CHAPTER 3

SOAP and WSDL

Web services are software components that expose their functionality to the network. To exploit that functionality, Web service consumers must be able to bind to a service and invoke its operations via its interface. To support this, we have two protocols that are the fundamental building blocks on which all else in the Web services arena is predicated: SOAP[1] and WSDL[2]. SOAP is the protocol via which Web services communicate, while WSDL is the technology that enables services to publish their interfaces to the network. In this chapter we cover both SOAP and WSDL in some depth and show how they can be used together with rudimentary tool support to form the basis of Web services-based applications.

The SOAP Model

Web services are an instance of the service-oriented architecture pattern that use SOAP as the (logical) transport mechanism for moving messages between services described by WSDL interfaces. This is a conceptually simple architecture, as shown in Figure 3-1, where SOAP messages are propagated via some underlying transport protocol between Web services.

1. In this chapter, unless otherwise explicitly stated, all references to SOAP and the SOAP Specification pertain to the SOAP 1.2 recommendation.
2. In this chapter, unless otherwise explicitly stated, all references to WSDL and the WSDL specification pertain to WSDL 1.1; see http://www.w3.org/TR/wsdl. The W3C's WSDL effort is less advanced than the latest SOAP work, though where possible we highlight new techniques from the WSDL 1.2 working drafts.

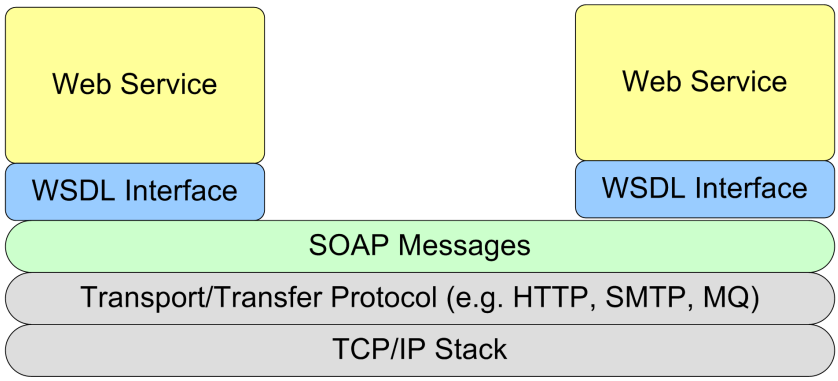


Figure 3-1 The logical Web services network.

A SOAP message is an XML document whose root element is called the *envelope*. Within the envelope, there are two child elements called the header and the body. Application payloads are carried in the body, while the information held in the header blocks usually contains data from the various Web services protocols that augment the basic SOAP infrastructure (and which is the primary subject of this book). The structure of a SOAP message is shown in Figure 3-2.

The SOAP message shown in Figure 3-2 provides the conceptual basis on which the whole SOAP model is based. Application payload travels in the body of the message and additional protocol messages travel in header blocks (which are optional, and may not be present if only application data is being transported). This permits a separation of concerns at the SOAP processing level between application-level messages and higher-level Web services protocols (e.g., transactions, security) whose payload travels in the SOAP header space.

The split between application and protocol data within SOAP messages allows the SOAP processing model to be a little more sophisticated than was suggested by the simple architecture shown in Figure 3-1. SOAP's distributed processing model outlines the fundamentals of the Web services architecture. It states (abstractly) how SOAP messages—including both the header and body elements—are processed as they are transmitted between Web services. In SOAP terms, we see that an application is comprised of *nodes* that exchange messages. The nodes are free to communicate in any manner they see fit, including any message-exchange pattern from one-way transmission through bilateral conversations. Furthermore, it is assumed in SOAP that messages may pass through any number of intermediate nodes between the sender and final recipient.

More interestingly however, the SOAP specification proposes a number of *roles* to describe the behavior of nodes under certain circumstances, which are shown in Figure 3-3. As a message progresses from node to node through a SOAP-based network, it encounters nodes that play the correct role for that message. Inside message elements, we may find role declarations that match these roles (or indeed other roles produced by third parties), and where we find a node and message part that match, the node executes its logic against the message. For example,

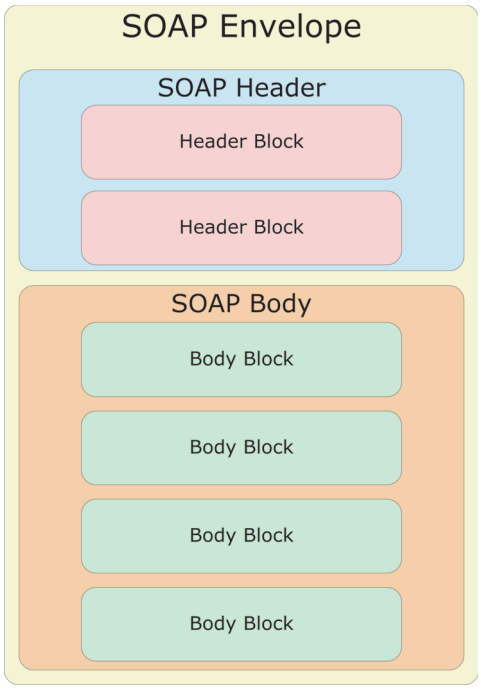


Figure 3-2 The structure of a SOAP message.

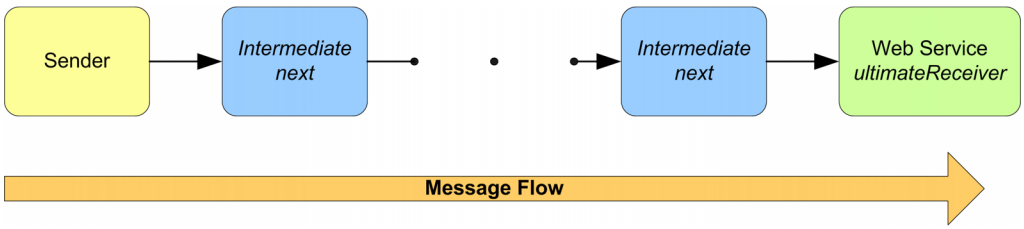


Figure 3-3 SOAP node roles.

where a node receives a message that specifies a role of `next` (and every node except the sender is always implicitly `next`), the node must perform the processing expected of that role or fault. In Figure 3-3, we see that the nodes labeled “intermediate” all play the role `next`. The Web service that finally consumes the message plays the role `ultimateReceiver`, and so each processes only the parts of the SOAP message which are (either implicitly or explicitly) marked as being for that role.

The processing model shown in Figure 3-3 is supported in software by SOAP servers. A SOAP server is a piece of middleware that mediates between SOAP traffic and application components, dealing with the message and processing model of the SOAP specification on a Web service's behalf. Therefore, to build Web services, it is important to understand how a SOAP server implements the SOAP model.

While it is impossible to cover every SOAP server platform here, we will examine the architecture of a generalized SOAP server (whose characteristics are actually derived from popular implementations such as Apache Axis and Microsoft ASP.Net) so that we have a mental model onto which we can hang various aspects of SOAP processing. An idealized view of a SOAP server is presented in Figure 3-4. This shows a generic SOAP server architecture. Inbound messages arrive via the physical network and are translated from the network protocol into the textual SOAP message. This SOAP message passes up the SOAP request stack where information stored in SOAP headers (typically context information for other Web services protocols like security, transactions and so forth) are processed by handlers that have been registered with the Web service. Such handlers are considered to be intermediate nodes in SOAP terms.

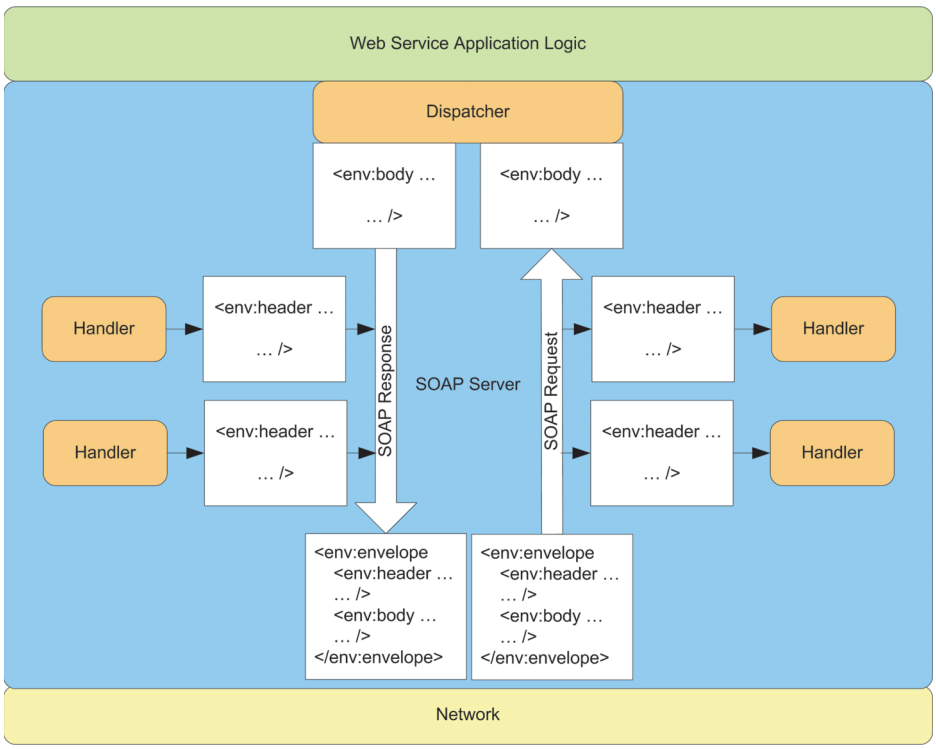


Figure 3-4 The architecture of a generalized SOAP server.

The handlers that operate on the headers are not generally part of the SOAP server by default, but are usually third-party components registered with the server to augment its capabilities. This means that SOAP servers are themselves extensible and can be upgraded to include additional protocol support over their lifetime as Web services' needs evolve.

At some later point, provided the header processing has not caused the service invocation to fail, the application payload of the message (carried in the SOAP body) reaches a dispatch mechanism where it causes some computation to occur within the back-end service implementation. The application logic then performs some computation before returning data to the dispatcher, which then propagates a SOAP message back down the SOAP response stack. Like the request stack, the response stack may have handlers registered with it which operate on the outgoing message, inserting headers into messages as they flow outward to be consumed by other Web services. Again, these handlers are considered to be nodes in SOAP terms.

Eventually, the outgoing message reaches the network level where it is marshaled into the appropriate network protocol and duly passes on to other SOAP nodes on the network, to be consumed by other SOAP nodes.

SOAP

Having understood the SOAP model and seen how this model is supported by SOAP servers, we can now begin to discuss the details of SOAP itself. SOAP is the oldest, most mature, and the single most important protocol in the Web services world. The SOAP specification defines this protocol as "[an] XML-based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined datatypes, and a convention for representing remote procedure calls and responses."[3]

In its earlier incarnations, the acronym SOAP used to stand for "Simple Object Access Protocol," though that meaning has ceased to exist in the SOAP 1.2 specification. This is undoubtedly a good thing since SOAP isn't especially simple, it's not exclusively designed for object access and it is more a packaging mechanism than a protocol per se.

In the following sections, we examine SOAP in some depth—from its basic use pattern and XML document structure, encoding schemes, RPC convention, binding SOAP messages, transport protocols, to using it as the basis for Web services communication.

