Summary of Contents

3

SOAP Basics

With Web Services, we are on the verge of a new programming model. A set of standards has been developed that gives us programmatic access to the application logic of the web. This application logic is accessible to clients on every platform, and in every programming language. Using this model, we can build applications that integrate components using standard Internet protocols. As has already been touched upon in Chapter 1, at the core of the Web Services model is **SOAP** (**Simple Object Access Protocol**), the protocol that allows messages to be transmitted as XML documents and invokes the capabilities of Web Services. The SOAP standard is the key to Web Services.

This chapter delves into SOAP 1.1, and the concepts needed to start using SOAP in applications. We will cover the fundamentals of SOAP and its design, and then we will drill down into the details of SOAP messages, transports, and conventions.

Note that this chapter, and the majority of the rest of the book will focus on SOAP version 1.1, because this is the newest final version, which has support available for it, so is currently the relevant version to learn about.

This chapter will not cover SOAP 1.2, because at the time of writing, it is currently a Working Draft on the W3C, and therefore prone to significant change. SOAP 1.2 is briefly discussed in Chapter 1.

To track the progress of SOAP 1.2, go to the W3C SOAP 1.2 Working Draft document at http://www.w3.org/TR/soap12/, and visit the XML Protocol working group main page at http://www.w3.org/2000/xp/.

SOAP Fundamentals

SOAP is a specification for using XML documents as messages. The SOAP Specification contains:

- A syntax for defining messages as XML documents, which we refer to as SOAP messages
- A model for exchanging SOAP messages
- A set of rules for representing data within SOAP messages, known as SOAP encoding (or **section 5 encoding** due to the section of the specification it appears in)
- A guideline for transporting SOAP messages over HTTP
- A convention for performing remote procedure calls (RPC) using SOAP messages

SOAP and Web Services

With all the buzz and acronyms surrounding the topic of Web Services, it can get a little confusing. The list of protocols and technologies related to Web Services grows everyday. Of all the Web Services acronyms, SOAP is probably the most important. It is rapidly becoming the standard protocol for accessing a Web Service, and accessing the service is key. For Web Services to work as a technology, there must be well-defined approaches for discovering a service (UDDI – Universal Description Discovery and Integration) and determining its capabilities (WSDL – Web Service Definition Language). For any individual Web Service to succeed, however, these technologies are optional: written documentation or even a conversation over coffee can define the location of a service and its methods. However, without a protocol to access the methods, the service is useless. SOAP is the best choice today for that protocol.

Although SOAP is a great choice for a Web Services messaging protocol, it is not the only choice. Web Services can simply operate on HTTP GET, or only expose functionality through XML-RPC. This does not make these components any less of a Web Service than a component that works with SOAP. Generally, however, SOAP is the messaging protocol of choice for Web Services. There is widespread acceptance of SOAP both by vendors and independent developers, and the tools and implementations that work with SOAP are improving all the time.

The first version of the SOAP Specification that was available to the public was released in 1999, and it was a result of collaboration between developers at Microsoft, DevelopMentor, and UserLand Software. The current version, SOAP 1.1, was released on May 8th 2000 as a Note by the W3C with additional contributions from IBM and Lotus. Since then more than twenty different implementations have been started covering a wide variety of languages and platforms.

For a complete list of SOAP implementations, go to http://www.soapware.org/directory/4/implementations. *Here you will be able to find a SOAP implementation that fits your needs, or if there isn't one yet, you will find the resources to help you build it.*

The SOAP Message Exchange Model

The SOAP specification defines a model for exchanging messages. It relies on three basic concepts: messages are XML documents, they travel from a sender to a receiver, and receivers can be chained together. Working with just these three concepts, it is possible to build sophisticated systems that rely on SOAP.

XML Documents As Messages

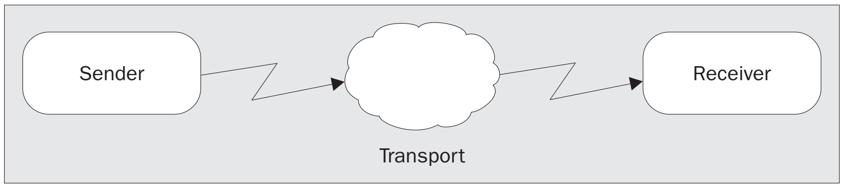
The most fundamental concept of the SOAP model is the use of XML documents as messages. SOAP messages are XML. This provides several advantages over other messaging protocols. XML messages can be composed and read by a developer with a text editor, so it makes the debugging process much more simple than that of a complex binary protocol. As XML has achieved such widespread acceptance, there are tools to help us work with XML on most platforms.

