Information Technology
Important Questions 3

(C# and .Net Framework)

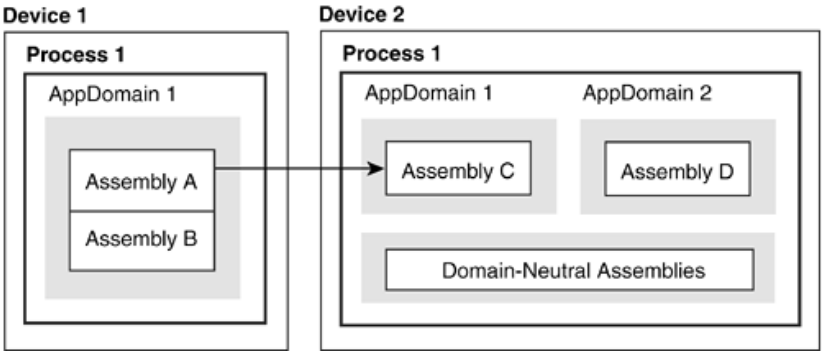
1) Application Domains
- Remoting
- Leasing and Sponsorship
- .NET Coding Design Guidelines
- Assemblies
- Security
- Application Development
- Web Services
- Building an XML Web Service
- Web Service Client
- WSDL and SOAP
- Web Service with Complex Data Types
- Web Service Performance.

(1) What are application Domains?

The AppDomain is one of the key architectural features supporting the managed environment.

Most operating systems see the world in terms of processes that provide the resources, such as memory and tables required by applications. Because a .NET application cannot run directly in the unmanaged process, .NET partitions a process into one or more logical areas in which assemblies execute. These logical areas are AppDomains.

As shown in Figure 14-1, a process may contain more than one AppDomain and an AppDomain may contain one or more assemblies. A default AppDomain is created when the Common Language Runtime (CLR) initializes, and additional ones are created by the CLR as needed. An application may also instruct the CLR to create a new AppDomain.



2) **What are the advantages of Application Domains?**

Advantages of AppDomains

Aside from the need for a managed environment, the use of AppDomains provides several advantages over the traditional process-based architecture:

- Code Isolation. AppDomains institute a level of fault isolation that prevents a code failure in one AppDomain from causing another AppDomain to crash. .NET achieves this code separation in two ways: by preventing an AppDomain from directly referencing objects in another AppDomain, and by having each AppDomain load and maintain its own copy of key assemblies that allow it to run independently. As a by-product of this, an AppDomain can be selectively debugged and unloaded without directly affecting other AppDomains in the process.
- Performance. Implementing an application to run in multiple AppDomains can produce better performance than a comparable design that relies on multiple processes. This efficiency derives from several factors: A physical process requires more memory and resources than AppDomains, which share the resources of a single process; creating and disposing of processes is much more time consuming than comparable operations on AppDomains; and making a call between processes is slower and requires more overhead than making a call between AppDomains residing in the same process.
- Security. By its very nature, an AppDomain presents a security boundary between its contained resources and assemblies attempting to access them. To cross this boundary, an outside assembly must rely on remoting , which requires cooperation between the AppDomains. In addition, an AppDomain has its own security policy that it can impose upon assemblies to restrict their permissible operations. This "sandbox" security model allows AppDomains to ensure that assemblies are well behaved. In Chapter 15, "Code Refinement, Security, and Deployment," we'll look at examples of using AppDomains to enforce code security.

3) **What is remoting?**

Remoting

At its core , remoting is as a way to permit applications in separate AppDomains to communicate and exchange data. This is usually characterized as a client-server relationship in which the client accesses resources or objects on a remote server that agrees to provide access. The way in which this agreement between client and server is implemented is what remoting is all about. The physical proximity of the AppDomains does not matter: They may be in the same process, in different processes, or on different machines on different continents.

4) **What are the steps required to enable a client to access an object on a remote server?**

Consider the steps required to enable a client to access an object on a remote server:

- Create a TCP or HTTP connection between the client and server.

- Select how the messages sent between server and client are formatted.
- Register the type that is to be accessed remotely.
- Create the remote object and activate it from the server or the client.

.NET takes care of all the details. You don't have to understand the underlying details of TCP, HTTP, or ports—just specify that you want a connection and what port to use. If HTTP is selected, communications use a Simple Object Access Protocol (SOAP) format; for TCP, binary is used. The registration process occurs on both the server and client. The client selects a registration method and passes it a couple of parameters that specify the address of the server and the type (class) to be accessed. The server registers the types and ports that it wants to make available to clients, and how it will make them available. For example, it may implement the object as a singleton that is created once and handles calls from all clients ; or it may choose to create a new object to handle each call.

5) **Explain in detail remoting Architecture.**

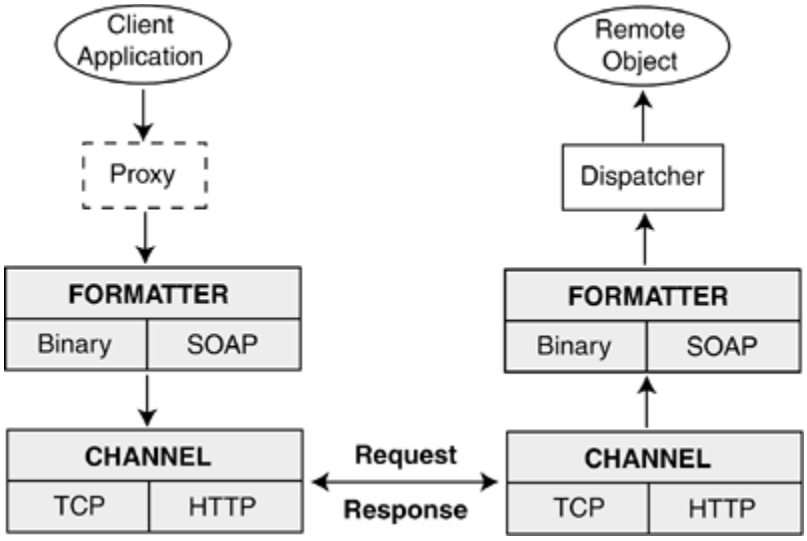
Remoting Architecture

When a client attempts to invoke a method on a remote object, its call passes through several layers on the client side. The first of these is a proxy —an abstract class that has the same interface as the remote object it represents. It verifies that the number and type of arguments in the call are correct, packages the request into a message, and passes it to the client channel . The channel is responsible for transporting the request to the remote object. At a minimum, the channel consists of a formatter sink that serializes the request into a stream and a client transport sink that actually transmits the request to a port on the server. The sinks within a channel are referred to as a sink chain . Aside from the two standard sinks, the channel may also contain custom sinks that operate on the request stream.

On the server side, the process is reversed , as the server transport sink receives the message and sends it up the chain. After the formatter rebuilds the request from the stream, .NET creates the object on the server and executes the requested method.

Figure 14-4 illustrates the client-server roles in a remoting architecture. Let's examine its three key components : proxies, formatter classes, and channel classes.

Figure 14-4. High-level view of .NET remoting architecture



6) **What are proxies?**

Proxies

When a client attempts to communicate with a remote object, its reference to the object is actually handled by an intermediary known as a proxy . For .NET remoting, there are two types of proxies: a transparent proxy that the client communicates with directly, and a real proxy that takes the client request and forwards it to the remote object.

The transparent proxy is created by the CLR to present an interface to the client that is identical to the remote class. This enables the CLR to verify that all client calls match the signature of the target method—that is, the type and number of parameters match. Although the CLR takes care of constructing the transparent proxy, the developer is responsible for ensuring that the CLR has the metadata that defines the remote class available at compile time and runtime. The easiest way is to provide the client with a copy of the server assembly that contains the class. But, as we discuss later, there are better alternatives.

After the transparent proxy verifies the call, it packages the request into a message object—a class that implements the `IMessage` interface. The message object is passed as a parameter to the real proxy's `Invoke` method, which passes it into a channel. There, a formatter object serializes the message and passes it to a channel object that physically sends the message to the remote object.

7) **Explain two types of remoting.**

Types of Remoting

Recall that the parameters in a C# method may be passed by value or by reference . Remoting uses the same concept to permit a client to access objects—although the terminology is a bit different. When a client gets an actual copy of the object, it is referred to as marshaling by value (MBV) ; when the client gets only a reference to the remote object, it is referred to as marshaling by reference (MBR) . The term marshaling simply refers to the transfer of the object or request between the client and server.

Marshaling by Value

When an object is marshaled by value, the client receives a copy of the object in its own application domain. It can then work with the object locally and has no need for a proxy. This approach is much less popular than marshaling by reference where all calls are made on a remote object. However, for objects that are designed to run on a client as easily as on a server, and are called frequently, this can reduce the overhead of calls to the server.

As an example, consider an object that calculates body mass index (BMI). Instead of having the server implement the class and return BMI values, it can be designed to return the BMI object itself. The client can then use the object locally and avoid further calls to the server. Let's see how to implement this.

