

CHAPTER 7

Simple Object Access Protocol (SOAP)

IN THIS CHAPTER:

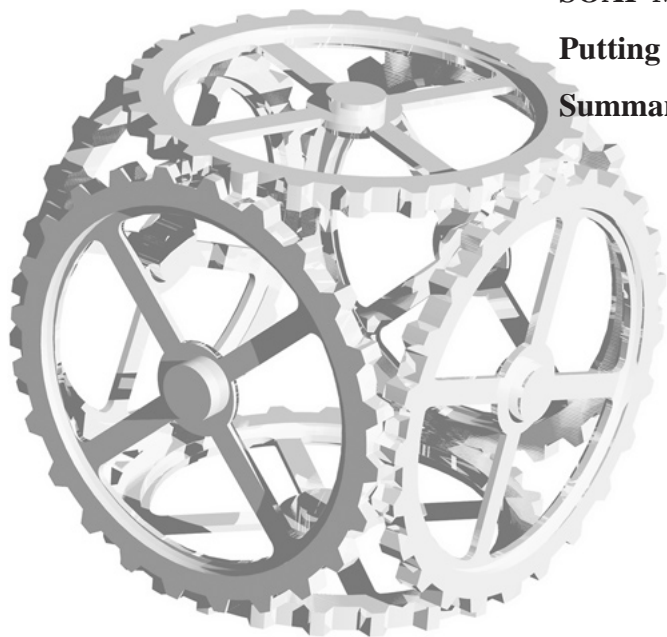
What Is SOAP?

SOAP Encoding

SOAP Messages

Putting SOAP to Use

Summary



2 Building Web Services and .NET Applications

This chapter provides an in-depth look at the Simple Object Access Protocol (SOAP) 1.1 specifications. Instead of simply regurgitating the specifications, this chapter approaches them from a slightly different angle. In addition, all the actual schema definitions that make up the SOAP encoding, serialization, and message rules are examined. We start with a high-level overview of messages, jump straight into the encoding and serialization rules, followed by an in-depth look at message rules. Even within the discussion on encoding and serialization, there's a high-level discussion of serialization, followed by encoding, and then an in-depth look at serialization. This approach was used to minimize the need to reread different sections, making it easier to grasp the specifications completely in a shorter time. At the end of the chapter, we discuss a sample application that uses SOAP services in an ASP page to handle requests from an HTML client.

What Is SOAP?

Most people tend to think of SOAP as nothing more than a protocol for Remote Procedure Calls (RPC) over Hypertext Transfer Protocol (HTTP). However, this is only one implementation of *SOAP*, which is defined as a lightweight protocol for passing structured and typed data between two peers using XML. The specification doesn't require the use of HTTP or even a request/response type of conversation. Instead, SOAP can be used with any protocol that supports the transmission of XML data from a sender to a receiver. In fact, both Microsoft and IBM have implemented SOAP messages over SMTP, which means SOAP messages can be routed through e-mail servers.

The bottom line is SOAP is nothing more than a lightweight messaging protocol, which can be used to send messages between peers. The main goals of SOAP are focused on providing a common way to package message data and define encoding rules used to serialize and deserialize the data during transmission. Another goal was to provide a model that can be used to implement RPC operations using SOAP. All these goals are considered "orthogonal" in the specification, which means they are independent, but related. For example: A SOAP message should define encoding rules, but these rules needn't be the same rules defined in the specification.

The Message Exchange Model

Even though SOAP is basically a one-way protocol, it can be used to transmit XML data using any type of conversation supported by the transport protocol used. This means SOAP can be used for one-way, multicast, or request/response operations as

Chapter 7: Simple Object Access Protocol (SOAP) 3

long as the transport protocol supports them. These different operations are referred to as “message exchange models” in the SOAP specification. The patterns are actually implemented by combining SOAP messages. For example: The request/response model is implemented with two SOAP messages: one that carries the request and one that carries the response.

Another important concept to understand with SOAP is it’s intended to be transmitted along a message path from the sender to the receiver. This message path can pass through multiple stops before reaching the final receiver. This allows for processing of the message at one or more stops along the way. The actual implementation of transmitting the data across the message path is again based on the transport protocol used.

One good analogy to use with SOAP is to think of it in terms of an interoffice message. Most people are familiar with the concept of interoffice envelopes, which are used to send information between people or groups in an organization. On the front of the envelope are a series of lines, used to enter sender and receiver information, such as the department and name. When the receiver gets the message, that person crosses their name off the list on the envelope. The next step is based on information found on the envelope or in the message. If the message was one way, it stops there, and the envelope can be reused for another message. If the sender needs a response, then the receiver reverses the sender and receiver information on the envelope’s next line and sends it back. The message could also be intended for multiple people, in which case the receiver passes it on to the next person on the list.

We mentioned previously that one of the goals of SOAP is to define how messages are packaged for transmission. The name of the XML element used to package the SOAP data is actually called the *SOAP envelope*. Inside the envelope, we can find header information and the actual message data. This is explored in more detail when we look at the SOAP envelope later in this chapter. However, the main point here is the SOAP message contains the information used to control its movement along the message path. This information is independent of the actual transport protocol used to transmit the SOAP message, which is similar to the way interoffice mail is handled.

HTTP Bindings

Another set of goals the SOAP team wanted to accomplish was to provide information on how to bind SOAP with a transport protocol. Because SOAP was primarily developed to support distributed services over the Internet, they chose the HTTP protocol. Two different bindings have been defined in the specification: one for standard HTTP and one for the HTTP Extension Framework. For this discussion, we only look at the standard HTTP implementation. Information about the HTTP

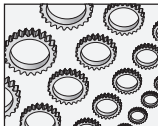
4 Building Web Services and .NET Applications

Extension Framework can be found in the SOAP specifications. Also important to understand is the SOAP specification doesn't override HTTP semantics. Instead, the binding of SOAP over HTTP maps to existing HTTP semantics.

Few requirements exist when using SOAP over HTTP. The main restriction is this: a SOAP message can only be sent using POST operations. For both an HTTP Request and Response, the Content-Type of the document must be set to "text/xml." This is important to remember because, if the content type isn't set correctly, the data could be transformed into a format that isn't readable by XML parsers. For instance, if the type was set to "text/html," invalid HTML characters would be converted to escape sequences that a parser couldn't use. The specific requirements for a request or response are detailed in the following paragraphs.

HTTP Request

The HTTP Request is required to have at least one header field with the name "SOAPAction." This field is mandatory and contains a URI that's used to indicate the intent of the SOAP message. The specification also allows for more than one SOAPAction field to support multiple intentions. No restrictions are imposed on the format of the URI that's used. In addition, the SOAPAction field isn't meant to replace the HTTP Request-URI, which typically maps to a FORM's action field. However, if the SOAPAction is initialized with an empty string, this means the intent can be found in the Request-URI. If the field doesn't contain any value, this means no indication of intent exists.



NOTE

A URI is a unique name recognized by the processing application that identifies a particular resource. URIs include Uniform Resource Locators (URL) and Uniform Resource Name (URN).

The SOAPAction field is a little confusing and there's been a lot of discussion about how to use it on news servers. The field can be used in a variety of different ways. The following list shows some examples:

- ▶ SOAP HTTP Servers can use the SOAPAction field to route the message to different receivers.
- ▶ Firewalls or other servers can use the SOAPAction field to filter the SOAP message.
- ▶ Messaging implementations can use the SOAPAction field to indicate the message being sent.
- ▶ RPC implementations can use the SOAPAction field to identify the method name.

