Simple Object Access Protocol (SOAP)

IN THIS CHAPTER

- Evolution of XML Protocols 115
- Simple Object Access Protocol (SOAP) 117
- Doing Business with SkatesTown 121
- Inventory Check Web Service 125
- SOAP Envelope Framework 133
- Taking Advantage of SOAP Extensibility 137
- SOAP Intermediaries 145
- Error Handling in SOAP 151
- SOAP Data Encoding 153
- Architecting Distributed Systems with Web Services 166
- Purchase Order Submission Web Service 175
- SOAP Protocol Bindings 187
There is a lot more to Web services than Simple Object Access Protocol (SOAP). Chapter 1, “Web Services Overview,” introduced the Web services interoperability stack that went several levels higher than SOAP. SOAP is synonymous with Web services, however, because since its introduction in late 1999, it has become the de facto standard for Web services messaging and invocation. With competitive and market pressures driving the Web services industry in a hard race to provide meaningful solutions to cross-enterprise integration problems, SOAP is the go-to-market technology of choice.

What is SOAP all about, you ask? Will it save you from failure (and keep you clean) while you toil 80-hour work weeks on a business-to-business (B2B) integration project from hell? Will it support your extensibility needs as requirements change, and provide you with interoperability across multi-vendor offerings? Will it be the keyword on your resume that will guarantee you a big raise as you switch jobs? In short, is it the new new thing? Well, maybe.

SOAP is so simple and so flexible that it can be used in many different ways to fit the needs of different Web service scenarios. This is both a blessing and a curse. It is a blessing because chances are that SOAP can fit your needs. It is a curse because you probably won’t know how to make it do that. This is where this chapter comes in. When you are through with it, you will know not only how to use SOAP straight out of the box, but also how to extend SOAP in multiple ways to support your diverse and changing needs. You will also have applied design best practices to build several meaningful e-commerce Web services for our favorite company, SkatesTown. Last but not least, you will be ready to handle the rest of the book and climb still higher toward the top of the Web services interoperability stack. To this end, the chapter will discuss the following topics:

- The evolution of XML protocols and the history and motivation behind SOAP’s creation
- The SOAP envelope framework, complete with discussions of versioning, header-based vertical extensibility, intermediary-based horizontal extensibility, error handling, and bindings to multiple transport protocols
- The various mechanisms for packaging information in SOAP messages, including SOAP’s own data-encoding rules and a number of heuristics for putting just about any kind of data in SOAP messages
- The use of SOAP within multiple distributed system architectures such as RPC- and messaging-based systems in all their flavors
- Building and consuming Web services using the Java-based Apache Axis Web services engine

One final note before we begin. The SOAP 1.1 specification is slightly over 40 pages long. This chapter is noticeably longer, because the purpose of this book is to be something more than an annotated spec or a tutorial for building Web services. We’ve tried hard to create a
thorough treatment of Web services for people who want answers to questions that begin not
only with “what” and “how” but also with “why.” To become an expert at Web services, you
need to be comfortable dealing with the latter type of questions. We are here to help.

So, why SOAP? As this chapter will show, SOAP is simple, flexible, and highly extensible.
Because it is XML based, SOAP is programming language, platform, and hardware neutral.
What better choice for the XML protocol that is the foundation of Web services? To prove this
point, let’s start the chapter by looking at some of the earlier work that inspired SOAP.

Evolution of XML Protocols

The enabling technology behind Web services is built around XML protocols. XML protocols
govern how communication happens and how data is represented in XML format on the
wire. XML protocols can be broadly classified into two generations. First-generation protocols
are based purely on XML 1.0. Second-generation protocols take advantage of both XML
Namespaces and XML Schema. SOAP is a second-generation XML protocol.

First-Generation XML Protocols

There were many interesting first-generation XML protocol efforts. They informed the com-
munity of important protocol requirements and particular approaches to satisfying these
requirements. Unfortunately, very few of the first-generation XML protocols achieved multi-
vendor support and broad adoption. Two are worth mentioning: Web Distributed Data
Exchange (WDDX) and XML-RPC.

WDDX

WDDX provides a language- and platform-neutral mechanism for data exchange between
applications. WDDX is perfect for data syndication and remote B2B integration APIs because
it is all about representing data as XML. For example, Moreover Technologies, the Web feed
company, exposes all its content through a WDDX-based remote API. Access http://
mindle.com/cgi-local/page?index+wddx with an XML-aware browser such as Internet
Explorer and you will get a WDDX packet with current headline news. A simplified version of
the packet is shown in the following example. You can see from it that the data format is a
recordset (tabular data) with three fields containing the URL to the full article, its headline
text, and the publishing source:

```
<wddxPacket version="1.0">
  <header/>
  <data>
    <recordset rowCount="2" fieldNames="url,headline_text,source">
      <field name="url">
        <string>http://c.mindle.com/click/here.pl?x22535276</string>
      </field>
      <field name="headline_text">
        <string>Latest headline news</string>
      </field>
      <field name="source">
        <string>Moreover Technologies</string>
      </field>
    </recordset>
  </data>
</wddxPacket>
```
Allaire Corporation (now Macromedia, Inc.) created WDDX in 1998. WDDX is currently supported in many environments and is flexible enough to handle most useful datatypes (strings, numbers, booleans, date/time, binary, arrays, structures, and recordsets), but it cannot represent arbitrary data in XML. It is an epitome of the 80/20 rule: flexible enough to be useful yet simple enough to be broadly supported. Because WDDX is not bound to any particular transport, applications can exchange WDDX packets via HTTP, over e-mail, or by any other means. Many applications persist data as XML in a relational database using WDDX.