For an object to be marshaled by value, it must be serializable. This means that the class must either implement the `ISerializable` interface or—the easier approach—have the [`Serializable]` attribute. Here is the code for the class on the server:

```
[Serializable]
public class BMICalculator
{
   // Calculate body mass index
   public decimal inches;
   public decimal pounds;
   public decimal GetBMI()
   {
      return ((pounds*703* 10/(inches*inches))/10);
   }
```

```
}
```

The `HealthTools` class that is marshaled by reference returns an instance of `BMICalculator` :

```
public class HealthTools: MarshalByRefObject
{
   // Return objects to calculate BMI
   public BMICalculator GetBMIObj(){
      return new BMICalculator();
   }
```

The client creates an instance of `HealthTools` and calls the `GetBMIObj` method to return the calculator object:

```
HealthMonitor remoteObj = new HealthMonitor();
BMICalculator calc= remoteObj.GetBMIObj();
calc.pounds= 168M;
calc.inches= 73M;
Console.WriteLine(calc.GetBMI());
```

It is important to understand that this example uses both marshaling by value and marshaling by reference: an MBR type (`HealthTools`) implements a method (`GetBMIObj`) that returns an MBV type (`BMICalculator`). You should recognize this as a form of the factory design pattern discussed in Chapter 4, "Working with Objects in C#."

Marshaling by Reference

Marshaling by reference (MBR) occurs when a client makes a call on an object running on a remote server. The call is marshaled to the server by the proxy, and the results of the call are then marshaled back to the client.

Objects accessed using MBR must inherit from the `MarshalbyRefObject` class. Its most important members , `InitializeLiIfetimeServices` and `GetLifeTimeServices` , create and

retrieve objects that are used to control how long a remoting object is kept alive on the server. Managing the lifetime of an object is a key feature of remoting and is discussed later in this section.

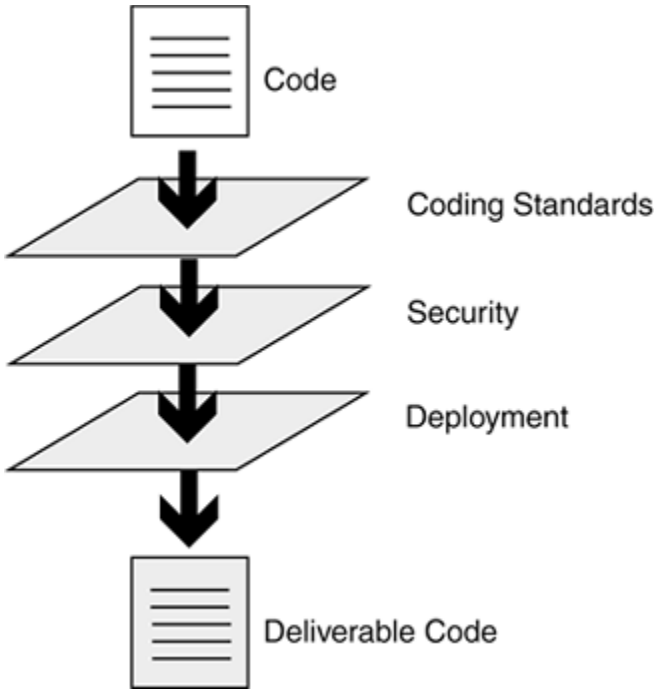
`MarshalByRefObject s` come in two flavors: client-activated objects (CAO) and server-activated objects (SAO)—also commonly referred to as well-known objects (WKO). Server-activated objects are further separated into single call and singleton types. A server may implement both client-activated and server-activated objects. It's up to the client to choose which one to use. If the client selects SAO, the server makes the determination as to whether to use server-activated single call or server-activated singleton objects.

The choice of activation mode profoundly affects the overall design, performance, and scalability of a remoting application. It determines when objects are created, how many objects are created, how their lifecycle is managed, and whether objects maintain state information. Let's look at the details.

8) **What are the issues and steps involved in producing deliverable .NET software?**

This chapter looks at the issues and steps involved in producing a deliverable .NET software product. It breaks the process down into the three categories shown in Figure 15-1: code refinement, which looks at how code is tested against best practice rules; code security, which ensures that code is accessed only by other code that has permission to do so; and code deployment, which looks at how an application or component is packaged and made available for deployment.

Figure 15-1. Deliverable software should meet coding standards, be secure, and be easily deployed

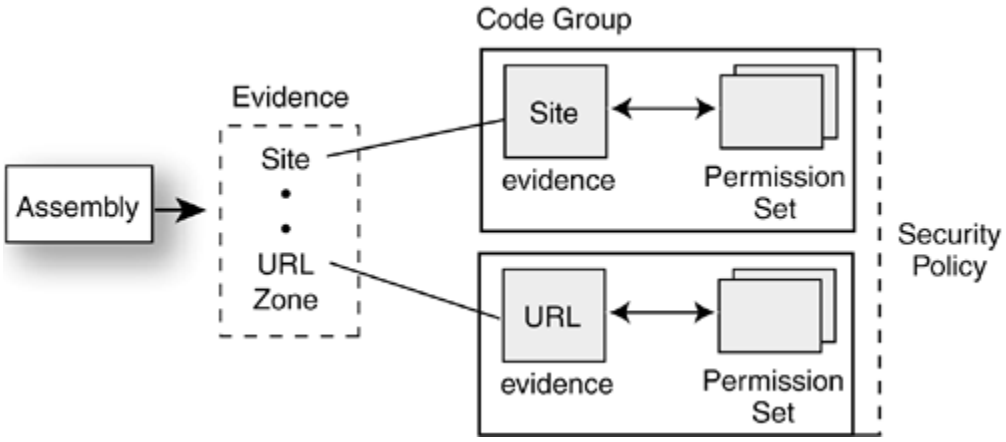


9) **Explain .NET Security features.**

Security

The centerpiece of .NET security is the Code Access Security model. As the name implies, it is based on code access—not user access. Conceptually, the model is quite simple. Before an assembly or component within an assembly may access system resources (files, the registry, event log, and others), the CLR checks to ensure that it has permission to do so. It does this by collecting evidence about the assembly—where is it located and its content. Based on this evidence, it grants the assembly certain permissions to access resources and perform operations. Figure 15-4 illustrates the key elements in this process and introduces the terms that you must know in order to administer security.

Figure 15-4. An assembly is matched with code groups whose evidence it satisfies



When an assembly is loaded, the CLR gathers its evidence and attempts to match it with code groups whose evidence it satisfies. A code group is a binding between a set of permissions and a single type of evidence. For example, a code group may be defined so that only assemblies from a particular application directory are allowed to have Web access. If an assembly's site evidence indicates it is from that directory, it is part of the code group and has Web access. An assembly can be a member of multiple code groups, and, consequently, can have multiple permissions.

.NET provides predefined evidence, permissions, code groups, and security policies—a collection of code groups. Although code can be used to hook into and modify some aspects of security, an administrator performs the bulk of security configuration and management using .NET tools. In most cases, the predefined elements are all that an administrator needs. However, the security model is flexible, and permits an administrator to create security policies from custom evidence, permissions, and code groups.

This abstract representation of .NET security shown in Figure 15-4 is implemented in concrete types: The `Evidence` class is a collection that holds evidence objects; permission classes grant access to a resource or the right to perform some action; and a `PermissionSet` is a collection class that groups permissions and contains methods to manipulate them. The following sections take a close look at evidence, permissions, and how they are related . You'll then see how to implement a security policy, both as an administrator and by accessing the permission classes through code.

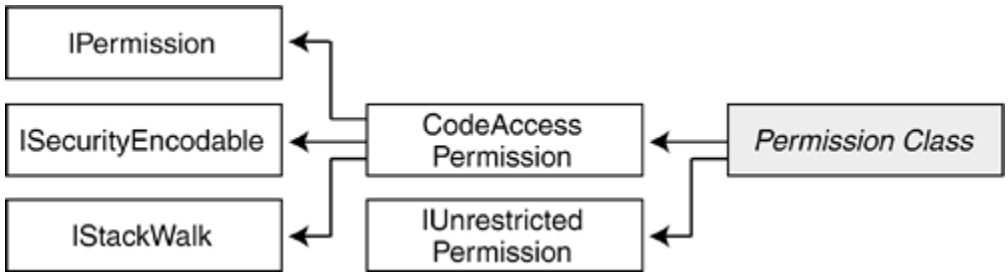
10) **Explain Built-in Security permissions available in .NET.**

The individual permissions shown in Figure 15-5 are implemented in .NET as built-in permission classes. In addition to being accessed by the security configuration tools, they can also be accessed by code to implement a finer-grained security than can be configured using administrative tools.

Table 15-1 summarizes the more important built-in permission classes.

All permission classes inherit and implement the interfaces shown in Figure 15-7. Of these, `IPermission` and `IStackWalk` are the most useful. `IPermission` defines a `Demand` method that triggers the stack walk mentioned earlier; `IStackWalk` contains methods that permit a program to modify how the stack walk is performed. This proves to be a handy way to ensure that a called component does not perform an action outside of those that are requested . We'll look at these interfaces in more detail in the discussion of programmatic security.

Figure 15-7. Interfaces inherited by permission classes



Identity Permissions

Recall that when the CLR loads an assembly, it matches the assembly's evidence against that required by code groups and grants permissions from the code groups whose criteria it meets. These code group derived permissions are either custom permissions or built-in permissions as described in Table 15-1.

The CLR also grants another set of permissions that correspond directly to the identity evidence provided by the assembly. For example, there are `ZoneIdentityPermission` and `StrongNamedIdentityPermission` classes that demand an assembly originate from a specific zone or have a specific strong name identity. Table 15-2 lists the origin-based identity classes.

Table 15-2. Identity Permission Classes

Class	Identity Represented
`PublisherIdentityPermission`	The digital signature of the assembly's publisher.
`SiteIdentityPermission`	Web site where the code comes from.
`StrongNamedIdentityPermission`	Strong name of the assembly.
`URLIdentityPermission`	URL where the code comes from. This includes the protocols HTTP, HTTPS, and FTP.
`ZoneIdentityPermission`	Zone where the code originates: `Internet` , `Intranet` , `MyComputer` , `NoZone` , `TRusted` , `Untrusted` .

