEnvelope` to send and then send this to a configured `SoapSender` instance.

Whether we're sending across HTTP or TCP, we send the message the same way. The only difference is the address to which we send the message. For HTTP, we'd specify the web address as:

```
string toAddress = "http://remoteServer/Receiver";
```

And for TCP, we'd specify the address as follows:

```
string toAddress = "soap.tcp://remoteServer/Receiver";
```

We first create the `EndpointReference` to send to and configure the `SoapSender` object:

```
EndpointReference myEndpoint = new EndpointRefernce(new Uri(toAddress));
SoapSender mySender = new SoapSender(myEndpoint);
```

We can then create a `SoapEnvelope` to send specifying the action and the content (the message) for the envelope:

```
SoapEnvelope myEnvelope = new SoapEnvelope();
myEnvelope.Context.Address.Action = new Action(toAddress);
myEnvelope.SetBodyObject("Message to send");
```

We can then send the message by calling the `Send` method of the `SoapSender` object:

```
mySender.Send(myEnvelope);
```

Create a Class to Receive Messages

To receive messages, we need to create an instance of the `SoapReceiver` class and override the `Receive` method to handle the incoming messages:

```
namespace SoapReceivers
{
  public class MyReceiver : Microsoft.Web.Services3.Messaging.SoapReceiver
  {
    protected override void Receive (SoapEnvelope message)
    {
      // handle the incoming message
    }
  }
}
```

Receiving the Message across HTTP

If receiving the message across HTTP, we can simply register an ASP.NET handler in IIS. The `SoapReceiver` class implements the `IHttpHandler` interface so it can be used directly within the `<httpHandlers>` section of `web.config`:

```
<httpHandlers>
  <add verb="*" path="Receiver.ashx"
    type="SoapReceivers.MyReceiver" />
</httpHandlers>
```

Receiving the Message across TCP

To receive the message across TCP, we need to register the `SoapReceiver` object to receive messages for the required address. WSE 3.0 implements a static `SoapReceivers` class that has an `Add` method we can call to register the receiver.

First, we create the correct `EndpointReference` for the `SoapReceiver`:

```
string toAddress = "soap.tcp://remoteServer/Receiver";
EndpointReference myEndpoint = new EndpointRefernce(new Uri(toAddress));
```

And then we register the receiver:

```
SoapReceivers.Add(myEndpoint, typeof(SoapReceivers.MyReceiver));
```

Implement Bi-directional SOAP Messaging

Bi-directional messaging is implemented in a similar way to one-way messaging, except that you make use of the `SoapClient` and `SoapService` classes.

These classes inherit from `SoapSender` and `SoapReceiver` that we've just looked at for one-way messaging, so both these classes can be used to implement one-way and bi-directional messaging. To send, create a class that derives from `SoapClient` and call its `Send` method to send the message and derive a class from `SoapService`, overriding its `Receive` method to receive the messages.

The process for implementing bi-directional messaging is only slightly more complex.

Create a Class to Send Messages

As the `SoapClient` class is abstract, we need to create an instance of it in order to send a message. We must also configure the class, so we need to accept an instance of an `EndpointReference` in the constructor and pass this to the base class:

```
public class MyMessageSender : Microsoft.Web.Services3.Messaging.SoapClient
{
  public MyMessageSender (EndpointReference myDestination)
    :  base (myDestination)
  {
  }
}
```

We could now, to send a one-way message, use this class as is and call the `Send` method passing in the `SoapEnvelope` to send.

To add a bi-directional method, we need to define another method that we can use (or several methods if we require them). There is no method to overload and we're free to call the method whatever we want as long as we mark the method using the `SoapMethod` attribute specifying the method to call in the Web Service:

```
[SoapMethod("MyRemoteMethod")]
public SoapEnvelope RemoteMethodSend(SoapEnvelope message)
{
  return (base.SendRequestResponse("MyRemoteMethod", message);
}
```

Create a Class to Receive Messages

In order to receive bi-directional messages at the Web Service, we need to create a class derived from `SoapService`:

```
public class MyMessageReceiver : Microsoft.Web.Services3.Messaging.
SoapService
{
}
```

If we want to handle one-way messages, we can override the `Receive` method in the same was as we saw when we derived the class from `SoapReceiver`.

To handle the bi-directional method, we need to create a new method and add the `SoapMethod` attribute specifying the name of the method called:

```
[SoapMethod("MyRemoteMethod")]
public SoapEnvelope RemoteMethodReceive(SoapEnvelope message)
{
  return (base.SendRequestResponse("MyRemoteMethod", message);
}
```

The name specified in the `SoapMethod` attribute at the server must match the method name passed to the `SendRequestResponse` method in the client. The actual names of the methods in the class are irrelevant — it's the `SoapMethod` attribute that determines which method in the server receives the message.

Configuring the Sender and Receiver

Configuring the sender and receiver for bi-directional messaging is exactly the same as for one-way messaging. For the sender, pass the correct destination address to the constructor of the `SoapClient` derived class; for the receiver, either register the HTTP handler or create an instance of the `SoapService` derived class and add it to the `SoapReceivers` collection.

Adding Attachments to Method Calls

Previous versions of WSE used Direct Internet Message Encapsulation (DIME) to handle attachments for Web Services. In WSE 3.0, attachments are handled using the SOAP Message Transmission Optimization Mechanism (MTOM) specification (http://www.w3.org/TR/soap12-mtom/).