3. http://www.w3.org/TR/SOAP/

SOAP Messages

We have already seen the overall structure of a SOAP message, as defined by the SOAP `Envelope`, in Figure 3-4. All SOAP messages, no matter how lengthy or complex, ultimately conform to this structure. The only caveat is there must be at least one body block within the SOAP body element in a message and there does not necessarily have to be a SOAP header or any SOAP header blocks. There is no upper limit on the number of header or body blocks, however. A sample SOAP message is presented here in Figure 3-5:

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope
    xmlns:env="http://www.w3.org/2002/06/soap-envelope" >
 <env:Header>
    <tx:transaction-id
        xmlns:tx="http://transaction.example.org"
     env:encodingStyle="http://transaction.example.org/enc"
        env:role=
"http://www.w3.org/2002/06/soap-envelope/role/ultimateReceiver"
        env:mustUnderstand="true">
           decd7461-4ef2138d-7b52e370-fed8a006-ca7ea17
    </tx:transaction-id>
 </env:Header>
 <env:Body xmlns:bank="http://bank.example.org">
    <bank:credit-account env:encodingStyle=
                  "http://www.w3.org/2002/06/soap-encoding">
        <bank:account>12345678</bank:account>
        <bank:sort>10-11-12</bank:sort>
        <bank:amount currency="usd">123.45</bank:amount>
    </bank:credit-account>
    <bank:debit-account>
        <bank:account>87654321</bank:account>
        <bank:sort>12-11-10</bank:sort>
        <bank:amount currency="usd">123.45</bank:amount>
    </bank:debit-account>
 </env:Body>
</env:Envelope>
```

Figure 3-5 A simple SOAP message.

The structure of all SOAP messages (including that shown in Figure 3-5) maps directly onto the abstract model shown in Figure 3-2. Figure 3-5 contains a typical SOAP message with a single header block (which presumably has something to do with managing transactional integrity), and a body containing two elements (which presumably instructs the recipient of the message to perform an operation on two bank accounts). Both the `Header` and `Body` elements are contained within the outer `Envelope` element, which acts solely as a container.

SOAP Envelope

The SOAP `Envelope` is the container structure for the SOAP message and is associated with the namespace `http://www.w3.org/2002/06/soap-envelope`. An example is shown in Figure 3-6 where the namespace is associated with the prefix `env`:

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope
    xmlns:env="http://www.w3.org/2002/06/soap-envelope" >
  <!-- Optional header blocks -->
  <env:Header>
    ...
  </env:Header>
  <!-- Single mandatory body element -->
  <env:Body xmlns:bank="http://bank.example.org">
    ...
  </env:Body>
</env:Envelope>
```

Figure 3-6 The SOAP `envelope` element.

The `Envelope` contains up to two child elements, the `Header` and the `Body` (where the `Body` element is mandatory). Aside from acting as a parent to the `Header` and the `Body` elements, the `Envelope` may also hold namespace declarations that are used within the message.

SOAP Header

The `Header` element provides a mechanism for extending the content of a SOAP message with out-of-band information designed to assist (in some arbitrary and extensible way) the passage of the application content in the `Body` section content through a Web services-based application.

> The SOAP header space is where much of the value in Web services resides, since it is here that aspects like security, transactions, routing, and so on are expressed. Every Web services standard has staked its claim on some part of the SOAP header territory, but in a mutually compatible way. The fact that SOAP headers are extensible enough to support such diverse standards is a major win, since it supports flexible protocol composition tailored to suit specific application domains.

A SOAP header has the local name `Header` associated with the `http://www.w3.org/2002/06/soap-envelope` namespace. It may also contain any number of namespace qualified attributes and any number of child elements, known as *header blocks*. In the absence of any such header blocks, the `Header` element itself may be omitted from the `Envelope`. A sample header block is shown in Figure 3-7.

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope
    xmlns:env="http://www.w3.org/2002/06/soap-envelope" >
  <!-- Optional header blocks -->
  <env:Header>
    <tx:transaction-id
        xmlns:tx="http://transaction.example.org"
     env:encodingStyle="http://transaction.example.org/enc"
        env:role=
"http://www.w3.org/2002/06/soap-envelope/role/ultimateReceiver"
        env:mustUnderstand="true">
           decd7461-4ef2138d-7b52e370-fed8a006-ca7ea17
    </tx:transaction-id>
  </env:Header>

  <!-- Single mandatory body element -->
  <env:Body xmlns:bank="http://bank.example.org">
    ...
  </env:Body>
</env:Envelope>
```

Figure 3-7 A SOAP `header` element.

If present, each header block must be namespace qualified (according to the rules set out in the SOAP schema), may specify how it has been encoded (i.e., which schema constrains it) through the `encodingStyle` attribute, may specify its consumer through the `role` attribute, and may demand that it is understood by SOAP infrastructure that encounters its message through the `mustUnderstand` attribute. The SOAP specification stipulates that it is illegal for the `role` and `mustUnderstand` attributes to appear anywhere other than in header block declarations.

The sender of a SOAP message should not place them anywhere else, and a receiver of such a malformed message must ignore these attributes if they are out of place. These attributes are of fundamental importance to SOAP processing (and thus Web services) and warrant further discussion. The vehicle for this discussion is the example SOAP message shown in Figure 3-5 where we see a header block called `transaction-id` that provides the necessary out-of-band information for the application payload to be processed within a transaction (using a hypothetical transaction processing protocol).

The `role` Attribute

The `role` attribute controls the targeting of header blocks to particular SOAP nodes (where a SOAP node is an entity that is SOAP-aware). The `role` attribute contains a URI that identifies the `role` being played by the intended recipient of its header block. The SOAP node receiving the message containing the header block must check through the headers to see if any of the declared roles are applicable. If there are any matches, the header blocks must be processed or appropriate faults generated.

Although any URI is valid as a role for a SOAP node to assume, the SOAP specification provides three common roles that fit into the canonical SOAP processing model as part of the standard:

- `http://www.w3.org/2002/06/soap-envelope/role/none`: No SOAP processor should attempt to process this header block, although other header blocks may reference it and its contents, allowing data to be shared between header blocks (and thus save bandwidth in transmission).
- `http://www.w3.org/2002/06/soap-envelope/role/next`: Every node must be willing to assume this role since it dictates that header block content is meant for the next SOAP node in the message chain. If a node knows in advance that a subsequent node does not need a header block marked with the "next" role, then it is at liberty to remove that block from the header.
- `http://www.w3.org/2002/06/soap-envelope/role/ultimateReceiver`: The ultimate receiver is the final node in the message chain. Header blocks referencing this `role` attribute (or equivalently referencing no `role` attribute) should be delivered to this last node. It always implicitly plays the role of "next" given that the last node always comes after some other node—even in the simplest case where the last node comes immediately after the initiator.

Figure 3-8 highlights the role attribute from our example SOAP message in Figure 3-5:

```
<env:Header>
  <tx:transaction-id
      xmlns:tx="http://transaction.example.org"
    env:encodingStyle="http://transaction.example.org/enc"
      env:role=
"http://www.w3.org/2002/06/soap-envelope/role/ultimateReceiver"
      env:mustUnderstand="true">
         decd7461-4ef2138d-7b52e370-fed8a006-ca7ea17
  </tx:transaction-id>
</env:Header>
```

Figure 3-8 The `role` attribute.

The role attribute in Figure 3-8 has the value http://www.w3.org/2002/06/soap-envelope/role/ultimateReceiver, which means the contents of the header block are intended for the final SOAP processing node in this interaction (i.e., the recipient Web service). According to the SOAP processing model, this Web service must be capable of processing the application payload (in the SOAP body) in accordance with the transaction processing specification in the header block.

The mustUnderstand Attribute

If the mustUnderstand attribute is set to true, it implies that any SOAP infrastructure that receives the message containing that header block must be able to process it correctly or issue an appropriate fault message. Those header blocks that contain the mustUnderstand="true" attribute are known as *mandatory header blocks* since they must be processed by any nodes playing the matching roles. Header blocks missing their mustUnderstand attribute should still be examined by nodes that play the appropriate role. If a failure to act on a role occurs, it is not deemed to be critical and further processing may occur since by missing the mustUnderstand attribute they are not considered mandatory, as shown in Figure 3-9.

```
<env:Header>
  <tx:transaction-id
      xmlns:tx="http://transaction.example.org"
     env:encodingStyle="http://transaction.example.org/enc"
      env:role="http://www.w3.org/2002/06/soap-envelope/role/
ultimateReceiver"
      env:mustUnderstand="true">
         decd7461-4ef2138d-7b52e370-fed8a006-ca7ea17
  </tx:transaction-id>
</env:Header>
```

Figure 3-9 The mustUnderstand attribute.

> The SOAP specification states that SOAP senders *should not* generate, but SOAP receivers *must* accept the SOAP mustUnderstand attribute information item with a value of "false" or "0". That is, a SOAP message should contain the literal values “true” and “false” in mustUnderstand attributes, not the characters “1” and “0”.

In our example shown in Figure 3-9, the mustUnderstand attribute is set to true because it is imperative that the processing node must perform the account debit-credit within a transaction. If it cannot support transactional processing, then we would prefer that it leaves the accounts well alone—particularly if it is our money being transferred.

The encodingStyle Attribute

The `encodingStyle` attribute is used to declare how the contents of a header block were created. Knowing this information allows a recipient of the header to decode the information it contains. SOAP allows many encoding schemes and provides one of its own as an optional part of the spec. However, we will not dwell on such matters since this attribute is used not only in header blocks but in the body as well, and is covered in much more depth later in this chapter.

SOAP Body

In contrast to the intricacies of the SOAP header space, the body section of a SOAP envelope is straightforward, being simply a container for XML application payload. In fact the SOAP specification states "[T]his specification mandates no particular structure or interpretation of these elements, and provides no standard means for specifying the processing to be done." In our example in Figure 3-5, the application content housed by the SOAP `Body` consists of two elements that are interpreted as commands to debit and credit a bank account, which collectively amount to a funds transfer. The only constraints the SOAP specification places on the SOAP body are that it is implicitly targeted at the `ultimateRecipient` of the application content and that the ultimate recipient must understand its contents.

SOAP Faults

By contrast to its standard role as the simple carrier of application payload, the SOAP Body also acts in a far more interesting way as the conduit for propagating exceptions between the parties in a Web services application. The SOAP `Fault` is a reserved element predefined by the SOAP specification whose purpose is to provide an extensible mechanism for transporting structured and unstructured information about problems that have arisen during the processing of SOAP messages or subsequent application execution. Since the fault mechanism is predefined by the SOAP specification, SOAP toolkits are able to use this mechanism as a standard mechanism for distributed exception handling.

The SOAP `Fault` element belongs to the same namespace as the SOAP `Envelope` and contains two mandatory child elements: `Code` and `Reason`, and three optional elements: `Node`, `Role`, and `Detail`. An example of a SOAP Fault is shown in Figure 3-10 below. The fault is generated in response to the message shown in Figure 3-5 where the message conveyed information on a bank account cash transfer. To understand precisely what has caused the fault, we must understand each of the elements of which it is composed.

The first child element of the `Fault` is the `Code` element, which contains two subelements: a mandatory element called `Value` and an optional element called `Subcode`. The `Value` element can contain any of a small number of fault codes as qualified names (some-

```
<?xml version="1.0" ?>
<env:Envelope
     xmlns:env="http://www.w3.org/2002/06/soap-envelope"
     xmlns:bank="http://bank.example.org">
    <env:Body>
        <env:Fault>
            <env:Code>
                <env:Value>env:Receiver</env:Value>
                <env:Subcode>
                    <env:Value>bank:bad-account</env:Value>
                </env:Subcode>
            </env:Code>
            <env:Reason lang="en-UK">
             The specified account does exist at this branch
            </env:Reason>
            <env:Detail>
                <err:myfaultdetails
                  xmlns:err= "http://bank.example.org/fault">
                    <err:invalid-account-sortcode>
                        <bank:sortcode>
                            10-11-12
                        </bank:sortcode>
                        <bank:account>
                            12345678
                        </bank:account>
                    </err:invalid-account-sortcode >
                </err:myfaultdetails>
            </env:Detail>
        </env:Fault>
    </env:Body>
</env:Envelope>
```

Figure 3-10 An example SOAP fault.

times abbreviated to `QName`) from the `http://www.w3.org/2002/06/soap-envelope` namespace, as per Figure 3-11, where each `QName` identifies a reason why the fault was generated.

In Figure 3-10 the contents of the `env:Value` element is `env:Receiver` (shown in Figure 3-12), which tells us that it was the SOAP node at the end of the message path (the receiver) that generated the fault and not an intermediate node dealing with the transaction header block.

As shown in Figure 3-13, the `Subcode` element contains a `Value` element that gives application-specific information on the fault through the qualified name `bank:bad-account`. This `QName` has significance only within the scope of the application that issued it, and as such the `Subcode` mechanism provides the means for propagating finely targeted application-level exception messages.

Fault Code	Description
`VersionMismatch`	Occurs when SOAP infrastructure has detected mutually incompatible implementations based on different versions of the SOAP specification.
`MustUnderstand`	Issued in the case where a SOAP node has received a header block has with its `mustUnderstand` attribute set to true, but does not have the capability to correctly process that header block – that is, it does not understand the protocol with which that header block is associated.
`DataEncodingUnknown`	Arises when the content of either a header or body block is encoded according to a schema that the SOAP node reporting the fault does not understand.
`Sender`	Occurs when the sender propagated a malformed message, including messages with insufficient data to enable the recipient to process it. It is an indication that the message is not to be resent without change.
`Receiver`	Generated when the recipient of the SOAP message could not process the message content because of some application failure. Assuming the failure is transient, resending the message later may successfully invoke processing.

Figure 3-11 SOAP fault codes.