Chapter 7: Simple Object Access Protocol (SOAP) 5

As you can see, this field can be used in many different ways. Currently, with RPC implementations, the SOAPAction field usually contains a namespace with the method name appended. The following code listing shows an HTTP Request header that's used to transmit a SOAP message.

```
L 7-1 POST /Pattern HTTP/1.1
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "urn:rdacustomsoftware-com:Employee.GetEmployees"
```

HTTP Response

The response header returned from an HTTP SOAP request doesn't return anything different than a normal HTTP response, however, restrictions exist on the status codes. For any successful receipt and processing of an HTTP SOAP message, the status code returned should be in the 2xx range. If errors occurred—system or application—then the status code returned must be 500, “Internal Server Error”. If a status code of 500 is returned, then the SOAP message must also contain a SOAP Fault element. The SOAP Fault is discussed in more detail later in this chapter.

RPC over SOAP

We already discussed the fact that SOAP supports the capability to implement Remote Procedure Calls (RPC). By taking advantage of the SOAP binding over HTTP, a request and response model supports the capability to invoke a method, and then to return the response. The actual implementation in SOAP is straightforward with few rules. The following is a description of the rules for implementing RPC in SOAP when using the SOAP encoding rules:

- ▶ The method is represented as a structure in the SOAP message. The structure, of course, is XML, and the element that represents the method must have the same name as the method. In addition, the element's datatype must match the datatype defined for the return value.
- ▶ Each parameter in the method is represented as a child element that's named after the parameter and defines the same datatype as the parameter. The parameter elements must follow the same order as the parameters in the method.
- ▶ Responses are structured the same way a method call is structured. The name of the response element isn't specified, but the standard implementation is to append “Response” to the method name.
- ▶ If the response returns a value, then the first child element contains the value. As with the response name, no specifications exist on what this element should

6 Building Web Services and .NET Applications

be named, but the datatype must match the type defined for the method's return value. You can use one of the SOAP defined elements we discuss later, you could name it Response, or you could name it anything else.

- ▶ Any out or in/out parameters are also returned in the response structure following the "Response" element. These parameter elements must be named the same as the method parameter with the datatype also matching. And, as with the method call, these return parameters must maintain the same order as the original method.
- ▶ If errors are in the method invocation, then the SOAP Fault element must be returned. We discuss SOAP Faults in more detail later in this chapter.

As you can see, these rules are easy to follow and make RPC over SOAP easy to understand and implement. However, the previous description was also based on the use of SOAP encoding rules, which we examine shortly, and different encoding rules might be used. In addition, nothing in the specifications prohibits the use of different rules. Regardless of the rules used, RPC is still supported and the SOAP RPC rules are still valid. Instead of elements containing data, however, attributes may contain the data. In other words, the actual SOAP RPC specification is more generic than the previously provided description.

SOAP Is More Than RPC

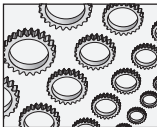
Because HTTP is a request/response protocol, it was also an ideal candidate for demonstrating RPC implementations using SOAP. Unfortunately, this one implementation has been the primary focus of most discussions on SOAP. SOAP was meant to provide a lightweight protocol for sending structured and typed data using XML. Instead of using method names and parameters, a SOAP message could be sent a server that routes it to multiple receivers in an organization. In addition, one of the changes between the SOAP 1.0 and 1.1 specifications was to move the HTTP binding information to the bottom of the document to take the focus off HTTP. By now, it should be obvious that SOAP is much more than an RPC protocol over HTTP.

SOAP Encoding

Encoding rules are used to describe how data should be formatted for a particular operation. One example is the character encoding rules identified by the UTF-8 value typically used for XML and HTML. This set of rules defines how characters

Chapter 7: Simple Object Access Protocol (SOAP) 7

are represented using binary numbers a computer can understand. The operation of converting a character from the letter *A* to a value of 0x41 is considered encoding the character. Converting the value of 0x41 back to a letter *A* is called *decoding*. The operation of encoding data, transporting that data from one location to another, and then decoding it is also called *serialization*.



NOTE

For this discussion, binary data is considered encoded data. Character data is the human-readable, decoded, format found in XML documents.

The XML encoding rules handle the conversion of character data to and from binary data. In addition, programming languages that use XML have different binary representations of the data contained in the document. Numbers are an example of datatypes that can have a wide range of binary representations. On some systems, an integer is a 16-bit value while, on other systems, it may be a 32-bit value. In XML, the data is always represented as character data, which means an integer needs to be converted into character data when it's included in an XML document. If the system that converts the XML character data back into binary data assumes the number is an integer, a corruption of data can occur. For example, if a 32-bit system sends the number 48,000 to a 16-bit system, the resulting conversion to an integer will be incorrect because the maximum positive value a 16-bit integer can hold is 32,767. What we need is something that describes what an integer is, so systems know how to convert it from binary data to character data and back again, which is where the SOAP encoding rules apply.

Understanding Serialization

Before we get into the SOAP encoding rules, it helps to get a solid understanding of what happens when data is serialized using SOAP. We previously mentioned the operation of encoding, sending, and then decoding data is referred to as serialization. This makes sense in the context of sending messages; however, another definition is “serialization is the process of writing or reading an object to or from a storage medium.” With SOAP, the objects are datatypes and the storage medium is an XML document. When the XML document is serialized, you can think of the document as an object and the storage medium for the document can be memory or file based.

With a SOAP message, several translations occur when data is sent from one system to another. Figure 7-1 shows the flow of an integer from a C++ application to a Java application. In the top-left corner is the C++ code that defines an integer and

8 Building Web Services and .NET Applications

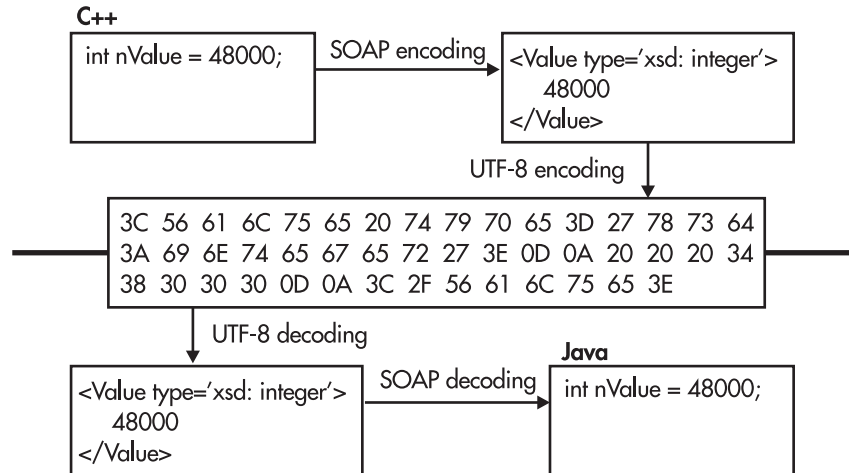


Figure 7-1 *Serialization*

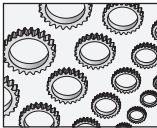
assigns it the value of 48000. Next, SOAP encoding rules are applied to the integer and we end up with an XML element that has a type of integer. The XML document is then encoded with UTF-8 encoding rules and transmitted to the receiver as binary data. Once it reaches the receiver, the binary data is decoded using UTF-8 rules to convert the data back into character data. Finally, the XML character data is decoded using SOAP rules to create a Java integer with the value of 48000. This is an example of a simple datatype. SOAP also defines rules that handle much more complex datatypes.