**XML-RPC**

XML-RPC is an RPC protocol introduced in the market in 1998 by Userland. XML-RPC supports a set of datatypes similar to that supported by WDDX and uses HTTP as the underlying transport protocol. Because of its simplicity, XML-RPC enjoyed good multi-vendor support. Here’s an example XML-RPC method call and response:

```xml
<methodCall>
  <methodName>NumberToText</methodName>
  <params>
    <param>
      <value><i4>28</i4></value>
    </param>
  </params>
</methodCall>

... 

<methodResponse>
  <params>
    <param>
      <value><string>twenty-eight</string></value>
    </param>
  </params>
</methodResponse>
```
First-Generation Problems

Although first-generation XML protocols have been and still are very useful, their simplicity and reliance on XML 1.0 alone causes some problems.

First-generation protocols are not very extensible. The protocol architects had to reach agreement before any changes were implemented, and the protocol version had to be revved up in order to let tools distinguish new protocol versions from old ones and handle the XML appropriately. For example, when XML-RPC and WDDX added support for binary data, both protocols had to update their specifications, and the protocol implementations on all different languages and platforms supporting the protocols had to be updated. The overhead of constantly revising specifications and deploying updated tools for handling the latest versions of the protocols imposed limits on the speed and scope of adoption of first-generation protocols. Second-generation protocols address the issue of extensibility with XML namespaces.

The second problem with first-generation protocols had to do with datatyping. First-generation XML protocols stuck to a single Document Type Definition (DTD) to describe the representation of serialized data in XML. In general, they used just a few XML elements. This approach made building tools supporting these protocols relatively easy. The trouble with such an approach is that the XML describing the data in protocol messages expressed datatype information and not semantic information. In other words, to gain the ability to represent data in XML, first-generation XML protocols went without the ability to preserve information about the meaning of the data. Second-generation XML protocols use XML schema as a mechanism to combine descriptive syntax with datatype information.

To sum things up, the need to provide broad extensibility without centralized standardization and the need to combine datatype information with semantic information were the driving forces behind the effort to improve upon first-generation efforts and to create SOAP, the de facto standard XML protocol for modern Web services and B2B applications.

Simple Object Access Protocol (SOAP)

This section looks at the history, design center, and core capabilities of SOAP as a means for establishing the base on which to build our understanding of Web services.

The Making of SOAP

Microsoft started thinking about XML-based distributed computing in 1997. The goal was to enable applications to communicate via Remote Procedure Calls (RPCs) on top of HTTP. DevelopMentor and Userland joined the discussions. The name SOAP was coined in early 1998. Things moved forward, but as the group tried to involve wider circles at Microsoft, politics stepped in and the process was stalled. The DCOM camp at the company disliked the idea
of SOAP and believed that Microsoft should use its dominant position in the market to push the DCOM wire protocol via some form of HTTP tunneling instead of pursuing XML. Some XML-focused folks at Microsoft believed that the SOAP idea was good but that it had come too early. Perhaps they were looking for some of the advanced facilities that could be provided by XML Schema and Namespaces. Frustrated by the deadlock, Userland went public with a cut of the spec published as XML-RPC in the summer of 1998.

In 1999, as Microsoft was working on its version of XML Schema (XML Data) and adding support for namespaces in its XML products, the idea of SOAP gained additional momentum. It was still an XML-based RPC mechanism, however. That’s why it met with resistance from the BizTalk (http://www.biztalk.org) team. The BizTalk model was based more on messaging than RPCs. It took people a few months to resolve their differences. SOAP 0.9 appeared for public review on September 13, 1999. It was submitted to the IETF as an Internet public draft. With few changes, in December 1999, SOAP 1.0 came to life.

On May 8, 2000 SOAP 1.1 was submitted as a Note to the World Wide Web Consortium (W3C) with IBM as a co-author—an unexpected and refreshing change. In addition, the SOAP 1.1 spec was much more extensible, eliminating concerns that backing SOAP implied backing some Microsoft proprietary technology. This change, and the fact that IBM immediately released a Java SOAP implementation that was subsequently donated to the Apache XML Project (http://xml.apache.org) for open-source development, convinced even the greatest skeptics that SOAP is something to pay attention to. Sun voiced support for SOAP and started work on integrating Web services into the J2EE platform. Not long after, many vendors and open-source projects were working on Web service implementations.

Right before the XTech 2000 Conference, the W3C made an announcement that it was looking into starting an activity in the area of XML protocols: “We’ve been under pressure from many sources, including the advisory board, to address the threat of fragmentation of and investigate the exciting opportunities in the area of XML protocols. It makes sense to address this now because the technology is still early in its evolution…” (http://lists.w3.org/Archives/Public/xml-dist-app/2000Feb/0006.html). On September 13, 2000 the XML Protocol working group at the W3C was formed to design the core XML protocol that was to become the core of XML-based distributed computing in the years to come. The group started with SOAP 1.1 as a foundation and produced the first working draft of SOAP 1.2 on July 9, 2001.