We won't examine a SOAP message in detail until later in the chapter, but here is an example of one:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
               soap:encodingStyle="http://schemas.xmlsoap.org/soap
                                   /encoding/">
   <soap:Body>
      <w:Greeting xmlns:w="http://www.wrox.com/helloworld/">
         <w:message>Hello world!</w:message>
      </w:Greeting>
   </soap:Body>
</soap:Envelope>
```

Senders and Receivers

When SOAP messages are exchanged, there are two parties involved: a sender, and a receiver. The message moves from the sender to the receiver. This operation is the basic building block of SOAP message exchanges, the smallest unit of work. The figure below illustrates this simple operation:



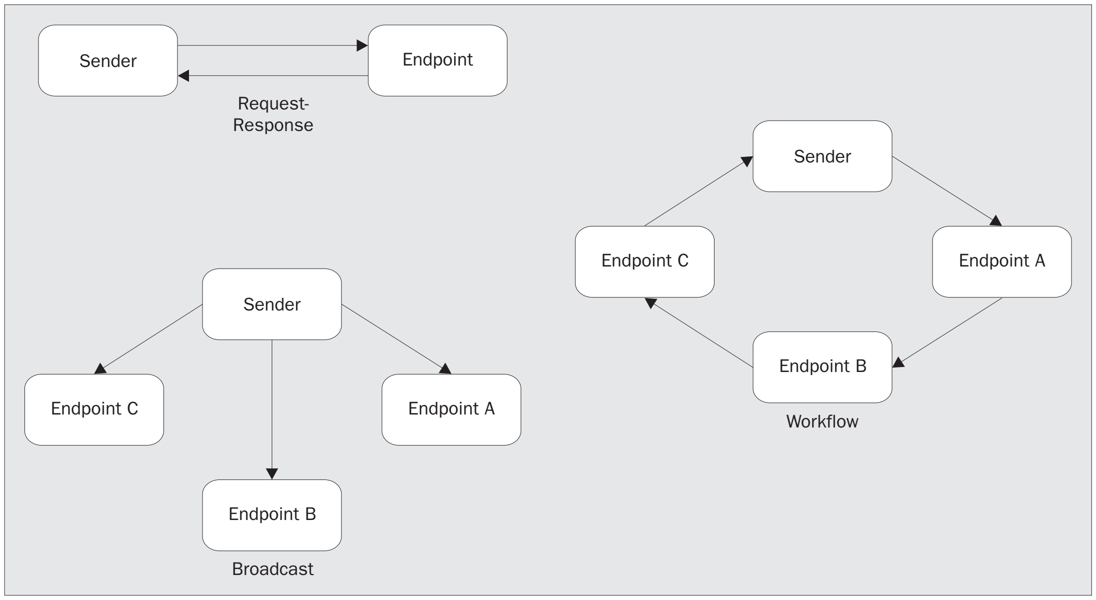
In many cases, however, this type of operation is not enough. A more common requirement would be for messages to be exchanged in request-response pairs. As we will see later in the chapter, this is the method SOAP uses with the HTTP transport and/or the RPC convention. Requiring that model, however, would make it difficult to design one-way message exchanges. By starting with the most basic operation, a one-way message exchange from sender to receiver, more complicated exchanges can be composed without preventing the simplest exchanges from occurring. This gives us the ability to construct message chains.

Message Chains

SOAP messages do not have to follow a traditional client-server model. Messages might be exchanged in this manner, as in the case of HTTP, or a chain of logical entities might process the messages. This concept of a logical entity that performs some processing of a SOAP message is referred to as an **endpoint**. Endpoints are receivers of SOAP messages. It is the responsibility of an endpoint to examine a message and remove the part that was addressed to that endpoint for processing.

It is worth mentioning here that despite the "O" in SOAP, there is nothing object-oriented about the SOAP model. Endpoints, as well as clients, can be written in any language, and there is no presumption of "objects" existing on either end of the wire.

As the model allows us to combine one-way messages into more complex operations, endpoints can function as both receiver and sender. This capability allows for a processing chain to be created, with messages being routed through the chain with some potential processing occurring at each step. Endpoints that function as both sender and receiver, passing messages that they receive on to another endpoint, are referred to as **intermediaries**. Intermediaries and the message chain concept allow developers the opportunity to construct sophisticated systems based on SOAP. The figures below show some examples of message patterns that can be achieved through chaining endpoints together:



Endpoint Behavior

Thinking of SOAP in terms of endpoints helps to understand the flexibility of SOAP messaging. No matter what route a message takes, or how many endpoints may process it, all endpoints must handle messages in a certain way. Here are the three steps that an endpoint must take to conform to the specification:

- Examine the SOAP message to see if it contains any information addressed to this endpoint
- Determine which of the parts addressed to this endpoint are mandatory, if any. If the endpoint can handle those mandatory parts, process the message. If not, reject it
- If this endpoint is an intermediary, then remove all the parts identified in the first step before sending the message to the next endpoint

We will revisit these steps later in the chapter. By conforming to these three requirements, endpoints can be chained together to form complex systems.

Modular Design

SOAP is open and extensible. That means that all of the following scenarios are acceptable and allowed by the SOAP specification:

- A desktop application composes a SOAP message that requests stock quotes and sends it as the body of an HTTP POST. A web server receives the POST, processes the message, and returns a SOAP message in the HTTP response
- A server process composes a SOAP message that describes a system event and broadcasts the message over named pipes to other servers on the network
- A software developer with limited social skills decides to compose a SOAP message declaring his love for a co-worker and sends it as an e-mail attachment (this is likely to be a one-way message)

How is it that SOAP can support such different scenarios with the same model? The answer is in SOAP's modular design. Throughout the specification, there are placeholders left open for future extensibility of the protocol. SOAP is designed to be extensible in all of the following areas:

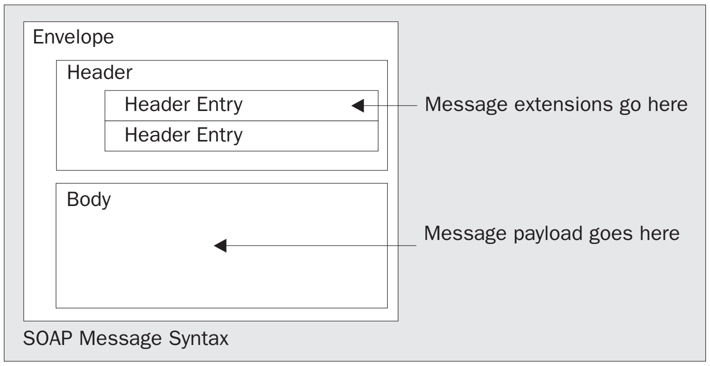
- **Message Syntax** – the SOAP message format does have an area set aside for extensions to be added (we will examine this area, the `Header` element, in the next section)
- **Data** – the SOAP payload can contain any type of data. SOAP provides one method for serializing data, but applications can define their own rules as well
- **Transport** – SOAP does not dictate how messages will be transported during the exchange. SOAP defines how messages should be exchanged over HTTP, but any communications protocol or method can be substituted for HTTP
- **Purpose** – SOAP does not define what you want to put into a message. Although this may sound like we are counting data twice, there is a difference between data and purpose, as we will see later in the chapter

Extensibility is important, but without some concrete implementations of these concepts, SOAP would be just a lot of interesting concepts. Luckily, the authors of SOAP provided a description of one implementation each of: data, transport, and purpose. For data, the Specification provides the SOAP encoding rules (section 5 encoding). For Transport, a transport binding for HTTP is defined. Finally, for purpose, the specification defines a convention for using SOAP messages for RPC.

We will cover each of these topics in more detail in this chapter. It is important to remember these four concepts, because the fact that they are separate from the protocol and therefore extensible is one of the biggest advantages of SOAP.

SOAP Messages

Now that we have covered SOAP at a high level, let's examine the most important detail of SOAP: the structure of a message. First and foremost, SOAP uses XML syntax for messages. The structure of a SOAP message is shown overleaf:



The diagram shows how a SOAP message can be broken down into components, and we will cover each of these in detail. A SOAP message contains a payload, the application-specific information. Here is an example of a SOAP message as an actual XML document:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
               soap:encodingStyle="http://schemas.xmlsoap.org/soap/
                                   encoding/">
   <soap:Header>
      <h:from xmlns:h="http://www.wrox.com/Header">SoapGuy@wrox.com</h:from>
   </soap:Header>
   <soap:Body>
      <w:GetSecretIdentity xmlns:w="http://www.wrox.com/heroes/">
         <w:codename>XSLT-Man</w:codename>
      </w:GetSecretIdentity>
   </soap:Body>
</soap:Envelope>
```

Before we go into the contents of the SOAP message, let's take a quick glance at the XML of the message. As can be seen, SOAP messages rely heavily on XML Namespaces. All of the elements in this document are prefixed with a namespace, and there is a good reason why the SOAP specification uses namespaces so extensively. In order for a SOAP message to carry any arbitrary XML payload, all the elements of the message must be scoped in some fashion to avoid conflicts in the names of elements.

The Namespaces in XML Recommendation can be found at http://www.w3.org/TR/REC-xml-names/.

The namespace prefix `soap` is used on most of the elements in the above message. In this example, the prefix is associated with the namespace URI `http://schemas.xmlsoap.org/soap/envelope/`, and it identifies the elements that are part of a standard SOAP message. Like all namespace prefixes, the choice of `soap` is irrelevant. The namespace prefix could have been something else entirely, as in this message:

```
<blah:Envelope xmlns:blah ="http://schemas.xmlsoap.org/soap/envelope/"
               blah:encodingStyle="http://schemas.xmlsoap.org/
                                   soap/encoding/">
   <blah:Header>
      <h:from xmlns:h="http://www.wrox.com/Header">SoapGuy@wrox.com</h:from>
   </blah:Header>
   <blah:Body>
      <w:GetSecretIdentity xmlns:w="http://www.wrox.com/heroes/">
```

```
            <w:codename>XSLT-Man</w:codename>
         </w:GetSecretIdentity>
      </blah:Body>
   </blah:Envelope>
```

The namespace prefix could also be eliminated completely if the namespace is the default namespace for the document. The default namespace is assigned using just the `xmlns` attribute, as shown here:

```
   <Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <Header>
         <h:from xmlns:h="http://www.wrox.com/Header">SoapGuy@wrox.com</h:from>
      </Header>
      <Body>
         <w:GetSecretIdentity xmlns:w="http://www.wrox.com/heroes/">
            <w:codename>XSLT-Man</w:codename>
         </w:GetSecretIdentity>
      </Body>
   </Envelope>
```

All three of these messages are acceptable and equivalent. For the sake of readability, it is better to use the `soap` namespace prefix for elements.

All of the elements in the message that are associated with the `soap` namespace are standard elements of a SOAP message, as are the attributes. Any other elements are either related to message extensions or the message payload. There are three standard SOAP elements that appear in this sample message: the `Envelope`, the `Body`, and the `Header`. There is also one other standard element that does not appear in this example message, the `Fault` element, which we will discuss later in this chapter.

Envelope

The `Envelope` element, as its name would suggest, serves as a container for the other elements of the SOAP message. As it is the top element, the `Envelope` is the message. The example below shows the same message we saw earlier, but this time, the `Envelope` element has been highlighted to stress its position in the message.

```
   <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
                  soap:encodingStyle="http://schemas.xmlsoap.org/soap/
                                          encoding/">
      <soap:Header>
         <h:from xmlns:h="http://www.wrox.com/Header">SoapGuy@wrox.com</h:from>
      </soap:Header>
      <soap:Body>
         <w:GetSecretIdentity xmlns:w="http://www.wrox.com/heroes/">
            <w:codename>XSLT-Man</w:codename>
         </w:GetSecretIdentity>
      </soap:Body>
   </soap:Envelope>
```

Envelope Namespace

SOAP messages indicate their version by the namespace of the `Envelope` element. The only version recognized by the 1.1 Note is the URI `"http://schemas.xmlsoap.org/soap/envelope/"`. Messages that do not use this namespace are invalid, and endpoints that receive messages with another namespace must return a "fault". We will discuss `Fault` elements later in this section.