When enabling a Web Service or client application for WSE 3.0, you automatically enable the handling of MTOM encoded messages. However, the default configuration settings will allow you to receive and handle MTOM encoded messages at a Web Service but not to send MTOM encoded messages from a client application.

From the Messaging tab of the WSE 3.0 Configuration tool, you have two main options that determine now MTOM is handled.

- Client Mode – defaults to `off` and determines whether requests to Web Service methods will be MTOM encoded if required. A value of `on` will allow attachments to be added to the request to the Web Service.
- Server Mode – defaults to `optional` and determines whether the response from the Web Service is MTOM encoded, with `optional` indicating that the response will match the request (it will be MTOM encoded if the request was MTOM encoded). A value of `always` indicates that the response will always be MTOM encoded and a value of `never` indicates that the response will never be MTOM encoded.

It is also possible to set these values directly in `web.config` as the `clientMode` and `serverMode` attributes of the `<mtom>` element of the `<messaging>` element in the `<microsoft.web.services3>` configuration section. Note, however, that the default values don't appear in `web.config` — it's only if you pick values that are non-default that you'll get a corresponding entry in `web.config`.

Handling Attachments

Attachments in WSE 3.0 are considered to be an array of bytes. We can pass a byte array as a parameter to a Web Service method from a client application and we can return a byte array from a Web Service method to the client application.

We can define a Web Service method that accepts a byte array as a parameter:

```
public int UploadImage(byte[] myFile)
```

Or, we can return a byte array from the Web Service method:

```
public byte[] RetreiveImage(int intImageID)
```

In either case, the byte array (the attachment) will be MTOM encoded and transmitted as part of the SOAP message.

Note, however, that as it's a byte array that we're transmitting there is no way to determine the type of the attachment from the byte array. We can't even retrieve the name of the file that we're transmitting. For these purposes, you will have to implement an alternative mechanism for returning any other information about the attachment.

Sending Attachments

If we're dealing with files, and this will be the most common type of attachment we will use, we can use some of the `static` members of the `File` class in the `System.IO` namespace to very easily read the entire contents of the file into a byte array:

```
byte[] myFile = System.IO.File.ReadAllBytes(@"C:\image.jpg");
```

We can then pass this to a Web Service method by adding it as a parameter to the Web Service method or returning it from a Web Service method.

Receiving Attachments

When receiving an attachment, we can use `static` members of the `System.IO.File` class to save the file to disk. Assuming that `myFile` is a byte array, we can use the `WriteAllBytes` method to save the file to disk:

```
System.IO.File.WriteAllBytes(@"C:\image.jpg", myFile);
```

However, as we've already seen, we have no way of returning the filename along with the byte array and you will need to return it using some alternative mechanism.

Route SOAP Messages Using a WSE Router

SOAP Routing allows you to route requests for the methods in one Web Service to another Web Service. This is for use in situations where your Web Service resides on a private network and you don't want to expose the network to the Internet. Routing also allows you to hide the internal implementation of your network and, through simple configuration changes, allow you to route requests to an alternative server — ideal if you need to perform maintenance on a particular server.

Create a WSE Router Application

A WSE Router is a class that inherits from the `SoapHttpRouter` class in the `Microsoft.Web.Services3.Messaging` namespace. This is an HTTP Handler that sits and processes incoming requests to determine where the request is to be redirected to.

To enable this processing you need to override the `ProcessRequestMessage` method, perform the necessary processing and return the `Uri` for the message to be routed to:

```
namespace SoapRouters
{
  public class ContentRouter
    : Microsoft.Web.Services3.Messaging.SoapHttpRouter
  {
    protected override Uri ProcessRequestMessage
      (Microsoft.Web.Services3.SoapEnvelope message)
    {
        // implement routing rules
    }
  }
}
```

When we override the `ProcessRequestMessage` method, we provide custom rules for routing. This is known as content-based routing as it is the content of the actual message that determines the routing of the message.

Once you've created your derived `SoapHttpRouter`, you then need to configure ASP.NET to handle requests for a specific Web Service in the `<httpHandlers>` element of `web.config`:

```
<httpHandlers>
  <add verb="*" path="RoutedService.asmx"
    type="SoapRouters.ContentRouter, ContentRouter" />
</httpHandlers>
```

Configure a Referral Cache for Routing

If you don't want to perform content based routing, it is necessary to create a referral cache that you can use to perform all the necessary routing. The default behavior of the `SoapHttpRouter` is to use a referral cache to perform the routing and it is only by overriding the `ProcessRequestMessage` method that we override this behavior.

We can then easily add a referral cache based WSE Router by using the standard `SoapHttpRouter` class to handle all requests for Web Services in your application. This can be configured very easily in the `<httpHandlers>` section of `web.config`:

```
<httpHandlers>
  <add verb="*" path="*.asmx"
    type="Microsoft.Web.Services3.Messaging.SoapHttpRouter,
      Microsoft.Web.Services3, Version-3.0.0.0, Culture=neutral,
      PublicKeyToken=31bf3856ad364e35" />
</httpHandlers>
```

All requests to Web Services will now be passed to the `SoapHttpRouter` class that, by default, looks for a referral cache file to determine where requests are handled. This is specified in the `<referrals>` element of `<microsoft.web.services3>` in `web.config`:

```
<microsoft.web.services3>
  <referrals>
    <cache name="referralCache.config" />
  </referrals>
</microsoft.web.services3>
```

The Referral Cache File

The referral cache file is used for each request that the `SoapHttpRouter` receives to determine the routing instructions for the request. Within the referral cache, there must be a routing instruction for every possible request that the WSE router is expected to handle. If there isn't a routing instruction for a received request, a SOAP Fault is sent back to the caller.