Unlike the built-in permissions described earlier, these classes cannot be administered using configuration tools. Instead, a program creates an instance of an identity permission class and uses its methods to demand that an assembly provide a specified identity to perform some action. This programmatic use of permission classes is referred to as imperative security .

Permission Attributes

All security permission classes have a corresponding attribute class that can be applied as an attribute to an assembly, class, and method to specify security equivalent to that provided by the permission class. This is referred to as declarative security , and serves two useful purposes: When applied at the assembly level, a permission attribute informs the runtime which permissions the assembly requires, and enables the runtime to throw an exception if it cannot grant these permissions; when applied to classes and methods within the code, the attribute specifies which permissions any calling assemblies must have to use this assembly. Examples using declarative security are provided later in the chapter.

Evidence

To qualify as a member of a code group and assume its privileges, an assembly must provide evidence that matches the evidence membership requirements of the code group. This evidence is based on either the assembly's origin or its signature. The origin identification includes `Site` , `Url` , and `Zone` evidence; the signature refers to an assembly's strong name, its digitally signed certificate (such as X.509), or a hash of the assembly's content. The Common Language Runtime provides seven predefined types of evidence. They are referred to by names used in the security administrative tools:

- `Strong Name` . An assembly with a `Strong Name` has a public key that can be used to identify the assembly. A class or method can be configured to accept calls only from an assembly having a specified public key value. The most common use for this is to identify third-party components that share the same public key. A `Strong Name` has two other properties, `Version` and `Name` , that also can be required as evidence by a host assembly.
- `Publisher` . This evidence indicates that an assembly has been digitally signed with a certificate such as X.509. Certificates are provided by a trusted certificate authority and are most commonly used for secure Internet transactions. When a signed assembly is loaded, the CLR recognizes the certificate and adds a Publisher object to the assembly.
- `Hash` . By applying a computational algorithm to an assembly, a unique identifier known as a hash is created. This hash evidence is automatically added to each assembly and serves to identify particular builds of the assembly. Any change in the compiled code yields a different hash value—even if the version is unchanged.
- `Application Directory` . This evidence is used to grant a permission set to all assemblies that are located in a specified directory or in a subdirectory of the running application.

- `Site` . `Site` evidence is the top-level portion of a URL that excludes the format and any subdirectory identifiers. For example, www.corecsharp.net is extracted as site evidence from http://www.corecsharp.net/code.
- `URL` . This evidence consists of the entire URL identifying where an assembly comes from. In the preceding example, http://www.corecsharp.net/code is provided as `URL` evidence.
- `Zone` . The `System.Security.SecurityZone` enumeration defines five security zones: `MyComputer` , `Intranet` , `Internet` , `trusted` , and `Untrusted` . An assembly's `zone` evidence is the zone from which it comes.

  - `MyComputer` . Code coming from the local machine.

  - `Intranet` . Code coming from computers on the same local area network.

  - `Internet` . Code coming from the Internet that is identified by an HTTP or IP address. If the local machine is identified as `http://localhost/` , it is part of the `Internet` zone.

  - `TRusted` . Identifies Internet sites that are trusted. These sites are specified using Microsoft Internet Explorer (IE).

  - `UnTRusted` . Sites specified in IE as being malicious or untrustworthy.

11) **What are Security Policies? How .NET applies security policies?**

Security Policies

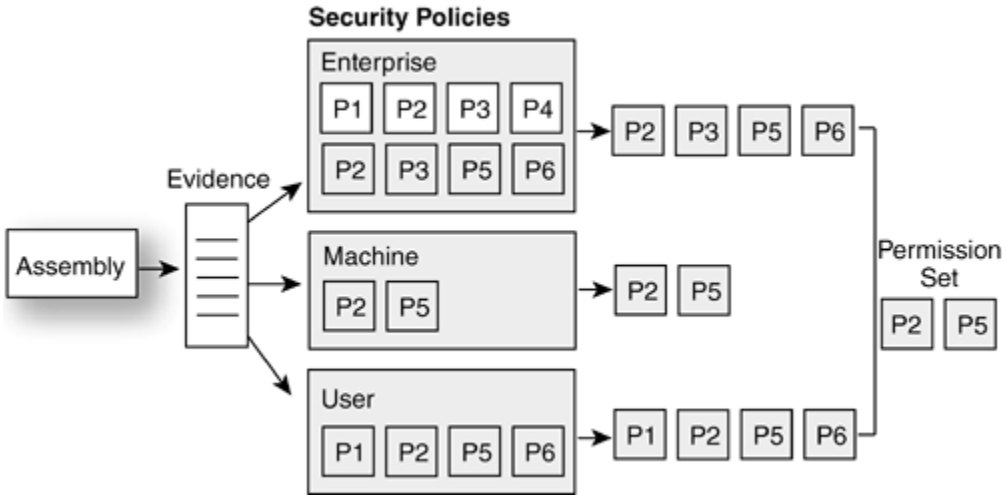
A .NET security policy defines how assembly evidence is evaluated to determine the permissions that are granted to the assembly. .NET recognizes four policy levels: `Enterprise` , `Machine` , `User` , and `Application Domain` . The policy-level names describe their recommended usage. `Enterprise` is intended to define security policy across all machines in the enterprise; `Machine` defines security for a single machine; `User` defines security policy for individual users; and `Application Domain` security is applied to code running in a specific AppDomain. `Enterprise` , `Machine` , and `User` policies are configured by an administrator. `AppDomain` policy, which is implemented only programmatically and used for special cases, is not discussed.

Despite their names, policies can be configured in any way an administrator chooses. The `User` policy could be set up to define enterprise security and the `Machine` policy to define user security. However, an administrator should take advantage of the names and use them to apply security to their intended target. As you will see in the discussion of the .NET Framework Configuration Tool (see "The .NET Framework Configuration Tool" on page 704), the security policy is granular enough to allow custom security policies on individual machines and users.

How .NET Applies Security Policies

Each security policy level is made up of one or more code sets. Each code set, in turn , contains a set of permissions that are mapped to a specific evidence type. Figure 15-8 illustrates how code sets and policy levels are combined to yield a permission set for an assembly.

Figure 15-8. A permission set is created from the intersection of policy level permissions



The .NET security manager is responsible for evaluating evidence and policy to determine the permissions granted. It begins at the enterprise level and determines the permissions in it that can be granted to the assembly. In this example, enterprise contains three code groups—two of which the assembly's evidence satisfies. The logical union of these permissions produces the permission set at this level. The other two policy levels are evaluated in the same way, yielding their associated permission set. The logical intersection of the three permission sets produces the permission set that is assigned to the assembly. In this case, the final set consists of permissions 2 and 5—the only permissions present on each level.

12) **Explain briefly XML Web Sevices**.

XML Web Services provide a relatively simple technique for accessing a method on an object that is running on a local or remote computer. Many embrace this lightweight approach to making Remote Procedure Calls (RPC) as technology that will spell the end to much heavier and complex solutions for distributed communications such as DCOM and CORBA. At the root of its appeal is the fact that Web Services are based on standardized technologies such as HTTP and XML that are designed to promote seamless interoperability among different operating systems.

13) **Define HTTP and SOAP.**

HTTP and SOAP—a protocol that codifies how XML is used to package the request and response data that comprise a Web Service operation—are the two cornerstones of Web Services. Other protocols such as GET and POST are available, but the Simple Object Access Protocol (SOAP) has fewer limitations and is used in the majority of real-world applications. This chapter takes a look at the SOAP format as well as several issues related to SOAP, including the handling of complex data types, exception handling, and security.

14) Explain the Architecture of Web Services and its constituent parts.

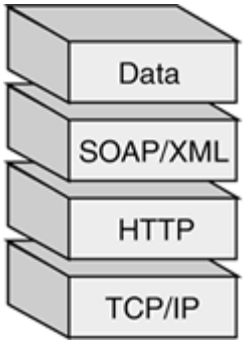
The Web Service architecture is a service-oriented architecture that enables applications to be distributed across a network or the Internet to clients using any language or operating system. As shown in Figure 18-1, Web Service communications are implemented using existing technology and standards that require no proprietary vendor support. This technology either formed the basis of the Internet or evolved from it. HTTP and XML have been discussed in earlier chapters, but let's take a brief look at them—from a Web Services perspective—along with TCP/IP and SOAP:

- TCP/IP (Transmission Control Protocol/Internet Protocol). A communications protocol suite that forms the basis of the Internet. It's an open system that governs the flow of data between computers by breaking data into chunks that are easily routed over a network. A Web Service user or developer rarely has direct contact with this layer.
- HTTP (Hypertext Transfer Protocol). Technically, this text-based protocol is a Remote Procedure Call (RPC) protocol that supports request/response communications. The .NET Framework as well as most production Web Services use it because it has wide support and generally allows information to sail unimpeded through firewalls. Both the HTTP `POST` and HTTP `GET` methods are supported in .NET as a way to call a Web Service. Most applications use `POST` , because it packages the request inside the request body (in a SOAP envelope), rather than as part of a less secure query string.
- XML (Extended Markup Language). We have seen in earlier chapters how data can be serialized into XML format for storage or transmission. The fact that it is text based makes it easy to work with. Just about every programming environment supports tools for encoding and decoding XML formatted data. Its inherent flexibility, extensibility, and validity checking make it attractive to Web Services that must deal with simple data types such as strings, as well as more complex data structures. There are currently two XML-based protocols used for delivering Web Services: XML-RPC and SOAP. Because .NET supports SOAP, this chapter focuses on it. After you understand SOAP, you should have no difficulty with XML-RPC if you encounter it.
- SOAP (Simple Object Access Protocol). SOAP is defined as "a lightweight protocol for exchange of information in a decentralized, distributed environment." [1] It is not designed specifically for Web Services, nor restricted to HTTP; but its RPC specifications define a model for invoking methods on a specified target machine, passing parameters, handling faults, and receiving a method response. We will look at SOAP in detail later in the chapter. For now, keep in mind that the details of XML and SOAP are typically handled transparently by .NET. The provider code only needs to focus on implementing the

method that returns the desired data; the requestor code simply places a method call and processes the returned data.