What Should SOAP Do?

SOAP claims to be a specification for a ubiquitous XML distributed computing infrastructure. It’s a nice buzzword-compliant phrase, but what does it mean? Let’s parse it bit by bit to find out what SOAP should do.
XML means that, as a second-generation XML protocol, SOAP is based on XML 1.0, XML Schema, and XML Namespaces.

_Distributed computing_ implies that SOAP can be used to enable the interoperability of remote applications (in a very broad sense of the phrase). Distributed computing is a fuzzy term and it means different things to different people and in different situations. Here are some “facets” you can use to think about a particular distributed computing scenario: the protocol stack used for communication, connection management, security, transaction support, marshalling and unmarshalling of data, protocol evolution and version management, error handling, audit trails, and so on. The requirements for different facets will vary between scenarios. For example, a stock ticker service that continuously distributes stock prices to a number of subscribers will have different needs than an e-commerce payment-processing service. The stock ticker service will probably need no support for transactions and only minimal, if any, security or audit trails (it distributes publicly available data). The e-commerce payment-processing service will require Cerberean security, heavy-duty transaction support, and full audit trails.

_Infrastructure_ implies that SOAP is aimed at low-level distributed systems developers, not developers of application/business logic or business users. Infrastructure products such as application servers become “SOAP enabled” by including a Web service engine that understands SOAP. SOAP works behind the scenes making sure your applications can interoperate without your having to worry too much about it.

_Ubiquitous_ means omnipresent, universal. On first look, it seems to be a meaningless term, thrown into the phrase to make it sound grander. It turns out, however, that this is the most important part. The ubiquity goal of SOAP is a blessing because, if SOAP-enabled systems are everywhere on the Internet, it should be easier to do distributed computing. After all, that’s what SOAP is all about. However, the ubiquity of SOAP is also a curse, because one technology specification should be able to support many different types of distributed computing scenarios, from the stock ticker service to the e-commerce payment-processing service. To meet this goal, SOAP needs to be a highly abstract and flexible technology. However, the more abstract SOAP becomes, the less support it will provide for specific distributed computing scenarios. Furthermore, greater abstraction means more risk that different SOAP implementations will fail to interoperate. This is the eternal tug-of-war between generality and specificity.

**What Is SOAP, Really?**

Like most new technologies that change the rules of how applications are being developed, Web services and SOAP have sometimes been over-hyped. Despite the hype, however, SOAP is still of great importance because it is the industry’s best effort to date to standardize on the infrastructure technology for cross-platform XML distributed computing.
Above all, SOAP is relatively simple. Historically, simplicity is a key feature of most successful architectures that have achieved mass adoption. The Web with HTTP and HTML at its core is a prime example. Simple systems are easier to describe, understand, implement, test, maintain, and evolve. At its heart, SOAP is a specification for a simple yet flexible second-generation XML protocol. SOAP 1.0 printed at about 40 pages. The text of the specification has grown since then (the authors have to make sure the specification is clear and has no holes), but the core concepts remain simple.

Because SOAP is focused on the common aspects of all distributed computing scenarios, it provides the following:

- A mechanism for defining the unit of communication. In SOAP, all information is packaged in a clearly identifiable SOAP message. This is done via a SOAP envelope that encloses all other information. A message can have a body in which potentially arbitrary XML can be used. It can also have any number of headers that encapsulate information outside the body of the message.
- A mechanism for error handling that can identify the source and cause of the error and allows for error-diagnostic information to be exchanged between participants of an interaction. This is done via the notion of a SOAP fault.
- An extensibility mechanism so that evolution is not hindered and there is no lock-in. XML, schemas, and namespaces really shine here. The two key requirements on extensions are that they can be orthogonal to other extensions and they can be introduced and used without the need for centralized registration or coordination. Typically, extensions are introduced via SOAP headers. They can be used to build more complex protocols on top of SOAP.
- A flexible mechanism for data representation that allows for the exchange of data already serialized in some format (text, XML, and so on) as well as a convention for representing abstract data structures such as programming language datatypes in an XML format.
- A convention for representing Remote Procedure Calls (RPCs) and responses as SOAP messages, because RPCs are the most common type of distributed computing interaction and because they map so well to procedural programming language constructs.
- A document-centric approach to reflect more natural document exchange models for business interactions. This is needed to support the cases in which RPCs result in interfaces that are too fine grained and, therefore, brittle.
- A binding mechanism for SOAP messages to HTTP, because HTTP is the most common communication protocol on the Internet.

Although solid consensus exists in the industry about the core capabilities of SOAP, there is considerably less agreement on how higher-level issues such as security and transaction-management should be addressed. Nearly everyone agrees that to tackle the broad spectrum of
interesting problems we are faced with, we need to work in parallel on a set of layered specifications for XML distributed computing. Indeed, many loosely coupled industry initiatives are developing standards and technologies around SOAP. Tracking these efforts is like trying to shoot at many moving targets. The authors of this book have tried our best to address the relevant efforts in this space and to provide you with up-to-date information. Chapter 1 showed how many of these efforts layered around the notion of the Web services interoperability stack. Chapter 5, “Using SOAP for e-Business,” goes into more detail about the set of standards surrounding SOAP that enable secure, robust, and scalable enterprise-grade Web services.