An example referral cache file is given below:

```
<?xml version="1.0" encoding="utf-8" ?>
<r:referrals xmlns:r="http://schemas.xmlsoap.org/ws/2001/10/referral">
  <r:ref>
    <r:for>
      <r:exact>http://localhost/ServiceA.asmx</r:exact>
    </r:for>
    <r:if/>
    <r:go>
      <r:via>http://localhost/ServiceB.asmx</r:via>
    </r:go>
  </r:ref>
</r:referrals>
```

As you can see, the file is quite simple. The `<r:ref>` element is used to hold routing instructions for a destination Web Service and the referral cache file can contain as many `<r:ref>` elements as required.

Within the `<r:ref>` element, there is always a `<r:for>` element that is used to determine which Web Services the routing instruction is for. The `<r:for>` element contains one of the following elements:

- `<r:exact>` - matches to an exact request URL
- `<r:prefix>` - matches the start of a request URL

The `<r:if>` element can be used to specify any conditional decisions that need to be made before the request can be routed.

The `<r:go>` element that is used to specify where the request is to be routed. There can be multiple `<r:via>` elements within the `<r:go>` element and each of these specifies a destination to be routed to. If there are multiple `<r:via>` elements, the selection of which `<r:via>` element to use is non-deterministic and one of the entries will be picked at random.

Applying a Policy to Incoming Requests

We saw earlier how to apply policy assertions to a Web Service using the `Policy` attribute. When we're using a WSE Router, we no longer have a direct connection to the Web Service and instead need to apply the policy assertion to the WSE Router.

We can't do this for a WSE Router using the Policy attribute; instead, we need to override the `GetRequestPolicy()` method in the `SoapHttpRouter` derived class to return an instance of the correct policy.

Assuming we're using the `ExamplePolicy` policy that we defined earlier, we can apply this to the WSE Router by content based router overriding the `GetRequestPolicy()` method as follows:

```
namespace SoapRouters
{
  public class ContentRouter
    : Microsoft.Web.Services3.Messaging.SoapHttpRouter
  {
    protected override Uri ProcessRequestMessage
      (Microsoft.Web.Services3.SoapEnvelope message)
    {
        // implement routing rules
    }
    protected override Microsoft.Web.Services3.Design.Policy
    GetRequestPolicy()
    {
        return new ExamplePolicy();
    }
  }
}
```

We can also add the same policy to a referral cache based WSE Router by deriving from the `SoapHttpRouter` class and keeping the default `ProcessRequestMessage` behavior:

```
namespace SoapRouters
{
  public class ReferralRouter
    : Microsoft.Web.Services3.Messaging.SoapHttpRouter
  {
    protected override Microsoft.Web.Services3.Design.Policy
    GetRequestPolicy()
    {
        return new ExamplePolicy();
    }
  }
}
```

Creating and Access a Serviced Component and Using Message Queuing

Create, Configure and Access a Serviced Component

Create a Serviced Component

There are several steps that must be completed to create a serviced component.

1. Add a reference to the `System.EnterpriseServices` namespace to your project.
2. Inherit from `ServicedComponent` and add a default constructor

```
using System.EnterpriseServices;
public class myServicedComponent : ServicedComponent
{
  public myServicedComponent()
  {
  }
}
```

3. Make the class visible to COM+ by adding the `ComVisible` attribute to the class definition. By default, all classes in an assembly are hidden from COM+ by the `ComVisible(false)` applied to the assembly in `AssemblyInfo.cs`. You need to apply `ComVisible(true)` to each class you want to make visible to COM+:

```
using System.EnterpriseServices;
using System.Runtime.InteropServices;


[ComVisible(true)]
public class myServicedComponent : ServicedComponent
{
  public myServicedComponent()
  {
  }
}
```

4. Specify how the components are activated by COM+. You can specify whether components are activated in the calling client's process (`Library`, the default setting) or whether the component is activated in its own process (`Server`). You can specify the type you're after by adding the `ApplicationActivation` attribute to `AssemblyInfo.cs`:

```
[assembly: ApplicationActivation(ActivationOption.Server)]
```

5. Give the assembly containing the serviced components a strong name. Without a strong name, the assembly cannot be used by COM+.

Add Attributes to a Serviced Component

Most COM+ functionality is added to serviced components by using attributes. These attributes are used by COM+ to configure the serviced component and the configuration is stored within COM+. Attributes specify the initial configuration for a serviced component but this can be modified using the COM+ plugin for MMC.

There are over 20 attributes specified in the `System.EnterpriseServices` namespace that can be applied to your class to control its interaction with COM+. We'll only look at a few of them here.

Transactions

Adding the requirements for transactions to a serviced component is accomplished using the `Transaction` attribute. There are several properties that we can pass to the attribute when it is defined:

- `Isolation` – the `TransactionIsolationLevel` enumeration allows you to set the isolation level of the transaction: `Chaos`, `ReadCommitted`, `ReadUncommitted`, `RepeatableRead`, `Serializable`, `Snapshot`, or `Unspecified`.
- `Timeout` – the time, in seconds, that the transaction will be allowed to run before it is timed out by the transaction coordinator.