[1] Simple Object Access Protocol (SOAP) 1.1—W3C Note, May 8, 2000.

Figure 18-1. Web Service transport



15) **How web service is discovered and used by the consumer?**

Discovering and Using a Web Service

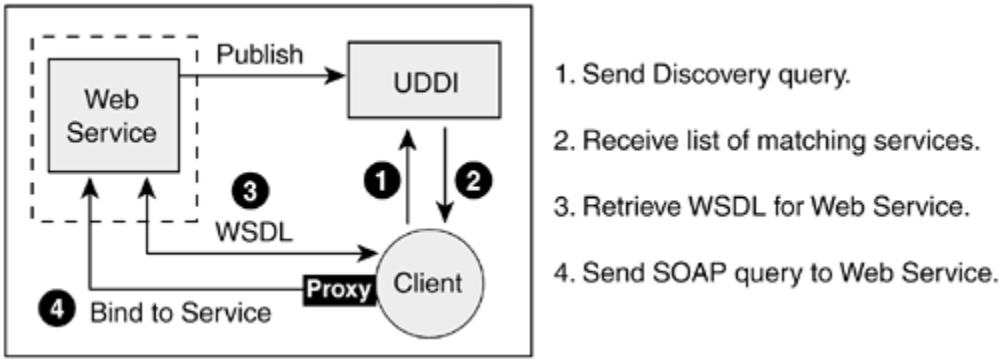
To use a Web Service, a client must have a description of how to access the service. This information is provided by a Web Services Description Language (WSDL) document that provides the name of the service, the signature of the method(s) that can be called, the address of the service (usually a URL), and binding information that describes how the transport operation will occur. In practical terms, the WSDL information contains the methods that actually call a Web Service. We implement these methods in our client code (as source or a DLL reference) and use them as a proxy to access the service. Section 18.2, "Building an XML Web Service," provides a concrete example of using .NET to retrieve WSDL information and incorporate it into a client application.

Introduction to UDDI

Web sites are identified by a domain name and IP address that are maintained by a distributed network service known as the Domain Name System (DNS). Servers in this network are responsible for controlling e-mail delivery and translating domain names into IP addresses. The combination of DNS and Web search engines enables users to quickly locate Web content. However, neither DNS servers nor search engines provide a formal way of identifying Web Services.

To organize Web Services into a publicly searchable directory, a public consortium (www.uddi.com) comprising hundreds of companies has defined a standard known as Universal Description Discovery and Integration (UDDI). This standard defines a SOAP-based interface that can be used to publish a service or inquire about services in a UDDI-compliant registry. The registry has a business-to-business flavor about it—containing information about a company, its services, and interface specifications for any Web Services it offers. Importantly, there is no single UDDI registry. IBM, SAP, and Microsoft maintain the most prominent registries. Users may query each separately by entering a business name or service as a search term . Figure 18-2 provides an overview of the inquiry process.

Figure 18-2. Discovering and accessing a Web Service



The dialog between the client and UDDI registry server is conducted using SOAP messages. Overall, UDDI defines approximately 40 SOAP messages for inquiry and publishing.

UDDI Discovery Example

To demonstrate how to use UDDI, w'e'll look at the SOAP messages sent between client and server as we seek to discover a Web Service that can provide a stock quote. A Web-based UDDI browser (described shortly) sends and receives the messages.

Step 1: Send Discovery Request

For our example, we'll search the UDDI registry provided by Microsoft:

```
http://uddi.microsoft.com/inquire
```

An inquiry may request business information or `tModel` information. The former contains information about a business, including contact information and a description of services. The `tModel` structure is much simpler: It consists of a unique key, optional description, and a URL or pointer to a Web page where information for using the service is described. The following

message requests a list of `tModel` keys for companies whose service includes providing a stock quote.

```
<Envelope>
   <Body>
      <find_tModel generic="1.0" maxRows="100">
         <findQualifiers/>
         <name>stock quote</name>
      </find_tModel>
   </Body>
</Envelope>
```

Step 2: Registry Service Responds with List of Services Matching Request

A list of `tModel` keys is returned. These keys are used for the subsequent query:

```
<soap:Envelope>
  <soap:Body>
    <tModelList generic="1.0" operator="Microsoft Corporation" >
      <tModelInfos>
        <tModelInfo
          tModelKey="uuid:7aa6f610-5e3c-11d7-bece-000629dc0a53">
          <name>Stock Quote</name>
        </tModelInfo>
        <tModelInfo
          tModelKey="uuid:265973ab-31cb-4890-83e0-34d9c1b385e5">
          <name>Stock Quotes and Information</name>
        </tModelInfo>
      </tModelInfos>
```

```
    </tModelList>
  </soap:Body>
</soap:Envelope>
```

Step 3: Retrieve Overview Document Containing WSDL

Send a request for `tModel` details for the service with the specified `tModelKey` :

```
<Envelope>
   <Body>
      <get_tModelDetail generic="1.0">
         <tModelKey>uuid:7aa6f610-5e3c-11d7-bece-000629dc0a53
         </tModelKey>
      </get_tModelDetail>
   </Body>
</Envelope>
```

The response message includes the `OverviewURL` element that points to a WSDL document. This document contains the information needed to create an application to access the service.

```
<overviewDoc>
   <description xml:lang="en">
      Get Stock quote for a company symbol
   </description>
   <overviewURL>
      http://www.webservicex.net/stockquote.asmx?WSDL
   </overviewURL>
</overviewDoc>
```

You can display the WSDL by pointing your browser to this URL. To invoke the Web Service, remove the query string (`?WSDL`) from the URL, and navigate to it with your browser.

16) Explain the steps required for building the web services by hand. Give a suitable example with code and diagrams.

Building an XML Web Service

In this section, we demonstrate how to build a Web Service by hand and then access it using a browser. We also show how to create the same Web Service using Visual Studio.NET. Although IIS is used, the examples run on any Web server.

Creating a Web Service by Hand

The first step is to select or create a virtual directory under IIS that will hold the Web Service source code file(s). Any physical directory can be mapped to an IIS virtual directory. You can use the Internet Service Manager or simply right-click the directory and select Sharing-Web Sharing. Then, assign it an alias that will be used in its URL. In our example, we will place the service in the `\ws` subdirectory.

After the directory has been set up, the next step is to use a text editor to create a file in this directory with the `.asmx` extension to contain the Web Service code. Listing 18-1 contains the code for our simple Web Service. This service exposes a method `GetdayBorn` that accepts three integer parameters that represent the month, day, and year for birth date. A string value containing the day of the week (Monday, Tuesday, and so on) for this date is returned. The service performs rudimentary error checking and returns an error message if the date is invalid.

Listing 18-1. Web Service to Return a Date's Day of Week— `BirthDayWS.asmx`

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

using System;

namespace BirthDayWS

{

   public class BirthDay

   {

      [System.Web.Services.WebMethod

      (Description="Return day of week for a date")]

      public string GetDayBorn(int mo, int day, int yr)

      {
```

```
         bool err = false;
         string dob;
         if (mo <1 || mo >12) err=true;
         if (day < 1 || day > 31) err=true;
         if (err)
            dob = "Invalid Date";
         } else {
            DateTime dt = new DateTime(yr,mo,day);
            dob = dt.ToString("dddd"); // Get day
         }
         return(dob);
      }
   }
}
```

The code consists of a single class and a method that is invoked by a client to return the day-of-week value. In addition to the C# code that implements the logic of the service, two other elements are required: a `WebService` directive and a `WebMethod` attribute.

WebService Directive

The `WebService` directive identifies the file as defining a Web Service:

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

The directive specifies the class implementing the XML Web Service and the programming language used in the implementation. In this example, the directive and the code for the class are present in the `BirthDayWS.asmx` file. Note, however, that the class can be in a separate assembly. In that case, the separate assembly is placed in a `\bin` directory below the Web application where the Web Service resides. If the class were in `bdAssembly.dll` , the `WebService` directive would look like this:

```
<%@ WebService Language="C#"
      Class="BirthDayWS.BirthDay, bdAssembly" %>
```

This statement would be the only line needed in the `.asmx` file.

WebMethod Attribute

The `WebMethod` attribute identifies a method as being accessible to clients making HTTP requests —that is, as an XML Web Service. Although not required, it is a good practice to include the `Description` property to describe the purpose of the method:

```
[System.Web.Services.WebMethod
       (Description="Return day of week for a date")]
```

The description is added to the WSDL for the service and—as we will see next—is displayed when a Web Service is accessed via a browser. The `WebMethod` attribute has other optional parameters, which are described later in this chapter.

Testing the Web Service

A quick way to test the newly developed Web Service is to point a browser to its location. For this example, we enter the address:

```
http://localhost/ws/BirthDayWS.asmx
```

This brings up a Web page that lists all the services (methods) available through this `.asmx` file as well as a Service Description link that displays WSDL information. For our example, there is only one method, `GetdayBorn` . Clicking it yields the page shown in Figure 18-3. This page contains the name of the class implementing the Web Service, the name of the method to be invoked, a description of the method, and text boxes for entering values to be passed to the Web Service method.

Figure 18-3. Using a browser to access the `BirthDay` Web Service

17) Explain the steps in building an XML Web Service client.

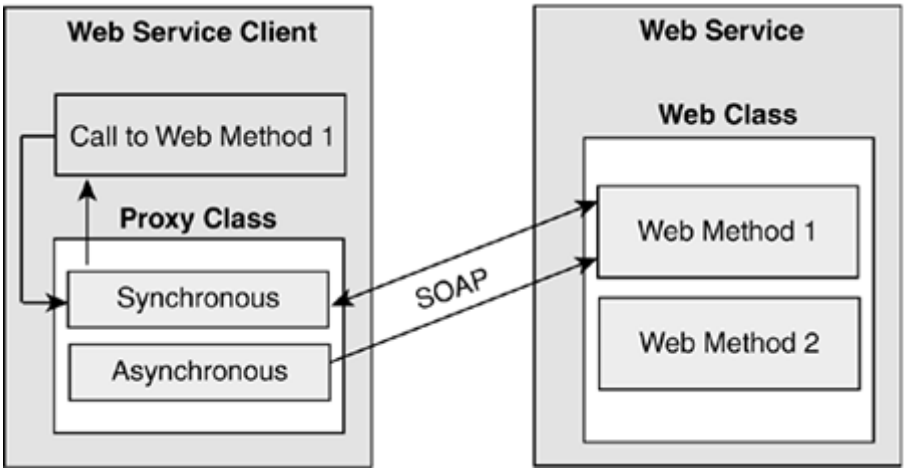
Building an XML Web Service Client

This section describes how to create a client application that consumes a Web Service. The sobjective is to make the client's call to the remote Web method as simple as calling a method in its own code. Although the actual implementation does not reach quite that level of simplicity, the code is straightforward, and much of it can be generated automatically using .NET tools.

Before delving into details, let's first take a high-level view of how a .NET Web Services client interacts with a Web Service.

The most important thing to observe in Figure 18-6 is that the client does not directly invoke the Web Service method. Instead, it calls a proxy object that performs this task. The proxy class is created from the WSDL information provided by the Web Service. We'll see how to do this shortly. The proxy code is a class that has the same class name and method names as the Web Service class does. It contains the transport logic that allows it to make the actual connection to the Web Service. It may do this either synchronously (without receiving confirmation) or asynchronously. The messages exchanged between the proxy and server are bundled within an HTTP request and transmitted using either the HTTP or the SOAP wire protocol .

Figure 18-6. Overview of how a client accesses a Web Service



Creating a Simple Client to Access the Web Service Class

To demonstrate the basic principles involved in creating a Web Service client, let's develop a console client application to access the `BirthDayWS` Web Service (refer to Figure 18-1). The client passes three arguments to the service and prints the string it returns. Recall that the service consists of the Web class `Birthday` and a single method `GeTDayBorn` :

```
public class BirthDay

{

   [System.Web.Services.WebMethod

        (Description="Return day of week for a date")]

public string GetDayBorn(int mo, int day, int yr)
```

Although the Web Service method can reside on a remote machine anywhere on the Internet, we can approach the client design as if we were within the same assembly. Here is the client code contained in the file `bdClient.cs` :

```
using System;

using System.Web.Services;

public class BirthDayClient

{

   static void Main(string[] args)

   {

      BirthDay bd = new BirthDay();

      string dayOfWeek = bd.GetDayBorn(12,20,1963);

      Console.WriteLine(dayOfWeek);

   }

}
```

Compiling this, of course, results in an error stating that `BirthDay` and `bd` cannot be found. We resolve this by creating a proxy class that performs the remote call, yet can be accessed locally by the client. The code for the proxy class is obtained by feeding the WSDL information that defines the Web Service into the .NET `wsdl.exe` utility.

Using wsdl.exe to Create a Proxy

The `wsdl.exe` utility reads the WSDL describing a Web Service and generates the source code for a proxy class to access the service; it can also use the information to create the skeleton code for a Web Service. This latter feature is designed for developers who prefer to design the WSDL

as the first step in creating a Web Service. We do not take that approach in this chapter, but you should be aware that there are WSDL editors available for that task.

The `wsdl.exe` utility is run from the command line and has numerous flags or options to govern its execution. Table 18-1 lists those most commonly used.

Table 18-1. wsdl.exe Command-Line Options

Option	Description
`/appsettingurlkey` `/urlkey:`	Specifies a key within the client's `*.config` file that contains the URL of the Web Service. The default is to hardcode the URL within the proxy class.
`/language` `/l:`	Specifies the language to use for generating the proxy class. Choices are: `CS` (C#), `VB` (Visual Basic), `JS` (JScript), and `VJS` (Visual J#). C# is the default.
`/namespace` `/n:`	Specifies the namespace for the proxy.
`/out`	Specifies file in which the generated proxy code is placed. The default is to use the XML Web Service name with an extension reflecting the language used for the code.
`/protocol`	The wire protocol to use within the proxy code: `/protocol:SOAP` SOAP 1.1 is generated `/protocol:SOAP12` SOAP 1.2 is generated `/protocol:HttpGet` or `/protocol:HttpPost`
`/server`	Generates an abstract class for the Web Service. This is used when the WSDL document is used to create a Web Service.
`/serverinterface`	Generates interfaces—rather than abstract classes—for the Web Service. Available only in 2.0 and later.

The following statement creates a proxy class from the `BirthDayWS.asmx` Web Service and

places it in `c:\BDProxy.cs` :

```
wsdl.exe /out:c:\BDProxy.cs http://localhost/ws/
BirthDayWS.asmx?WSDL
```

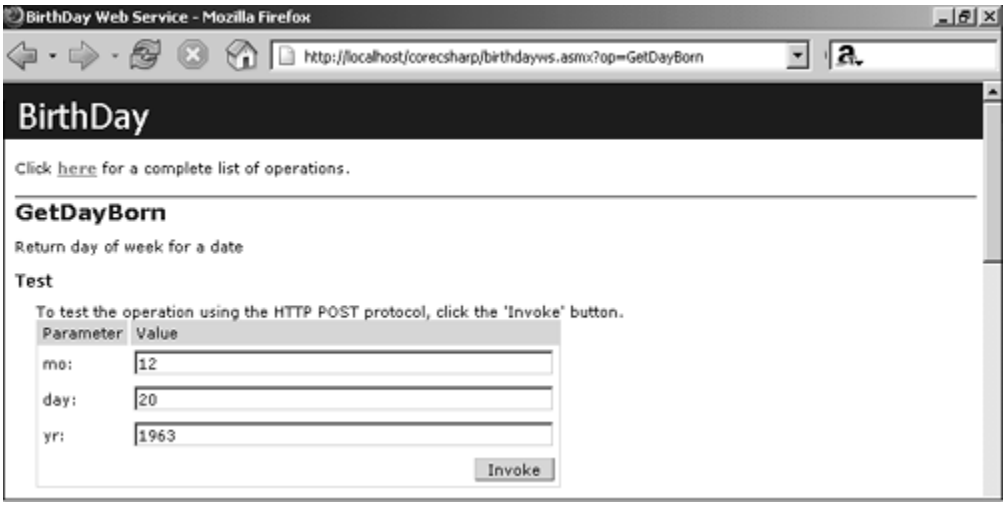
This proxy source code can be used in two ways: include it in the client's source code, or compile it into a DLL and add a reference to this DLL when compiling the client code. Let's look first at the DLL approach. This command line compiles the source code and links two DLL files containing classes required by the proxy:

```
csc /t:library /r:system.web.services.dll /r:system.xml.dll
         BDProxy.cs
```

We are now ready to compile the client source code into an executable file, `bdClient.exe` :

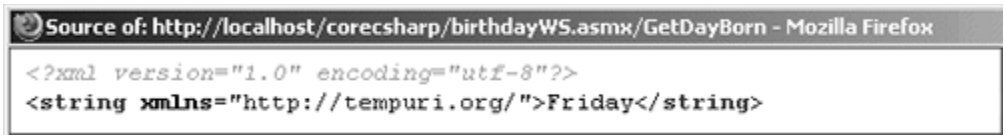
```
csc /r:BDProxy.dll  bdClient.cs
```

If we add the proxy code directly to the `bdClient.cs` file and compile it, the result is a module that produces the same output as a version that links the proxy as a separate DLL file.



To use the Web Service, fill in the parameter values and select the Invoke button. This causes the parameters to be sent to the Web Service using the HTTP `POST` protocol. The output received from the service is an XML document shown in Figure 18-4.

Figure 18-4. `BirthDayWS` output



The output from the method is included in the `string` element of the XML wrapper. Fortunately, we do not have to parse the XML to retrieve this value when writing our own SOAP client. The WSDL contract provides information that allows our client to treat the remote Web Service as a method that returns data conforming to the method's type—not as XML.

18) **Explain in detail WSDL**.

Web Services Description Language (WSDL)

WSDL is broadly defined as "an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information." [2] In our case, the endpoints are the client and a Web Service, and WSDL defines how the client interacts with the service.

[2] Web Services Descriptive Language (WSDL) 1.1—W3C Note, March 15, 2002.

When developing a Web Service from the ground up, it is good practice to develop the interface definition first—in the form of WSDL—and then map it to the implementation code. Although the sheer complexity of WSDL demands that WSDL editors be used for the task, there is still a need for the developer to have a general understanding of the WSDL structure. The same is true even if you work only on the client side and use `wsdl.exe` to create source code directly from the WSDL. Problems can arise using utilities and editors that require a familiarity with the format in order to perform troubleshooting. For example, a well-designed WSDL interface is often built from multiple documents tied together by an XML `import` element. Being familiar with the semantics and existence of the `import` element goes a long way toward solving import issues—a not uncommon source of WSDL errors.

This section introduces the basic structure of WSDL, which should satisfy the curiosity of the occasional Web Service consumer or developer. Those interested in learning more should refer to the specification that is published by W3C at http://www.w3.org/TR/wsdl.

The WSDL Structure

Specifications for the WSDL grammar define six major elements: definitions, types, message, port type, binding, and service. Let's discuss these within the context of the WSDL file that describes the sample `BirthDayWS` Web Service.

<Definitions>

This is the root element of the WSDL document. It declares multiple namespaces used throughout the document, and contains all of the other elements:

```
<definitions xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:s="http://www.w3.org/2001/XMLSchema"
   xmlns:s0="http://tempuri.org/"
   xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
   xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
   targetNamespace="http://tempuri.org/"
   xmlns="http://schemas.xmlsoap.org/wsdl/">
```

Namespaces are used to distinguish elements because it is possible that elements from different namespaces could have the same name .

<Types>

This element contains an XSD (XML Schema Definition Language) schema that describes the data types publicly exposed by the service: the parameters passed in the Web Service request, and the response:

```
<types>
   <s:schema elementFormDefault="qualified"
       targetNamespace="http://tempuri.org/">
   <s:element name="GetDayBorn">
```

```
   <s:complexType>
   <s:sequence>
   <s:element minOccurs="1" maxOccurs="1" name="mo"
      type="s:int" />
   <s:element minOccurs="1" maxOccurs="1" name="day"
      type="s:int" />
   <s:element minOccurs="1" maxOccurs="1" name="yr"
      type="s:int" />
   </s:sequence>
   </s:complexType>
   </s:element>
   <s:element name="GetDayBornResponse">
   <s:complexType>
   <s:sequence>
   <s:element minOccurs="0" maxOccurs="1" name="GetDayBornResult"
      type="s:string" />
   </s:sequence>
   </s:complexType>
   </s:element>
   </s:schema>
</types>
```

<**Message**>

Defines the data that is exchanged between the Web Service provider and consumer. Each message is assigned a unique name and defines its parameters—if any—in terms of names provided by the `types` element:

```
<message name="GetDayBornSoapIn">
```

```
<part name="parameters" element="s0:GetDayBorn" />
</message>
<message name="GetDayBornSoapOut">
<part name="parameters" element="s0:GetDayBornResponse" />
</message>
```

<PortType>

Each `<portType>` element defines the `<Message>` elements that belong to a communications transport. The `name` attribute specifies the name for the transport. The `<portType>` element contains `<operation>` elements that correspond to the methods in the Web Service. The `<input>` and `<output>` elements define the messages associated with the operation. Four types of operations are supported: one-way , in which the service receives a message; request-response , in which the client sends a request; solicit-response , in which the service first sends a message to the client; and notification , where the service sends a message to clients .

```
<portType name="BirthDaySoap">
<operation name="GetDayBorn">
<documentation>Return day of week for any date</documentation>
<input message="s0:GetDayBornSoapIn" />
<output message="s0:GetDayBornSoapOut" />
</operation>
</portType>
```

<Binding>

A set of rules that describe how the `<portType>` operation is transmitted over the wire. Wire protocols available are HTTP `GET` , HTTP `POST` , and SOAP. This example demonstrates how SOAP is specified.

As acknowledgement of the importance of SOAP as a transport protocol, the WSDL 1.1 specification includes extensions for SOAP 1.1. These extension elements include `<binding>` , `<operation>` , and `<body>` .

```
<binding name="BirthDaySoap" type="s0:BirthDaySoap">
   <soap:binding transport=
      "http://schemas.xmlsoap.org/soap/http"
      style="document" />
   <operation name="GetDayBorn">
      <soap:operation soapAction="http://tempuri.org/GetDayBorn"
         style="document" />
      <input>
         <soap:body use="literal" />
      </input>
      <output>
         <soap:body use="literal" />
      </output>
   </operation>
</binding>
```

Note that the `<operation>` element specifies the entry point for the Web method that is called on the server. One other thing to be aware of is the `style` attribute in the binding element. This value, which may be `document` or `rpc` , specifies how an operation is formatted. By default, .NET sets this value to `document` . To specify `rpc` , you must apply the `SoapRpcMethodAttribute` to the Web method:

```
[SoapRpcMethod][WebMethod]
public string GetDayBorn(string month, int day, int yr)
```

Although there is a rather spirited debate among WSDL purists as to which is better, you can safely ignore the histrionics and use the .NET default. However, knowing your options will enable you to easily work with third parties that may have a preference.

<Service>

Identifies the location of the Web Service. Specifically, it lists the name of the Web Service class, the URL, and references the binding for this endpoint.

```
<service name="BirthDay">
   <port name="BirthDaySoap" binding="s0:BirthDaySoap">
      <soap:address location=
      "http://localhost/ws/BirthDayWs.asmx" />
   </port>
</service>
```

19) Explain in detail SOAP.

Simple Object Access Protocol (SOAP)

SOAP is a platform-neutral protocol for exchanging information. Its cross-platform capabilities are attributable to its use of XML to define the data being passed and support for HTTP as a communications protocol. SOAP is the most popular and flexible protocol for the exchange of information between Web Service consumers and providers. Its format allows it to define complex data structures not supported by the competing HTTP `GET` and `POST` protocols.

Our discussion of SOAP follows the same approach used with WSDL: We examine the basic features of SOAP using the request/response messages generated from the `BirthDayWS` Web Service example. The format of these messages is described on the Web page containing the desired method(s) of the Web Service.

A SOAP Request Message

The header for a SOAP request reveals that the SOAP request is packaged as an HTTP `POST` request to the server designated by the `Host` field. The `length` field specifies the number of characters in the body of the `POST` , and `SOAPAction` indicates the namespace and method to be contacted.

```
POST /ws/BirthDayWS.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
```

```
SOAPAction: "http://tempuri.org/GetDayBorn"
```

Listing 18-3 shows the XML template for the SOAP message that is sent to the server.

Listing 18-3. `GetdayBorn` SOAP Request Content

```
<?xml version="1.0" encoding="utf-8"?>

<soap:Envelope

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001

➥/XMLSchema" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:tns="http:/

➥/tempuri.org/" xmlns:types="http://tempuri.org/encodedTypes"
xmlns:soap="http://schemas

➥.xmlsoap.org/soap/envelope/">

   <soap:Body

      soap:encodingStyle=

         "http://schemas.xmlsoap.org/soap/encoding/">

      <tns:GetDayBorn>

         <mo xsi:type="xsd:int">int</mo>

         <day xsi:type="xsd:int">int</day>

         <yr xsi:type="xsd:int">int</yr>

      </tns:GetDayBorn>

   </soap:Body>

</soap:Envelope>
```

The overall structure of a SOAP message is not complex. It is an XML document that has a mandatory root element, `<Envelope>` , an optional `<Header>` element, and a mandatory `<Body>` .

A SOAP envelope, as the name implies, is conceptually a container for the message. The SOAP header represents a way to extend the basic message. It may contain additional information about

the message and—as we will see later—can be used to add a measure of security. The SOAP body contains what one would regard as the actual data: the arguments sent to the service and the response. The contents of the `<Body>` element in this example consist of the method name and its three parameters that correspond to the call made within the client code:

```
string dayOfWeek = bd.GetDayBorn(12,20,1963);
```

A SOAP Response Message

The SOAP body of the response includes a `<GetDayBornResult>` element (see Listing 18-4) that contains the response from the Web Service and identifies it as a string type.

Listing 18-4. `GeTDayBorn` SOAP Response Content

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"xmlns:soap-
enc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:tns="http://tempuri.org/" xmlns:types="http://tem-
puri.org/encodedTypes" xmlns:soap="http://schemas.xml-
soap.org/soap/envelope/">
   <soap:Body
      soap:encodingStyle=
         "http://schemas.xmlsoap.org/soap/encoding/">
      <tns:GetDayBornResponse>
         <GetDayBornResult
            xsi:type="xsd:string">string</GetDayBornResult>
      </tns:GetDayBornResponse>
   </soap:Body>
</soap:Envelope>
```

Using the SOAP Header for User Authentication

The optional SOAP header is available for adding miscellaneous information about its associated SOAP message. One popular use for this header is to include identification information about the user making the request. This enables user authentication to be performed by the methods within the Web Service.

20) **What are the issues of building web services with complex data types.**

Using Web Services with Complex Data Types

The `BirthDayWS` Web Service used throughout this chapter accepts integers as input and returns a `string` value. This is useful for introducing Web Service principles because HTTP `GET` , HTTP `POST` , and SOAP all support it. However, Web Services also have the capability of serving up more complex data types such as data sets, hash tables, images, and custom objects.

Before data can be sent to or from a Web Service, it is serialized using XML serialization. Conversely, it is deserialized on the receiving end so it can be restored to its original type. As we saw in Chapter 4, "Working with Objects in C#," not all data can be serialized. Thus, when designing a Web Service, it is important to understand restrictions that apply to serialization:

- XML serialization can only be used with classes that contain a public parameterless constructor. For example, you may have a Web Service that returns a hash table because it has the constructor `public Hashtable()` . On the other hand, the `Bitmap` class does not have a parameterless constructor and cannot be used as a return type.
- Read-only properties in a class cannot be serialized. The property must have a `get` and `set` accessor and be public.
- Fields must be public to be serialized; private ones are ignored.