Now, let’s take a look at how SkatesTown is planning to use SOAP and Web services.

**Doing Business with SkatesTown**

When Al Rosen of Silver Bullet Consulting first began his engagement with SkatesTown, he focused on understanding the e-commerce practices of the company and its customers. After a series of conversations with SkatesTown’s CTO Dean Carroll, he concluded the following:

- SkatesTown’s manufacturing, inventory management, and supply chain automation systems are in good order. These systems are easily accessible by SkatesTown’s Web-centric applications.
- SkatesTown has solid consumer-oriented online presence. Product and inventory information is fed into the online catalog that is accessible to both direct consumers and SkatesTown’s reseller partners via two different sites.
- Although SkatesTown’s order processing system is sophisticated, it is poorly connected to online applications. This is a pain point for the company because SkatesTown’s partners are demanding better integration with their supply chain automation systems.
- SkatesTown’s purchase order system is solid. It accepts purchase orders in XML format and uses XML Schema-based validation to guarantee their correctness. Purchase order item stock keeping units (SKUs) and quantities are checked against the inventory management system. If all items are available, an invoice is created. SkatesTown charges a uniform 5% tax on purchases and the highest of 5% of the total purchase or $20 for shipping and handling.

Digging deeper into the order processing part of the business, Al discovered that it uses a low-tech approach that has a high labor cost and is not suitable for automation. He noticed one area that badly needed automation: the process of purchase order submission. Purchase orders are sent to SkatesTown by e-mail. All e-mails arrive in a single manager’s account in operations. The manager manually distributes the orders to several subordinates. They have to open the e-mail, copy only the XML over to the purchase order system, and enter the order there. The system writes an invoice file in XML format. This file must be opened, and the XML must be
copied and pasted into a reply e-mail message. Simple misspellings of e-mail addresses and cut-and-paste errors are common. They cost SkatesTown and its partners both money and time.

Another area that needs automation is the inventory checking process. SkatesTown’s partners used to submit purchase orders without having a clear idea whether all the items were in stock. This often caused delayed order processing. Further, purchasing personnel from the partner companies would engage in long e-mail dialogs with operations people at SkatesTown. This situation was not very efficient. To improve it, SkatesTown built a simple online application that communicates with the company’s inventory management system. Partners could log in, browse SkatesTown’s products, and check whether certain items were in stock. The application interface is shown in Figure 3.1. (You can access this application as Example 1 under Chapter 3 in the example application on this book’s Web site.) This application was a good start, but now SkatesTown’s partners are demanding the ability to have their purchasing applications directly inquire about order availability.

![Welcome to SkatesTown!](image)

**FIGURE 3.1**
*SkatesTown’s online inventory check application.*

Looking at the two areas that most needed to be improved, Al Rosen chose to focus on the inventory checking process because the business logic was already present. He just had to enable better automation. To do this, he had to better understand how the application worked.
Interacting with the Inventory System

The logic for interacting with the inventory system is very simple. Looking through the Java Server Pages (JSPs) that made up the online application, Al easily extracted the key business logic operations from /ch3/ext/inventoryCheck.jsp. Here is the process for checking SkatesTown’s inventory:

```java
import bws.BookUtil;
import com.skatestown.data.Product;
import com.skatestown.backend.ProductDB;

String sku = ...;
int quantity = ...;

ProductDB db = BookUtil.getProductDB(...);
Product   p = db.getBySKU(sku);

boolean isInStock = (p != null && p.getNumInStock() >= quantity);
```

Given a SKU and a desired product quantity, an application needs to get an instance of the SkatesTown product database and locate a product with a matching SKU. If such a product is available and if the number of items in stock is greater than or equal to the desired quantity, the inventory check succeeds. Because most of the examples in this chapter talk to the inventory system, it is good to take a deeper look at its implementation.

**NOTE**

A note of caution: this book's sample applications demonstrate realistic uses of Java technology and Web services to solve real business problems while, at the same time, remaining simple enough to fit in the book's scope and size limitations. Further, all the examples are directly accessible in many environments and on all platforms that have a JSP and servlet engine without any sophisticated installation. To meet these somewhat conflicting criteria, something has to give. For example:

- To keep the code simple, we do as little data validation and error checking as possible without allowing applications to break. You won’t find us defining custom exception types or producing long, readable error messages.
- To get away from the complexities of external system access, we use simple XML files to store data.
- To make deployment easier, we use the BookUtil class as a place to go for all operations that depend on file locations or URLs. You can tune the deployment options for the example applications by modifying some of the constants defined in BookUtil.
- All file paths are relative to the installation directory of the example application.
SkatesTown’s inventory is represented by a simple XML file stored in /resources/products.xml (see Listing 3.1). By modifying this file, you can change the behavior of many examples. The Java representation of products in SkatesTown’s systems is the com.skatestown.data.Product class. It is a simple bean that has one property for every element under product.

Listing 3.1  SkatesTown Inventory Database

```xml
<?xml version="1.0" encoding="UTF-8"?>
<products>
  <product>
    <sku>947-TI</sku>
    <name>Titanium Glider</name>
    <type>skateboard</type>
    <desc>Street-style titanium skateboard.</desc>
    <price>129.00</price>
    <inStock>36</inStock>
  </product>
  ...
</products>
```

SkatesTown’s inventory system is accessible via the ProductDB (for product database) class in package com.skatestown.backend. Listing 3.2 shows the key operations it supports. To construct an instance of the class, you pass an XML DOM Document object representation of products.xml. (BookUtil.getProductDB() does this automatically.) After that, you can get a listing of all products or you can search for a product by its SKU.