- `Value` – the `TransactionOption` enumeration allows you to determine how the serviced component deals with transactions: `Disabled`, `NotSupported`, `Required`, `RequiresNew`, or `Supported`.

Object Pooling

Using the `ObjectPooling` attribute you can specify that only a set number of objects can be created. There are several properties that we can pass to the attribute when it is defined:

- `CreationTimeout` – the time, in milliseconds, that a client application will wait for an object to become available before it times out.
- `MaxPoolSize` – the maximum number of objects that will ever exist in the pool.
- `MinPoolSize` – the minimum number of objects that are contained in the pool. This specifies how many are created on startup and how many are maintained in the pool.

Queued Components

In many cases, it will be necessary to use components that make use of Microsoft Message Queuing (MSMQ). There are two attributes, `InterfaceQueuing` and `ApplicationQueuing`, which can be used to instruct COM+ that your application makes use of MSMQ. We'll look at both of these attributes in more detail later.

Register a Serviced Component

Before you can use your component, it must be registered with COM+. There are several ways that you can install serviced components. Two of these are by using the Microsoft Management Console or the Services Installation Tool.

Microsoft Management Console

The Component Services snap in for MMC can be accessed from the Administrative Tools section of the Control Panel. Expanding the Component Services -> Computers -> My Computer node allows you to see the various objects of COM+. Expanding the COM+ Applications node allows you to view all of the applications on the computer, as shown in Figure 6-1.

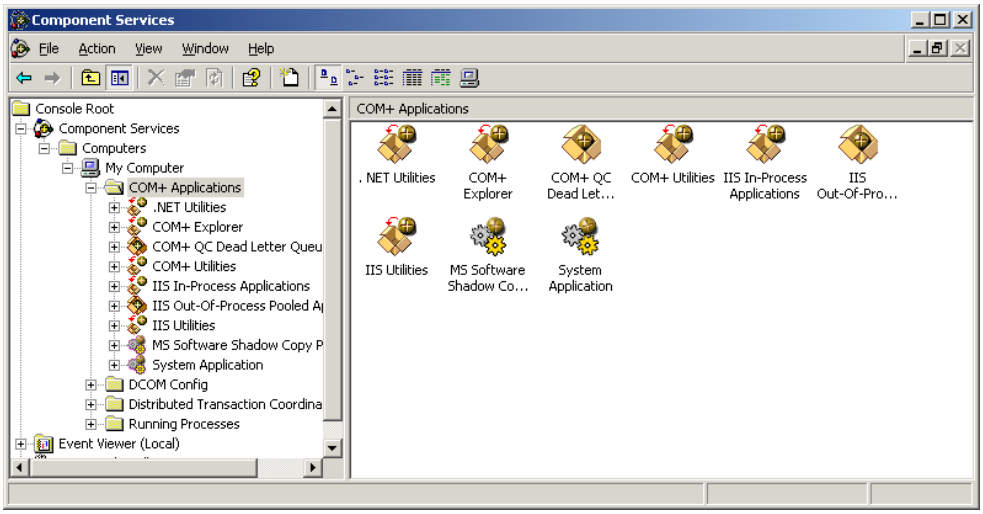

Figure 6-1 – Components Services console

You can add a new application to COM+ by selecting New -> Application from the context menu of the COM+ Applications node. Once you've created the application, you can drag and drop the assemblies containing the services components into the application's Components folder.

Services Installation Tool

The .NET SDK provides an installation tool, `regsvcs.exe`, that you can use to register an assembly with COM+.

From the command line, you specify the name of the assembly and the application name:

```
regsvcs MyAssembly.dll MyComApplication
```

This will create a COM+ application called `MyComApplication` and add the serviced components from then `MyAssembly.dll` assembly to the application.

Implement Security

If you extend a COM+ application in MMC, you'll see that there are three "folders" contained within the application. Two of these, Components and Legacy Components, contain the serviced components for the application. The third, Roles, is used to configure security of the COM+ application.

Roles, in COM+, are completely unrelated to Windows security and you must configure the roles within COM+ to contain users and groups from the Windows security model. These can then be used within your serviced component by making use of several other attributes:

- `ApplicationAccessControl` – this attribute is applied to the assembly containing your serviced components. Passing `true` to the attribute enables roles based security for all the serviced components in the assembly.
- `ComponentAccessControl` – this attribute is applied to each serviced component. Passing `true` to the attribute enables role based security for the serviced component.
- `SecurityRole` – this attribute is also added to the serviced component and takes at least one parameter — the name of the role to associate with the serviced component (the role will be created within COM+ if it doesn't already exist). You can also pass a Boolean, optional second parameter that specifies whether the Everyone Windows security group is added to the role in COM+.

Within your application, you can then make use of roles by using the properties and methods of the `ContextUtil` class:

- `IsSecurityEnabled` – this property returns `true` if role-based security has been enabled for the current serviced component.
- `IsCallerInRole ("role")` – this method returns `true` if the caller is in the specified role.

Using a Serviced Component

Using a serviced component that is registered with COM+ is no different to using a component that isn't:

- Register the component as you would any other component. In Visual Studio, select Add Reference from the project's content menu or from the main Project menu.
- Create an instance of the serviced component using the `new` keyword.
- Access the properties and methods of the serviced component.