```
<env:Fault>
  <env:Code>
    <env:Value>env:Receiver</env:Value>
```

Figure 3-12 Identifying the faulting SOAP node.

```
<env:Subcode>
  <env:Value>bank:bad-account</env:Value>
</env:Subcode>
```

Figure 3-13 Application-specific fault information.

Though it isn't used in this fault, the `Subcode` element also makes the SOAP fault mechanism extensible. Like the `Code` element, the `Subcode` element also contains a mandatory `Value` child element and an optional `Subcode` element, which may contain further nested `Subcode` elements. The `Value` element of any `Subcode` contains a qualified name that consists of a prefix and a local name that references a particular `QName` within the application-level XML message set.

The `Reason` element associated with a `Code` is used to provide a human readable explanation of the fault, which in Figure 3-10 tells us that "`The specified account does not exist at this branch`". SOAP toolkits often use the contents of the `Reason` element when throwing exceptions or logging failures to make debugging easier. However, the `Reason` element is strictly meant for human consumption and it is considered bad practice to use its content for further processing.

The optional `Node` element provides information on which node in the SOAP message's path caused the fault. The content of the `Node` element is simply the URI of the node where the problem arose. In Figure 3-10 we do not have a `Node` element because it is the ultimate recipient of the message that caused the fault, and clearly the sender of the message already knows the URI of the recipient. However if, for example, an intermediate node dealing with the transactional aspects of the transfer failed, then we would expect that the `Node` element would be used to inform us of the intermediary's failure (and as we shall see, we would not expect a `Detail` element).

The `Node` element is complemented by the also optional `Role` element that provides information pertaining to what the failing node was doing at the point at which it failed. The `Role` element carries a URI that identifies the operation (usually some Web services standard) and that the party resolving the fault can use to determine what part of the application went wrong. Thus, the combination of `Node` and `Role` provides valuable feedback on exactly what went wrong and where.

The SOAP `Detail` element, as recapped in Figure 3-14, provides in-depth feedback on the fault if that fault was caused as a by-product of processing the SOAP `Body`.

```
<env:Detail>
  <err:myfaultdetails
    xmlns:err= "http://bank.example.org/fault">
    <err:invalid-account-sortcode>
      <bank:sortcode>
        10-11-12
      </bank:sortcode>
      <bank:account>
        12345678
      </bank:account>
    </err:invalid-account-sortcode >
  </err:myfaultdetails>
</env:Detail>
```

Figure 3-14 Fault detail in an application-specific form.

The presence of the `Detail` element provides information on faults arising from the application payload (i.e., the `Body` element had been at least partially processed by the ultimate recipient), whereas its absence indicates that the fault arose because of out-of-band information carried in header blocks. Thus we would expect that if a `Detail` block is present, as it is in Figure 3-10 and Figure 3-11, the `Node` and `Role` elements will be absent and vice versa.

The contents of the `Detail` element are known as *detail entries* and are application-specific and consist of any number of child elements. In Fault detail in an application-specific form. we see the `invalid-account-sortcode` element which describes the fault is some application specific fashion.

SOAP Encoding

The `encodingStyle` attribute appears in both header blocks and the body element of a SOAP message. As its name suggests, the attribute conveys information about how the contents of a particular element are encoded. At first this might seem a little odd since the SOAP message is expressed in XML. However, the SOAP specification is distinctly hands-off in specifying how header and body elements (aside from the SOAP `Fault` element) are composed, and defines only the overall structure of the message. Furthermore, XML is expressive and does not constrain the form of document a great deal and, therefore, we could imagine a number of different and mutually uninteroperable ways of encoding the same data, for example:

```
<account>
    <balance>
        123.45
    </balance>
</account>
```

and `<account balance="123.45"/>`

might both be informally interpreted in the same way by a human reader but would not be considered equivalent by XML processing software. Ironically, this is one of the downfalls of XML—it is so expressive that, given the chance, we would all express ourselves in completely different ways. To solve this problem, the `encodingStyle` attribute allows the form of the content to be constrained according to some schema shared between the sender and recipient.

One potential drawback is that senders and receivers of messages may not share schemas—indeed the senders and receivers may be applications that do not deal with XML at all—and thus the best intentions of a SOAP-based architecture may be laid to waste. To avoid such problems, the SOAP specification has its own schema and rules for converting application-level data into a form suitable for embedding into SOAP messages. This is known as SOAP Encoding, and is associated with the namespace `env:encodingStyle="http://www.w3.org/2002/06/soap-encoding"`.

The rules for encoding application data as SOAP messages are captured in the SOAP specification as the SOAP Data Model. This is a straightforward and concise part of the specification that describes how to reduce data structures to a directed, labeled graph. While it is outside of the scope of this book to detail the SOAP Data Model, the general technique is shown in Figure 3-15. This SOAP encoding example, highlights the fact that there are two aspects to the encoding. The first of these is to transform a data structure from an application into a form suitable for expressing in XML via the rules specified in the SOAP Data Model. The other aspect is to ensure that all the data in the subsequent XML document is properly constrained by the SOAP schema. It is worth noting that SOAP provides low entry point through SOAP encoding since a SOAP toolkit will support the serialization and deserialization of arbitrary graphs of objects via this model, with minimal effort required of the developer. In fact, coupled with the fact that SOAP has a packaging mechanism for managing message content, and a means (though SOAP encoding) of easily creating message content we are close to having an XML-based Remote Procedure Call mechanism.

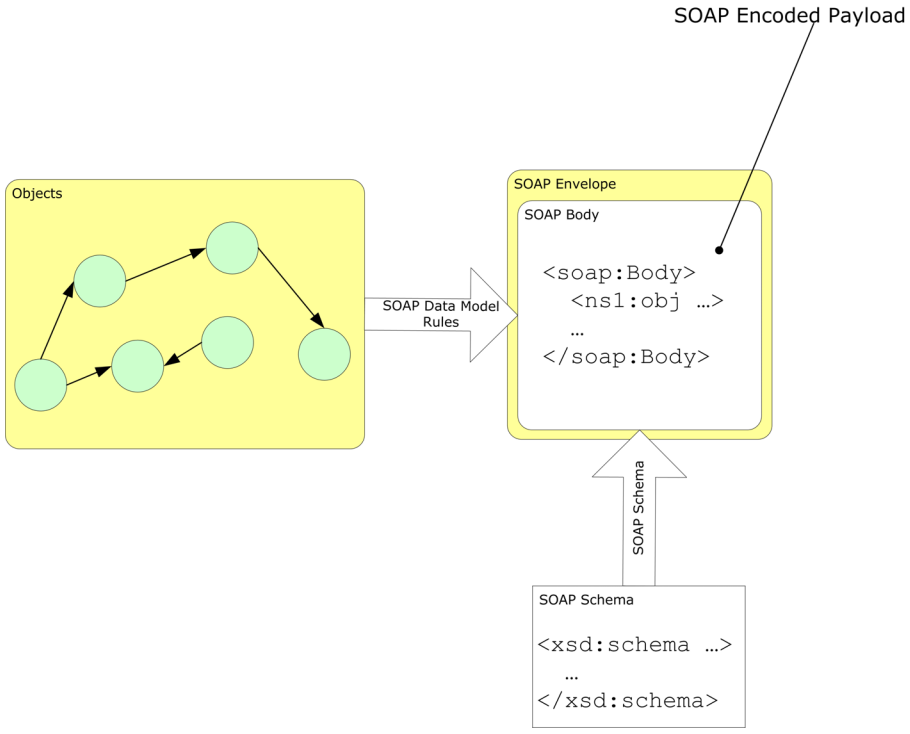


Figure 3-15 SOAP encoding application-level objects.

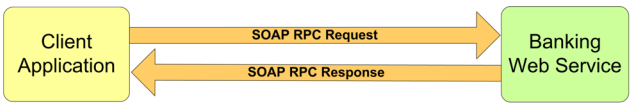
SOAP RPC

As it happens, the SOAP specification is useful straight "out of the box." The fact that it provides both a message format and marshalling naturally supports RPC, and indeed millions of developers worldwide will by now have seen how easy it is to run SOAP RPC-based Web services on a myriad of platforms. It's probably not the case that SOAP RPC will be the dominant paradigm for SOAP in the long term, but it is easy to achieve results with SOAP RPC quickly because all the major toolkits support it and RPC is a pattern many developers are familiar with.

> Note that although SOAP RPC has enjoyed some prominence in older Web services toolkits, there is a majority consensus of opinion in the Web services community that more coarse-grained, document-oriented interactions should be the norm when using SOAP.

SOAP RPC provides toolkits with a convention for packaging SOAP-encoded messages so they can be easily mapped onto procedure calls in programming languages. To illustrate, let's return to our banking scenario and see how SOAP RPC might be used to expose account management facilities to users. Bear in mind throughout this simple example that it is an utterly insecure instance whose purpose is to demonstrate SOAP RPC only.

Figure 3-16 shows a simple interaction between a Web service that offers the facility to open bank accounts and a client that consumes this functionality on behalf of a user. The Web service supports an operation called `openAccount(…)` which it exposes through a SOAP server and advertises as being accessible via SOAP RPC (SOAP does not itself provide a means of describing interfaces, but as we shall see later in the chapter, WSDL does). The client inter-




```
// Client implementation
Bank b = new Bank();
String account =
b.openAccout(title,
                  surname,
                  firstname,
                  postcode,
                  telephone);
```

```
// Service implementation
class Bank
{
    public String openAccout(title,
                  surname,
                  firstname,
                  postcode,
                  telephone)
    {
        // Some account processing
        return accoutNumber;
    }
}
```

Figure 3-16 Interacting with a banking service via SOAP RPC.

acts with this service through a stub or proxy class called `Bank` which is toolkit-generated (though masochists are free to generate their own stubs) and deals with the marshalling and unmarshalling of local variables into SOAP RPC messages.

In this simple use case, the SOAP on the wire between the client and Web service is similarly straightforward. Figure 3-17 shows the SOAP RPC request sent from the client to the Web service:

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope
    xmlns:env="http://www.w3.org/2002/06/soap-envelope">
    <env:Body>
        <bank:openAccount env:encodingStyle=
         "http://www.w3.org/2002/06/soap-encoding"
         xmlns:bank="http://bank.example.org/account"
         xmlns:xs="http://www.w3.org/2001/XMLSchema"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
            <bank:title xsi:type="xs:string">
                Mr
            </bank:title>
            <bank:surname xsi:type="xs:string">
                Bond
            </bank:surname>
            <bank:firstname xsi:type="xs:string">
                James
            </bank:firstname>
            <bank:postcode xsi:type="xs:string">
                S1 3AZ
            </bank:postcode>
            <bank:telephone xsi:type="xs:string">
                09876 123456
            </bank:telephone>
        </bank:openAccount>
    </env:Body>
</env:Envelope>
```

Figure 3-17 A SOAP RPC request.

There is nothing particularly surprising about the RPC request presented in Figure 3-17. As per the RPC specification, the content is held entirely within the SOAP body (SOAP RPC does not preclude the use of header blocks, but they are unnecessary for this example), and the name of the element (`openAccount`) matches the name of the method to be called on the Web service. The contents of the `bank:openAccount` correspond to the parameters of the `openAccount` method shown in Figure 3-16, with the addition of the `xsi:type` attribute to help recipients of the message to convert the contents of each parameter element to the correct kind of variable in specific programming languages. The response to the original request follows a slightly more intricate set of rules and conventions as shown in Figure 3-18.