Listing 3.2  SkatesTown’s Product Database Class

```java
public class ProductDB {

  private Product[] products;

  public ProductDB(Document doc) throws Exception {
    // Load product information
  }

  public Product getBySKU(String sku) {
    Product[] list = getProducts();
    for (int i = 0; i < list.length; i++)
```
if ( sku.equals( list[i].getSKU() ) ) return( list[i] );
return( null );
}

public Product[] getProducts()
{
    return  products;
}

This was all Al Rosen needed to know to move forward with the task of automating the inventory checking process.

**Inventory Check Web Service**

SkatesTown’s inventory check Web service is very simple. The interaction model is that of an RPC. There are two input parameters: the product SKU (a string) and the quantity desired (an integer). The result is a simple boolean value—true if more than the desired quantity of the product are in stock and false otherwise.

**Choosing a Web Service Engine**

Al Rosen decided to host all of SkatesTown’s Web services on the Apache Axis Web service engine for a number of reasons:

- The open-source implementation guaranteed that SkatesTown will not experience vendor lock-in in the future. Further, if any serious problems were discovered, you could always look at the code to see what is going on.
- Axis is one of the best Java-based Web services engines. It is better architected and much faster than its Apache SOAP predecessor. The core Axis team includes some of the great Web service gurus from companies such as HP, IBM, and Macromedia.
- Axis is also probably the most extensible Web service engine. It can be tuned to support new versions of SOAP as well as the many types of extensions for which current versions of SOAP allow.
- Axis can run on top of a simple servlet engine or a full-blown J2EE application server. SkatesTown could keep its current J2EE application server without having to switch.

This combination of factors leads to an easy sell. SkatesTown’s CTO agreed to have all Web services developed on top of Axis. Al spent some time on [http://xml.apache.org/axis](http://xml.apache.org/axis) learning more about the technology and its capabilities. He learned how to install Axis on top of SkatesTown’s J2EE server by reading the Axis installation instructions.
Service Provider View

To expose the Web service, Al Rosen had to do two things: implement the service backend and deploy it into the Web service engine.

Building the backend for the inventory check Web service was simple because the logic was already available in SkatesTown’s JSP pages (see Listing 3.3).

Listing 3.3  Inventory Check Web Service Implementation

```java
import org.apache.axis.MessageContext;
import bws.BookUtil;
import com.skatestown.data.Product;
import com.skatestown.backend.ProductDB;

/**
 * Inventory check Web service
 */
public class InventoryCheck
{

/**
 * Checks inventory availability given a product SKU and
 * a desired product quantity.
 *
 * @param msgContext    This is the Axis message processing context
 *                      BookUtil needs this to extract deployment
 *                      information to load the product database.
 * @param sku           product SKU
 * @param quantity      quantity desired
 * @return              true|false based on product availability
 * @exception Exception most likely a problem accessing the DB
 */
public static boolean doCheck(MessageContext msgContext,
        String sku, int quantity)
throws Exception
{
    ProductDB db = BookUtil.getProductDB(msgContext);
    Product prod = db.getBySKU(sku);
    return (prod != null && prod.getNumInStock() >= quantity);
}
}
```

One Axis-specific feature of the implementation is that the first argument to the doCheck() method is an Axis message context object. You need the Axis context so that you can get to the product database using the BookUtil class. From inside the Axis message context, you can get
access to the servlet context of the example Web application. (Axis details such as message context are covered in Chapter 4, “Creating Web Services.”) Then you can use this context to load the product database from `resources/products.xml`. Note that this parameter will not be “visible” to the requestor of a Web service. It is something Axis will provide you with if it notices it (using Java reflection) to be the first parameter in your method. The message context parameter would not be necessary in a real-world situation where the product database would most likely be obtained via JNDI.

Deploying the Web service into Axis is trivial because Axis has the concept of a Java Web Service (JWS) file. A JWS file is a Java file stored with the `.jws` extension somewhere in the externally accessible Web applications directory structure (anywhere other than under `/WEB-INF`). JWSs are to Web services somewhat as JSPs are to servlets. When a request is made to a JWS file, Axis will automatically compile the file and invoke the Web service it provides. This is a great convenience for development and maintenance.

In this case, the code from Listing 3.3 is stored as `/ch3/ex2/InventoryCheck.jws`. This automatically makes the Web service available at the application URL `appRoot/ch3/ex2/InventoryCheck.jws`. For the example application deployed on top of Tomcat, this URL is `http://localhost:8080/bws/ch3/ex2/InventoryCheck.jws`.

**Service Requestor View**

Because SOAP is language and platform neutral, the inventory check Web service can be accessed from any programming environment that is Web services enabled. There are two different ways to access Web services, depending on whether service descriptions are available. Service descriptions use the Web Services Description Language (WSDL) to specify in detail information about Web services such as the type of data they require, the type of data they produce, where they are located, and so on. WSDL is to Web services what IDL is to COM and CORBA and what Java reflection is to Java classes. Web services that have WSDL descriptions can be accessed in the simplest possible manner. Chapter 6, “Describing Web Services,” introduces WSDL, its capabilities, and the tools that use WSDL to make Web service development and usage simpler and easier. In this chapter, we will have to do without WSDL.