As you can see, there is no change to the way that you reference, instantiate and access the service component. Deriving from the `ServicedComponent` class, and using the attributes from the `System.EnterpriseServices` namespace, shields you from all the complexities of COM+.

Create, Delete and Set Permissions on a Message Queue

Message queuing allows you to send messages to an application and, even if the application isn't running, expect those messages to be delivered. Microsoft Message Queuing (MSMQ) acts almost like a postal system, in that it accepts messages for delivery and ensures that the messages are delivered.
MSMQ defines two types of queue:

- Public – a public queue is registered in Active Directory and can be discovered by browsing the network. Public queues are named as `MachineName\QueueName`.
- Private – a private queue has exactly the same functionality as a public queue except that it isn't discoverable through Active Directory. Private queues are named as `MachineName\Private$\QueueName`.

In addition to the public and private queues that you may define for your applications, there are two other queues that may be available: journal queues and dead-letter queues.

Queues can also be marked as transactional. Making use of the queue will cause the transaction coordinator to manage the distributed transaction. You can then interact with previous stages of the transaction ensuring that any transactional business logic in your application is handled correctly.

Create a Message Queue Manually

To manually create a message queue, you need to launch the Computer Management console from the Administrative Tools section of the Control Panel. Expanding the Message Queuing node under Services and Applications allows you to view all the message queues on the computer, as shown in Figure 6-2.

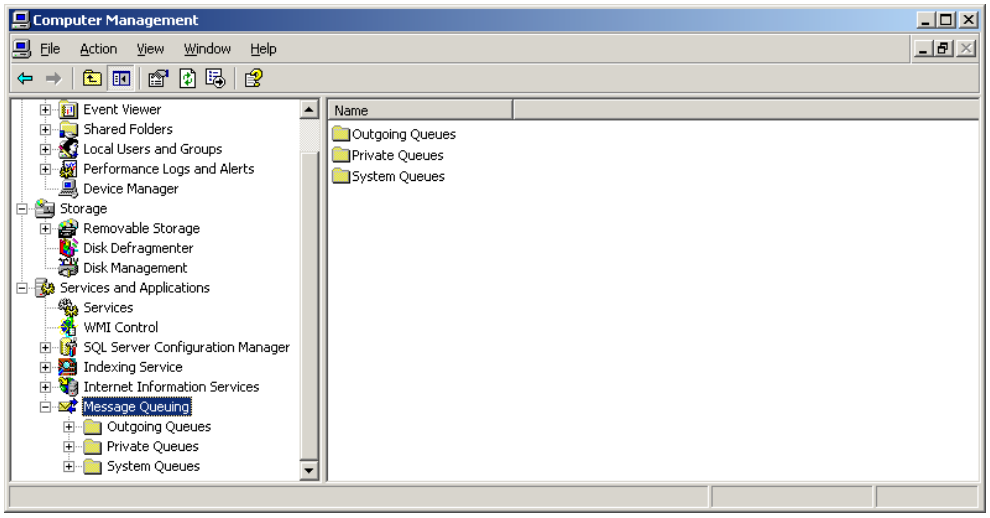

Figure 6-2 – Message Queuing in the Computer Management console

You can create a new private queue by clicking on the Private Queues node and selecting New -> Private from the context menu. From the wizard, you can specify the name of the message queue and whether you want the queue to be transactional.

Create a Message Queue Programmatically

All of the functionality for queuing is contained in the `System.Messaging` namespace and there are two static methods of the `MessageQueue` class that you'll use to create a new message queue:

- `MessageQueue.Exists("queuename")` – returns `true` if the queue already exists and you can access the queue by simply creating a new instance of the `MessageQueue` class and specifying the name of the queue to the constructor. If it returns `false`, then you may need to create the message queue before you can use it.
- `MessageQueue.Create("queuename")` – creates a non-transactional message queue with the specified name and returns the queue for immediate use. An overloaded `Create` method accepts a Boolean second parameter that indicates whether the queue is to be transactional or not.

Delete a Message Queue

You can delete a message queue in two ways:

- Select the queue in the Computer Management console and select Delete from the queue's context menu.
- Call the static `MessageQueue.Delete` method passing the name of the queue you want to delete.

Set Permissions for a Message Queue

By default everyone (the Everyone Windows security group) has full access to message queues. You can set the permissions for a message queue in the Computer Management console by selecting the Properties option from the queue's context menu. The Security tab allows you to specify permissions for the queue based on Windows security users, groups and computers.

It is also possible to set permissions on a message queue programmatically by using the `SetPermissions` method of the queue itself:

```
MessageQueue myQueue = new MessageQueue(".\private$\myQueue");
myQueue.SetPermissions("BUILTIN\Administrators",
  MessageQueueAccessRights.FullControl);
```

Here, we've granted the BUILTIN\Administrators group full control on the queue. There are several values in the `MessageQueueAccessRights` enumeration that allow you to define very granular permissions on the queue.

It is also possible to reset the permissions on the queue to their default values (the Everyone Windows security group having full control) by calling the `ResetPermissions` method of the queue.

Sending and Receiving Messages to a Message Queue and Delete Messages from a Message Queue

Create a Message

To add a message to a message queue, you create an instance of the `Message` class in the `System.Messaging` namespace and configure various properties before posting the message to the queue.

There are over 50 properties that can be set for a Message but the ones that you'll most likely come across are as follows:

- `Body` – an `Object` representing the message that is being sent via the queue.
- `Label` – a label for the message that can be used for several purposes. You can, perhaps, think of it as a title or name of the message.
- `Priority` – a member of the `MessagePriority` enumeration indicating the relative priority of the message.