The SOAP specification also defines a comprehensive set of rules for serialization, which defines how the application data should be handled. These serialization rules aren't the same as encoding rules, which are also found in the specification. Encoding rules define how the data should be converted from binary form to text and back again. Serialization rules define how the XML data should be structured for SOAP messages, which represents a restriction on how XML is used. Normally, data can be carried in XML as an attribute or as element content. The SOAP serialization rules only allow for data to be carried as element content.

Before we get into the serialization rules, several things are associated with encoding rules that you need to understand first. This is one area that makes the actual SOAP specifications somewhat confusing. The serialization rules are presented before the encoding rules, which means you need to read through everything several times before it makes sense. Instead of repeating that here, we look at the encoding rules next, and then follow up with the serialization rules.

Encoding Rules

The SOAP encoding rules are based on XML Schema specifications, which were discussed in Chapter 3. These specifications cover everything from simple datatypes, such as integers, to complex datatypes, such as structures with nested structures. However, some restrictions and extensions exist to the XML Schema defined in the SOAP specifications. When using SOAP encoding, any restrictions defined in the SOAP specification take precedence over the XML Schema specifications. In addition, it isn't mandatory for an implementation to use the SOAP encoding rules. Nothing is stopping a company from defining its own encoding rules for SOAP implementations. To ensure compatibility with a wide audience, however, the use of SOAP encoding rules are encouraged.



NOTE

When discussing the SOAP encoding rules two terms are used throughout the specifications, which the reader needs to understand: Accessor and value. Accessor is the reference used to access application data in a message. With SOAP, the element names are typically used as accessors. Value is the actual application data being encoded.

The SOAP encoding rules are contained in a W3C schema that uses the following namespace declaration:

```
L 7-2 xmlns:SOAP-ENC='http://schemas.xmlsoap.org/soap/encoding/'
```

The URL identified in this namespace also represents the physical location where the schema can be found. These rules are based on W3C schema specifications with additional attributes, elements, and datatypes designed to support the previously mentioned SOAP serialization rules. The majority of elements defined in the schema is based on simple `xsd:type` definitions with additional attributes added. For instance, the SOAP encoding rules define an element named `SOAP-ENC:string` based on the `xsd:string` type.

The specification also identifies two different types of elements that can be used in SOAP messages. The first type is called an *independent element*, which is defined as a top-level element. Independent elements can be a named element or one of the SOAP defined elements with an `id` used to access the element. The second type of element is called an *embedded element*, which are all other elements not at the top level. At this point the need to distinguish formally between independent and embedded elements may not be obvious, but this will make more sense as we discuss the encoding and serialization rules. As mentioned earlier, the encoding rules define elements and

10 Building Web Services and .NET Applications

datatypes designed to support serialization. The next step is to examine some of these datatypes and describe how the SOAP elements are defined.

Common Attributes

One of the first definitions in the SOAP encoding schema is an attribute group named `commonAttributes`, shown in Listing 7-1. This group is used to add common attributes to all the elements defined in the schema. These common attributes are used to provide multireference access to elements in a SOAP message. An element can be defined with an `id` and then referenced from other elements. In addition, elements can reference other elements in the same schema or elements defined in an external source, using the `href` attribute. Finally, by adding the `anyAttribute` definition, it is valid to include attributes from other namespaces to SOAP defined elements without violating the schema.

Listing 7-1 Common Attributes Defined

```
L 7-3 <attributeGroup name='commonAttributes'>
  <attribute name='id' type='ID' />
  <attribute name='href' type='uriReference' />
  <anyAttribute namespace='##other' />
</attributeGroup>

<element name='string' type='tns:string' />
<complexType name='string' base='string' content='textOnly'>
  <attributeGroup ref='tns:commonAttributes' />
</complexType>
```

Listing 7-1 also shows the element definition for a string type that includes the `commonAttributes` group in the type definition. The result is a SOAP string type that can be identified using the `id` attribute. This same element could also reference another element by using the `href` attribute. One rule when using the `href` attribute is the element shouldn't contain data, but no schema definitions enforce this rule. Finally, other attributes can be added to this element, which is done with arrays and the `SOAP-ENC:position` attribute, discussed later in this chapter.

The string type shown in Listing 7-1 is only one example of a definition that includes common attributes. All the simple types found in the W3C schema specification—along with the majority of SOAP datatypes—have also been defined as elements with a `commonAttributes` group. Understanding how these

Chapter 7: Simple Object Access Protocol (SOAP) 11

attributes are used is important in understanding the encoding and serialization rules. Listing 7-2 shows several XML snippets that use the SOAP defined elements and common attributes. The first line in Listing 7-2 defines the namespace SOAP-ENC, which references the SOAP schema. Throughout the rest of this chapter, we assume SOAP-ENC is always defined this way. The next block of code shows schema definitions for three different elements. And, finally, the last two blocks of XML data show different ways to include the data in a SOAP message.

Listing 7-2 *Using Common Attributes*

```
L 7-4 xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"

<element name="EmployeeFirstName" type="SOAP-ENC:string" />
<element name="LastName" type="SOAP-ENC:string" />
<element name="EmployeeLastName" type="SOAP-ENC:string" />

<SOAP-ENC:string id="FirstName">John</SOAP-ENC:string>
<EmployeeFirstName href="#FirstName" />

<LastName id="LastName">Doe</LastName>
<EmployeeLastName href="#LastName" />
```

The next-to-last group of XML data in Listing 7-2 shows the use of a SOAP defined element as an independent element with an `id` of "FirstName". As mentioned previously, an element in a SOAP message can be named or it can use the predefined SOAP elements based on SOAP datatypes. When using one of the SOAP elements, you also need to use the `id` attribute to access the element when processing the message. Remember, the SOAP string datatype, shown in Listing 7-1, was defined with the `commonAttributes` attribute group. The second element, `EmployeeFirstName`, is defined as a SOAP string that uses the `href` attribute to reference the first element.

The last group of XML data in Listing 7-2 shows the use of a named element defined as a SOAP string. This element is also assigned an `id` of `LastName`. The second element, `EmployeeLastName`, was also defined as a SOAP string that uses the `href` attribute to access the `LastName` element. This implementation isn't much different from the first implementation. The only difference is this: `LastName` was defined as part of the message's schema, while `SOAP-ENC:string` is defined as part of the SOAP schema.

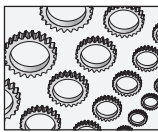
12 Building Web Services and .NET Applications

Compound Datatypes

The SOAP encoding schema also defines two compound types for handling structured data named Struct and Array. The *Struct* datatype is used to handle data where the values are accessed by name. This also means the accessor names must be unique within a Struct. The *Array* datatype is used to handle data where the values are accessed by ordinal position within a sequence. The Array datatype also supports the capability to handle partial arrays and sparse arrays. In addition, the SOAP specifications support generic structured data similar to Structs and Arrays, but they don't use the SOAP compound types as a base.

SOAP-ENC:Struct

Listing 7-3 shows the definition of a Struct from the SOAP encoding schema. The first line defines an element named Struct based on the Struct complexType, which is defined in the same schema. The next block of XML data is a model group used to define a sequence of elements that can have any name. The last block of XML data is used to define the actual Struct datatype. This definition includes the Struct group with a maxOccurs of one, along with the commonAttributes attribute group that's included in most SOAP datatypes. The only constraint for a Struct datatype is it can contain multiple elements, but each element name must be unique.



NOTE

The use of tns as a namespace for types in the code samples represents the target Namespace of the schema being defined. For example, the use of "tns:Employee" refers to a datatype named Employee defined in the same schema.