Listing 3.4 shows the prototypical model for building Web service clients in the absence of a formal service description. The basic class structure is simple:

- A private member stores the URL where the service can be accessed. Of course, this property can have optional getter/setter methods.
- A simple constructor sets the target URL for the service. If the URL is well known, it can be set in a default constructor.
- There is one method for every operation exposed by the Web service. The method signature is exactly the same as the signature of the Web service operation.
**Listing 3.4** Inventory Check Web Service Client

```java
package ch3.ex2;

import org.apache.axis.client.ServiceClient;

/*
 * Inventory check web service client
 */
public class InventoryCheckClient
{
    /**
     * Service URL
     */
    private String url;

    /**
     * Point a client at a given service URL
     */
    public InventoryCheckClient(String targetUrl)
    {
        url = targetUrl;
    }

    /**
     * Invoke the inventory check web service
     */
    public boolean doCheck(String sku, int quantity) throws Exception
    {
        ServiceClient call = new ServiceClient(url);
        Boolean result = (Boolean) call.invoke(
                "",
                "doCheck",
                new Object[]{ sku, new Integer(quantity) } );
        return result.booleanValue();
    }
}
```

This approach for building Web service clients by hand insulates developers from the details of XML, the SOAP message format and protocol, and the APIs for invoking Web services using some particular client library. For example, users of `InventoryCheckClient` will never know that you have implemented the class using Axis. This is a good thing.

Chapter 4 will go into the details of the Axis API. Here we'll briefly look at what needs to happen to access the Web service. First, you need to create a `ServiceClient` object using the service URL. The service client is the abstraction used to make a Web service call. Then, you call
the `invoke()` method of the `ServiceClient`, passing in the name of the operation you are trying to invoke and an object array of the two operation parameters: a `String` for the SKU and an `Integer` for the quantity. The result will be a `Boolean` object.

That’s all there is to invoking a Web service using Axis.

**Putting the Service to the Test**

Figure 3.2 shows a simple JSP page (`/ch3/ex2/index.jsp`) that uses `InventoryCheckClient` to access SkatesTown’s Web service. You can experiment with different SKU and quantity combinations and see how SkatesTown’s SW responds. You can check the responses against the contents of the product database in `/resources/products.xml`.

![SOAP Inventory Check](image)

**SOAP Inventory Check**

Given a SKU and a desired product quantity, this example will make a SOAP call to the inventory check web service.

**FIGURE 3.2**

*Putting the SkatesTown inventory check Web service to the test.*

The inventory check example demonstrates one of the promises of Web services—you don’t have to know XML to build them or to consume them. This finding validates SOAP’s claim as an infrastructure technology. The mechanism that allows this to happen involves multiple abstraction layers (see Figure 3.3). Providers and requestors view services as Java APIs. Invoking a Web service requires one or more Java method invocations. Implementing a Web service requires implementing a Java backend (a class or an EJB, for example). The Web service view is one of SOAP messages being exchanged between the requestor and the provider.
These are both logical views in that this is not how the requestor and provider communicate. The only “real” view is the wire-level view where HTTP packets containing SOAP messages are exchanged between the requestor’s application and the provider’s Web server. The miracle of software abstraction has come to our aid once again.

**Figure 3.3**
Layering of views in Web service invocation.

### SOAP on the Wire

The powers of abstraction aside, really understanding Web services does require some knowledge of XML. Just as a highly skilled Java developer has an idea about what the JVM is doing and can use this knowledge to write higher performance applications, so must a Web service guru understand the SOAP specification and how SOAP messages are moved around between requestors and providers. This does not mean that to build or consume sophisticated or high-performance Web services you have to work with raw XML—layers can be applied to abstract your application from SOAP. However, knowledge of SOAP and the way in which a Web service engine translates Java API calls into SOAP messages and vice versa allows you to make educated decisions about how to define and implement Web services.

**TCPMon**

Luckily, the Apache Axis distribution comes with an awesome tool that can monitor the exchange of SOAP messages on the wire. The aptly named `TCPMon` tool will monitor all traffic on a given port. You can learn how to use TCPMon by looking at the examples installation section in `/bws/readme.html`.

TCPMon will either do its work as a proxy or redirect all traffic to another host and port. This ability makes TCPMon great not only for monitoring SOAP traffic but also for testing the book’s examples with a backend other than Tomcat. Figure 3.4 shows TCPMon in action on the inventory check Web service. In this case, the backend is running on the Macromedia JRun J2EE application server. By default, JRun’s servlet engine listens on port 8100, not on 8080 as Tomcat does. In the figure, TCPMon is set up to listen on 8080 but to redirect all traffic to 8100. Essentially, with TCPMon you can make JRun (or IBM WebSphere or BEA Weblogic) appear to listen on the same port as Tomcat and run the book’s examples without any changes.
The SOAP Request

Here is the information that passed on the wire as a result of the inventory check Web service request. Some irrelevant HTTP headers have been removed and the XML has been formatted for better readability but, apart from that, no substantial changes have been made:

POST /bws/inventory/InventoryCheck.jws HTTP/1.0
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: 426

SOAPAction: ""

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <SOAP-ENV:Body>
        <doCheck>
            <arg0 xsi:type="xsd:string">947-TI</arg0>
            <arg1 xsi:type="xsd:int">1</arg1>
        </doCheck>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
Later in the chapter, we will look in detail at all parts of SOAP. For now, a quick introduction will suffice.