Send a Message

Once you've created the `Message` that you wish to send, you can add it to the message queue by calling the `Send` method of the `MessageQueue` instance:

```
// create the message queue
MessageQueue myQueue = new MessageQueue(".\private$\myQueue");

// create and configure the message
Message myMessage = new Message();
myMessage.Body = "This is a test message";
myMessage.Label = "Message test";
myMessage.Priority = MessagePriority.Normal;

// send the message
myQueue.Send(myMessage);
```

In addition to sending an instance of a `Message` object to a `MessageQueue`, it is also possible to send any serializable object using the `Send` method. Passing any `Object` other than a `Message` to the `Send` method internally creates a new `Message` and places the passed in `Object` as the `Body` of the message. It is also possible to set the `Label` of a message passed in this way by using an overloaded version of the `Send` method passing the required `Label` as the second parameter to the `Send` method.

Receive a Message

Once you've sent a message to the message queue, you must then retrieve the message from the queue. All queue actions are asynchronous. In other words, you send a message to the queue and there is no requirement for the receiver of the queue to be running at the time the message is sent. And, if the receiver isn't running when the message is sent, it will be queued until the receiver asks for the message from the queue.

Messages are received from the queue using the `Receive` method of the `MessageQueue` object.

This method blocks until a message is received and returns a `Message` object corresponding identical to the message that was sent to the queue:

```
// create the message queue
MessageQueue myQueue = new MessageQueue(".\private$\myQueue");

// receive the message
Message myMessage = myQueue.Receive();

// process the message
```

A blocking method that waits forever is not particularly usable and it is possible to specify that the `Receive` method return after a given period of time:

```
// create the message queue
MessageQueue myQueue = new MessageQueue(".\private$\myQueue");

try
{
  // receive the message
  Message myMessage = myQueue.Receive(new TimeSpan(0,1,0);

  // process the message
}
catch (MessageQueueException ex)
{
  if (ex.MessageQueueErrorCode == MessageQueueErrorCode.IOTimeout)
  {
    // no message in queue
  }
}
```

In this example, we wait 1 minute for a message to appear in the queue. If there is already a message in the queue, or one is added to the queue during that time, the `Receive` method returns immediately and returns the `Message` instance. If there is no message received during the specified `TimeSpan`, the `Receive` method will throw a `MessageQueueException` with the error code specified as `IOTimeout`.

Decide Which Formatter to Use

When sending messages, the message is, by default, formatted using the `XmlMessageFormatter` from the `System.Messaging` namespace. There are three formatters, all implementing the `IMessageFormatter` interface:

- `ActiveXMessageFormatter` – serializes the `Message` to a stream.
- `BinaryMessageFormatter` – serializes the `Message` in binary format.
- `XmlMessageFormatter` – serializes the `Message` as XML.

In order to use a formatter other than `XmlMessageFormatter`, or if you want to do the formatting yourself using the `XmlMessageFormatter`, you must create an instance of the formatter you require and call its `Write` method to format the message correctly. Assuming we have an `Object` called `myObject` that contains an arbitrary class, we can format this using the `BinaryMessageFormatter` as follows:

```
// create the message queue
MessageQueue myQueue = new MessageQueue(".\private$\myQueue");

// create and configure the message
Message myMessage = new Message();
myMessage.Label = "Message test";
myMessage.Priority = MessagePriority.Normal;

// create the correct formatter and format the object
BinaryMessageFormatter myFormatter = new BinaryMessageFormatter();
myFormatter.Write(myMessage, myObject);

// send the message
myQueue.Send(myMessage);
```

As you've formatted the `Message` when sending, you must un-format the `Message` when receiving. You can no longer directly use the `Body` property of the `Message`; instead, must use the `Read` method of the same formatter to retrieve the object correctly. So, to retrieve the `Object` we've just sent using the `BinaryMessageFormatter`, we'd have to do the following:

```
// create the message queue
MessageQueue myQueue = new MessageQueue(".\private$\myQueue");

// receive the message
Message myMessage = myQueue.Receive();

// create the correct formatter and un-format the object
BinaryMessageFormatter myFormatter = new BinaryMessageFormatter();
Object myObject = myFormatter.Read(myMessage);
```

Delete Queued Messages

It is not possible to delete queued messages from a queue. There is no `Delete` method on the `MessageQueue` class. Once a message is retrieved from the queue using the `Receive` method, it is deleted from the queue.

Handle Acknowledgements

Using message queues is, by definition, asynchronous. If you want to have an acknowledgement that your message has been delivered successfully, you need to make use of a second message queue and several properties of the `Message` that you're going to send.

When receiving acknowledgements, you need some way of knowing which message the acknowledgement is for. This is accomplished by using the `Id` of the sent message and returning it as the `CorrelationId` of the acknowledgement `Message`.