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope
     xmlns:env="http://www.w3.org/2002/06/soap-envelope">
    <env:Body>
        <bank:openAccountResponse env:encodingStyle=
         "http://www.w3.org/2002/06/soap-encoding" xmlns:rpc=
         "http://www.w3.org/2002/06/soap-rpc" xmlns:bank=
         "http://bank.example.org/account" xmlns:xs=
         "http://www.w3.org/2001/XMLSchema" xmlns:xsi=
          "http://www.w3.org/2001/XMLSchema-instance">
            <rpc:result>bank:accountNo</rpc:result>
            <bank:accountNo xsi:type="xsd:int">
                10000014
            </bank:accountNo>
        </bank:openAccountResponse>
    </env:Body>
</env:Envelope>
```

Figure 3-18 A SOAP RPC response.

The SOAP RPC response is slightly more complex and interesting than the request, and there are two noteworthy aspects of the SOAP RPC response. The first is that by convention the name of the response element is the same as the request element with *Response* appended (and toolkits make use of this convention so it's practically standard now).

The second interesting aspect is that the response is capable of matching the procedure call semantics of many languages since it supports in, out, and in/out parameters as well as return values where an "in" parameter sources a variable to the procedure call; an "out" parameter sources nothing to the procedure but is populated with data at the end of the procedure call. An "in/out" parameter does both, while a return value is similar to an out parameter with the exception that its data may be ignored by the caller.

In this example, we have five "in" parameters (title, surname, first name, post code, and telephone number) which we saw in the SOAP request and expect a single return value (account number) which we see in the SOAP response. The return value is also interesting because, due to its importance in most programming languages, it is separated from out and in/out parameters by the addition of the `rpc:result` element that contains a `QName` that references the element which holds the return value. Other elements which are not referenced are simply treated as out or in/out parameters. This behavior is different from previous versions of SOAP where the return value was distinguished by being first among the child elements of the response. This was rectified for SOAP 1.2 because of the inevitable ambiguity that such a contrivance incurs—what happens when there is no return value?

Of course in a textbook example like this, everything has worked correctly and no problems were encountered. Indeed, you would be hard pressed to find a reader who would enjoy a book where the examples were a set of abject failures. However, like paying taxes and dying, computing systems failures seem inevitable. To cover those cases where things go wrong, SOAP

RPC takes advantage of the SOAP fault mechanism with a set of additional fault codes (whose namespace is `http://www.w3.org/2002/06/soap-rpc`), which are used in preference to the standard SOAP fault codes in RPC-based messages shown in Figure 3-19, in decreasing order of precedence.

Fault	SOAP Encoding for Fault
Transient fault at receiver (e.g. out of memory error).	Fault with value of `env:Receiver` should be generated.
Receiver does not understand data encoding (e.g. encoding mechanism substantially different at sender and receiver)	A fault with a `Value` of `env:DataEncodingUnknown` for `Code` should be generated.
The service being invoked does not expose a method matching the name of the RPC element.	A fault with a `Value` of `env:Sender` for `Code` and a `Value` of `rpc:ProcedureNotPresent` for `Subcode` may be generated.
The receiver cannot parse the arguments sent. There may be too many or too few arguments, or there may be type mismatches.	A fault with a `Value` of `env:Sender` for `Code` and a `Value` of `rpc:BadArguments` for `Subcode` must be generated.

Figure 3-19 SOAP RPC faults.

Finally, in Figure 3-20 we see a SOAP RPC fault in action where a poorly constructed client application has tried to invoke an operation on the bank Web service, but has populated its request message with nonsense. In this figure, the bank Web service responds with a SOAP RPC fault that identifies the faulting actor (the sender) as part of the `Code` element. It also describes what the faulting actor did wrong (sent bad arguments) by specified the `QName` `rpc:BadArguments` as part of the `subcode` element. It also contains some human-readable information to aid debugging (missing surname parameter), in the `Reason` element.

```
<?xml version="1.0"?>
<env:Envelope
     xmlns:env="http://www.w3.org/2002/06/soap-envelope"
     xmlns:rpc="http://www.w3.org/2002/06/soap-rpc">
    <env:Body>
        <env:Fault>
            <env:Code>
                <env:Value>env:Sender</env:Value>
                <env:Subcode>
                    <env:Value>rpc:BadArguments</env:Value>
                </env:Subcode>
            </env:Code>
            <env:Reason>
                Missing surname parameter
            </env:Reason>
        </env:Fault>
    </env:Body>
</env:Envelope>
```

Figure 3-20 A SOAP RPC fault.

Using Alternative SOAP Encodings

Of course some applications already deal with XML natively, and there are currently XML-based vocabularies in use today supporting a plethora of B2B applications. SOAP-based messaging can take advantage of the pre-existence of schemas to craft message exchanges that compliment existing systems using so-called document-style SOAP.

The way in which alternative SOAP encodings are handled is straightforward. Instead of encoding header or body content according to the SOAP Data Model, we simply encode according to the rules and constraints of our data model and schema. In essence, we can just slide our own XML documents into a SOAP message, providing we remember to specify the `encodingStyle` attribute (and of course ensuring that the intended recipients of the message can understand it). This style of SOAP encoding is known as literal style and naturally suits the interchange of business-level documents based on their existing schemas.

This is a definite boon to SOAP use and by our estimation, the future dominant paradigm for SOAP use. Its plus points include not only the ability to re-use existing schemas, but by dint of the fact that we are now dealing with message exchanges and not remote procedure calls, we are encouraged to design Web service interactions with much coarser granularity. In essence, we are changing from a fine-grained model that RPC encourages (you send a little bit of data, get a little bit back and make further calls until your business is completed), to a much coarser-grained model where you send all the data necessary to get some business process done at the recipient end, and expect that the recipient may take some time before he gets back to you with a complete answer.

One particularly apt view of the fine- versus coarse-grained view of Web services interactions is that of a phone call versus a facsimile transmission.[4] Where we interact with a business over the phone there is a great deal of back-and-forth between ourselves and the agent of the business to whom we are talking. We both have to establish contexts and roles for each other, and then enter into a socially and linguistically complex conversation to get business transacted. Small units of data are exchanged like "color" and "amount" that are meaningless without the context, and if the call is lost we have to start over. This is fine-grained interaction.

While we would not seek to undermine the value of good old human-to-human communication, sometimes we just don't have time for this. It's even worse for our computing systems to have to communicate this way since they don't get any of the social pleasures of talking to each other. A better solution is often to obtain a catalog or brochure for the business that we want to trade with. When we have the catalog, we can spend time pouring over the contents to see what goods or services we require. Once we are certain of what we want, we can just fill in and fax the order form to the company and soon our products arrive via postal mail.

This system is eminently preferable for business processes based on Web services. For a start, complex and meaningful data was exchanged that does not rely on context. A catalog and an order form are descriptive enough to be universally understood and the frequency of data exchange was low, which presents less opportunity for things to go astray. This system is also loosely coupled when the systems are not directly communicating (which only happens twice: once for catalog delivery and once while the order is being faxed). They are not affected by one another and do not tie up one another's resources—quite the contrary to the telephone-based system.

Of course, we don't necessarily get something for nothing. The price that we must pay as developers is that we must write the code to deal with the encoding schemes we choose. In the SOAP RPC domain where the encoding is fixed and the serialization from application-level data structure to XML is governed by the SOAP Data Model, toolkits could take care of much of this work. Unfortunately, when we are working with our own schemas, we cannot expect SOAP toolkits to be able to second-guess its semantics and, thus, we have to develop our own handler code to deal with it, as shown in Figure 3-21.

[4] See "The 7 Principles of Web Services Business Process Management" at http://www.iona.com/whitepapers/Principles-of-Web-Services-and-BPM.pdf

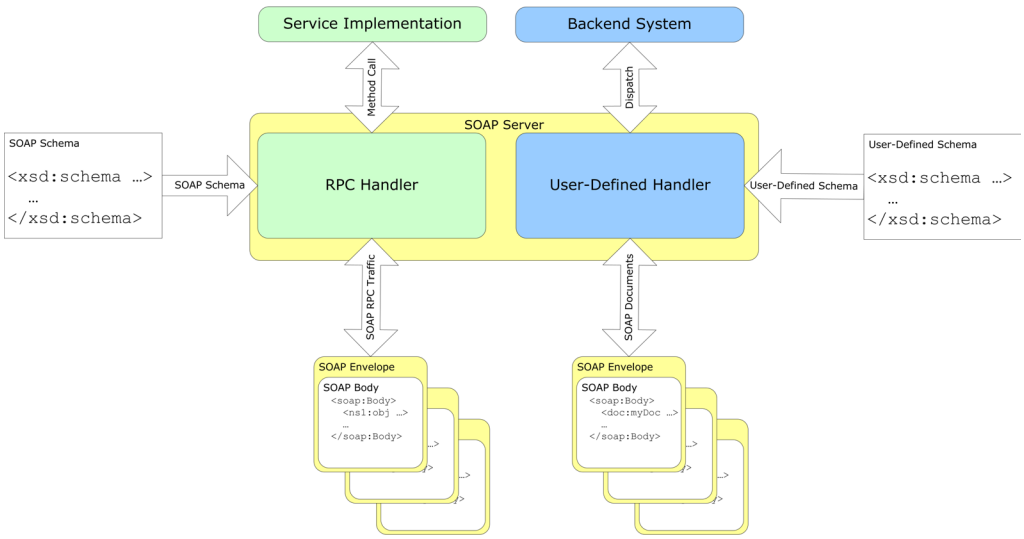


Figure 3-21 Document-oriented SOAP processing.

The user-defined handler in Figure 3-21 is one of potentially many handlers deployed onto the SOAP server to provide the functionality to deal with SOAP messages encoded with arbitrary schemas. Where the SOAP RPC handler simply dispatches the contents of the SOAP RPC messages it receives to appropriate method calls, there are no such constraints on a Web service which uses document-style SOAP. One valid method would be to simply pick out the important values from the incoming document and use them to call a method, just like the SOAP RPC handler. However, as more enterprises become focused on XML as a standard means for transporting data within the enterprise boundary, it is more likely that the contents of the SOAP body will flow directly onto the intranet. Once delivered to the intranet, the messages may be transformed into proprietary XML formats for inclusion with in-house applications, or may be used to trigger business processes without the need to perform the kind of marshalling/unmarshalling required for SOAP RPC.

Note that irrespective of whether the application payload in SOAP messages is SOAP-encoded, or encoded according to a third-party schema, the way that header blocks are used to convey out-of-band information to support advanced Web services protocols is unaffected. Headers are an orthogonal issue to the application-level content.

Document, RPC, Literal, Encoded

Much of the confusion in understanding SOAP comes from the fact that several of the key terms are overloaded. For example, both SOAP RPC (meaning the convention for performing remote procedure calls via SOAP) and RPC style are both valid pieces of SOAP terminology. In this section we clarify the meaning of each of these terms so they do not cause further confusion as we begin discussing WSDL.

Document

Document-style SOAP refers to the way in which the application payload is hosted within the SOAP `Body` element. When we use document style, it means the document is placed directly as a child of the `Body` element, as shown on the left in Figure 3-22, where the application content is a direct child of the `<soap:Body>` element.

	Document	RPC

Literal	`<soap:Body>`
	`  <inv:invoice ...>`
	`  <inv:orderNo ...`
	`  </inv:invoice>`
	`</soap:Body>`
		`<soap:Body>`
		`  **<m:purchase>**`
		`  <inv:invoice ...>`
		`  <inv:orderNo ...`
		`  </inv:invoice>`
		`  **</m:purchase>**`
		`</soap:Body>`

Encoded	`<soap:Body>`
	`  <ns1:invoice>`
	`  <ns1:orderNo...`
	`  </ns1:invoice>`
	`</soap:Body>`
		`<soap:Body>`
		`  **<m:purchase>**`
		`  <ns1:invoice>`
		`  <ns1:orderNo...`
		`  </ns1:invoice>`
		`  **</m:purchase>**`
		`</soap:Body>`

Figure 3-22 Document, RPC, Literal, and Encoded SOAP messages.

RPC

RPC-style SOAP wraps the application content inside an element whose name can be used to indicate the name of a method to dispatch the content to. This is shown on the right-hand side of Figure 3-22, where we see the application content wrapped `<m:purchase>` element.

Literal

Literal SOAP messages use arbitrary schemas to provide the meta-level description (and constraints) of the SOAP payload. Thus when using literal SOAP, we see that it is akin to taking an instance document of a particular schema and embedding it directly into the SOAP message, as shown at the top of Figure 3-22.

Encoded

SOAP-encoded messages are created by transforming application-level data structures via the SOAP Data Model into a XML format that conforms to the SOAP Schema. Thus, encoded messages tend to look machine-produced and may not generally resemble the same message expressed as a literal. Encoded messages are shown at the bottom of Figure 3-22.

SOAP RPC and SOAP Document-Literal

The SOAP specification provides four ways in which we could package SOAP messages, as shown in Figure 3-22. However, in Web services we tend to use only two of them: SOAP encoded-rpc (when combined with a request-response protocol becomes the SOAP RPC convention) and SOAP document-literal.

Document-literal is the preferred means of exchanging SOAP messages since it just packages application-level XML documents into the SOAP `Body` for transport without placing any semantics on the content.

As we have previously seen, with SOAP RPC the implied semantics are that the first child of the SOAP `Body` element names a method to which the content should be dispatched.

The remaining two options, document-encoded and rpc-literal, are seldom used since they mix styles to no great effect. Encoding documents is pointless if we already have schemas that describe them. Similarly, wrapping a document within a named element is futile unless we are going to use that convention as a remote procedure call mechanism. Since we already have SOAP RPC, this is simply a waste of effort.

SOAP, Web Services, and the REST Architecture

The World Wide Web (WWW) is unquestionably the largest and, by implication, the most scalable distributed system ever built. Though its original goal of simple content delivery was modest, the way that the Web has scaled is nothing short of miraculous.

Given the success of the Web, there is a body of opinion involved in designing the fundamental Web services architecture (that includes the SOAP specification) for which the means to

achieving the same level of application scalability through Web services mirrors that of content scalability in the WWW.

The members of this group are proponents of the REST (*REpresentational State Transfer*) architecture, which it is claimed is "Web-Friendly." The REST architecture sees a distributed system as a collection of named resources (named with URIs) that support a small set of common verbs (GET, PUT, POST, DELETE) in common with the WWW.

> The REST idea of defining global methods is similar to the UNIX concept of pipelining programs. UNIX programs all have three simple interfaces defined (STDIN, STDOUT, STDERR) for every program, which allows any two arbitrary programs to interact. The simplicity of REST as compared to custom network interfaces is analogous to the simplicity of UNIX pipelines vs. writing a custom application to achieve the same functionality. REST embraces simplicity and gains scalability as a result. The Web is a REST system, and the generic interfaces in question are completely described by the semantics of HTTP.[5]

What this means to the SOAP developer is that certain operations involving the retrieval of data without changing the state of a Web resource should be performed in a manner that is harmonious with the Web. For example, imagine that we want to retrieve the balance of our account from our bank Web service. Ordinarily we might have thought that something like that shown in Figure 3-23 would be ideal. If this message was sent as part of a HTTP POST, then it would be delivered to the SOAP server, which would then extract the parameters and deliver the results via the `getBalanceResponse` message.