The use of the Envelope namespace to indicate message versions is a good example of how much the SOAP specification relies on XML Namespaces. Without XML Namespaces, it would be extremely difficult to define an open XML format for messages that did not result in name conflicts with the payload XML of the message.

encodingStyle attribute

The specification defines an attribute called `encodingStyle` that can be used to describe how data will be represented in the message. Encoding is the method used to represent data. The `encodingStyle` attribute can appear on any element in the message, but in the case of SOAP encoding, it often appears on the `Envelope` element. We will discuss the `encodingStyle` attribute and encoding in general in more detail later in the chapter.

Body

The `Body` element of a SOAP message is the location for application-specific data. It contains the payload of the message, carrying the data that represents the purpose of the message. It could be a remote procedure call, a purchase order, a stylesheet, or any XML that needs to be exchanged using a message. The `Body` element is highlighted in the message below:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
               soap:encodingStyle="http://schemas.xmlsoap.org/soap/
                                   encoding/">
   <soap:Header>
      <h:from xmlns:h="http://www.wrox.com/Header">SoapGuy@wrox.com</h:from>
   </soap:Header>
   <soap:Body>
      <w:GetSecretIdentity xmlns:w="http://www.wrox.com/heroes/">
         <w:codename>XSLT-Man</w:codename>
      </w:GetSecretIdentity>
   </soap:Body>
</soap:Envelope>
```

The `Body` element must appear as an immediate child of the `Envelope` element. If there is no `Header` element, then the `Body` element is the first child; if a `Header` element does appear in the message, then the `Body` element immediately follows it. The payload of the message is represented as child elements of `Body`, and is serialized according to the chosen convention and encoding. Most of this chapter deals with the contents of the `Body` and how to build payloads.

Header

The SOAP message structure of Envelope and Body elements is an open one that maps well to many messaging scenarios. The Body element encapsulates the payload for the message, but in some instances, the payload data is not enough. Perhaps a message is part of a set of messages that must be treated as a single logical transaction, or the message should be executed on a persistent object that resides at the server. Issues like transactions and object references are vital to the message, but are separate from the payload.

It is unrealistic to think that we can predict every type of extension that will be needed by a SOAP message. So, in a wise design choice, the authors created the Header element. The purpose of the Header element is to encapsulate extensions to the message format without having to couple them to the payload or to modify the fundamental structure of SOAP. This allows extensions like transactions, encryption, object references, billing, and countless others to be added over time without breaking the specification. The text below illustrates our original example message with an additional Header entry. The entire Header element is highlighted in the example below.

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
               soap:encodingStyle="http://schemas.xmlsoap.org/soap
                                    /encoding/">
   <soap:Header>
      <h:from xmlns:h="http://www.wrox.com/Header">SoapGuy@wrox.com</h:from>
      <h:report xmlns:h="http://www.wrox.com/Header">1</h:report>
   </soap:Header>
   <soap:Body>
      <w:GetSecretIdentity xmlns:w="http://www.wrox.com/heroes/">
         <w:codename>XSLT-Man</w:codename>
      </w:GetSecretIdentity>
   </soap:Body>
</soap:Envelope>
```

In this example, the header entry h:from could be used to send a report via e-mail to the address that is indicated. By agreeing upon a set of extensions, a sender and receiver can build additional capabilities into a message exchange without requiring additional features from SOAP.

As this chapter is being written, there is still little detail out about SOAP 1.2, the very likely successor to SOAP 1.1. However, what information is available suggests that the XMLP working group will be leveraging the extensibility of the Header element to build additional capabilities on top of SOAP 1.1. If SOAP 1.2 develops in this manner, it should help early adopters of SOAP significantly.

Just like the Body element, the Header element must appear as an immediate child of the Envelope element. It is optional, but if it does appear, it must appear as the first child. The Header element contains one or more child elements known as entries. Header entries can be used to add any type of extension to a message, and by default, an endpoint will ignore the extension unless it can understand it. This allows the extensions to be developed over time without breaking existing endpoints. Some extensions have additional requirements, however, that are dealt with using the mustUnderstand and actor attributes.

actor attribute

As the SOAP model allows for endpoints to be chained together, it is necessary to identify what parts of a message are meant for what endpoint on the chain. In the case of the payload, it is not necessary to do this: the final endpoint on the chain is the target of the payload. However, in the case of Header elements, the issue of addressing becomes important.

The actor attribute can be used to address a Header element to a particular endpoint. The value of the actor attribute is a URI that identifies the endpoint the Header element entry is targeted for. SOAP attaches special significance to two values of the actor attribute to address issues of message chaining. If the value is "http://schemas.xmlsoap.org/soap/actor/next", then the entry is targeted for the first endpoint that finds it. Omitting the actor attribute indicates that the entry is intended for the final endpoint, the same endpoint that will process the payload.

The issue of the actor attribute becomes important when it comes to intermediaries and the Header element. There has been quite a bit of debate on the behavior of intermediaries towards Header elements. In the end, the consensus is that well-behaved intermediaries know whether or not they are the last endpoint in the chain, and if they are not, they should not modify Header elements that have no actor attribute. In addition, intermediaries must remove the Header elements they process.

mustUnderstand attribute

In the example of a Header element that represents a transaction, developers will not be able to accept the endpoint ignoring the extension. If a message is part of a transaction, it must be part of a transaction, and endpoints that cannot support transactions should not try to process a message. This is where the mustUnderstand attribute is useful.

The mustUnderstand attribute can be used anywhere in a SOAP message, but it commonly appears on a Header element. The value of the attribute is either a 1, indicating that the element is mandatory, or a 0, indicating that it is optional. The absence of the attribute is equivalent to the 0 value.

Let's take another look at the example message, this time with a mandatory Header element that is addressed to the final endpoint.

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
               soap:encodingStyle="http://schemas.xmlsoap.org/soap
                                    /encoding/">
   <soap:Header>
      <h:from xmlns:h="http://www.wrox.com/Header" soap:mustUnderstand="1">
         SoapGuy@wrox.com
      </h:from>
   </soap:Header>
   <soap:Body>
      <w:GetSecretIdentity xmlns:w="http://www.wrox.com/heroes/">
         <w:codename>XSLT-Man</w:codename>
      </w:GetSecretIdentity>
   </soap:Body>
</soap:Envelope>
```

Fault

Everything we have discussed about the SOAP message format so far covers how to build good clean messages that are successfully sent to and processed by the receiver every time. Of course, that is not a realistic view of how a real application will behave. Just as SOAP messages have a specified location and format for versioning, encoding style, payload, and extensions, they also have a location and format for errors. The element in a SOAP message that represents an error is the Fault element. You can think of the Fault element as exceptions for Web Services, a standard way to throw back a report on unexpected behavior to the originator of the message.

Faults are typically associated with a response message. Although the specification does not rule out `Fault` elements in requests, do not expect existing server implementations to behave well in the face of such requests!

If the `Fault` element appears, it must be in the payload of the SOAP message, which means that it must appear as a child element of the `Body`.

The example message below is a response that contains a `Fault element`.

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
               soap:encodingStyle="http://schemas.xmlsoap.org/soap/
                                    encoding/">
   <soap:Body>
      <soap:Fault>
         <faultcode>soap:MustUnderstand</faultcode>
         <faultstring>Mandatory Header error.</faultstring>
         <faultactor>http://www.wrox.com/heroes/endpoint.asp</faultactor>
         <detail>
            <w:source xmlns:w="http://www.wrox.com/">
               <module>endpoint.asp</module>
               <line>203</line>
            </w:source>
         </detail>
      </soap:Fault>
   </soap:Body>
</soap:Envelope>
```

faultcode element

The `faultcode` element contains a value that identifies the error condition to the application. This means this value is for machine use and is not intended for display to potential users. The `faultcode` element value must be a qualified name, as if it were an element in the message itself. In the above example, the `faultcode` element's value is `soap:MustUnderstand`, indicating that the `MustUnderstand` fault is a SOAP standard fault. This allows us to define our own values for the `faultcode` element and identify them by their namespace.

The following standard `faultcode` element values are defined in the SOAP 1.1 specification:

- **VersionMismatch** – this value indicates that the namespace of the SOAP `Envelope` element was not `http://schemas.xmlsoap.org/soap/envelope/`. Currently that value is the only acceptable version of a SOAP message, and it indicates that the message conforms to the 1.1 Note.
- **MustUnderstand** – this value is returned in a `faultcode` element when the endpoint encounters a mandatory `Header` element entry (one with a `mustUnderstand` attribute set to `1`) that it does not recognize.
- **Client** – this value should be used in the `faultcode` element when a problem is found in the message that was received. This could be anything from a missing element to an incorrect namespace in the body, but this `faultcode` element value states that the message that was received was to blame for the error.