Listing 7-3 SOAP-ENC:Struct

```
L 7-5 <element name='Struct' type='tns:Struct' />
      <group name='Struct'>
        <any minOccurs='0' maxOccurs='*' />
      </group>
      <complexType name='Struct'>
        <group ref='Struct' minOccurs='0' maxOccurs='1' />
        <attributeGroup ref='tns:commonAttributes' />
      </complexType>
```

Listing 7-4 shows two different schema definitions that can be used to describe an Employee structure with two elements. Both of these definitions represent the same

Chapter 7: Simple Object Access Protocol (SOAP) 13

encoding rules, however, the first one is much more condensed. By using the SOAP-ENC:Struct datatype, no need exists to include the sequence declaration and reference attributes. These definitions also support the capability to define structures with an `id` attribute, which can then be included in other structures with `href` attributes. The last definition in Listing 7-4 shows a generic XML structure that contains an element of type `Employee` and another element of type `SOAP-ENC:Struct`.

Listing 7-4 *Defining Structures*

```
L 7-6 <element name='Employee' type='tns:Employee' />
      <complexType name='Employee' base='SOAP-ENC:Struct'>
        <element name='FirstName' type='string' />
        <element name='LastName' type='string' />
      </complexType>

      <element name='Employee' base='tns:Employee'>
        <complexType name='Employee'>
          <sequence minOccurs='0' maxOccurs='1'>
            <element name='FirstName' type='string' />
            <element name='LastName' type='string' />
          </sequence>
          <attribute name='href' type='uriReference' />
          <attribute name='id' type='ID' />
          <anyAttribute namespace='##other' />
        </complexType>

      <complexType name='EmployeeDetails'>
        <element name='Employee' type='tns:Employee' />
        <element name='Manager' type='SOAP-ENC:Struct' />
      </complexType>
```

Now that you have a good handle on how structures should be defined, let's put it to use. Listing 7-5 shows three blocks of data that could be found in a SOAP message. The first block is an independent element that uses the `SOAP-ENC:Struct` element with an `id` of `Manager-1`. The second block of data is another independent element that contains `EmployeeDetails` information and conforms to the schema definition in Listing 7-4. Because the `Employee` element is based on the `Employee` type, it also contains the `First` and `Last` name elements. The `Manager` element was defined as a `SOAP-ENC:Struct` and it uses the `href` attribute to reference the first block, which contains the actual manager data. The last block of data is another `EmployeeDetails`

14 Building Web Services and .NET Applications

block with different employee information, but the same manager. By taking advantage of references in SOAP messages, the amount of duplicate data sent between peers can be minimized.

Listing 7-5 Using Structured Data

```
L 7-7 <SOAP-ENC:Struct id="Manager-1">
    <FirstName>John</FirstName>
    <LastName>Hancock</LastName>
    <Title>Project Manager</Title>
</SOAP-ENC:Struct>

<e:EmployeeDetails xmlns:e="Employee-URI" >
    <Employee>
        <FirstName>Jane</FirstName>
        <LastName>Doe</LastName>
    </Employee>
    <Manager href="#Manager-1" />
</e:EmployeeDetails>

<e:EmployeeDetails xmlns:e="Employee-URI" >
    <Employee>
        <FirstName>Jack</FirstName>
        <LastName>Quick</LastName>
    </Employee>
    <Manager href="#Manager-1" />
</e:EmployeeDetails>
```

SOAP-ENC:Array

Once we begin moving from Structs to Arrays, things become more complicated. The Array datatype is designed to handle large amounts of data. It wouldn't make sense to use a Struct for multiple rows of data returned from a database. Instead, arrays of Structs or arrays of simple datatypes are more suited to handle these types of datasets. In addition, because arrays can handle large datasets, they need to provide flexible ways to manage that data. As with anything that must be flexible or generic, the complexity of implementation gets more difficult as flexibility is increased.

The Array datatype definition shown in Listing 7-6 is much more complex than the Struct definition shown in Listing 7-3. The first block of data shows a datatype definition used for array position attributes. The second set of data defines several attributes and an attribute group named *arrayAttributes* that uses these attributes.

Chapter 7: Simple Object Access Protocol (SOAP) 15

This attribute group is included in the Array datatype definition, which makes the `arrayType` attribute mandatory for Arrays. The `arrayType` attribute is used to define the type and size of an array, while the `offset` attribute is designed to support the transmission of partial arrays. The third block of data defines an attribute and an attribute group used to identify array elements by position and to support the capability to transmit sparse arrays. The `arrayMemberAttributes` group isn't actually added to any datatype definitions. Instead, the attribute is used when needed, which is supported by the `anyAttributes` definition found in the `commonAttributes` group. The final block of data is the actual SOAP-ENC:Array definition, which is similar to the Struct definition with the addition of the `arrayAttributes` attribute group.

Listing 7-6 SOAP-ENC:Array

```
L 7-8 <simpleType name='arrayCoordinate' base='string' />

<attribute name='arrayType' type='string' />
<attribute name='offset' type='tns:arrayCoordinate' />
<attributeGroup name='arrayAttributes'>
  <attribute ref='tns:arrayType' minOccurs='1' />
  <attribute ref='tns:offset' />
</attributeGroup>

<attribute name='position' type='tns:arrayCoordinate' />
<attributeGroup name='arrayMemberAttributes'>
  <attribute ref='tns:position' />
</attributeGroup>

<element name='Array' type='tns:Array' />
<group name='Array'>
  <any minOccurs='0' maxOccurs='*' />
</group>
<complexType name='Array' content='elementOnly'>
  <group ref='Array' minOccurs='0' maxOccurs='1' />
  <attributeGroup ref='tns:arrayAttributes' />
  <attributeGroup ref='tns:commonAttributes' />
</complexType>
```

As you can see, the Array datatype provides support to handle sets of data, while the Struct datatype is designed for individual structures of data. Although this isn't a goal, one reason for defining a structure like this is to provide the capability to transmit

16 Building Web Services and .NET Applications

subsets of data. Most presentation components or applications have a limit on the amount of data they can handle. If you tried sending a data set with 10,000 records to a browser, the performance would be bad and some browser versions would crash with this much data. Typically, data sets that contain rows of data from a table are structured like an array. By using the Array datatype, it's possible to send only a block of 100 rows, which represents a partial array. In addition, you could send a few selected rows spread throughout a data set, which represents a sparse array. Finally, the SOAP specification also allows for multidimensional arrays and arrays within arrays.

Listing 7-7 shows several schema definitions that use the SOAP Array datatype. As you can see, the definitions are similar to the Struct type, but the implementation is different. The first definition shows a single-dimension array that contains Employee elements. The second is also a single-dimension array, but this one is defined as allowing any element name and type within the array. The last definition is a multidimensional array that contains the two arrays defined previously. Remember what we said about flexibility adding complexity: the real complexity with the Array datatype comes in the variety of different ways arrays can be implemented.