```
<?xml version="1.0" ?>
<env:Envelope
     xmlns:env="http://www.w3.org/2002/06/soap-envelope" >
  <env:Body>
    <bank:getBalance
  env:encodingStyle="http://www.w3.org/2002/06/soap-encoding"
  xmlns:bank="http://bank.example.org/">
      <bank:accountNo>
           12345678
      </bank:accountNo>
    </bank:getBalance>
  </env:Body>
</env:Envelope>
```

Figure 3-23 A "Web-Unfriendly" message.

5. See RESTwiki, http://internet.conveyor.com/RESTwiki/moin.cgi/FrontPage

However, this is now discouraged by the SOAP specification and instead we are encouraged to use HTTP directly to retrieve the information, rather than "tunneling" SOAP through HTTP to get the information. A "Web-friendly" equivalent of Figure 3-23 is shown in Figure 3-24 where the HTTP request directly identifies the information to be retrieved and informs the Web service that it wants the returned information delivered in SOAP format.

```
GET /account?no=12345678  HTTP/1.1
Host: bank.example.org
Accept: application/soap+xml
```

Figure 3-24 A "Web-Friendly" message.

Figure 3-24 is certainly Web friendly since it uses the Web's application protocol (HTTP). However, there are a number of obstacles that have not yet been overcome at the time of writing that may prove detrimental to this approach:

1. A service may be exposed over other protocols than HTTP (e.g., SMTP that does not support the GET verb).
2. This scheme cannot be used if there are intermediate nodes that process SOAP header blocks.
3. There is no guidance yet provided by the SOAP specification authors on how to turn an RPC definition into its Web-friendly format.
4. Too much choice for little gain since we have to support the "Web-Unfriendly" approach anyway for those interactions that require the exchange of structured data.

While these techniques may yet come to fruition, it may be a long time before resolution is reached. When architecting applications today, the best compromise that we can offer is to be aware of those situations where you are engaged in pure information retrieval, and ensure that your architecture is extensible enough to change to a Web-friendly mechanism for those interactions tomorrow. Make sure the code that deals with Web services interactions is modular enough to be easily replaced by Web-friendly modules when the W3C architectural recommendations become more specific.

Looking Back to SOAP 1.1

While Web services will migrate toward SOAP 1.2 in the near future, the most prevalent Web services technology today is the now deprecated SOAP 1.1. Although there isn't a great deal that has changed between the two revisions, there are some caveats we must be aware of when dealing with SOAP 1.1-based systems. To ensure that the work we've invested in understanding

SOAP 1.2 isn't lost on SOAP 1.1 systems, we shall finish our coverage of SOAP with a set of notes that should make our SOAP 1.2 knowledge backwardly compatible with SOAP 1.1.[6]

Syntactic Differences between SOAP 1.2 and SOAP 1.1

- SOAP 1.2 does not permit any element after the body. The SOAP 1.1 schema definition allowed for such a possibility, but the textual description is silent about it. However, the Web Services Interoperability Organization (WS-I) has recently disallowed this practice in its basic profile and as such we should now consider that no elements are allowed after the SOAP body, since any other interpretation will hamper interoperability.
- SOAP 1.2 does not allow the `encodingStyle` attribute to appear on the SOAP Envelope, while SOAP 1.1 allows it to appear on any element.
- SOAP 1.2 defines the new `Misunderstood` header element for conveying information on a mandatory header block that could not be processed, as indicated by the presence of a `mustUnderstand` fault code. SOAP 1.1 provided the fault code, but no details on its use.
- In the SOAP 1.2 infoset-based description, the `mustUnderstand` attribute in header elements takes the (logical) value `true` or `false` while in SOAP 1.1 they are the literal value `1` or `0`, respectively.
- SOAP 1.2 provides a new fault code `DataEncodingUnknown`.
- The various namespaces defined by the two protocols are different.
- SOAP 1.2 replaces the attribute `actor` with `role` but with essentially the same semantics.
- SOAP 1.2 defines two new roles, `none` and `ultimateReceiver`, together with a more detailed processing model on how these behave.
- SOAP 1.2 has removed the dot notation for fault codes, which are now simply of the form `env:name`, where `env` is the SOAP envelope namespace.
- SOAP 1.2 replaces `client` and `server` fault codes with `Sender` and `Receiver`.
- SOAP 1.2 uses the element names `Code` and `Reason`, respectively, for what is called `faultcode` and `faultstring` in SOAP 1.1.
- SOAP 1.2 provides a hierarchical structure for the mandatory SOAP `Code` element, and introduces two new optional subelements, `Node` and `Role`.

6. These notes are abridged from the SOAP 1.2. Primer document which can be found at: http://www.w3.org/TR/2002/WD-soap12-part0-20020626/

Changes to SOAP-RPC

Though there was some feeling in the SOAP community that SOAP RPC has had its day and should be dropped in favor of a purely document-oriented protocol, the widespread acceptance of SOAP RPC has meant that it persists in SOAP 1.2, but with a few notable differences:

- SOAP 1.2 provides a `rpc:result` element accessor for RPCs.
- SOAP 1.2 provides several additional fault codes in the RPC namespace.
- SOAP 1.2 allows RPC requests and responses to be modeled as both structs as well as arrays. SOAP 1.1 allowed only the former construct.
- SOAP 1.2 offers guidance on a Web-friendly approach to defining RPCs where the method's purpose is purely a "safe" informational retrieval.

SOAP Encoding

Given the fact that SOAP RPC is still supported in SOAP 1.2 and that there have been some changes to the RPC mechanism, some portions of the SOAP encoding part of the specification have been updated to either better reflect the changes made to SOAP RPC in SOAP 1.2, or to provide performance enhancements compared to their SOAP 1.1 equivalents.

- An abstract data model based on a directed edge-labeled graph has been formulated for SOAP 1.2. The SOAP 1.2 encodings are dependent on this data model. The SOAP RPC conventions are dependent on this data model, but have no dependencies on the SOAP encoding. Support of the SOAP 1.2 encodings and SOAP 1.2 RPC conventions are optional.
- The syntax for the serialization of an array has been changed in SOAP 1.2 from that in SOAP 1.1.
- The support provided in SOAP 1.1 for partially transmitted and sparse arrays is not available in SOAP 1.2.
- SOAP 1.2 allows the inline serialization of multi-ref values.
- The `href` attribute in SOAP 1.1 of type `anyURI`, is called `ref` in SOAP 1.2 and is of type `IDREF`.
- In SOAP 1.2, omitted accessors of compound types are made equal to NILs.
- SOAP 1.2 provides several fault subcodes for indicating encoding errors.
- Types on nodes are made optional in SOAP 1.2.

While most of these issues are aimed at the developers of SOAP infrastucture, it is often useful to bear these features in mind for debugging purposes, especially while we are in the changeover period before SOAP 1.2 becomes the dominant SOAP version.

WSDL

Having a means of transporting data between Web services is only half the story. Without interface descriptions for our Web services, they are about as useful as any other undocumented API—very little! While in theory we could simply examine the message schemas for a Web service and figure out for ourselves how to interoperate with it, this is a difficult and error-prone process and one which could be safely automated if Web services had recognizable interfaces. Fortunately, WSDL provides this capability and more for Web services.

The Web Service Description Language or WSDL—pronounced "Whiz Dull"—is the equivalent of an XML-based IDL from CORBA or COM, and is used to describe a Web service's endpoints to other software agents with which it will interact. WSDL can be used to specify the interfaces of Web services bound to a number of protocols including HTTP GET and POST, but we are only interested in WSDL's SOAP support here, since it is SOAP which we consider to support the (logical) Web services network. In the remainder of this chapter we explore WSDL and show how we can build rich interfaces for Web services that enable truly dynamic discovery and binding, and show how WSDL can be used as the basis of other protocols and extended to other domains outside of interface description.

WSDL Structure

A WSDL interface logically consists of two parts: the abstract parts that describe the operations the Web service supports and the types of messages that parameterize those operations; and the concrete parts that describe how those operations are tied to a physical network endpoint and how messages are mapped onto specific carrier protocols which that network endpoint supports. The general structure of a WSDL document is shown in Figure 3-25.

The foundation of any WSDL interface is the set of messages that the service behind the interface expects to send and receive. A `message` is normally defined using XML Schema types (though WSDL allows other schema languages to be used) and is partitioned into a number of logical parts to ease access to its contents.

Messages themselves are grouped into WSDL `operation` elements that have similar semantics to function signatures in an imperative programming language. Like a function signature, an operation has input, output, and fault messages where WSDL supports at most a single input and output message, but permits the declaration of an arbitrary number of faults.

The `portType` is where what we think of as a Web service begins to take shape. A `portType` is a collection of operations that we consider to be a Web service. However, at this point the operations are still defined in abstract terms, simply grouping sets of message exchanges into operations.

The `binding` section of a WSDL interface describes how to map the abstractly defined messages and operations onto a physical carrier protocol. Each `operation` from each `portType` that is to be bound to a specific protocol (and thus ultimately be made available to the net-

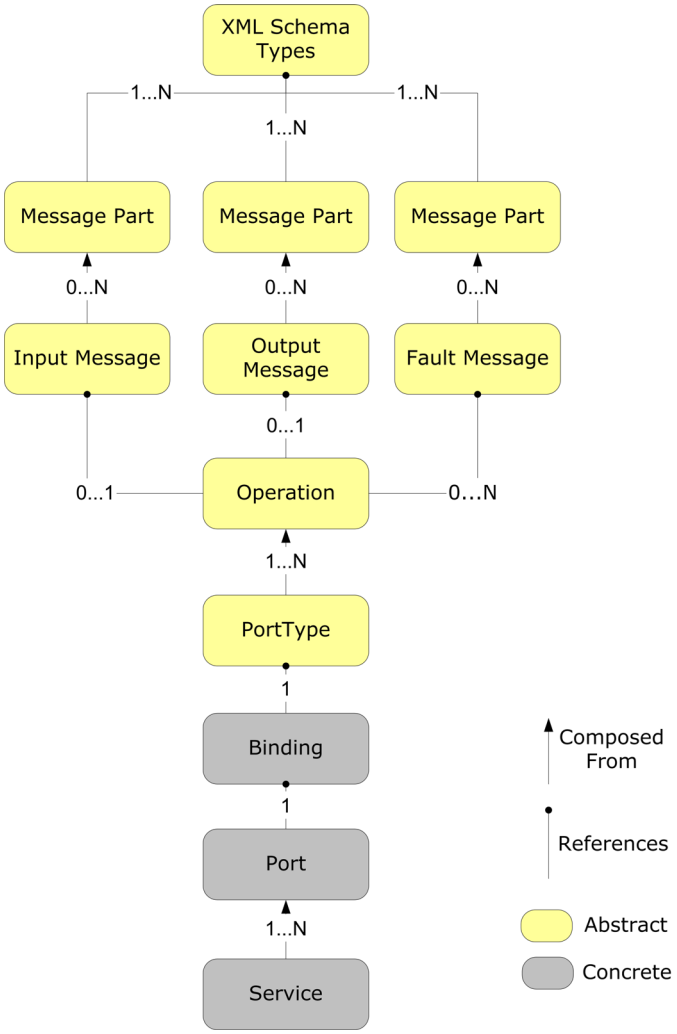


Figure 3-25 WSDL structure.

work) is augmented with binding information from the binding part of the WSDL specification—WSDL supports SOAP, HTTP GET and POST, and MIME—to provide a protocol-specific version of the original `portType` declaration.

Finally, a `port` is declared that references a particular `binding`, and along with addressing information is wrapped together into a `service` element to form the final physical, network addressable Web service.

As we saw in Figure 3-25, the abstract components of a WSDL description are the `types`, `message`, and `portType` elements, while the concrete elements are `binding` and `service`.

The split between abstract and concrete is useful, because it allows us to design interfaces in isolation from eventual deployment environments, using only the abstract definitions in WSDL. Once we are happy with the abstract aspects of the Web service interface, we can then write the concrete parts to tie the service down to a specific location, accessible over a specific protocol.

The Stock Quote WSDL Interface

Having seen WSDL from a theoretical perspective, we can concretize that theory by considering a specific example. The classic Web services application is the stock ticker example where a Web service provides stock quotes on request. Throughout the remainder of this discussion, we shall use a simple Web service which supports a single operation that has an equivalent signature to the following C# code:

```
double GetStockQuote(string symbol);
```

We examine WSDL stage by stage and show how we can turn this simple method signature into a true Web service interface.

Definitions

The opening element of any WSDL document is `definitions`, which is the parent for all other elements in the WSDL document. As well as acting as a container, the `definitions` element is also the place where global namespace declarations are placed.

```
<wsdl:definitions
  targetNamespace="http://stock.example.org/wsdl"
  xmlns:tns="http://stock.example.org/wsdl"
  xmlns:stockQ="http://stock.example.org/schema"
  xmlns:wsdl="http://www.w3.org/2003/02/wsdl">
  <!-- Remainder of WSDL description omitted -->
</wsdl:definitions>
```

Figure 3-26 The WSDL `definitions` element.

A typical WSDL `definitions` element takes the form shown in Figure 3-26, where the element declares the target namespace of the document, a corresponding prefix for that namespace, and a namespace prefix for the WSDL namespace (or alternatively it is also common to use the WSDL namespace as the default namespace for the whole document). Other

namespaces may also be declared at this scope, or may be declared locally to their use within the rest of the document. Good practice for declaring namespaces to WSDL documents is to ensure the namespaces that are required for the abstract parts of the document are declared at this level, while namespaces required for the concrete parts of a WSDL document (like the bindings section) are declared locally to make factoring and management of WSDL documents easier.

The Types Element

The `types` element is where types used in the interface description are defined, usually in XML Schema types, since XML Schema is the recommended schema language for WSDL. For instance in our simple stock quote Web service, we define types that represent traded stocks and advertise those types as part of its WSDL interface as illustrated in Figure 3-27.

```
<wsdl:definitions … >
  <wsdl:import namespace="http://stock.example.org/schema"
    location="http://stock.example.org/schema"/>
  <wsdl:types xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="stock-quote">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="symbol" ref="stockQ:symbol"/>
          <xs:element name="lastPrice" ref="stockQ:price"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <!-- Other schema type definitions -->
  <wsdl:types>
</wsdl:definitions>
```

Figure 3-27 Defining `types` in a WSDL interface.

Before writing the `types` section, we first `import` some types declared by an external schema that make the types within that schema available to this WSDL document to build on. Those schema types (symbol and price) are used to create a new complex type (`stock-price`) which the WSDL interface will use to advertise its associated Web service.

The orthodox view is to use XML Schema to provide constraints and type information in Web services-based applications. However it is not necessarily the case that XML Schema is the right choice for every application domain, particularly those domains that have already chosen a different schema language on which to base their interoperability infrastructure. Recognizing this requirement, WSDL 1.2 supports the notion of other schema languages being used in place of the recommended XML Schema. Although the WSDL 1.2 specification does not provide as wide coverage for other schema languages, it does allow for their use within WSDL interfaces.

Message Elements

Once we have our types, we can move on to the business of specifying exactly how consumers can interact with the Web service. The `message` declarations compose the types that we have defined (and those that we are borrowing from other schemas) into the expected input, output and fault messages that the Web service will consume and produce. If we take our simple stock ticker Web service as an example, we can imagine a number of messages the Web service would be expected to exchange as shown in Figure 3-28.

```
<wsdl:message name="StockPriceRequestMessage">
  <wsdl:part name="symbol" element="stockQ:symbol"/>
</wsdl:message>
<wsdl:message name="StockPriceRespnseMessage">
  <wsdl:part name="price" element="stockQ:StockPriceType"/>
</wsdl:message>
<wsdl:message name="StockSymbolNotFoundMessage">
  <wsdl:part name="symbol" element="stockQ:symbol"/>
</wsdl:message>
```

Figure 3-28 The `message` elements.

As we see in Figure 3-28, a WSDL `message` declaration describes the (abstract) form of a message that a Web service sends or receives. Each message is constructed from a number of (XML Schema) typed `part` elements—which can come from the `types` part of the description or an external schema that has been `import`ed into the same WSDL document—and each `part` is given a `name` to ease the insertion and extraction of particular information from a message. The `name` given to a `part` is unconstrained by WSDL but it is good practice to make the part name descriptive as one would when naming programming language variables.

In this example we have three possible messages: `StockPriceRequestMessage`, `StockPriceResponseMessage`, and `StockSymbolNotFoundMessage`, each of which carries some information having to do with stock prices and, because it is good practice to do so, whose name is indicative of its eventual use in the Web service.

PortType Elements

A `portType` defines a collection of `operations` within the WSDL document. Each operation within a `portType` combines `input`, `output`, and `fault` messages taken from a set of messages like those defined in Figure 3-28.

In the example shown in Figure 3-29, the `StockBrokerQueryPortType` declares an `operation` called `GetStockPrice` which is designed to allow users' systems to ask for the price of a particular equity.

The input to this operation is provided by a `StockPriceRequestMessage` message. The contents of this message are understood by the implementing Web service, which formu-

```
<wsdl:portType name="StockBrokerQueryPortType">
  <wsdl:operation name="GetStockPrice">
    <wsdl:input message="tns:StockPriceRequestMessage"/>
    <wsdl:output message="tns:StockPriceResponseMessage"/>
    <wsdl:fault name="UnknownSymbolFault"
      message="tns:StockSymbolNotFoundMessage"/>
</wsdl:portType>
```

Figure 3-29 Defining `portType` elements.

lates a response in an `output` `StockPriceRequestMessage` message that contains the details of the stock price for the equity requested.

Any exceptional behavior is returned to the caller through a fault called `UnknownSymbolFault` which is comprised from a `StockSymbolNotFoundMessage` message. Note that `portType` fault declarations have an additional `name` attribute compared to input and output messages, which is used to distinguish the individual faults from the set of possible faults that an operation can support.

Of course not all operations are so orthodox with a single input, output, and fault message, and so we have a variety of possible message exchange patterns described by the `operation` declarations within a `portType`, as follows:

- Input-Output: When the input message is sent to the service, either the output message is generated or one of the fault messages listed is generated instead.
- Input only: When a message is sent to the service, the service consumes it but does not produce any output message or fault. As such no output message or fault declarations are permitted in an operation of this type.
- Output-Input: The service generates the output message and in return the input message or one of the fault messages must be sent back.
- Output-only: The service will generate an output message, but does not expect anything in return. Fault messages are not allowed in this case.

Note that WSDL 1.2 changes the syntax of the `portType` declaration, renaming it `interface`. It also supports a useful new feature in the form of the `extends` attribute, which allows multiple `interface` declarations to be aggregated together and further extended to produce a completely new `interface`. For example, consider the situation where our simple stock Web service needs to evolve to support basic trading activities in addition to providing stock quotes. Using the `extends` mechanism, a new `interface` can be created which possesses all of the operations from each `interface` that it extends, plus any additional operations the developer chooses to add to it as exemplified in Figure 3-30.

The only potential pitfall when extending an `interface` is where names clash. For instance, an extending `interface` should take care not to call its operations by the same name

```
<wsdl:message name="BuyStockRequestMessage">
  <wsdl:part name="symbol" element="stockQ:symbol"/>
  <wsdl:part name="amount" element="xs:positiveInteger"/>
  <wsdl:part name="bid" element="stockQ:StockPriceType"/>
</wsdl:message>
<wsdl:message name="BuyStockResponseMessage">
  <wsdl:part name="symbol" element="stockQ:symbol"/>
  <wsdl:part name="amount" element="xs:positiveInteger"/>
  <wsdl:part name="price" element="stockQ:StockPriceType"/>
</wsdl:message>
<wsdl:message name="BidRejectedMessage">
  <wsdl:part name="symbol" element="stockQ:symbol"/>
  <wsdl:part name="amount" element="xs:positiveInteger"/>
  <wsdl:part name="bid" element="stockQ:StockPriceType"/>
  <wsdl:part name="asking" element="stockQ:StockPriceType"/>
</wsdl:message>

<wsdl:interface name="StockBrokerQueryPurchaseInterface"
  extends="tns:StockBrokerQueryInterface" >
  <wsdl:operation name="BuyStock">
    <wsdl:input message="tns:BuyStockRequestMessage"/>
    <wsdl:output message="tns:BuyStockRequestMessage"/>
    <wsdl:fault name="UnknownSymbolFault"
      message="tns:StockSymbolNotFoundMessage"/>
    <wsdl:fault name="BidRejectedFault"
      message="tns:BidRejectedMessage"/>
</wsdl:interface>
```

Figure 3-30 Extending `interface` definitions.

as operations from any `interface` that it extends unless the operations are equivalent. Furthermore, the designer of a new `interface` that extends multiple existing `interface` declarations must take care to see that there are no name clashes between any of the `interface` declarations as well as with the newly created `interface`.

Bindings

The `bindings` element draws together the `portType` and `operation` elements into a form suitable for exposing to the network. Bindings contain information that dictates how the format of the abstract messages is mapped onto the features of a particular network-level protocol.

While WSDL supports bindings for a number of protocols including HTTP GET and POST, and MIME, we are primarily interested in the SOAP binding for our simple stock quote `portType` from Figure 3-29, which is presented in Figure 3-31.