In this section, we work with two Web Service examples that illustrate the use of complex data. Our first example creates a Web Service that accepts the name of an image and returns it as a byte stream. The second example creates a client to use the Amazon Web Services provided by Amazon.com, Inc. These services offer a rich—but practical—sampling of accessing multiple Web methods and processing a wide variety of custom classes.

21) **Explain with an example of a web service which returns images**.

A Web Service to Return Images

Image manipulation usually requires representing an image as a `Bitmap` object. However, because bitmaps cannot be serialized and transferred directly, we must find an indirect way to

transport an image. The not-so-difficult solution is to break the image into bytes and return a byte stream to the client, who is responsible for transforming the stream to an image.

The logic on the server side is straightforward: a `FileStream` is opened and associated with the image file. Its contents are read into memory and converted to a byte array using

```
tempStream.ToArray()
```

This byte array is then sent to the Web client (see Listing 18-6).

Listing 18-6. Web Service to Return an Image as a String of Bytes

```
<%@ WebService Language="C#" Class="WSImages" %>
using System;
using System.Web.Services;
using System.IO;
using System.Web.Services.Protocols;
public class WSImages: System.Web.Services.WebService {
   [WebMethod(Description="Request an Image")]
   public byte[] GetImage(string imgName) {
      byte[] imgArray;
      imgArray = getBinaryFile("c:\"+imgName+".gif");
      if (imgArray.Length <2)
      {
         throw new SoapException(
           "Could not open image on server.",
            SoapException.ServerFaultCode);
      } else
      {
         return(imgArray);
```

```
      }
   }
   public byte[] getBinaryFile(string filename)
   {
      if(File.Exists(filename)) {
         try {
            FileStream s = File.OpenRead(filename);
            return ConvertStreamToByteBuffer(s);
         }
         catch(Exception e)
         {
            return new byte[0];
         }
      } else { return new byte[0]; }
   }
   // Write image to memory as a stream of bytes
   public byte[] ConvertStreamToByteBuffer(Stream imgStream) {
      int imgByte;
      MemoryStream tempStream = new MemoryStream();
      while((imgByte=imgStream.ReadByte())!=-1) {
         tempStream.WriteByte(((byte)imgByte));
      }
      return tempStream.ToArray();  // Convert to array of bytes
   }
}
```

Our client code receives the byte stream representing an image and reassembles it into a `Bitmap`

object. Because the `Bitmap` constructor accepts a stream type, we convert the byte array to a `MemoryStream` and pass it to the constructor. It can now be manipulated as an image.

```
WSImages myImage = new WSImages();
try {
   // Request an image from the Web Service
   byte[] image = myImage.GetImage("stanwyck");
   MemoryStream memStream = new MemoryStream(image);
   Console.WriteLine(memStream.Length);
   // Convert memory stream to a Bitmap
   Bitmap bm = new Bitmap(memStream);
   // Save image returned to local disk
   bm.Save("c:\bstanwyck.jpg",
        System.Drawing.Imaging.ImageFormat.Jpeg);
}
catch (WebException ex)
{
   Console.WriteLine(ex.Message);
}
```

22) **Explain the steps in using AMAZON web services**.

Using Amazon Web Services

To use the Amazon E-Commerce Service, you must register for a developer's token, which is required as part of all requests made to the Web Services. In addition, you should download (http://www.amazon.com/ webservices) the developer's kit that contains the latest documentation, examples, and—most importantly—a WSDL file defining all the services.

An examination of the WSDL file reveals that `AmazonSearchService` is the Web Service class that contains the numerous search methods available to clients . These methods provide the ability to search the Amazon product database by keyword, author, artist, ISBN number, manufacturer, actor, and a number of other criteria. Each search method takes a search object as

an argument that describes the request to the server and returns a `ProductInfo` object. For example, a request to search by keywords looks like this:

```
AmazonSearchService amazon = new AmazonSearchService();

KeywordRequest kwRequest = new KeywordRequest();

// Set fields for kwRequest

ProductInfo products = amazon.KeywordSearchRequest(kwRequest);
```

Sending a Request with the AmazonSearchService Class

Table 18-3 contains a sampling of the methods available for searching Amazon products. These methods are for accessing the Web Service synchronously. An asynchronous form of each method is also available that can be accessed using the techniques discussed earlier in this chapter.

Table 18-3. Selected Methods of AmazonSearchService Class

Method	Description
`ProductInfo KeyWordSearchRequest (KeywordRequest req)`	Method to return items that contain one or more keywords provided in request.
`ProductInfo AsinSearchRequest (AsinRequest req)`	Method to return a book having a requested Amazon Standard Identification Number (ASIN) that is the same as the book's ISBN. Represented as a 10-digit string.
`ProductInfo AuthorSearchRequest (AuthorRequest req)`	Method to return names of all books by requested author.
`ProductInfo ActorSearchRequest`	Method to return video titles of movies in which a specified actor or actress was a cast member.

Method	Description
`(ActorRequest req)`	
`ProductInfo PowerSearchRequest (PowerRequest req)`	Method to retrieve book information based on a Boolean query that may include a combination of title, subject, author, keyword, ISBN, publisher, language, and publication date (pubdate).

Each call to a Web method passes an object that describes the search request. This object is different for each method—for example, `AuthorSearchRequest` requires an `AuthorRequest` object, whereas `KeyWordSearchRequest` requires an instance of the `KeywordRequest` class. These classes expose almost identical fields. Each contains a unique string field that represents the search query, five other required fields common to each class, and some optional fields for sorting or specifying a locale. Table 18-4 lists unique and shared fields for each method listed in Table 18-3.

Table 18-4. Selected Fields for Classes That Define a Search Request

Using the ProductInfo Class to Process the Web Service Response

The Web Service responds to the search request with a `ProductInfo` object containing results from the search. This object exposes three important fields: a `TotalResult s` string contains the number of products retrieved by the request, a `TotalPages` string that indicates how many pages these results are displayed in, and the important `Details` array that contains a detailed description of products that constitute one returned page of results. This array is of the `Details` type. Table 18-5 shows the fields that are related to books.

Table 18-5. Selected Fields of the Details Class

Field	Description
`string ProductName`	Name of a single product.
`string SalesRank`	Ranking of product based on sales of items of its type.
`string Publisher`	Publisher of book.
`String ListPrice` `string OurPrice`	List and sales price of book.

`Reviews[] Reviews`	The `Reviews` class contains several fields relating to
	`reviews of the book:`
	`string AvgCustomerRating`
	`string TotalCustomerReviews`
	`CustomerReview CustomerReviews`
	`string Comment`
	`string Rating`

`String[] Authors`	One or more authors for the book.

This is only a small sample of the fields available in the `Details` class. There is a particularly rich set of fields worth exploring that define video products.

23) **Explain the steps in creating a proxy for the Amazon Web Services**.

Creating a Proxy for the Amazon Web Services

Our first step is to create a proxy class from the Amazon WSDL information. The downloadable kit includes a WSDL file, and it can also be retrieved from the Internet, as we do here. Using the VS.NET command line, we place the proxy source code in the file `AZProxy.cs` .

```
wsdl.exe  /out:c:\client\AZProxy.cs
      http://soap.amazon.com/schema3/AmazonWebServices.wsdl
```

Next, we create an assembly, `AZProxy.dll` , containing the proxy that will be used by client code. It is linked to assemblies containing .NET Web classes required by the application.

```
csc/t:library /r:system.web.services.dll /r:system.xml.dll
      AZProxy.cs
```

You can make a quick test of the service using this barebones application, `azclient.cs` :

```
using System;
using System.Web.Services;
namespace webclient.example {
public class AmazonClient
{
   static void Main(string[] args)
   {
      // Search for books matching keyword "butterflies"
      AmazonSearchService amazon = new AmazonSearchService();
      KeywordRequest kwRequest = new KeywordRequest();
      kwRequest.keyword = "butterflies";
      kwRequest.type = "heavy";
      kwRequest.devtag= "**********"; // your developer token
```

```
      kwRequest.mode = "books";       // search books only
      kwRequest.tag = "webservices-20";
      kwRequest.page = "1";          // return first page
      ProductInfo products =
          amazon.KeywordSearchRequest(kwRequest);
      Console.WriteLine(products.TotalResults);  // Results count
   }
}
}
```

Compile and execute this from the command line:

```
csc /r:AZProxy.dll azclient.cs
azclient
```

When `azclient.exe` is executed, it should print the number of matching results.