Listing 7-7 Defining Arrays

```
L 7-9 <element name='Employees' type='tns:Employees' />
<complexType name='Employees' base='SOAP-ENC:Array' >
  <element name='Employee' type='tns:Employee' />
</complexType>

<element name='Projects' type='tns:Projects' />
<complexType name='Projects' base='SOAP-ENC:Array' >
  <any type='ur-type' />
</complexType>

<complexType name='CompanyInfo' base='SOAP-ENC:Array' >
  <element name='Employees' type='tns:Employees' />
  <element name='Projects' type='tns:Projects' />
</complexType>
```

Listing 7-8 shows several different ways to describe single-dimension arrays within SOAP messages. The first example uses the Array datatype defined in the SOAP-encoding schema. The `arrayType` attribute is used to identify the type and number of elements in the array with the number located inside square brackets. For example, the first line in Listing 7-8 shows `arrayType='xsd:string[2]'`,

Chapter 7: Simple Object Access Protocol (SOAP) 17

which means the array type is `xsd:string` and has two elements. Also possible is to use an empty array size value, also known as the *asize value*, when the number of elements is unknown. The element names are arbitrary and can be anything, which is consistent with the Array type declaration in Listing 7-6.

Listing 7-8 *Single-Dimension Arrays*

```
L 7-10 <SOAP-ENC:Array SOAP-ENC:arrayType='xsd:string[2]' >
      <Item>StringOne</Item>
      <Item>StringTwo</Item>
    </SOAP-ENC:Array>

    <SOAP-ENC:Array SOAP-ENC:arrayType='xsd:ur-type[3]' >
      <Item xsi:type='xsd:int'>123</Item>
      <Item xsi:type='xsd:long'>654321</Item>
      <Item xsi:type='xsd:String'>StringData</Item>
    </SOAP-ENC:Array>

    <Projects id='Projects-1' SOAP-ENC:arrayType='xsd:ur-type[2]' >
      <xsd:int>123</xsd:int>
      <SOAP-ENC:String'>StringData</SOAP-ENC:String>
    </SOAP-ENC:Array>

    <Employees id='Employees-1' SOAP-ENC:arrayType='tns:Employee[2]' >
      <Employee>
        <FirstName>Jane</FirstName>
        <LastName>Doe</LastName>
      </Employee>
      <Employee>
        <FirstName>Jack</FirstName>
        <LastName>Quick</LastName>
      </Employee>
    </Employees>
```

The second and third examples in Listing 7-8 show the use of an `xsd:ur-type`, which can represent any datatype. When the `arrayType` is defined as an `xsd:ur-type`, the elements must identify the type of data they contain by using `xsi:type` definitions or the actual type definition as the element name. The `xsd:ur-type` examples in Listing 7-8 show both ways that are valid for defining the element type. The second `xsd:ur-type` example also uses the `Projects` element defined in Listing 7-7.

18 Building Web Services and .NET Applications

The final example in Listing 7-8 shows an array of Employees also defined in Listing 7-7. Note, the `arrayType` is still required to indicate the type of elements included in the array. This means the original definition of Employees could have left out the element definition; however, by defining the element in the schema, the Employees array only supports Employee elements.

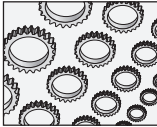
The next section of XML code, shown in Listing 7-9, represents two different types of multidimensional arrays. The first is a two-dimensional array of strings: the first size attribute represents rows and the second represents columns. The specification also allows for multiple dimensions and arrays within arrays. For example, the `arrayType` could have been defined as `'xsd:string[2,3,2]'`, which would indicate each column had two items.

Listing 7-9 Multidimension Arrays

```
L 7-11 <SOAP-ENC:Array SOAP-ENC:arrayType='xsd:string[2,3] '>
    <Item>Row1Col1</Item>
    <Item>Row1Col2</Item>
    <Item>Row1Col3</Item>
    <Item>Row2Col1</Item>
    <Item>Row2Col2</Item>
    <Item>Row2Col3</Item>
</SOAP-ENC:Array>

<CompanyInfo SOAP-ENC:arrayType='xsd:ur-type[ ][2] '>
    <Employees href='#Employees-1' />
    <Projects href='#Projects-1' />
</CompanyInfo>
```

The second block of XML data in Listing 7-9 shows an array that contains multiple arrays. This example uses the Employees and Projects elements populated in Listing 7-8 and defined in Listing 7-7. When an array is contained within a second array, its dimensions are listed first. In this example, the size of both Employees and Projects are unknown, which is indicated by the use of an empty setting. If the inner arrays were also multidimensional, the definition might look something like `'xsd:ur-type[,][2]'`, which identifies them as multidimensional arrays of unknown size. The example in Listing 7-9 also uses an array type of `xsd:ur-type`, which is a super type that includes all other types. If the innermost elements were all defined as the same simple type, such as an int or string, then that type would be used in the `arrayType`.

**NOTE**

SOAP Array indexes are zero-based, which means the first element in an array of five elements would be at position zero and the last element would be at position four.

The last set of XML data, shown in Listing 7-10, represents partially transmitted and sparse arrays. The first block of data defines an array of six strings with an offset value of four. This indicates only the data starting at position four in the array will be transmitted. Because arrays are zero-based, position four represents the second to last, or the fifth element out of six. As a result, only the last two elements are transmitted.

Listing 7-10 *Partially Transmitted and Sparse Arrays*

```
L 7-12 <SOAP-ENC:Array SOAP-ENC:arrayType='xsd:string[6]'  
      SOAP-ENC:offset='[4] '>  
  <Item>String Five</Item>  
  <Item>String Six</Item>  
</SOAP-ENC:Array>  
  
<SOAP-ENC:Array SOAP-ENC:arrayType='xsd:string[,][6] '>  
  <SOAP-ENC:Array href='#Array-5' SOAP-ENC:position='[4] '/>  
</SOAP-ENC:Array>  
  
<SOAP-ENC:Array id='Array-5' SOAP-ENC:arrayType='xsd:string[8,8] '>  
  <Item SOAP-ENC:position='[1,5] '>Row2Col6</Item>  
  <Item SOAP-ENC:position='[2,6] '>Row3Col7</Item>  
  <Item SOAP-ENC:position='[3,7] '>Row4Col8</Item>  
</SOAP-ENC:Array>
```

The last two blocks of data in Listing 7-10 represent sparse arrays. The first is a multidimensional array that contains multidimensional arrays of strings. In this example, only one of the outermost elements are included, which is the fifth element at array position four. This element contains a reference to the multidimensional array of strings that actually contains the data. The inner array could also have been added by using an `arrayType` instead of an `href` and inserting the subelements directly under the outermost element.

The innermost element in our multidimensional array is defined as another multidimensional array of strings, which is the last block of data in Listing 7-10. In this example, we only transmit three of the elements from the array. The `position` attribute uses the same syntax as dimension size attributes to define the index of a specific element in the array.

20 Building Web Services and .NET Applications

Serialization Rules

We're back to serialization again, and, we hope by now it makes sense why these weren't discussed before encoding rules and the SOAP datatypes. To be honest, some of the serialization rules were already discussed, specifically when arrays were covered. In addition, before looking into the encoding rules, we discussed what serialization is and how it works. With the encoding rules, we saw how different datatypes are defined and referenced within SOAP. The serialization rules further define how SOAP data must be structured when using the SOAP-encoding rules.

The following list is a summary of the serialization rules found in the SOAP 1.1 specifications:

- ▶ All values are represented as element content.
- ▶ Multireference values must be defined as independent elements. Single-reference values can be independent or embedded.
- ▶ All elements that contain values must have a type definition by using either `xmlns:type` attributes, schema-defined attributes, or elements. The datatype can also be defined by reference with array definitions.
- ▶ Multireference values must be identified with an `id` attribute defined as an ID type and accessible through the use of `href` attributes defined as uri-reference types. If an `href` attribute is used, then the element isn't expected to have content.
- ▶ Two types of data exist: simple and compound. *Simple datatypes* are defined as elements that don't contain subelements. *Compound datatypes* are defined as elements that contain subelements.
- ▶ Arrays can be multidimensional and they can contain single-reference or multireference values.
- ▶ All arrays must define an `arrayType` attribute, which has both type and size components, with the following type and dimension information.
 - ▶ The type must be the same as the element type or a supertype of all elements in the array. If the array is multidimensional, then the type must represent the innermost element.
 - ▶ The size component of the `arrayType` attribute should contain the sizes of all arrays, starting with the outermost array on the left.