The HTTP packet begins with the operation, a POST, and the target URL of the Web service (/bws/inventory/InventoryCheck.jws). This is how the requestor identifies the service to be invoked. The host is localhost (127.0.0.1) because you are accessing the example Web service that comes with the book from your local machine. The content MIME type of the request is text/xml. This is how SOAP must be invoked over HTTP. The content length header is automatically calculated based on the SOAP message that is part of the HTTP packet’s body. The SOAPAction header pertains to the binding of SOAP to the HTTP protocol. In some cases it might contain meaningful information. JWS-based Web service providers don’t require it, however, and that’s why it is empty.

The body of the HTTP packet contains the SOAP message describing the inventory check Web service request. The message is identified by the SOAP-ENV:Envelope element. The element has three xmlns: attributes that define three different namespaces and their associated prefixes: SOAP-ENV for the SOAP envelope namespace, xsd for XML Schema, and xsi for XML Schema instances. One other attribute, encodingStyle, specifies how data in the SOAP message will be encoded.

Inside the SOAP-ENV:Envelope element is a SOAP-ENV:Body element. The body of the SOAP message contains the real information about the Web service request. In this case, this element has the same name as the method on the Web service that you want to invoke—doCheck(). You can see that the Axis ServiceClient object auto-generated element names—arg0 and arg1—to hold the parameters passed to the method. This is fine, because no external schema or service description specifies how requests to the inventory check service should be made. In lieu of anything like that, Axis has to do its best in an attempt to make the call. Both parameter elements contain self-describing data. Axis introspected the Java types for the parameters and emitted xsi:type attributes, mapping these to XML Schema types. The SKU is a java.lang.String and is therefore mapped to xsd:string, and the quantity is a java.lang.Integer and is therefore mapped to xsd:int. The net result is that, even without a detailed schema or service description, the SOAP message contains enough information to guarantee successful invocation.

The SOAP Response
Here is the HTTP response that came back from Axis:

HTTP/1.0 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: 426

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
The HTTP response code is 200 OK because the service invocation completed successfully. The content type is also text/xml. The SOAP message for the response is structured in an identical manner to the one for the request. Inside the SOAP body is the element doCheckResponse. Axis has taken the element name of the operation to invoke and added Response to it. The element contained within uses the same pattern but with Result appended to indicate that the content of the element is the result of the operation. Again, Axis uses xsi:type to make the message’s data self-describing. This is how the service client knows that the result is a boolean. Otherwise, you couldn’t have cast the result of call.invoke() to a java.lang.Boolean in Listing 3.4.

If the messages seem relatively simple, it is because SOAP is designed with simplicity in mind. Of course, as always, some complexity lurks in the details. The next several sections will take an in-depth look at SOAP in an attempt to uncover and explain all that you need to know about SOAP to become a skilled and successful Web service developer and user.

**SOAP Envelope Framework**

The most important part that SOAP specifies is the envelope framework. Although it consists of just a few XML elements, it provides the structure and extensibility mechanisms that make SOAP so well suited as the foundation for all XML-based distributed computing. The SOAP envelope framework defines a mechanism for identifying what information is in a message, who should deal with the information, and whether this is optional or mandatory. A SOAP message consists of a mandatory envelope wrapping any number of optional headers and a mandatory body. These concepts are discussed in turn in the following sections.

**SOAP Envelope**

SOAP messages are XML documents that define a unit of communication in a distributed environment. The root element of the SOAP message is the Envelope element. In SOAP 1.1, this element falls under the http://schemas.xmlsoap.org/soap/envelope/ namespace. Because the Envelope element is uniquely identified by its namespace, it allows processing tools to immediately determine whether a given XML document is a SOAP message.
This certainly is convenient, but what do you trade off for this capability? The biggest thing you have to sacrifice is the ability to send arbitrary XML documents and perform simple schema validation on them. True, you can embed arbitrary XML inside the SOAP Body element, but naïve validation will fail when it encounters the Envelope element at the top of the document instead of the top document element of your schema. The lesson is that for seamless validation of arbitrary XML inside SOAP messages, you must integrate XML validation with the Web services engine. In most cases, the Web services engine will have to separate SOAP-specific from application-specific XML before validation can take place.

The SOAP envelope can contain an optional Header element and a mandatory Body element. Any number of other XML elements can follow the Body element. This extensibility feature helps with the encoding of data in SOAP messages. We’ll discuss it later in this chapter in the section “SOAP Data Encoding Rules.”

**SOAP Versioning**

One interesting note about SOAP is that the Envelope element does not expose any explicit protocol version, in the style of other protocols such as HTTP (HTTP/1.0 vs. HTTP/1.1) or WDDX (<wddxPacket version="1.0"> ... </wddxPacket>). The designers of SOAP explicitly made this choice because experience had shown simple number-based versioning to be fragile. Further, across protocols, there were no consistent rules for determining what changes in major versus minor version numbers truly mean. Instead of going this way, SOAP leverages the capabilities of XML namespaces and defines the protocol version to be the URI of the SOAP envelope namespace. As a result, the only meaningful statement that you can make about SOAP versions is that they are the same or different. It is no longer possible to talk about compatible versus incompatible changes to the protocol.