Any acknowledgements that you wish to receive must be handled by a second queue. So, the first thing the sender needs to do is create two message queues (we'll assume that they've already been created):

```
// create the two message queues
MessageQueue myQueue = new MessageQueue(".\private$\myQueue");
MessageQueue myAckQueue = new MessageQueue(".\private$\myAckQueue");
```

We then need to create the `Message` we're going to send:

```
// create and configure the message
Message myMessage = new Message();
myMessage.Body = "This is a test message";
myMessage.Label = "Message test";
myMessage.Priority = MessagePriority.Normal;
```

And, before we send the message, we must inform the `Message` that we'd like it to be acknowledged. In this case, we're only after an acknowledgement that it has been processed and removed from the queue. We also specify that the acknowledgement will be sent on the acknowledgement queue, `myAckQueue`:

```
// configure the acknowledgement details
myMessage.AcknowledgeType = AcknowledgeTypes.PositiveReceive;
myMessage.AdministrationQueue = myAckQueue;
```

We can then send the message as we normally would:

```
// send the message and store the Id
myQueue.Send(myMessage);
string myMessageId = myMessage.Id;
```

We can then wait for the acknowledgement queue, `myAckQueue`, to return the acknowledgement for the message we've just sent:

```
// wait for the acknowledgement message
Message myAckMessage = myAckQueue.ReceiveByCorrelationId(myMessageId);
```

By using the `ReceiveByCorrelationId` method, we're waiting for a specific message to be returned (there is also an overloaded version that allows us to specify a `TimeSpan` that we wish to block for), one that corresponds to the message we requested the acknowledgement for. We can check that this is indeed the correct acknowledgment message by checking that the `CorrelationId` property matches the id that we're expecting and that it is the correct type by checking that the `Acknowledgment` property is the correct value from the `AcknowledgeTypes` enumeration.

Acknowledgement of messages is handled automatically by MSMQ — the sender is completely responsible for configuring the second queue for receiving the acknowledgement. The receiver of the message is completely unaware that any acknowledgement of messages has occurred.

Peek at Messages

The `Receive` method that we saw earlier is used to return a `Message` from the queue and can be configured to timeout if no message is received in the given `TimeSpan`. The `Receive` method actually removes the message from the `MessageQueue`.

The `MessageQueue` also provides a `Peek` method that can be used to return the `Message` from the queue but leave the message in the queue. Subsequent calls to the `Peek` method will (unless a higher priority message is received) return the same `Message` object.

As with the `Receive` method, there is both a blocking and non-blocking version of `Peek`. If we don't specify a `TimeSpan` to the `Peek` method, it will block indefinitely:

```
// create the message queue
MessageQueue myQueue = new MessageQueue(".\private$\myQueue");

// peek the message
Message myMessage = myQueue.Peek();

// process the message
```

We can also specify a `TimeSpan` that we're prepared to wait for the `Peek` method to return before a `MessageQueueException` is thrown:

```
// create the message queue
MessageQueue myQueue = new MessageQueue(".\private$\myQueue");


try
{
  // peek the message
  Message myMessage = myQueue.Peek(new TimeSpan(0,1,0);

  // process the message
}
```

```
catch (MessageQueueException ex)
{
  if (ex.MessageQueueErrorCode == MessageQueueErrorCode.IOTimeout)
  {
    // no message in queue
  }
}
```

Receive a Message Asynchronously

The two methods that we've looked at so far for retrieving messages from the message queue, `Receive` and `Peek`, have both been synchronous — that is, we've made the request to the queue and blocked until we've retrieved a `Message` or until we've timed out.

We can also wait for messages to appear message queues by using the `BeginReceive/EndReceive` methods and `ReceiveCompleted` event or the corresponding `Peek` versions, `BeginPeek/EndPeek` and `PeekCompleted`. In either case, the paradigm is exactly the same.

Use BeginReceive/EndReceive and ReceiveCompleted

In order to asynchronously receive a `Message` from the `MessageQueue`, we must first create a `ReceiveCompleted` that will receive the asynchronous call. In here, we'll use the `AsyncResult` to call the corresponding `EndReceive` method to return the message that we can then process:

```
private static void myReceivedCompleted(object e,
  ReceiveCompletedEventArgs args)
{
  // connect to the queue (the first parameter)
  MessageQueue myQueue = (MessageQueue)e;

  // get the message (returns immed
  Message myMessage = myQueue.EndReceive(args.AsyncResult);

  // process the message
}
```

We can then attach this event handler to our message queue and call the BeginReceive method to start the event handling process:

```
// create the message queue
MessageQueue myQueue = new MessageQueue(".\private$\myQueue");

// attach the message handler
myQueue.ReceiveCompleted += new ReceiveCompletedEventHandler(
  myReceiveCompleted);

// begin the event handling
myQueue.BeginReceive();

// do some other work
```

After calling the `BeginReceive` method, we're free to continue execution of our application and the `myReceivedCompleted` event will be fired every time a message is added to the queue.

Message Security

MSMQ provides two levels of security for messages:

- Authentication – signing a message proves that the sender of the message is who they say they are.
- Encryption – encrypting a message ensures that the communication between the sender and the received is secure.

Signing a Message

Message authentication and, by definition, signing a message, is only possible if you're using a computer that is connected to an Active Directory domain. Active Directory is used to handle the certificates necessary to authenticate users.

Signing a message can be performed using two different types of certificate:

- Internal certificates are created automatically by MSMQ and are used to authenticate the Windows user. The certificate is based upon the Windows security identifier (SID) of the user. You can only use internal certificates if you're connected to an Active Directory domain as the domain manages all of the users and SID for users is consistent across all machines in the domain.
- External certificates are proved by an external certificate authority. You must use external certificates if you're sending messages to non-Windows machines or if you're not connected to an Active Directory domain.