Chapter 7: Simple Object Access Protocol (SOAP) 21

- ▶ If an array contains other arrays, then the contained array's size attribute should be included in the arrayType definition to the left of the outer array's size attribute. For example, an array of strings may define the following arrayType: 'SOAP-ENC:arrayType='xsd:string[6]'. However, if the array contained arrays of strings, the definition would be different: 'SOAP-ENC:arrayType='xsd:string[][6]'.
 - ▶ If a size isn't defined, then the size is unknown. However, size is still determined by examining the data.
- ▶ SOAP arrays can use an `offset` attribute to transmit partial arrays.
- ▶ Array member elements can use a `position` attribute to include specific elements in a transmission.

As you can see, these rules are based on the use of the datatypes and constructs defined by the SOAP encoding rules. By using these rules along with the SOAP encoding rules, interfaces are much more available to a larger audience. Remember this about the SOAP specifications we touched on at the beginning of this chapter: each of the specification parts are orthogonal to each other, which means implementations aren't required to use SOAP encoding and serialization rules to be compliant with SOAP specifications. Once again, however, following standards makes more sense than inventing your own set of rules.

One possible reason for developing your own set of rules would be for internal business applications with unique data or transport constraints. For instance, large data sets perform badly when the structure is element-based instead of attribute-based. A company that handles a lot of large data sets internally could define an encoding scheme that defines attribute accessors instead of element accessors. The only drawback is the SOAP servers that process these messages must know how to handle the datatypes. This isn't difficult to do. XML is easy to use, but the consumers of the SOAP interface would be limited.

If you're familiar with the SOAP specifications, you might have notice we haven't discussed the actual structure of SOAP messages in detail yet. We did cover the message structure at a high level, but left the details for later. The specifications start with the structure of SOAP messages, and then get into the encoding rules. The only problem is when you first read through the message specifications a lot doesn't make sense. However, we've now gone through the majority of encoding and serialization rules found in the specification. Now, it's finally time to look at the details of how SOAP messages are actually structured.

22 Building Web Services and .NET Applications

SOAP Messages

The interoffice message analogy used at the beginning of this chapter is a close representation of SOAP messages. Each SOAP message is contained in an element named *Envelope*. Inside the envelope, the message can include an element named *Header*, which carries additional information about the message. The message data is found in an element named *Body*, which, if included, follows the *Header*. The only thing that doesn't fit the analogy is an element named *Fault*. This element is added to the *Body* element if an error occurs when processing a SOAP request.

We've already gone through an in-depth look at the rules and structures associated with the schema for handling SOAP data. In this section, we examine the schema for handling SOAP messages. This schema can be considered an *Envelope* schema that uses the following namespace definition:

```
L 7-13 xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'
```

Similar to the namespace for SOAP Encoding rules, the URL in this namespace can be used to access the actual schema definition. Before getting into the *Envelope*, *Header*, and *Body* definitions, several global definitions are in the schema that we should consider.

Specifically, three global attributes, shown in Listing 7-11, are defined in the SOAP-ENV namespace. Both the `mustUnderstand` and `actor` attributes are meant to be used only by elements within the *Header*. The specification defines them at a global level, so they can be used by elements within the *Header* element. The *Header* element itself doesn't reference these attributes. We discuss their use in more detail when we discuss the *Header* element.

Listing 7-11 Envelope Schema Globals

```
L 7-14 <attribute name='mustUnderstand' default='0'>
  <simpleType base='boolean'>
    <pattern value='0|1' />
  </simpleType>
</attribute>

<attribute name='actor' type='uri-reference' />

<simpleType name='encodingStyle' base='uri-reference' derivedBy='list' />
<attributeGroup name='encodingStyle'>
  <attribute name='encodingStyle' type='tns:encodingStyle' />
</attributeGroup>
```

Chapter 7: Simple Object Access Protocol (SOAP) 23

The `encodingStyle` attribute, shown in Listing 7-11, is used to define the encoding rules that should be used. This attribute can be used on any element and applies to all children of that element unless a child defines a new `encodingStyle`. Normally, this would be set to the SOAP encoding namespace as the following shows

```
L 7-15 SOAP-ENV:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'
```

Earlier, we discussed the possibility that a company might want to define its own set of encoding rules. If a company has its own encoding rules, then the `encodingStyle` attribute should reference that set of rules.

The `encodingStyle` attribute isn't meant to define schema locations, which seems a little confusing when all the samples use URLs. Instead, this is treated as a reference name the SOAP server can use to determine how to process the message. In addition, the SOAP specifications don't define a default set of rules, which means a SOAP engine parsing the message would use its own default rules, which would normally be the SOAP encoding rules. This attribute can also contain multiple URI references and it can be defined with an empty value. If the attribute is defined as an empty value, then all encoding style claims from preceding definitions are turned off.

Throughout all the previous descriptions we used message specific schemas to define the message data. But we haven't discussed how that schema data is actually accessed to validate the structures. At first, the `encodingStyle` attribute looked like a good candidate for defining schema location. But, as discussed previously, this is only used as a reference name. The truth is, a variety of ways exist in which schema data can be accessed.

The most common method for accessing schema data is to use a URL in the namespace definition that references the actual schema location. The SOAP specifications also support the W3C Schema specifications and the use of an `xsi:schemaLocation` attribute, which is described in Chapter 3. In all cases, the schema specification states the validating parser should first look for an embedded schema or one that's loaded in a schema repository before attempting to access external resources. The following guidelines describe how the SOAP parser should look for the schema:

1. If the schema information is embedded in the XML data, then use that.
2. Attempt to locate the schema, either in the document or in a local repository using the `schemaLocation` URI.
3. Attempt to locate the schema, either in the document or in a local repository using the namespace URI.
4. Attempt to locate the schema in an external resource using the namespace URI.

24 Building Web Services and .NET Applications

5. Attempt to locate the schema in an external resource using the schemaLocation URI.
6. The last ditch effort is to attempt to locate a schema using only the namespace name.

The previous list also represents the order of priority to be used when looking for a schema. With Web Services, discussed in Chapter 8, the schema data is typically located in a local schema repository. One important aspect to note is this: if a schema cannot be located locally, the namespace URI takes precedence over the schemaLocation URI when attempting to locate an external resource.

SOAP Envelope

Listing 7-12 shows the schema definition of a SOAP Envelope. As you can see, this isn't at all complex. References exist to the Header and Body elements, along with support to handle any additional elements or attributes. Important to note is the Header element has a minOccurs of zero, while the Body element has a minOccurs of one. In other words, the Header element is optional, but the Body element is required. One thing not defined in the schema, which is a SOAP rule, is any additional attributes or elements must be namespace-qualified. Other rules undefined in the schema are these: the Header element must be an immediate child of the Envelope if present, and the Body element must immediately follow the Header. If a Header element isn't used, then the Body element must be the immediate child.