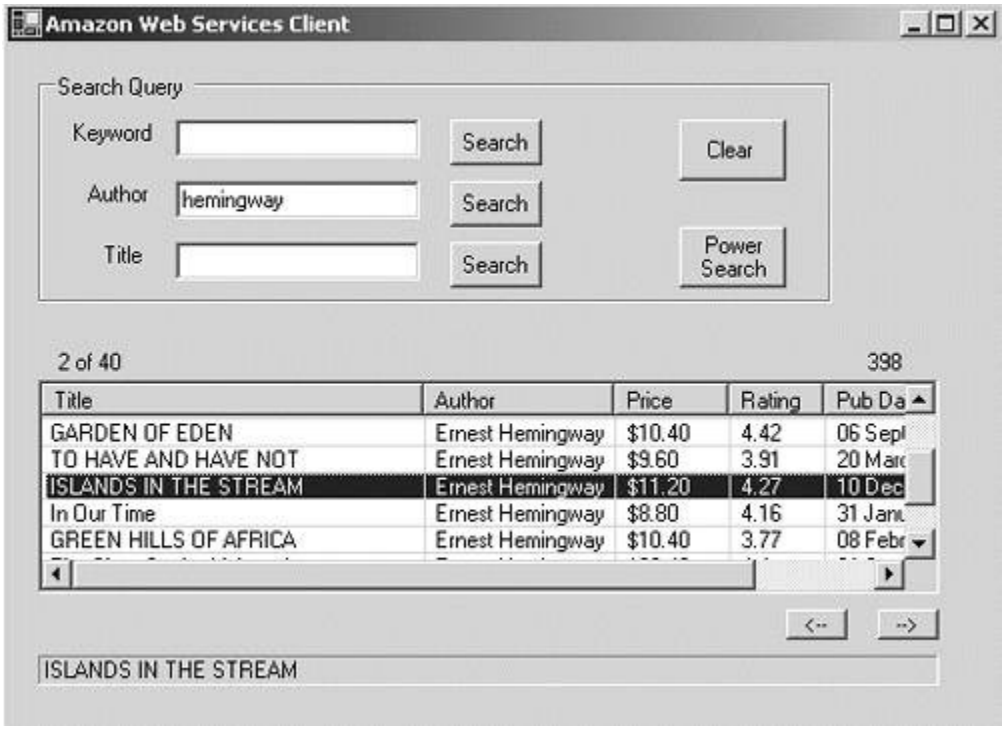
24) **Explain with an example for building a WinForms Web Service Client.**

Building a WinForms Web Service Client

Let's design a Windows Forms application that permits a user to perform searches on books using multiple search options. Open VS.NET and select a Windows application. After this is open, we need to add a reference to the proxy assembly `AZProxy.dll` . From the menu, select Project - Add Reference - Browse. Click the assembly when it is located and then click OK to add it as a reference. You also need to add a reference to `System.Web.Services.dll` , which contains the required Web Service namespaces.

The purpose of the application is to permit a user to search the Amazon book database by keyword, author, or title. The search can be on a single field on a combination of fields. Figure 18-7 shows the interface for entering the search values and viewing the results. The Search buttons submit a search request based on the value in their corresponding text box. The Power Search button creates a query that logically "ands" any values in the text boxes and submits it.

Figure 18-7. Overview of how a client accesses a Web Service



A single page of results is displayed in a `ListView` control. Beneath the control are buttons that can be used to navigate backward and forward through the results pages.

Each Search button has a `Click` event handler that calls a method to create an appropriate request object and send it to the Amazon Web Service. A successful call returns a `ProductInfo` object containing information about up to 10 books meeting the search criteria. Listing 18-7 displays code that creates an `AuthorRequest` object, sends it to the Web Service, and calls `FillListView` to display the results in the `ListView` control.

Listing 18-7. Client Code to Display Results of Author Search— `azwsclient.cs`

```
// Fields having class-wide scope
int CurrPg;               // Current page being displayed
string SearchMode = "";   // Current search mode
int MaxPages =1;          // Number of pages available
// This method is called when author Search button is clicked
```

```
private bool AuthorReq()
{
   AmazonSearchService amazon = new AmazonSearchService();
   AuthorRequest auRequest = new AuthorRequest();
   auRequest.author = textBox2.Text;   // Get author from GUI
   auRequest.type = "heavy";
   auRequest.devtag= "****KLMJFLGV9";  // Developer token
   auRequest.mode = "books";
   auRequest.tag = "webservices-20";
   auRequest.page = CurrPg.ToString();
   try
   {
      // Call Web Service with author query
      ProductInfo products =
            amazon.AuthorSearchRequest(auRequest);
      FillListView(products);
      return(true);
   }
   catch (SoapException ex)
   {
      MessageBox.Show(ex.Message);
      return(false);
   }
}
private void FillListView(ProductInfo products)
   listView1.Items.Clear();   // Remove current entries
   label6.Text="";            // Clear any title
```

```
   label4.Text = products.TotalResults;
   label5.Text = CurrPg.ToString()+" of "+products.TotalPages;
{
   MaxPages =  Convert.ToInt32(products.TotalPages);
   ListViewItem rowItem;
   string auth,rev;
   for (int i=0; i< products.Details.Length; i++)
   {
      rowItem = new
            ListViewItem(products.Details[i].ProductName);
      // Add Author. Make sure author exists.
      object ob = products.Details[i].Authors;
      if (ob != null)  auth =
            products.Details[i].Authors[0]; else auth="None";
      rowItem.SubItems.Add(auth);
      // Add Price
      rowItem.SubItems.Add(products.Details[i].OurPrice);
      // Add Average Rating
      ob = products.Details[i].Reviews;
      if (ob != null)  rev =
            products.Details[i].Reviews.AvgCustomerRating;
            else rev="None";
      rowItem.SubItems.Add(rev);
      // Add Date Published
      rowItem.SubItems.Add(
            products.Details[i].ReleaseDate);
      listView1.Items.Add(rowItem);
```

```
    }
}
```

The keyword, title, and power searches use an identical approach: Each has a routine comparable to `AuthorReq` that creates its own request object. The only significant difference pertains to the power search that creates a Boolean query from the search field values. The format for this type query is `field:value AND field2:value AND field3:value.` For example:

```
"author:hemingway AND keywords:Kilimanjaro"
```

This application was designed as a Windows Forms application. It could just as easily be set up as Web page under ASP.NET. The code to reference the assembly `AZProxy.dll` is identical. The `ListView` control is not supported on a Web Form, but you could easily substitute a `DataGrid` for it.

25) **What are the factors affecting performance of a web service?**

Web Services Performance

The performance of a Web Service from both the client and server side is affected by a variety of factors. Some are .NET related and others are inherent in the technology. The solutions for improving Web Services performance range from shaving a few milliseconds off the way .NET sends a request to simply eschewing Web Services for an alternate protocol when transferring large amounts of data. We'll look at all of these.

Configuring the HTTP Connection

Connections (HTTP) to Internet resources are managed in .NET by the `ServicePoint` class. This class includes properties to specify a connection's timeout interval, set its security protocol, and manage the use of server security certificates. It also includes properties that directly affect how much delay is incurred before a Web Service request is transmitted over a network: `UseNagleAlgorithm` and `Expect100Continue` . Despite the awkward sounding names , setting their properties is as easy as assigning a `TRue` or `false` value to them. The default for both is `true` .

Expect100Continue

This property determines whether a `POST` request should expect to receive a 100-Continue response from the server to indicate that the data can be posted. If this value is set to `TRue` , only the request header portion of a request is sent to the server. If the server finds no problem with the header, it returns a 100-Continue response, and the data is then sent. Two trips are required. If the property is set to `false` , the initial request includes the headers and data. If it is rejected by the server, the data has been sent unnecessarily; if it is accepted, a second trip is not necessary.

Because Web Service calls tend to pass small amounts of data, it can be beneficial to turn this property off. Even if a request is rejected, only a small amount of data will have been sent.

The Nagle Algorithm

One way to improve network efficiency is to reduce the number of small data packets sent across a network. To accomplish this, the software layer controlling the underlying TCP (Transmission Control Protocol) connection attempts to accumulate, or buffer, small messages into a larger TCP segment before they are sent. The technique to do this is based on the Nagle algorithm . [4]

[4] RFC 896, "Congestion Control in IP/TCP Internetworks," by John Nagle, 1984.

The crux of the algorithm is that small amounts of data should continue to be collected by TCP until it receives acknowledgment to send the data. .NET institutes a delay of up to 200 milliseconds to collect additional data for a packet. For a typically small Web Service request, there may be no reason to include this delay. It's an option you can experiment with.

To set the `Expect100Continue` and `UseNagleAlgorithm` properties, it is necessary to get a reference to the `ServicePoint` object being used to handle the Web request. This is done in the proxy code on the client side. Refer to Listing 18-2, and you'll see that the proxy code consists of a class derived from the base `SoapHttpClientProtocol` class. By overriding the inherited `GetWebRequest` method, you can customize the `WebRequest` object before the request is sent to the Web Service.

Add the following code to override the `GetWebRequest` method. Inside the method, you use the `Uri` object to get the `ServicePoint` . Its properties can then be turned off or on to test performance:

```
// Using System.Net must be added

protected override WebRequest GetWebRequest(Uri uri)

{

   // Locate ServicePoint object used by this application

   ServicePoint sp =
```

```
        ServicePointManager.FindServicePoint(uri);
   sp.Expect100Continue = false;
   sp.UseNagleAlgorithm = false;
   return WebRequest.Create(uri);
}
```

Working with Large Amounts of Data

Although the XML format offers a great deal of flexibility in representing data, it can place a potentially large burden on a network because of the large files that can be generated. Moreover, processing XML data can require extensive memory resources on the client and server. Finally, there is the nature of a Web Service: If the transmission fails at any point, the entire response must be resent . This is in contrast to FTP and the HTTP `GET` verb that allow partial data transmission.

For large amounts of data, consider these options:

- Use FTP or a Web client as described in Section 17.6, "Creating a Web Client with WebRequest and WebResponse."
- Avoid calls as much as possible by caching data rather re-requesting it.
- Look at compression techniques such as HTTP transport compression or the SOAP extensions that can compress part of a Web Service message.

Wait for new Web Service standards. Of particular note is Message Transmission Optimization Mechanism (MTOM), a new W3C recommendation pushed by Microsoftthat details a method for attaching large binary data to a SOAP message.