- **Server** – in contrast to the `Client` fault code, `Server` indicates that a problem occurred during processing that was not directly related to the content of the message. An example of this type of fault would be that the database used by the endpoint to return information is down.

The standard `faultcode` element values listed here represent classes of faults rather than a single error. They are extensible in that more specific codes that fit into these classes can be defined. This is done by appending a period to the code and adding an additional name to the code. For example, if the machine the endpoint is running on were to run out of memory, the endpoint could potentially return a `Server.OutOfMemory` fault code.

faultstring element

If the `faultcode` element contains the fault information that is meant for the machine, then the `faultstring` element value is what is meant for the user. The `faultstring` element contains a string value that briefly describes the fault that occurred in a way that would make sense if it were displayed to the user in an error dialog. That is not to suggest that it could not be technical in nature.

faultactor element

It is often just as important to know where the error occurred as it is to know what error occurred. This is especially true in systems that involve SOAP intermediaries. If a message must pass through a dozen endpoints before it can reach its final destination for payload processing, the developer needs to know at what point on the message routing chain an error occurred. The `faultactor` element is a placeholder for that type of information. The `faultactor` element contains a URI that identifies the endpoint where the fault originated.

The `faultactor` element is only mandatory for intermediaries. If a fault occurs at an intermediary, then that fault must have a `faultactor` element. If the fault occurs at the final destination, then the endpoint is not required to populate that value (although it may choose to do so, and it would probably be the nice thing to do for developers who are using our endpoints). This means that a fault with no `faultactor` element can be assumed to have originated from the final endpoint.

detail element

It is possible to provide descriptive error information with just the three elements above, but additional information would be helpful, if not necessary. For instance, we might want to include in the `Fault` element the module and source code line of the error while still debugging the application. In this case, the additional error information can be included as `detail` element entries, as seen in the example below.

The specification allows us to define any `detail` element entries we choose to, but it does define one case in which the endpoint returning the `Fault` element must return information in the detail entries: when an error occurs because the server could not process the message correctly. This is an important requirement, especially in the development of SOAP, because it helps to debug problems that arise from poorly formed messages. This example shows a SOAP message that might have resulted from such a message:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
               soap:encodingStyle="http://schemas.xmlsoap.org/soap
                                   /encoding/">
   <soap:Body>
      <soap:Fault>
         <faultcode>soap:Client.MissingParameter</faultcode>
         <faultstring>A parameter was missing</faultstring>
```

```
            <faultactor>http://www.wrox.com/heroes/endpoint.asp</faultactor>
            <detail>
               <w:error xmlns:w="http://www.wrox.com/">
                  <code>178</code>
                  <desc>The codename parameter was missing.</desc>
               </w:error>
            </detail>
         </soap:Fault>
      </soap:Body>
   </soap:Envelope>
```

Endpoint Behavior Revisited

Now that we know more about the structure of a SOAP message, let's consider what is really involved in the three steps of message processing:

- Examine the SOAP message to see if it contains any information addressed to this endpoint. Examine the header for entries addressed to this endpoint, either by position (next or last endpoint) or by URI. If this endpoint is the last, look for the body as well.
- Examine the header entries targeted at this endpoint. If any are marked with `mustUnderstand="1"` and are not recognized by the endpoint, a fault code of `MustUnderstand` must be returned.
- If this endpoint is an intermediary, then remove the processed header entries before sending the message to the next endpoint. This does not apply to the body, since only the last endpoint can process that.

These steps, and other requirements of SOAP, will be transparent to most developers. The various SOAP tools and implementations will take into account these requirements when generating endpoints. Part of the beauty of SOAP, however, is the minimal requirements for endpoints. If our application meets these three, it is a valid SOAP endpoint.

Body-Conscious

The elements that make up the SOAP message structure provide a framework for a message. Working with those elements, we know where to put our data, where to extend the message, and how to report errors, among other things. Using the `Envelope`, `Header`, `Body`, and `Fault` elements, we can assemble a SOAP message to accomplish what we need. Other than the `Fault` element, which appears inside the body, we have not discussed the actual payload of the message. The rest of the chapter deals with specific uses for the body, how we can represent data in the body, and what XML we can place in the body.

Data

In order to build SOAP messages from our language of choice, we need to know how to serialize data. We need to know the rules for representing an integer, string, or floating point number in a SOAP message so that we can exchange messages freely between languages and platforms. The serialization of data inside a SOAP message is referred to as **encoding**. As this section focuses on the payload of the message, the XML in the examples represents only a SOAP message payload and not a complete message.

Encoding Style

The ability to decide on a set of rules for representing data in a message is very important to the open nature of SOAP. It doesn't do much good of course to define a set of rules if we cannot tell what encoding rules were used to serialize a particular SOAP message. The `encodingStyle` attribute defined by the SOAP specification is used to identify the encoding rules used in a particular message.

The `encodingStyle` attribute is commonly located on the `Envelope` element. It is possible, however, to use the `encodingStyle` attribute on any element in a SOAP message. Although the SOAP specification defines a set of encoding rules that map well to programming constructs, **there is no default encoding**. This means that if the `encodingStyle` attribute does not appear in the message, the receiver cannot make any assumptions about how data will be represented within the message. By the same token, the zero length `encodingStyle=""`, is equivalent to a missing `encodingStyle` attribute. In either case, no `encodingStyle` means that the implementation will have to try to figure out how to deserialize data without any help.

What About XML Schemas?

Those familiar with XML Schemas may be wondering what relationship exists between encoding and XML Schemas. Encoding can make use of XML Schemas. In the case of SOAP encoding, the URI used in the `encodingStyle` attribute points to a schema. As we will see in the next section, the SOAP encoding rules use XML Schemas heavily, relying on the XML Schema `datatypes` namespace and the `type` attribute. The key difference is that encoding does not mandate XML Schemas. Encoding rules are simply identified by a URI. The rules implied by that URI could be backed up by nothing more than a verbal agreement, or possibly some written documentation. This allows developers who do not necessarily need the capabilities of XML Schemas to forego their use and start sending messages with encoding rules based on an accepted URI.

Although we can have encoding without a corresponding schema, it's not recommended. Most XML parsers will soon be schema-aware (in fact, some like Xerces already are), and we can save ourselves a lot of trouble when parsing messages if we rely on the parser and schema to validate and convert data instead of doing it manually.

SOAP Encoding

The SOAP Specification defines a single set of encoding rules that are referred to as **SOAP encoding**. SOAP encoding is based on XML Schemas and as such it closely models many of the standard types and constructs that developers would be familiar with. The value of the `encodingStyle` attribute for SOAP encoding is `http://schemas.xmlsoap.org/soap/encoding/`, which points to the XML Schema that defines the encoding rules.

Simple Data Types

In SOAP encoding, simple types are always represented as single elements in the body. SOAP encoding exposes all the simple types that are built into the XML Schemas Specification. If we are using a simple type with SOAP encoding, then it must come from XML Schemas, or be derived from a type that does. The namespace associated with the XML Schemas data types is `http://www.w3.org/1999/XMLSchema`. This provides the common types that many programmers will expect, like: `string`, `integer`, `float`, `date`, and so on. If we assume the `xsd` prefix is associated with the XML Schemas URI, and the `soapENC` prefix is associated with the SOAP encoding URI, then both of these payload values work with strings. This refers to XML Schemas:

```
<codename xsi:type="xsd:string">Hulk</codename>
```

while this refers to SOAP encoding:

```
<codename xsi:type="soapENC:string">Hulk</codename>
```

For a XML Schemas tutorial, including a list of the types available in XML Schemas (and therefore, SOAP encoding), go to http://www.w3.org/TR/xmlschema-0/.

xsi:type attribute

SOAP tries to make it possible for a wide variety of languages to communicate, and not all languages are created equal. In many scripting languages, type is a loose concept. To help level the playing field, SOAP borrows from XML Schemas once again and uses the `xsi:type` attribute.

The `xsi:type` attribute is a way for elements in the payload to indicate their type. It is associated with XML Schemas, and the `xsi` prefix in this case is associated with the URI `http://www.w3.org/1999/XMLSchema-instance`. It can appear on any payload element.

Developers not familiar with namespaces and schemas in XML need to be aware that the `xsi` prefix of the `xsi:type` is insignificant. The attribute could easily appear in the message as `foo:type`, provided that the `foo` prefix is associated with the namespace URI `http://www.w3.org/1999/XMLSchema-instance`. Likewise, the `xsd` prefix commonly used on the values of the `type` attribute is assumed here to be associated with `http://www.w3.org/1999/XMLSchema`. Be aware of this and other namespace issues as you work with SOAP messages.

If the application knows what type is being sent and retrieved from some outside source (for example, a schema, a WSDL document, or other metadata), then the `xsi:type` attribute is not required. The fact remains that not all languages will be supporting WSDL in the near future, if ever, and so the good neighbor approach suggests that including `xsi:type` will help make our SOAP messages more interoperable with "type-challenged" languages like XSLT.

So what does this mean for the SOAP message as a whole? In order to use the `xsi:type` attribute and the `xsd` prefix for data types, we must define what these prefixes mean inside the message. Let's consider another example message with encoding in mind.

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
               soap:encodingStyle="http://schemas.xmlsoap.org/soap
                                   /encoding/">
   <soap:Body>
      <m:MixedMessage xmlns:m="http://www.wrox.com/mix/">
         <param1>OU812</param1>
         <param2>2001</param2>
         <param3>3.14159</param3>
      </m:MixedMessage>
   </soap:Body>
</soap:Envelope>
```