Message authentication is enabled by checking the Authenticated option from the General properties for the message queue, as shown in Figure 6-3.

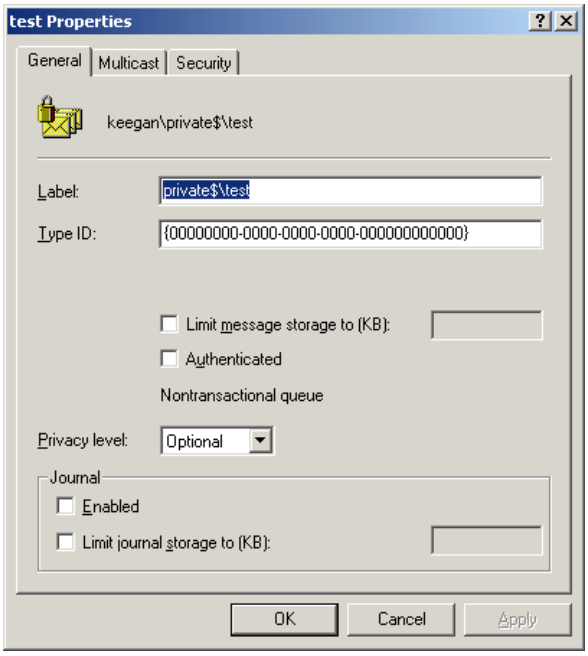

Figure 6-3 – Configuring security for a message queue.

When sending messages to a queue that requires authentication, any message that cannot be authenticated, or fails authentication, is immediately discarded by the queue. It is never placed in the queue.

In order for a message to be authenticated, the certificate needs to be attached to the message. For an internal certificate, this actually occurs automatically and the certificate of the user executing the current thread is automatically attached to a message as the `SenderId` property. You can override this behavior by setting the `AttachSenderId` property of the `Message` to false. If `AttachSenderId` is set to `false` the message will still be authenticated correctly but the sender will be anonymous.

The only difference between using internal and external certificates to sign a message is the process of attaching the certificate to the message. When using an internal certificate, this occurs automatically using the `SenderId` property, whereas with an external certificate you need to attach the certificate using the `SenderCertificate` property.

When using an internal certificate, all that is required is to set two properties of the `Message` for the message to be signed:

```
// create the message queue
MessageQueue myQueue = new MessageQueue(".\private$\myQueue");

// create and configure (not shown) the message Message
myMessage = new Message();

// sign the message
myMessage.UseAuthentication = true;
```

```
// send the message
myQueue.Send(myMessage);
```

Because the `SenderId` is automatically entered (as `AttachSenderId` has a default value of true), all you need to do is set the `UseAuthentication` property to `true` and the message will be authenticated before it is added to the queue.

External certificates are a little trickier, as you need to create an instance of the certificate that you're going to use and attach that to the `SenderCertificate` property of the message before it is attached to the queue:

```
// create an instance of the correct certificate
X509Certifcate2 myCertificate = new X509Certifcate2();

// create the message queue
MessageQueue myQueue = new MessageQueue(".\private$\myQueue");

// create and configure (not shown) the message
Message myMessage = new Message();

// sign the message and attach the certificate
myMessage.UseAuthentication = true;
myMessage.SenderCertificate = myCertificate.GetRawCertData();

// send the message
myQueue.Send(myMessage);
```

If you're not attached to an Active Directory domain, in workgroup mode, you must set the `AttachSenderId` property to `false` — MSMQ cannot associated a SID with a certificate when running in workgroup mode.

Verify a Message

When running attached to an Active Directory domain, it isn't necessary to check the authentication of a message — the authentication of the message was checked as it was added to the queue. When running in workgroup mode, however, you do need to check that the sender of the message, using the `SenderCertificate` property, is valid. You can retrieve the certificate from a message as follows:

```
// create the message queue
MessageQueue myQueue = new MessageQueue(".\private$\myQueue");

// receive the message
Message myMessage = myQueue.Receive();

// create the certificate
X509Certifcate2 myCertificate = new ~CCC
  X509Certifcate2(myMessage.SenderCertificate);
```

You can then inspect the properties of the `X509Certificate2` instance to determine if the certificate is to be trusted.

Encrypt a Message

If attached to an Active Directory domain, encrypting a message is performed seamlessly by MSMQ with very little external work required. If you look at Figure 6-3, you'll see that there is a Privacy level drop down box — this controls the encryption required for the message queue and can take one of three values:

- None – the queue accepts only non-encrypted messages.
- Body – the queue accepts only encrypted messages.
- Optional – the queue will accept both encrypted and non-encrypted messages.

To encrypt a message, all that is required is to set the `UseEncryption` property for the message:

```
// create the message queue
MessageQueue myQueue = new MessageQueue(".\private$\myQueue");

// create and configure (not shown) the message Message
myMessage = new Message();

// encrypt the message
myMessage.UseEncryption = true;

// send the message
myQueue.Send(myMessage);
```

The message will be automatically encrypted before it is sent to the message queue.

If you're running in workgroup mode, there is no automatic encryption of messages. If you want to use encryption, you must manually encrypt the body of the message before you send the message to the queue.

Decrypt a Message

Decryption of the message is handled automatically when connected to an Active Directory domain and all messages retrieved from the queue are un-encrypted.

If you're running in workgroup mode, and have manually encrypted the message, you'll need to manually de-encrypt the body of the message after it has been retrieved from the message queue.