Listing 7-12 SOAP Envelope

```
L 7-16 <element name="Envelope" type="tns:Envelope"/>
<complexType name='Envelope'>
  <element ref='tns:Header' minOccurs='0'/>
  <element ref='tns:Body' minOccurs='1'/>
  <any minOccurs='0' maxOccurs='*'/>
  <anyAttribute/>
</complexType>

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'>
  ...
</SOAP-ENV:Envelope>
```

Chapter 7: Simple Object Access Protocol (SOAP) 25

Another important rule to remember is the SOAP Envelope must be the top-level element in the XML document that contains the message. In addition, the XML document must not contain Document Type Definitions (DTDs) or processing instructions. Other than this, SOAP messages follow standard guidelines for XML documents, with the addition of XML Schema and XML Linking Language support.

Listing 7-12 also has an XML snippet that shows the definition of a SOAP Envelope element. As mentioned previously, the namespace is defined as SOAP-ENV and uses a URI that references the schema. This is also the only method currently available for versioning the Envelope schema. When new schemas are defined, then this namespace URI will be updated with a different reference. This new reference represents a new version and is a standard way of versioning schemas. The W3C schema specification already has several different versions: the 1999 version, <http://www.w3.org/1999/XMLSchema>, and the 2001 version, <http://www.w3.org/2001/XMLSchema>, are examples. The only change is in the year, but they represent two different versions.

SOAP Header

The *SOAP Header* is used to hold additional information about a message without affecting the message. As mentioned previously, if the Header is included, it must be an immediate child of the SOAP Envelope element. Listing 7-13 shows the schema definition of a Header element. As you can see, no specific elements or attributes are defined for it. The definition does support the capability to add any element or attribute, however. And, as with the Envelope, the SOAP rules specify that any elements or attributes added to this element must be namespace qualified.

Listing 7-13 SOAP Header

```
L7-17 <element name="Header" type="tns:Header" />
      <complexType name='Header'>
        <any minOccurs='0' maxOccurs='*' />
        <anyAttribute/>
      </complexType>
```

As you can guess, the SOAP Header can be used in many different ways. The following is a short list of some different uses:

- ▶ The SOAP Header can contain authentication information, such as a user name and password or a session key.

26 Building Web Services and .NET Applications

- ▶ It can contain transaction information, such as a transaction key used to group database operations.
- ▶ Different sections of the SOAP message can be identified for a specific server.
- ▶ It can be used to implement a multicast transmission, where the header contains information about the next receiver of the message.
- ▶ It can contain state information carried between requests.

These are only a few different implementations that give you an idea of how powerful headers are.

When a SOAP server encounters a header in the SOAP message, it's required to look at the Header and determine if any actions are required and what those actions are. The Header itself doesn't contain data; instead, the header information is placed in subelements called Header elements. The SOAP header may also contain multiple Header elements. To help a server determine if it should process a Header element, the two global attributes—`actor` and `mustUnderstand`—are used. These attributes are only valid on Header elements, which are immediate children of the Header.

actor Attribute

At the beginning of this chapter, we said a message travels along a message path from a sender to a receiver. This path could include multiple intermediate servers that handle a message along the way. The `actor` attribute is used on a Header element to indicate who should handle the information. If a server determines it should handle a Header element, then it must remove that element from the SOAP message. The server can add a new Header element before passing it on to the next server.

If a Header element doesn't contain an `actor` attribute, this indicates the final recipient of the message is the actor. However, this doesn't mean the final recipient must process or understand the Header element.

mustUnderstand Attribute

This attribute is used on Header elements to indicate if the receiver of that element must understand the element. As shown in Listing 7-11, this is a Boolean attribute that can only contain a value of 0 or 1. If the value is 0, this indicates the recipient isn't required to understand the Header element. If the value is 1, the receiver is required to understand the message.

If the `mustUnderstand` attribute isn't used, the default value is 0. In other words, if the attribute isn't used, then the receiver isn't required to understand the Header element.

SOAP Body

This is where the actual message data is added to a SOAP message and, like the Header, the Body element doesn't contain any data itself. Instead the Body element can contain multiple child elements, which are called *Body entries*. Each Body entry must be encoded as an independent element with `id` and `href` attributes. The Body entry elements must also be namespace qualified, and they can contain an `encodingRules` attribute. The Body entries don't contain `actor` and `mustUnderstand` attributes like the Header element. Instead, all Body entries are meant for the final receiver of the message. In addition, the final receiver must be able to understand and process the message.

Although no direct relationship exists between Header elements and Body elements, a Body element is equivalent to a Header element with the default `actor` and a `mustUnderstand` value of 1. In other words, if a Header element doesn't define an `actor` and defines a `mustUnderstand` value of 1, then it must be processed with the Body entries. This could be used to force authentication or transactions associated with the Body entries.

RPC Method Call

Listing 7-14 shows a SOAP message that represents an RPC method named "GetDetails," which contains a single parameter named "EmployeeID." This is a valid SOAP message and is used in the sample program discussed at the end of this chapter. As we discussed, the top element is named *Envelope* and is namespace qualified with the SOAP-ENV namespace. The Envelope element also contains the `encodingStyle` attribute, which is then applied to all child elements.

Listing 7-14 SOAP Body with Method Call

```
L 7-18 <SOAP-ENV:Envelope
SOAP-ENV:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'
  xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'>
  <SOAP-ENV:Body>
    <rda:GetDetails xmlns:rda='urn:rdacustomsoftware-com:Employee'>
      <EmployeeID>14</EmployeeID>
    </rda:GetDetails>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

28 Building Web Services and .NET Applications

The first immediate child element of the Envelope in Listing 7-14 is the SOAP Body. The Body contains one entry named “GetDetails,” which is namespace qualified with the rda namespace. This is the RPC method element described previously. In addition, the GetDetails element contains the one RPC parameter element previously described. This message represents a method call sent to SOAP server using an HTTP request.

RPC Method Response

The result of the GetDetails call is shown in Listing 7-15. This example is an actual response from the SOAP server in the sample application for this chapter. The response uses more of the SOAP attributes than we’ve discussed in this chapter. For example, the Envelope defines both SOAP namespaces and the xsi namespace used by W3C schema specifications. The response entry in the Body uses the xsi namespace and schemaLocation attribute to identify the location of the rda schema. The Projects element is defined as a SOAP:Array in the schema and this example shows the use of the arrayType attribute.

Listing 7-15 SOAP Body with Method Response

```
L 7-19 <SOAP-ENV:Envelope
      SOAP-ENV:encodingStyle=http://schemas.xmlsoap.org/soap/encoding/
      xmlns:xsi=http://www.w3.org/1999/XMLSchema-instance
      xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/
      xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/" >
  <SOAP-ENV:Body>
    <rda:GetDetailsResponse
      xsi:schemaLocation="urn:rdacustomsoftware-com:Employee
      http://localhost/Chapter7Web/Employee.xsd"
      xmlns:rda="urn:rdacustomsoftware-com:Employee" >
      <Employee>
        <EmployeeID>11</EmployeeID>
        <LastName>Barley</LastName>
        <FirstName>Mike</FirstName>
        <MiddleInitial>C</MiddleInitial>
        <OfficePhone>4105550485</OfficePhone>
        <CellPhone/>
        <Beeper>4105550839</Beeper>
      </Employee>
      <Manager>
        <LastName>Arcaro</LastName>
        <FirstName>Kim</FirstName>
```

Chapter 7: Simple Object Access Protocol (SOAP) 29

```
<Title>Office Manager</Title>
</Manager>
<Projects SOAP-ENC:arrayType="rda:Project[2]" >
  <Project>
    <ProjectID>3</ProjectID>
    <Name>November Banking</Name>
  </Project>
  <Project>
    <ProjectID>5</ProjectID>
    <Name>Zulu Market</Name>
  </Project>
</Projects>
</rda:GetDetailsResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

One other point about the example in Listing 7-15 is this: the response contains a result set with mixed data structures, which represents the power of RPC over SOAP or, more precisely, the power of XML. By using XML, we can send structured data across loosely coupled binary interfaces without the dependency problems caused by tight coupling. However, this loose coupling also makes sending incorrect data structures easy on a request or response. Web Services, discussed in Chapter 8, implements tighter coupling through service contracts. But application errors can still occur in any method call. As a result, the SOAP specifications also include an element meant to handle error information named *SOAP Fault*.