```
<wsdl:binding name="StockBrokerServiceSOAPBinding"
  type="tns:StockBrokerQueryPortType">
<soap:binding  styleDefault="document"
  transport="http://www.w3.org/2002/12/soap/bindings/HTTP/"
  encodingStyleDefault="http://stock.example.org/schema" />
  <wsdl:operation name="GetStockPrice">
    <soap:operation
      soapAction="http://stock.example.org/getStockPrice"/>
    <wsdl:input>
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal"/>
    </wsdl:output>
    <wsdl:fault>
      <soap:fault name="StockSymbolNotFoundMessage"/>
    </wsdl:fault>
  </wsdl:operation>
</wsdl:binding>
```

Figure 3-31 A SOAP binding.

The binding shown in Figure 3-31 binds the abstract `portType` defined in Figure 3-29 to the SOAP. It states how each of the message components of the operation defined in the `StockBrokerQueryPortType` is mapped onto its SOAP equivalent.

Starting from the top, we see a name for the binding (which is later used to tie a binding to a physical network endpoint) and the `portType` for which this binding is specified.

We then use elements from the WSDL SOAP binding specification to declare a binding for SOAP document-style exchanges, which is expressed as the default mode for this binding through the `styleDefault="document"` attribute. The encoding of the documents exchanged is defined by the stock broker schema `encodingStyleDefault="http://stock.example.org/schema"`. The fact that the service uses document-style SOAP and has its own schema means that it is a document-literal Web service.

Finall,y we see that the binding is for SOAP over the HTTP protocol as specified by the `transport="http://www.w3.org/2002/12/soap/bindings/HTTP/"` attribute. Each of these options is set as the default for the entire binding though both the style and encoding can be changed, if necessary, on a per-message basis.

This binding contains a single operation, namely `GetStockPrice`, which maps each of the input, output, and fault elements of the `GetStockPrice` operation from the `StockBrokerQueryPortType` to its SOAP on-the-wire format. The `soapAction` part of the operation binding is used to specify the HTTP SOAPAction header, which in turn can be used by SOAP servers as an indication of the action that should be taken by the receipt of the message at runtime—which usually captures the name of a method to invoke in a service implementation.

The soap:body elements for both the wsdl:input and wsdl:output elements provide information on how to extract or assemble the different messages inside the SOAP body. Since we have chosen literal encoding and document style for our messages (via the use="literal" and styleDefault="document" attribute), each part of a corresponding message is simply placed as a child of the soap:body element of the SOAP envelope. Had we been using RPC-style SOAP, then the direct child of the soap:body would be an element with the same name as the operation, with each message part as a child, as per SOAP RPC style, as contrasted with document style in Figure 3-32.[7]

```
<!-- RPC style -->
<soap:body>
  <GetStockPrice xmlns:gsp="http://stock.example.org/wsdl"
    xmlns:stockQ="http://stock.example.org/schema">
    <stockQ:symbol>MSFT</stockQ:symbol>
  </GetStockPrice>
</soap:body>

<!-- Document style -->
<soap:body>
  <stockQ:symbol
    xmlns:stockQ="http://stock.example.org/schema">
      MSFT
  </stockQ:symbol>
</soap:body>
```

Figure 3-32 Example SOAP RPC-style "Wrapping" element.

> Note that the WS-I basic profile has mandated that only messages defined with element can be used to create document-oriented Web services, and messages defined with type cannot.

Of course, the value of SOAP is not only that it provides a platform-neutral messaging format, but the fact that the mechanism is extensible through headers. To be of use in describing SOAP headers, the WSDL SOAP binding has facilities for describing header content and behavior. For example, imagine that the query operation for which we have already designed a SOAP binding in Figure 3-31 evolves such that only registered users can access the service and must authenticate by providing some credentials in a SOAP header block as part of an invocation. The WSDL interface for the service obviously needs to advertise this fact to users' applications or no one will be able to access the service.

7. Note: this is not SOAP-encoded, just RPC-style (i.e., wrapped in an element that is named indicatively of the method that the message should be dispatched to).

The WSDL fragment shown in Figure 3-33 presents a hypothetical `soap:header` declaration within the `wsdl:input` element which mandates that a header matching the same namespace as the `userID` message (as declared earlier in the document) is present, and will be consumed by the ultimate receiver of the incoming SOAP message.

```
<wsdl:message name="UserID"
  targetNamespace="http://security.example.org/user">
  <wsdl:part name="signature" type="xs:string"/>
  <wsdl:part name="session" type="xs:anyURI"/>
</wsdl:message>

<wsdl:input>
  <soap:body use="literal"/>
  <soap:header use="literal" message="tns:UserIDMessage"/>
</wsdl:input>

<wsdl:output
  xmlns:sec="http://security.example.org/user">
  <soap:body use="literal"/>
  <soap:headerfault message="sec:UserID" part="signature"/>
</wsdl:output>
```

Figure 3-33 Describing SOAP headers.

Correspondingly, a `soap:headerfault` element is present in the `wsdl:output` element to report back on any faults that occurred while processing the incoming header. If a fault does occur while processing the header, this `soap:headerfault` element identifies the user's signature that caused the problem. This information, which amounts to a "user unknown" response, can then be used at the client end to perhaps prompt the end user to re-enter a pass phrase.

Note that an error such as an incorrect signature is propagated back through the header mechanism and not through the body, since the SOAP specification mandates that errors pertaining to headers must be reported likewise through header blocks.

Services

The services element finally binds the Web service to a specific network-addressable location. It takes the bindings declared previously and ties them to a `port`, which is a physical network endpoint to which clients bind over the specified protocol.

Figure 3-34 shows a service description for our stockbroker example. It declares a service called `StockBrokerService`, which it defines in terms of a port called `StockBrokerServiceSOAPPort`. The port is itself defined in terms of the `StockBrokerServiceSOAPBinding` binding, which we saw in Figure 3-31, and is exposed to the network at the address `http://stock.example.org/` to be made accessible through the endpoint specified at the `soap:address` element.

```
<wsdl:service name="StockBrokerService">
  <wsdl:port name="StockBrokerServiceSOAPPort"
   binding="tns:StockBrokerServiceSOAPBinding">
    <soap:address
        location="http://stock.example.org/"/>
  </wsdl:port>
</wsdl:service>
```

Figure 3-34 A `service` element declaration.

Managing WSDL Descriptions

While the service element is the final piece in the puzzle as far as an individual WSDL document goes, that's not quite the end of the story. For simple one-off Web services, we may choose to have a single WSDL document that combines both concrete and abstract parts of the interface. However, for more complex deployments we may choose to split the abstract parts into a separate file, and join that with a number of different concrete bindings and services to better suit the access pattern for those services.

For example, it may be the case that a single abstract definition (`message`, `portType`, and `operation` declarations) might need to be exposed to the network via a number of protocols, not just SOAP. It might also be the case that a single protocol endpoint might need to be replicated for quality of service reasons or perhaps even several different organizations each want to expose the same service as part of their Web service offerings. By using the WSDL import mechanism, the same abstract definition of the service functionality can be used across all of these Web services irrespective of the underlying protocol or addressing. This is shown in Figure 3-35 where MIME, HTTP, and SOAP endpoints all share the same abstract functionality yet expose that functionality to the network each in their own way. Additionally, the SOAP protocol binding has been deployed at multiple endpoints which can be within a single administrative domain or spread around the whole Internet and yet each service, by dint of the fact that they share the same abstract definitions, is equivalent.

If a WSDL description needs to include features from another WSDL description or an external XML Schema file, then the `import` mechanism is used. It behaves in a similar fashion to the XML Schema `include` feature where it can be used to include components from other WSDL descriptions. We have already seen how the WSDL import mechanism is used in Figure 3-27 where the XML Schema types from the stockbroker schema were exposed to the stock broking WSDL description, as follows:

```
<wsdl:import namespace="http://stock.example.org/schema"
  location="http://stock.example.org/schema"/>
```

The `import` feature of WSDL means that a WSDL description can leverage existing XML infrastructure—previously defined schemas for in-house documents, database schemas, existing Web services, and the like—without having to reproduce those definitions as part of its own description.

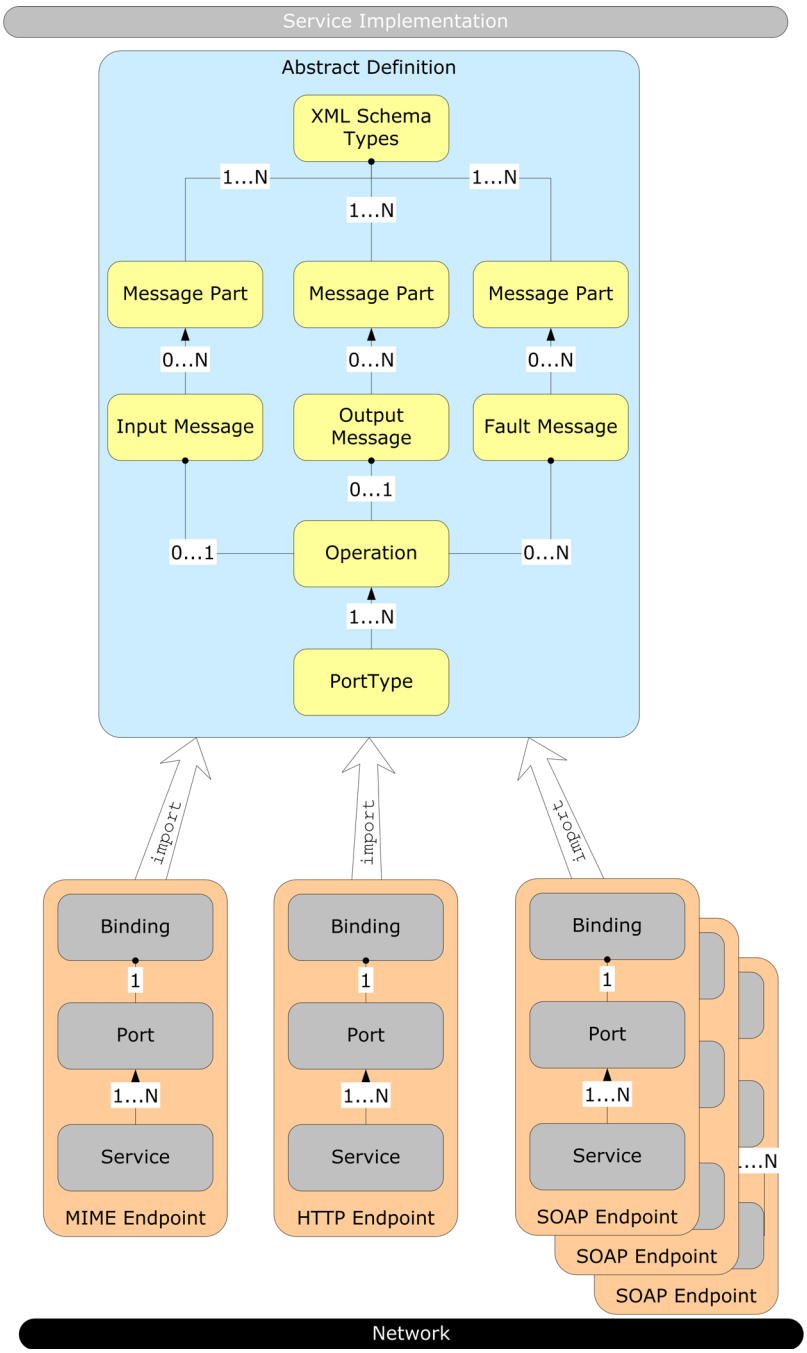


Figure 3-35 Including abstract WSDL descriptions for concrete endpoints.

Extending WSDL[8]

As Web services technology has advanced and matured, WSDL has begun to form the basis of higher-level protocols that leverage the basic building blocks that it provides, to avoid duplication of effort. Many of the technologies that we are going to examine throughout this book extend WSDL via such means to their own purpose. However, where SOAP offers header blocks as its extensibility mechanism for higher-level protocols to use, WSDL offers extension elements based on the XML Schema notion of substitution groups (see Chapter 2).

In the WSDL schema, several (abstract) global element declarations serve as the heads of substitution groups. In addition, the WSDL schema defines a base type for use by extensibility elements as a helper to ensure that the necessary substitution groups are present in any extensions. While it is outside the scope of this book to present the WSDL schema in full, there exists in the schema extensibility elements which user-defined elements can use to place themselves at *any* point within a WSDL `definition`. There are extensibility elements that allow extensions to appear at global scope, within a `service` declaration, before the `port` declaration, in a `message` element before any `part` declarations and any other point in a WSDL description, as shown in Figure 3-36.

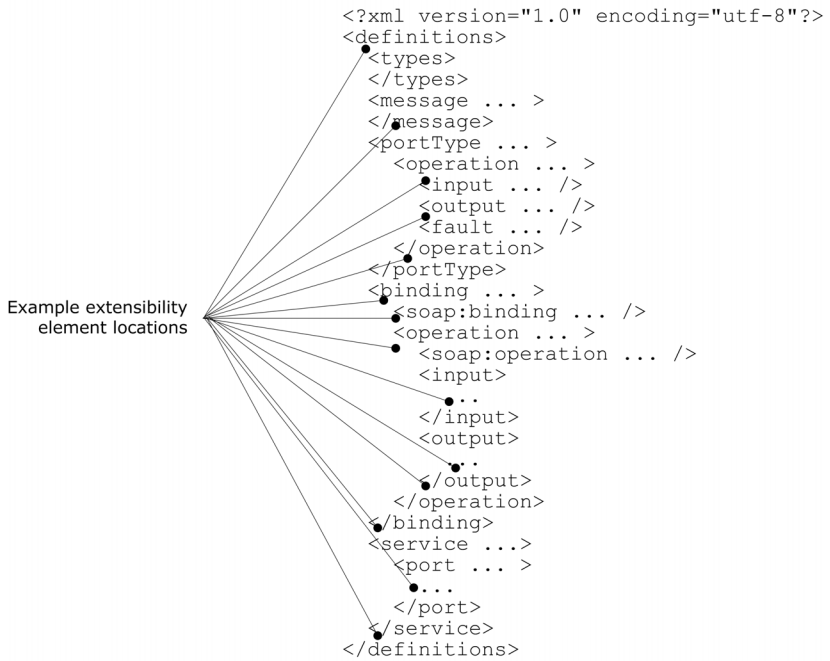


Figure 3-36 WSDL substitution group heads.

8. This section based on a draft version of the WSDL 1.2 specification.

For example, the `soap` elements that we have seen throughout the bindings section of our WDSL description are extensibility elements. In the schema for those elements, they have been declared as being part of the substitution group `bindingExt` which allows them to legally appear as part of the WSDL `bindings` section.

Additionally, third-party WSDL extensions may declare themselves as mandatory with the inclusion of a `wsdl:required` attribute in their definitions. Once a `required` attribute is set, any and all validation against an extended WSDL document must include the presence of the corresponding element as a part of the validation.

> Extensibility elements are commonly used to specify some technology-specific binding. They allow innovation in the area of network and message protocols without having to revise the base WSDL specification. WSDL recommends that specifications defining such protocols also define any necessary WSDL extensions used to describe those protocols or formats.[9]

Using SOAP and WSDL

While many of the more advanced features of the emerging Web services architecture are still being built into many of the platforms, support for SOAP and WSDL in most vendors' Web services toolkits is widespread and makes binding to and using Web services straightforward. In this section, we investigate how a typical application server and can be used to deploy our simple banking example, and how it can be later consumed by a client application. The overall architecture can be seen in Figure 3-37.

The architecture for this sample is typical of Web services applications that routinely combine a variety of platforms. In Figure 3-37, we use Microsoft's .Net and Internet Information Server to host the service implementation, but we use the Java platform and the Apache AXIS Web service toolkit to consume this service and drive the application.

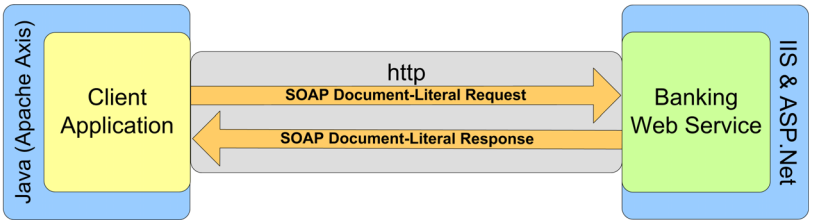


Figure 3-37 Cross-platform banking Web service example.

9. From WSDL 1.2 specification, http://www.w3.org/TR/wsdl12/.

Service Implementation and Deployment

The implementation of our banking service is a straightforward C# class, and is shown in Figure 3-38.

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Web;
using System.Web.Services;

[WebService(Namespace="http://bank.example.org")]
public class BankService : System.Web.Services.WebService
{
  [WebMethod]
  public string openAccount(string title,
                            string surname,
                            string firstname,
                            string postcode,
                            string telephone)
  {
    BankEndSystem bes = new BackEndSystem();
    string accountNumber = bes.processApplication(title,
                                                  surname,
                                                  firstname,
                                                  postcode,
                                                  telephone);
    return accountNumber;
  }
}
```