That message meets all the requirements of SOAP, but many implementations would not be able to process it because they would not be able to map the values in the payload to types in the target language. We don't want to require a language to use a union type like the variant in COM, or to try to map the type by trial and error. Therefore, we add a little information to our message to make it more readable:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
               soap:encodingStyle="http://schemas.xmlsoap.org/soap
                                    /encoding/"
               xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
               xmlns:xsd="http://www.w3.org/1999/XMLSchema">
   <soap:Body>
      <m:MixedMessage xmlns:m="http://www.wrox.com/mix/">
         <param1 xsi:type="xsd:string">OU812</param1>
         <param2 xsi:type="xsd:integer">2001</param2>
         <param3 xsi:type="xsd:double">3.14159</param3>
      </m:MixedMessage>
   </soap:Body>
</soap:Envelope>
```

Now all the data in the payload is identified by type, and it becomes much easier for a SOAP implementation to process.

Enumerations

SOAP encoding allows us to define enumerated types. It borrows once again from XML Schemas, which also has the concept of an enumeration. An enumeration is a named set of values, based on a basic type. For example, we could define an enumeration that represented geographical locations ("North, "South", etc). To define an enumeration, we must use XML Schemas.

Here is an example of an enumeration that defines a set of geographical regions.

```
<simpleType name="Region" base="xsd:string">
   <enumeration value="North"/>
   <enumeration value="South"/>
   <enumeration value="East"/>
   <enumeration value="West"/>
</simpleType>
```

If this enumeration appeared in a referenced schema, we could then use this type in a SOAP message just as we would any other type.

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
               soap:encodingStyle="http://schemas.xmlsoap.org/soap
                                    /encoding/"
               xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance">
   <soap:Body>
      <m:GetSalesTotals xmlns:m="http://www.wrox.com/sales/">
         <m:reg xsi:type="m:Region">East</m:reg>
      </m:GetSalesTotals>
   </soap:Body>
</soap:Envelope>
```

Binary Data

As part of the simple types it supports, SOAP and XML Schemas provide a type for representing binary data. One approach for working with binary data is to use the `base64` type. We can represent binary data, such as an image file, as an array of bytes in the message. The `base64` type converts binary data to text using the base64-encoding algorithm of XML Schemas. There is no relationship between SOAP and base64-encoding; if we use it, our application (or implementation of SOAP for your platform) must be able to understand and work with base64-encoding.

Catch All

In addition to the simple types, many languages have a "universal" data type or placeholder, something that can represent a variety of types within that language. In COM, the `variant` serves this purpose, as does the `any` type in CORBA. SOAP accounts for this possibility with the **polymorphic accessor**. If we are serializing a value in the form of a polymorphic accessor, we must provide the `type` attribute.

The polymorphic accessor is more difficult to pronounce than to use! Let's assume we are passing in a value representing a person's age, and that type could vary depending on how the information was to be used. If the value of our data is a `float`, it would appear like this:

```
<age xsi:type="xsd:float">3.5</age>
```

If is a `string`, it would appear like this:

```
<age xsi:type="xsd:string">3 and a half years old</age>
```

Both examples are legal if the `age` element has been defined as being a polymorphic accessor, meaning that its data type will vary.

What About XML?

Those frequenting the SOAP discussion lists and newsgroups will notice the recurring question: "How do I send XML in a SOAP payload?" or something to that effect. This is a general problem related to XML, but there are a couple of approaches we can use to transmit XML inside SOAP. We can:

❑ Rely on our toolkit or XML parser to properly encode the XML when we pass it in as a string parameter. If our implementation is based on RPC and does not encode the XML we pass in as a string properly, that is a bug. Let the implementation's author know.

❑ Consider why we are passing XML. If we are using SOAP RPC to pass XML as a string parameter, check to see if the implementation supports passing arbitrary XML in the payload. A good implementation should. As we will discuss later, SOAP does not have to be RPC, and if we are passing XML in string parameters, our application probably doesn't need RPC.

Compound Data Types

Sometimes, simple types are not enough. Just like the programming languages it must support, SOAP encoding provides structures for representing compound types. SOAP encoding handles two compound types: structs (records), and arrays. Complex types are serialized as payload elements, just like simple types, but they have child elements. The child elements are the fields or elements of the type. SOAP had to invent its own rules for structs and arrays because, as of this writing, XML Schemas does not pay special attention to these compound types.

Structs

Let's start with a struct (or structure, or record, whichever you prefer). Structs are easy to represent as XML because they have unique named members. Consider this C++ struct definition of a super-hero:

```
struct SuperHero
{
   string sCodename;
   string sFirstName;
   string sLastName;
   int nAge;
};

SuperHero hero = { "Hulk", "Bruce", "Banner", 32 };
```

We've chosen a simple struct to illustrate the basics of compound types. If we serialize the variable "hero" into a SOAP message payload using SOAP encoding, it would look like this:

```
<hero xsi:type="x:SuperHero">
   <sCodeName xsi:type="xsd:string">Hulk</sCodeName>
   <sFirstName xsi:type="xsd:string">Bruce</sFirstName>
   <sLastName xsi:type="xsd:string">Banner</sLastName>
   <nAge xsi:type="xsd:integer">32</nAge>
</hero>
```

As can be seen in this example, the `xsi:type` attribute is used on compound data types as well as simple types. In this case, the type is `x:SuperHero`, and the `x` namespace would point to a schema that represents our `SuperHero` struct.

Arrays

Arrays are compound types as well, and they are represented in much the same way that structs are. As we might expect, the difference between arrays and structs is in how we refer to their members. Structs have data that is identifiable by name, and array members are identified by position. The names of array elements are insignificant, so they cannot be used to look up a value.

In SOAP encoding, arrays are considered a special type. This type is indicated by their `xsi:type` attribute, which is `SOAP-ENC:Array`. As with all SOAP encoding, the namespace associated with the Array type is `http://schemas.xmlsoap.org/soap/encoding`. Elements with this `xsi:type` are declared as SOAP encoding arrays. The type of the array members is declared using another attribute, `SOAP-ENC:arrayType`. This attribute indicates the type and size of the array. Arrays in SOAP encoding can be confusing, so let's take a look at a simple array of five integers to see how these attributes are used to define an array:

```
<numbers xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:integer[5]">
   <item>10</item>
   <item>20</item>
   <item>30</item>
   <item>40</item>
   <item>50</item>
</numbers>
```

The numbers element is declared as a SOAP array, and the arrayType attribute states that it contains five elements of the integer type. This is accomplished by combining the values we used earlier in the type attribute (values from XML Schemas) and the square brackets [] with a size value. As can be seen, each of the array elements has the name item. This could have been any name as the member values are determined solely by the order of the elements.

Before we look at the more complex features of arrays, let's see another simple array. This array contains four names, each as a string. The differences occur in the arrayType attribute, and in the names of the members (which are irrelevant).

```
<names xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:string[4]">
   <e>John Doe</e>
   <e>John Q. Public</e>
   <e>John Smith</e>
   <e>John Elway</e>
</names>
```

By setting the arrayType attribute on a SOAP array, we are able to define the type of members that will appear. The arrayType attribute is the only restriction on member types; SOAP arrays do not place restrictions on member types by default, so we can mix types inside of an array. We can accomplish this by using an arrayType attribute value of SOAP-ENC:ur-type[]. The ur-type[] is a universal data type for the SOAP encoding data types, so arrays that use this can have mixed members. The only catch to using ur-type[] is that like the polymorphic accessor for simple types, we must use the xsi:type attribute on the accessors to indicate each element's type. Below is an example of a SOAP array that contains a mixed set of types as members. Notice that each member uses the xsi:type attribute to specify its type.

```
<mix xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="SOAP-ENC:ur-type[4]">
   <e xsi:type="xsd:string">John Elway</e>
   <e xsi:type="xsd:integer">7</e>
   <e xsi:type="xsd:string">Denver Broncos</e>
   <e xsi:type="xsd:date">1999</e>
</names>
```

Besides using mixed types, arrays have some other sophisticated features that we can take advantage of if our application needs them. Because arrays can be costly as parameters in remote procedure calls, SOAP defines two attributes that give us the flexibility to pass the portion of the array that we need to work with in our application. These attributes are the offset and position attributes.