SOAP Fault

As previously mentioned, this element is designed to handle errors in SOAP messages. There must only be one Fault element in a SOAP message and it must be contained in the SOAP Body. In addition, if a SOAP Fault has been added, then the Body shouldn't contain any other entries. Listing 7-16 shows the schema definition for a SOAP fault found in the Envelope schema. The schema defines four elements to contain the error information: *faultcode*, *faultstring*, *faultactor*, and *detail*. Both the *faultactor* and *detail* elements are optional.

Listing 7-16 SOAP Fault Schema

```
L 7-20 <complexType name='Fault' final='extension'>
      <element name='faultcode' type='qname' />
      <element name='faultstring' type='string' />
```

30 Building Web Services and .NET Applications

```
<element name='faultactor' type='uri-reference' minOccurs='0' />
<element name='detail' type='tns:detail' minOccurs='0' />
</complexType>

<complexType name='detail'>
  <any minOccurs='0' maxOccurs='*' />
  <anyAttribute />
</complexType>
```

Both the faultcode and faultstring are required elements and must be used to describe the error. In addition, the faultcode must be a namespace qualified value that can be used for processing. The fault string should describe the error and shouldn't be used for processing. Listing 7-17 shows an example of a Fault response in the sample program. This is generated by the *SoapMessageObject*, which is included in the Microsoft SOAP Toolkit version 2.0.

Listing 7-17 SOAP Fault Example

```
L 7-21 <SOAP-ENV:Envelope
      xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" >
  <SOAP-ENV:Body>
    <SOAP-ENV:Fault>
      <faultcode>SOAP-ENV:Client</faultcode>
      <faultstring>Invalid Method Name: GetFault</faultstring>
      <detail />
    </SOAP-ENV:Fault>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The SOAP specifications have also defined four fault codes to be used with SOAP fault entries. The specification also states these fault codes are required when describing faults defined in the specification. As you see, these codes are generic and are meant to be extended. The way you extend the code is by adding a more specific code separated from the generic code using (.)dot notation. For example, the sample in Listing 7-17 shows a generic code of SOAP-ENV:Client. A more specific code is SOAP-ENV:Client.Employee. Table 7-1 lists the four fault codes that are defined.

As mentioned previously, the faultactor and detail elements are both optional. In addition, they're meant for two different sections of a SOAP message. The faultactor is intended for errors with handling a Header element and indicates the actor that

Chapter 7: Simple Object Access Protocol (SOAP) 31

VersionMismatch	This is used if an invalid namespace was used.
MustUnderstand	This is returned if the target server is unable to process a header element marked with a mustUnderstand of 1.
Client	This is an indication the client sent invalid information. This could be a badly formed request or constraints from a header causing the server not to accept the message.
Server	This indicates an error with processing the message. This could be a system error or anything else that keeps the server from completing the request.

Table 7-1 SOAP Fault Codes

encountered the fault. Normally, a fault in the body wouldn't include this attribute, but it could be set to the receiver of the body. The detail element is meant to handle detailed information from an error with body entries. The detail element also works in a similar manner as Header and Body elements. It only contains subelements, which are called detail elements and must be included if an error is in the body. The SOAP specification also states that the detail elements must not contain any information about Header errors.

Putting SOAP to Use

We've gone through the SOAP specifications in detail, and now it's time to see how SOAP is really used. Right now, many different vendors and developers are producing SOAP applications. One of the biggest must be the .NET initiative from Microsoft. Although buried deep under the covers, the Web Service and Remoting methods used in .NET take advantage of the SOAP protocol. These interfaces can still use other types of transport protocols. However, SOAP is the default protocol for Web Services at all times, as well as for remote calls—called Remoting in .NET—when the calls cross machine and process boundaries.

Chapter 8 looks at Web Services and discusses more of the .NET implementation in detail. At this point, we focused on SOAP outside the context of Web Services and .NET. The reason for understanding SOAP at this level is many applications and platforms will need to integrate with Web Services and .NET using only the SOAP message protocol. In reality, a big mix of technologies is going to be in use over the foreseeable future. SOAP makes it easy for all of those technologies to work together. For example: An HTML page on a Web site running on NT 4.0 using older technologies might need to use data from a Web Service running on a .NET

32 Building Web Services and .NET Applications

platform. In contrast, a .NET application may need to access information from an older system running on an IIS 4.0 platform. In addition, either of these platforms could interact with a Web Service running on a Java platform. All of this is possible today, and we have developed an application that demonstrates how it can be accomplished.

Sample Application

The sample application can be downloaded from the following location: [TODO - add URL for sample applications]. This contains an HTML page that uses XML data islands to interact with an ASP page, which uses objects from the Microsoft SOAP toolkit version 2.0. The example also takes advantage of XML for SQL Server and uses XML templates to invoke queries against the database. XML template queries are explained in more detail in Chapter 11. Interesting to note, however, is the queries also use XSL templates, described in Chapter 3, to convert the query into a SOAP response.

Once you download the sample application, it should be expanded into a separate directory while maintaining the directory structure. A backed-up SQL Server database named Employee.bak is included; it should be restored to an existing SQL Server installation. This is the same database used in other examples, so restoring this database a second time isn't necessary if that has already been done. The HTML page is designed to be stand-alone and can run from any location with an IE 5.0 browser installed. The ASP page and XML templates must be running on a system that has IIS 5.0, SQL Server 2000 and the SOAP toolkit from Microsoft. Detailed information about setup and configuration is in the readme file.

The application itself is an Employee directory that can be used to return detailed information about each employee. The top of the page contains buttons to get the list of employees, get details, and get a Fault element. The bottom of the page contains the actual SOAP request and SOAP response messages that are sent back and forth. These messages were used to generate some of the sample code shown earlier.

When using the application, the employee list must be retrieved first before attempting to get details. The GetDetails call is also invoked if a user's name is double-clicked in the list displayed. As mentioned before, the HTML page can be run from any location. The only modification is to change the URLs in the JavaScript used to access server resources from "localhost" to the name of the server that contains the Web site. By working with this sample and looking at the code examples, you should get a good understanding of how SOAP works.

Summary

This chapter provided detailed information on the SOAP message protocol, along with examples on using SOAP. We began with a high-level view of SOAP messages, and then jumped into the SOAP encoding and serialization rules. While discussing serialization, we also started at a high level, and then provided more detail after getting a good understanding of the encoding rules. Once the low-level rules were discussed, we moved on to details of the actual SOAP message. We also discussed the use of SOAP in various implementations and provided a sample application that uses SOAP in HTML pages and ASP.

The following lists covers the main topics discussed:

- ▶ SOAP Message overview with a discussion of HTTP binding and RPC over SOAP
- ▶ SOAP Encoding rules and datatype definitions
- ▶ In-depth look at SOAP schemas that define the encoding rules
- ▶ SOAP Serialization rules along with an explanation of how serialization works
- ▶ Examples that only use the SOAP message protocol