Figure 3-38 A simple bank Web service implementation.

Most of the work for this service is done by some back-end banking system, to which our service delegates the workload. Our service implementation just acts as a kind of gateway between the Web service network to which it exposes our back-end business logic, and the back-end systems themselves to which it delegates work it receives from Web services clients. This pattern is commonplace when exposing existing systems via Web services, and makes good architectural sense since the existing system does not have to be altered just to add in Web service support.

The key to building a successful Web service, even one as simple as our bank account example, is to ensure that the orthogonal issues of service functionality and deployment are kept separate. That is, do not allow the implementation of your system to change purely because you intend to expose its functionality as a Web service.

It is a useful paradigm to treat your Web services as "user interfaces" through which users (in most cases other computer systems) will interact with your business systems. In the same way that you would not dream of putting business rules or data into human user interfaces, then you should not place business rules or data into your Web service implementations. Similarly, you would not expect that a back-end business system would be updated simply to accommodate a user interface, and you should assume that such mission-critical systems should not be altered to accommodate a Web service deployment.

When deployed into our Web services platform (in this example, Microsoft's IIS with ASP.Net), the associated WSDL description of the service is generated by inspection of the implementation's interface and made available to the Web. The resultant WSDL[10] is shown in Figure 3-39.

It is important to bear in mind, that although the WSDL shown in Figure 3-39 is intricate and lengthy for a simple service, the effort required to build it is practically zero because of tool support. The only issue that this should raise in the developer's mind is that their chosen platform and tools should handle this kind of work on their behalf. WSDL should only be handcrafted where there a specific need to do something intricate and unusual that tool support would not facilitate.

Binding to and Invoking Web Services

Once the service has been deployed and its endpoint known by consumers, clients can bind to it by using their client-side Web services toolkits to create proxies. A proxy is a piece of code that sits between the client application and the network and deals with all of the business of serializing and deserializing variables from the client's program into a form suitable for network transmission and back again. The client application, therefore, never has to be aware of any network activity and is simpler to build.

10. The WDSL description generated by ASP.Net is richer than that shown here since it also includes HTTP GET and HTTP POST bindings. However, we are predominantly interested in SOAP as the Web services transport, and so the HTTP bindings have been removed.

```
<?xml version="1.0" encoding="utf-8"?>
<definitions
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:bank="http://bank.example.org"
  targetNamespace="http://bank.example.org"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <xs:schema elementFormDefault="qualified"
       targetNamespace="http://bank.example.org">
      <xs:element name="openAccount">
        <xs:complexType>
          <xs:sequence>
            <xs:element minOccurs="0" maxOccurs="1"
               name="title" type="xs:string"/>
            <xs:element minOccurs="0" maxOccurs="1"
               name="surname" type="xs:string"/>
            <xs:element minOccurs="0" maxOccurs="1"
               name="firstname" type="xs:string"/>
            <xs:element minOccurs="0" maxOccurs="1"
               name="postcode" type="xs:string"/>
            <xs:element minOccurs="0" maxOccurs="1"
               name="telephone" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="openAccountResponse">
        <xs:complexType>
          <xs:sequence>
            <xs:element minOccurs="0" maxOccurs="1"
               name="openAccountResult" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="string" nillable="true"
         type="xs:string"/>
    </xs:schema>
  </types>
  <message name="openAccountSoapIn">
    <part name="parameters" element="bank:openAccount"/>
  </message>
  <message name="openAccountSoapOut">
    <part name="parameters"
     element="bank:openAccountResponse"/>
  </message>
  <portType name="BankServiceSoap">
```

Figure 3-39 Bank service auto-generated WSDL description.

```
    <operation name="openAccount">
      <input message="bank:openAccountSoapIn"/>
      <output message="bank:openAccountSoapOut"/>
    </operation>
  </portType>
  <binding name="BankServiceSoap"
   type="bank:BankServiceSoap">
    <soap:binding
         transport="http://schemas.xmlsoap.org/soap/http"
         style="document"/>
    <operation name="openAccount">
      <soap:operation
           soapAction="http://bank.example.org/openAccount"
           style="document"/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
  <service name="BankService">
    <port name="BankServiceSoap"
     binding="bank:BankServiceSoap">
      <soap:address
          location="http://localhost/dnws/BankService.asmx"/>
    </port>
  </service>
</definitions>
```

Figure 3-39 Bank service auto-generated WSDL description (continued).

In our example, the serialization and deserialization is to SOAP from Java and back again, and is handled by a proxy generated by the Apache AXIS `WSDL2Java` tool. This tool parses WSDL at a given location and generates a proxy class which allows client code to communicate with that service. For example, the proxy code generated by this tool when it consumed our bank example service is shown in Figure 3-40.

```
/**
 * This file was auto-generated from WSDL
 * by the Apache Axis WSDL2Java emitter.
 */

package org.example.bank;
import java.lang.String;

public class BankServiceSoapStub
                extends org.apache.axis.client.Stub
                implements org.example.bank.BankServiceSoap {

  // Data members removed for brevity

  public BankServiceSoapStub()
                          throws org.apache.axis.AxisFault {
     this(null);
  }

  // Other constructors removed for brevity

  private org.apache.axis.client.Call createCall()
                            throws java.rmi.RemoteException {
      // Implementation removed for brevity
      return _call;
    }
    catch (java.lang.Throwable t) {
      throw new org.apache.axis.AxisFault("Failure trying" +
                               " to get the Call object", t);
    }
  }

  public String openAccount(String title, String surname,
                            String firstname,
                            String postcode,
                            String telephone)
                            throws java.rmi.RemoteException {
     // Implementation removed for brevity
  }
}
```

Figure 3-40 Apache AXIS auto-generated proxy for the bank Web service.

The proxy class shown in Figure 3-40 allows the client of the Web service to call its functionality with a call as simple as the likes of:

```
bankAccountService.openAccount("Mr", "Aneurin", "Bevan",
                               "ABC 123", "0207 123 4567")
```

without having to worry about the fact that on the wire, the proxy has sent a SOAP message that looks like that shown in Figure 3-41 below:

```
<?xml version="1.0" encoding="utf-8" ?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/
envelope/" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soap:Body>
    <openAccount xmlns="http://bank.example.org">
      <title>Mr</title>
      <surname>Bevan</surname>
      <firstname>Aneurin</firstname>
      <postcode>ABC 123</postcode>
      <telephone>0207 123 4567</telephone>
    </openAccount>
  </soap:Body>
</soap:Envelope>
```

Figure 3-41 Proxy generated SOAP message.

At the receiving end, the bank service's SOAP server will retrieve this SOAP from the network and turn it into something meaningful (in our case C# objects) before passing it to the service implementation. The service implementation grinds away at its task, producing some result in its own proprietary format before passing it back to the underlying Web services platform to serialize its results into the appropriate network format (i.e., a SOAP message) and return it to the caller. At this point the service invocation has finished and the resources used during the execution of that service can be freed.

Where's the Hard Work?

For simple interactions, there isn't any hard work for the developer to do because SOAP toolkits are sufficiently advanced enough to automate this. For example, we didn't have to worry about the style of SOAP encoding or how the marshalling occurred in any of our bank account examples, even though we crossed networks, servers, and even languages and platforms.

Though it may seem from these examples that Web services is an automation utopia, it is not. While it is true that for the majority of cases, simple interactions can be automated (though auto-generation of WSDL from service implementation code and auto-generation of proxies from WSDL descriptions), this is about as far as toolkits have advanced.

Given that this book extends beyond this third chapter, it is safe to assume that we're going to have to roll up our shirt sleeves at some point and patch the gaps that the current set of Web services toolkits inevitably leaves. It is in these subsequent chapters where we will find the hard work!

Summary

SOAP is the protocol that Web services use to communicate. It is an XML-based protocol that specifies a container called an Envelope, which stores application payload in a second container, called the Body, and additional (usually contextual) information inside a third container called the Header. The SOAP specification describes a processing model where application messages (and their associated headers) can pass through intermediary processing nodes between the sender and receiver, where the information stored in the SOAP header blocks can be used by those intermediaries to provide various quality of service characteristics. For example, the headers may contain routing information, transaction context, security credentials, or any other protocol information.

WSDL is an interface description language for Web services and like SOAP, WSDL is currently popularized by its 1.1 version, which is due to be superseded by WSDL 1.2. A WSDL interface is composed from a number of elements, each building on the previous, from simple type and message declarations, culminating in a network addressable entity which uses the defined types and messages to expose operations onto the Web.

Though SOAP and WSDL are undoubtedly important protocols in their own right, when drawn together through tool support, their potential is significantly enhanced. Web services toolkits can consume the WSDL offered by a service and automatically generate the code to deal with messages in the format that the service expects, while providing a straightforward API to the developer.

Architect's Note

- SOAP 1.1 is the most widely adopted version of the SOAP specification. However, SOAP 1.2 has now reached W3C recommendation status and thus SOAP 1.1 is now considered deprecated.
- SOAP RPC is quick and easy, but may lead to applications with too tight a level of coupling. Exchanging larger documents is preferable, even if it means writing handler code to deal with them.
- XML-Native applications should not use SOAP-RPC; they should use the XML vocabularies that they have already developed, and use those vocabularies as the basis of their communication via document-oriented SOAP.
- Be prepared for a shift in the Web services architecture, and ensure your services can support "Web-friendly" access where appropriate.
- Do not deploy a Web service without its WSDL description—a service is naked without it.
- Use tool support—it is wasted effort to do for yourself what a tool can do more easily, more quickly, and more accurately.