The SOAP-ENC:offset attribute lets us specify where in the array we are beginning, so transmitting only part of the array. All elements before the offset are assumed to contain the default value, or NULL, depending on the application's behavior. The offset attribute appears on the array element, as shown below. In that case, the elements are the third and fourth of the array.

```
<names xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:string[4]"
                 SOAP-ENC:offset="[2]" >
   <e>John Smith</e>
   <e>John Elway</e>
</names>
```

The `SOAP-ENC:position` attribute specifies the position in the array of a particular member (like `offset`, the `position` attribute is zero based). As might be expected, that means that the position attribute must appear on the member itself rather than the array element. If the `position` attribute appears on one member, it must appear on all the members. This example shows how the `position` attribute can be used to pass a large array that is almost empty (this is referred to in the Specification as **sparse arrays**):

```
<names xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:string[100]">
   <e SOAP-ENC:position="[11]">John Smith</e>
   <e SOAP-ENC:position="[45]">John Elway</e>
</names>
```

These two attributes (`offset` and `position`) are to some extent interchangeable in that the `offset` attribute implies position for all the elements that appear. That means that it is possible to describe an array in minimal fashion using either technique.

Custom Encoding

SOAP encoding is just one example of a set of encoding rules. This is a topic that deserves a level of detail beyond the slope of this chapter. Suffice it to say that we can define our own encoding, and there are many reasons that we might want to. SOAP encoding will probably handle most needs because the rules for data representation match up well with the customary programming types.

Multi-reference Values

Whether a value is represented as a simple or compound type, it is not uncommon for the same value to appear multiple times in a single payload. Because XML is a verbose representation of data, there is an opportunity to write more efficient XML documents by eliminating redundant data. SOAP follows the lead of XML Schemas by allowing values to be referenced multiple times inside a document. SOAP uses the `id` and `href` attributes to allow values to be referenced inside of a message. This allows for redundant data to be eliminated, and for the payload to more accurately reflect the language models it represents (if we are serializing a reference, why not serialize it as a reference?).

Let's look at an example of multi-reference values in use. In this example, we have a struct that represents an employee. The employee struct contains the employee's name, identification number, and address. The address is also represented as a struct.

```
<m:Employee>
   <idno>12345</idno>
   <fname>Billy</fname>
   <lname>Batson</lname>
   <address>
      <street>1000 Sharon Drive</street>
      <city>Charlotte</city>
      <state>North Carolina</state>
      <zip>28211</zip>
   </address>
</m:Employee>
```

When we introduce a second employee record, it turns out that these two employees live at the same address. There is redundant data in the address member if both these employees appear in the message payload.

```
<m:Employee>
   <idno>12345</idno>
   <fname>Billy</fname>
   <lname>Batson</lname>
   <address>
      <street>1000 Sharon Drive</street>
      <city>Charlotte</city>
      <state>North Carolina</state>
      <zip>28211</zip>
   </address>
</m:Employee>
<m:Employee>
   <idno>23456</idno>
   <fname>Wally</fname>
   <lname>West</lname>
   <address>
      <street>1000 Sharon Drive</street>
      <city>Charlotte</city>
      <state>North Carolina</state>
      <zip>28211</zip>
   </address>
</m:Employee>
```

By using the `id` and `href` attributes, we can make the address field of these two structs a multi-reference value. The example below shows what the resulting payload would look like:

```
<m:Employee>
   <idno>12345</idno>
   <fname>Billy</fname>
   <lname>Batson</lname>
   <address href="#address1"/>
</m:Employee>
<m:Employee>
   <idno>23456</idno>
   <fname>Wally</fname>
   <lname>West</lname>
   <address href="#address1"/>
</m:Employee>
<m:address id="address1">
   <street>1000 Sharon Drive</street>
   <city>Charlotte</city>
   <state>North Carolina</state>
   <zip>28211</zip>
</m:address>
```

If we are working with a SOAP message with only two structs in the payload, multi-reference values might seem like overkill. The real advantage to using multi-reference values becomes obvious when we are working with large amounts of data, such as an array of structs. Let's assume our redundant address in this example is a high-rise apartment building one block away from our company. If we were passing an array of 100 structs in a payload, perhaps 50 or more employees might all have the same address. The reduction in message size by using multi-reference values would be significant.

For many SOAP implementations, supporting multi-reference values has come late, and some still do not support this capability. On the other hand, implementations like Microsoft's .NET Framework took the approach of using multi-reference values to represent every element of an array, whether the values appear more than once or not. Interoperability tests between SOAP implementations have made great progress in identifying these types of issues.

What's Simple About It?

At this point, many developers ask, "Whatever happened to 'Simple'?" It's fair to say that with advanced topics like multiref accessors and sparse arrays, the "Simple" part of SOAP seems like a distant memory, and it is tempting to use XML-RPC or even a home-grown solution. For SOAP to be able to function as a generic messaging protocol, it must be extensible, and this extensibility does not come without a price. This is an advantage of SOAP, not a handicap. SOAP is simple when the needs of the application allow it to be, and yet its open nature allows it to handle the complexities of more sophisticated systems. As yet, not all SOAP implementations handle the more advanced aspects of the specification. As SOAP implementations mature, complex topics like multirefs will be handled transparently for most users. Until then, be extra nice to those developers working on SOAP implementations.

For any developers out there who are working on an implementation of SOAP for their platform, we highly recommend checking out both the SOAPBuilders group at Yahoo (http://groups.yahoo.com/group/soapbuilders) *and the DevelopMentor SOAP discussion list* (http://discuss.develop.com). *They are both great sources of information on interoperability, advanced topics, and what the specification really means, as the best of the SOAP community and many of the authors of SOAP frequent them.*

Now that we have covered the details of what goes into a SOAP message, let's turn our attention to how we move a message from point A to point B. This mechanism for moving messages is called the transport.

Transports

Once we have a SOAP message, we will probably want to send it to someone. After all, what good is a message if it never goes anywhere? The transport is the method by which a SOAP message is moved from sender to receiver. One example of a transport is HTTP, the Hypertext Transfer Protocol.

Separation of Message and Transport

One of the best design decisions made by the authors of SOAP was to separate the message definition from the message transport. It may sound ridiculous, but there is nothing in the specification that requires computers be involved in the transport of a SOAP message. Given that, here is a list of possible transports for SOAP messages (some more likely than others):

- HTTP
- SMTP
- MQSeries
- Raw sockets
- Files
- Named Pipes
- Carrier Pigeon

Granted, not many developers will be exercising stock options after developing SOAP-enabled carrier pigeons, but this helps to illustrate the modular nature of the specification. Most developers are going to focus on HTTP as the standard transport for their SOAP messages, and that is the transport that we will focus on in this chapter. As SOAP support continues to grow, there will be SOAP transport bindings defined and implemented for any number of protocols.

HTTP

When many developers think of SOAP, they think of XML over HTTP. HTTP is an excellent transport for SOAP because of its wide acceptance. HTTP is the ubiquitous protocol for the Web, a constant reminder that standards can actually work. Combining HTTP, the standard transport protocol for the Web, and SOAP, the leading candidate for the standard messaging format, gives us a powerful tool. HTTP makes such a great transport for SOAP that the authors made sure that the rules for using HTTP as a transport are part of the SOAP specification.

There are only a couple of basic rules for using HTTP as a SOAP transport. The mechanism for sending a SOAP message over HTTP is the standard HTTP POST method. An HTTP POST sends a block of data to a particular URI on the web server. In the case of SOAP messages, this block of data is the SOAP message itself. Because the SOAP message is XML, the Content-Type header of the HTTP POST must be `text/xml`. If there is a response to the message, it is returned in the HTTP response.

Let's take another look at the example SOAP message we used earlier, this time transporting the message over HTTP.

```
POST /endpoint.asp HTTP/1.1
Content-Type: text/xml
Content-Length: ###
SOAPAction: "urn:wroxheroes"

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
               soap:encodingStyle="http://schemas.xmlsoap.org/soap
                                   /encoding/">
   <soap:Header>
      <h:from xmlns:h="http://www.wrox.com/Header">
         SoapGuy@wrox.com
      </h:from>
   </soap:Header>
   <soap:Body>
      <w:GetSecretIdentity xmlns:w="http://www.wrox.com/heroes/">
         <codename>XSLT-Man</codename>
      </w:GetSecretIdentity>
   </soap:Body>
</soap:Envelope>
```

The first four lines of this example are related to the HTTP transport. The first line contains the HTTP method, `POST`, and the URI indicating the location of the endpoint. This example might represent a SOAP message request sent to the URL http://www.wrox.com/endpoint.asp. The first line also indicates that version 1.1 of HTTP is being used in this request.

The next three lines are the HTTP headers. The first two are standard HTTP. `Content-Type` indicates the MIME type of the POST content. All SOAP messages must use `text/xml`. `Content-Length` tells the size in bytes of the content. The last header, `SOAPAction`, is SOAP specific. We will examine that next, but before that, notice the remainder of the HTTP request. There is an additional carriage-return/line feed that separates headers from the body, and the body content itself is a SOAP message. The message itself does not change because it is being transported by HTTP, and other than the `SOAPAction` header, HTTP does not change just because it is sending a SOAP message.

SOAPAction Header

The SOAP specification does define a single additional HTTP header to be used when SOAP messages are transported over HTTP, and that header is the `SOAPAction` header. The `SOAPAction` header provides a hint to servers that a particular HTTP POST contains a SOAP message, and the value of the header is a URI that indicates the intent of the SOAP message. This allows firewalls and other servers to perform conditional processing based on the presence of the `SOAPAction` header.

Version 1.1 stated that the `SOAPAction` header must be present in HTTP transports, although it may be blank. Since then, that requirement has been removed. A blank `SOAPAction` on an HTTP POST means that the intent of the message can be inferred from the target of the POST, the URI. Although this may sound confusing, it is actually pretty straightforward. There are some messages whose purpose is obvious by where they are going (what URL they are posted to). If our methods are implemented at one URL, we may need to use the `SOAPAction` header to make the intent of an incoming message clearer. The need for `SOAPAction` largely depends on how the endpoint is implemented.

Status Codes

You will recall that HTTP returns status information in the form of status codes. These codes are integers, and they are sectioned into classes of 100. For example, anything in the range 200-299 indicates success. SOAP places a requirement on the HTTP transport when it is used to exchange SOAP messages. If the response message contains a fault, then the status code of the HTTP response must be 500, which indicates an Internal Server Error. We will see examples of both success and failure status codes later in the chapter.

Let's take a look at the response to our `GetSecretIdentity` call (below) to see the relationship between request and response, as well as message and fault.

```
HTTP/1.1 200 OK
Content-Type: text/xml
Content-Length: ###

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
               soap:encodingStyle="http://schemas.xmlsoap.org/soap
                                   /encoding/"
               xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
               xmlns:xsd="http://www.w3.org/1999/XMLSchema">
   <soap:Body>
      <m:GetSecretIdentityResponse xmlns:m="http://www.wrox.com/heroes/">
         <return xsi:type="xsd:string">Michael Kay</return>
      </m:GetSecretIdentityResponse>
   </soap:Body>
</soap:Envelope>
```

The above response is successful, returning the identity of Wrox's resident XSLT super-hero. Notice the first line of the example response, which contains the status code 200. Now, let's call the `GetSecretIdentity` method again, but this time we will send a different request, one with problems:

```
POST /endpoint.asp HTTP/1.1
Content-Type: text/xml
Content-Length: ###
SOAPAction: "urn:wroxheroes"

<Envelope>
   <Body>
      <w:GetSecretIdentity xmlns:w="http://www.wrox.com/heroes/">
         <codename>XSLT-Man</codename>
      </w:GetSecretIdentity>
   </Body>
</Envelope>
```

In this case, the SOAP namespace is missing completely, so the endpoint must return a fault, which means it must use a status code of `500` on the response. The response message is shown below, and it contains a `VersionMismatch` fault as well as the appropriate status code:

```
HTTP/1.1 500 Internal Server Error
Content-Type: text/xml
Content-Length: ###

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
               soap:encodingStyle="http://schemas.xmlsoap.org/soap
                                   /encoding/"
               xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
               xmlns:xsd="http://www.w3.org/1999/XMLSchema">
   <soap:Body>
      <soap:Fault>
         <faultcode>soap:VersionMismatch</faultcode>
            <faultstring>The SOAP namespace is incorrect.</faultstring>
         <faultactor>http://www.wrox.com/endpoint.asp</faultactor>
         <detail>
            <w:errorinfo xmlns:w="http://www.wrox.com/">
               <desc>The SOAP namespace was blank.</desc>
            </w:errorinfo>
         </detail>
      </soap:Fault>
   </soap:Body>
</soap:Envelope>
```

These examples show how to use HTTP as the transport binding for SOAP. It is not difficult to use HTTP, which is one of the reasons for its popularity. For more information on HTTP, including the status codes, consult Chapter 2.

If we think of transport as "how" the message is sent, then the purpose of the message is the "why". HTTP is SOAP's "how" of choice for the time being, and our next topic, RPC, is the "why".

SOAP and RPC

We have spent much of this chapter trying to show that SOAP does not necessarily have to be used for remote procedure calls. The truth is that RPC is what gets most developers excited about SOAP, and with good reason. The idea that the complexities of CORBA and DCOM can be forgotten with a little XML is an attractive concept. The SOAP specification clearly describes how remote procedure calls should be represented in SOAP messages.

Saying that SOAP replaces CORBA or DCOM is an oversimplification, however. SOAP is missing most of the features that developers expect from a robust distributed object protocol, such as garbage collection or object pooling. Despite the letter "O" in SOAP, as the Specification stands today, there are no "objects" in the DCOM or CORBA sense. The SOAP Specification makes it clear that this was not a design goal of the authors. On the other hand, it is clear that SOAP was designed with RPC in mind.

SOAP RPC Convention

We already discussed the open nature of the SOAP specification as it relates to message transport. SOAP defines a message syntax and exchange model, but it does not try to define the "how" of message exchange. The **convention** is the set of rules applied to a particular use of SOAP messages. The Specification defines rules for a single convention: remote procedure calls, or RPC. The RPC convention can be defined as a way of serializing remote procedure calls and responses as SOAP messages. At the end of this chapter we will discuss the implications of using some alternative conventions.

Like its partner HTTP, SOAP RPC uses a request-response model for message exchanges. Making a remote procedure call with SOAP just involves building a SOAP message. The request SOAP message that is sent to the endpoint represents the call, and the response SOAP message represents the results of that call. Let's take a look at the rules for building method calls and returns in SOAP messages.

The Call

Making a remote procedure call with SOAP just involves building a SOAP message. The message that is sent to the endpoint represents the call. The payload of that request message contains a struct that is the serialized method call. The child elements of that struct are the inbound parameters of the method. The end result is an XML representation of a call that looks the way we would expect.

When RPC calls are serialized in a SOAP message, the name of the element must match the name of the method, as do the parameters. Consider this method signature:

```
// Return the current stock price, given the company symbol
double GetStockQuote ( [in] string sSymbol );
```

If our method namespace is `http://www.wroxstox.com/`, then the serialized method call that requests the stock quote using symbol `OU812` would look like this:

```
<q:GetStockQuote xmlns:q="http://www.wroxstox.com/">
   <q:sSymbol xsi:type="xsd:string">OU812</q:sSymbol>
</q:GetStockQuote>
```

The method name matches the element, as does the parameter. While the parameter names match the child elements, only inbound parameters appear in the serialized call. To illustrate this, let's look at three very similar methods signatures:

```
// Reverse the string, s, and return the new string.
string ReverseString ( [in] string s );

// Reverse the string, s, and return the new string.
void ReverseString ( [in] string s, [out] string sRev );

// Reverse the string, s, passed in by reference.
void ReverseString ( [in, out] string s );
```

All three methods would be represented by the same serialized call. If we called the `ReverseString()` method with the parameter `s` containing a value of `ROHT`, the call would be represented as shown (below):

```
<x:ReverseString xmlns:x="http://www.wrox.com/">
   <s xsi:type="xsd:string">ROHT</s>
</x:ReverseString>
```

Now we'll take a look at how the return from a remote procedure call is serialized, and also how the returns from the different forms of the `ReverseString()` method shown here, would look.

The Return

As we mentioned earlier, the RPC convention uses a request-response model. Just as the call is represented in the request SOAP message, the results of the call are returned in the response SOAP message. The payload in the response also contains a struct, and the child elements are the outbound parameters and/or the return value of the method.

The name of the method response struct can be anything, but it is a convention to append the word "Response" to the name of the method call struct. For example, `ReverseString` would result in `ReverseStringResponse`. Just as in the call, the names of parameter elements are significant and should match the parameters. If the method returns a value, the name is irrelevant, but it must be the first child of the method struct.

So how do we know the difference between a single out parameter and a return value? In one sense, there is no difference. Both are returning a single value as part of the method return. The real answer lies in the name of the parameter. If the name of the parameter matches a parameter of the method, then it is an out parameter. Return values cannot be identified by name, only position, but the name should not conflict with the parameters of the method.

Now that we know how a serialized return should look, let's continue to use our `ReverseString` example. As we saw before, all three method signatures for `ReverseString` result in the same serialized call. However, how do the serialized returns differ? The first version reverses the string and returns the result as the return value of the method:

```
<x:ReverseStringResponse xmlns:x="http://www.wrox.com/">
   <x:ret xsi:type="xsd:string">THOR</x:ret>
</x:ReverseString>
```

The second version has no return value, but instead uses an out parameter called `sRev`:

```
<x:ReverseStringResponse xmlns:x="http://www.wrox.com/">
   <x:sRev xsi:type="xsd:string">THOR</x:sRev>
</x:ReverseString>
```

The final version reverses the string after passing it by reference, so the parameter `s` is both an in and out parameter:

```
<x:ReverseStringResponse xmlns:x="http://www.wrox.com/">
   <x:s xsi:type="xsd:string">THOR</x:s>
</x:ReverseString>
```

As can be seen, the difference between values returned in parameters and the value of the method itself is all in the name. In the first case, the element was named `ret`, and in the second case, it was named `s`. We can access parameters by name or position as elements of the RPC struct, but the return value can only be accessed by position.

What if there are no out parameters or return values? In that case, we still have a response that represents a method return, but with no data:

```
<m:GetNothing xmlns:m="http://tempuri.org/"/>
```

Now that we know how to work with remote procedure calls, let's look at a more complete example that uses full messages and a transport.

RPC and HTTP

We have been looking at RPC in isolation, but in order to perform a remote procedure call, we need a way to move our message to the "remote" location. Here is where SOAP really shines: when we combine RPC and HTTP to make calls against Web Services.

Assume we need to call a remote procedure on a web server to validate a city and state combination and return a zip code. Our hypothetical Web Service will exist at www.livezipcodes.com (this is not a real endpoint, don't bother trying!). We do not know how the method is implemented; all we know is how to access it. The method can be invoked at the URL http://www.livezipcodes.com/call.asp, the method is associated with the namespace URI `http://www.livezipcodes.com/methods/`, and the `SOAPAction` for this method is `urn:livezipcodes`. The signature for the method is shown below:

```
string GetZipCode ( string city, string state );
```

In building the request message, we do not need any extensions, so the `Header` element can be left out. The payload will be a struct representing the method call. The method parameters are passed as child elements. Here is the HTTP request, including the request SOAP message, sent to www.livezipcodes.com.

```
POST /call.asp HTTP/1.1
Content-Type: text/xml
Content-Length: ###
SOAPAction: "urn:livezipcodes"
```

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
               soap:encodingStyle="http://schemas.xmlsoap.org/soap
                                    /encoding/"
               xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
               xmlns:xsd="http://www.w3.org/1999/XMLSchema">
   <soap:Body>
      <m:GetZipCode xmlns:m="http://www.livezipcodes.com/methods/">
         <city xsi:type="xsd:string">Modest Town</city>
         <state xsi:type="xsd:string">Virginia</state>
      </m:GetZipCode>
   </soap:Body>
</soap:Envelope>
```

Since there is actually a place named Modest Town, Virginia, the response from the endpoint would look like this.

```
HTTP/1.1 200 OK
Content-Type: text/xml
Content-Length: ###

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
               soap:encodingStyle="http://schemas.xmlsoap.org/soap
                                    /encoding/"
               xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
               xmlns:xsd="http://www.w3.org/1999/XMLSchema">
   <soap:Body>
      <m:GetZipCodeResponse xmlns:m="http://www.livezipcodes.com/methods/">
         <zip xsi:type="xsd:string">23412</zip>
      </m:GetZipCodeResponse>
   </soap:Body>
</soap:Envelope>
```

If we were to execute this same method, but the endpoint is unable to access its database of geographical information, the response would be more like this.

```
HTTP/1.1 500 Internal Server Error
Content-Type: text/xml
Content-Length: ###

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
               soap:encodingStyle="http://schemas.xmlsoap.org/soap
                                    /encoding/"
               xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
               xmlns:xsd="http://www.w3.org/1999/XMLSchema">
   <soap:Body>
      <soap:Fault>
         <faultcode>soap:Server.DatabaseDown</faultcode>
         <faultstring>The database is unavailable.</faultstring>
         <faultactor>http://www.livezipcodes.com/call.asp</faultactor>
      </soap:Fault>
   </soap:Body>
</soap:Envelope>
```

Beyond RPC - Working with XML Documents

Although RPC has certainly received the most attention, SOAP messages can be used to transfer arbitrary XML documents. For a given document type, we can define a new convention that describes the purpose of the message transfer.

Here's something we don't see everyday: a SOAP message that has nothing to do with RPC.

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
               soap:encodingStyle="http://schemas.xmlsoap.org/soap
                                   /encoding/"
               xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <soap:Body>
      <xsl:stylesheet version="1.0">
         <xsl:template match="/">
            <html>
               <body>
                  <p><xsl:value-of select="Envelope"/></p>
               </body>
            </html>
         </xsl:template>
      </xsl:stylesheet>
   </soap:Body>
</soap:Envelope>
```

This message contains a payload that is an XSLT stylesheet (one that happens to be written to manipulate a SOAP message). Consider the implications of having the SOAP `Body` element be an arbitrary XML document (in this case, an XSLT stylesheet) instead of RPC. In that sense, SOAP is a general-purpose mechanism for transporting an XML document. This is how SOAP is being applied in both Microsoft's BizTalk Server and in the ebXML protocol, but the potential uses do not stop there.

> *For more information on BizTalk, go to the BizTalk Home Page at* http://www.biztalk.org/home/default.asp*. Additional information about ebXML is provided in Chapter 1 of this book, and* http://www.ebxml.org/

For every potential application of XML, SOAP provides the mechanism for extending that use via messaging. This is an area of development that is still largely untapped because of the excitement surrounding RPC. **Web Services can be composed with SOAP but without RPC**. As more developers realize this, the excitement surrounding SOAP will continue to grow. Inserting an XSLT transform into a SOAP message gives us a mechanism to trigger an XML transformation on a remote machine, and that's a Web Service. A SOAP message that carries a Vector Markup Language (VML) document could be used to insert new graphical elements into a diagram that exists on another machine, and that's a Web Service too. The possibilities are endless!

Summary

In this chapter we have taken a look at SOAP and how it relates to Web Services. SOAP is a messaging protocol based on XML. The SOAP specification defines a modular architecture for messaging that allows any combinations of message routing, transports, and conventions to be used to build systems. SOAP can be used as a messaging protocol for Web Services, and it is the protocol of choice for most vendors because of its growing acceptance as a standard.

We examined the syntax of the SOAP message itself. SOAP messages have an `Envelope` element as the document element, which provides version information. The `Envelope` element contains a `Body` element that holds the payload of the message, and it may also contain a `Header` element. The `Header` element contains one or more entries that represent extensions to the message syntax. SOAP defines another standard element, `Fault`, which is used to carry error information inside the `Body` element.

SOAP messages contain data, and there are rules for how data types are represented in a message. SOAP defines one set of rules, called SOAP encoding, but new encoding rules can be defined. SOAP encoding relies on XML Schemas for most of its data types, and it adds structs and arrays as well.

Messages in SOAP can be transported by any mechanism, whether by socket or by hand. The specification defines a transport binding for HTTP, which does not stray far from the general mechanisms for XML transfer over HTTP. SOAP adds one twist, the `SOAPAction` header, to help servers route SOAP messages without needing to examine their contents.

The last area of SOAP we focused on is remote procedure calls, or RPC. SOAP messages can be used to perform remote procedure calls, and the Specification defines how calls and returns should be serialized in messages. RPC is just one convention for SOAP messages, and we saw how the future of SOAP might be in other XML document conventions.