What does this mean for Web service engines? It gives them a choice of how to treat SOAP messages that have a version other than the one the engine is best suited for processing. Because an engine supporting a later version of SOAP will know about all previous versions of the specification, it has a range of options based on the namespace of the incoming SOAP message:

- If the message version is the same as any version the engine knows how to process, the engine can just process the message.
- If the message version is older than any version the engine knows how to process, the engine can do one of two things: generate a version mismatch error and/or attempt to negotiate the protocol version with the client by sending some information regarding the versions that it can accept.
• If the message version is newer than any version the engine knows how to process, the engine can choose to attempt processing the message anyway (typically not a good choice) or it can go the way of a version mismatch error combined with some information about the versions it understands.

All in all, the simple versioning based on the namespace URI results in the fairly flexible and accommodating behavior of Web service engines.

SOAP Headers

Headers are the primary extensibility mechanism in SOAP. They provide the means by which additional facets can be added to SOAP-based protocols. Headers define a very elegant yet simple mechanism to extend SOAP messages in a decentralized manner. Typical areas where headers get involved are authentication and authorization, transaction management, payment processing, tracing and auditing, and so on. Another way to think about this is that you would pass via headers any information orthogonal to the specific information needed to execute a request.

For example, a transfer payment service only really needs from and to account numbers and a transfer amount to execute. In real-world scenarios, however, a service request is likely to contain much more information, such as the identity of the person making the request, account/payment information, and so on. This additional information is usually handled by infrastructure services (login and security, transaction coordination, billing) outside the main transfer payment service. Encoding this information as part of the body of a SOAP message will only complicate matters. That is why it will be passed in as headers.

A SOAP message can include any number of header entries (simply referred to as headers). If any headers are present, they will all be children of the SOAP Header element, which, if present, must appear as the first child of the SOAP Envelope element. The following example shows a SOAP message with two headers, Transaction and Priority. Both headers are uniquely identified by the combination of their element name and their namespace URI:

```xml
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
<SOAP-ENV:Header>
   <t:Transaction xmlns:t="some-URI" SOAP-ENV:mustUnderstand="1">
      12345
   </t:Transaction>
   <p:Priority xmlns:p="some-Other-URI">
      <ReallyVeryHigh/>
   </p:Priority>
</SOAP-ENV:Header>
<SOAP-ENV:Body>
...
```
The contents of a header (sometimes referred to as the header value) are determined by the schema of the header element. This allows headers to contain arbitrary XML, another example of the benefits of SOAP being an XML-based protocol. Compare it to protocols such as HTTP where header values must be simple strings, thus forcing any structured information to be somehow encoded to become a string. For example, cookie values come in a semicolon delimited format, such as cookie1=value1;cookie2=value2. It is easy to reach the limits of these simple encodings. XML is a much better way to represent this type of structured information.

Also, notice the SOAP mustUnderstand attribute with value 1 that decorates the Transaction element. This attribute indicates that the recipient of the SOAP message must process the Transaction header entry. If a recipient does not know how to process a header tagged with mustUnderstand=’1’, it must abort processing with a well-defined error. This rule allows for robust evolution of SOAP-based protocols. It ensures that a recipient that might be unaware of certain important protocol extensions does not ignore them.

Note that because the Priority header is not tagged with mustUnderstand=’1’, it can be ignored during processing. Presumably, this will be OK because a server that does not know how to process message priorities will assume normal priority.

You might have noticed that the SOAP body can be treated as a well-specified SOAP header flagged with mustUnderstand=’1’. Although this is certainly true, the SOAP designers thought that having a separation between the headers and body of a message does not complicate the protocol and is convenient for readability.

Before leaving the topic of headers, it is important to point out that, despite the obvious need for header extensions to support such basic distributed computing concepts such as authentication credentials or transaction information, there hasn’t been a broad standardization effort in this area, with the exception of some security extensions that we’ll review in Chapter 5. Some of the leading Web service vendors are doing interesting work, but the industry as a whole is some way away from agreeing on core extensions to SOAP. Two primary forces maintain this unsatisfactory status quo:

- Most current Web service engines do not have a solid extensibility architecture. Therefore, header processing is relatively difficult right now. At the time of this writing, Apache Axis is a notable exception to this rule.
- Market pressure is pushing Web service vendors to innovate in isolation and to prefer shipping software over coordinating extensions with partners and competitors.

Wider Web service adoption will undoubtedly put pressure on the Web services community to think more about interoperability and begin broad standardization in some of these key areas.
SOAP Body

The SOAP Body element immediately surrounds the information that is core to the SOAP message. All immediate children of the Body element are body entries (typically referred to simply as bodies). Bodies can contain arbitrary XML. Sometimes, based on the intent of the SOAP message, certain conventions will govern the format of the SOAP body. The conventions for representing RPCs are discussed later in the section "SOAP-based RPCs." The conventions for communicating error information are discussed in the section "Error Handling in SOAP."

Taking Advantage of SOAP Extensibility

Let's take a look at how SkatesTown can use SOAP extensibility to its benefit. It turns out that SkatesTown's partners are demanding some type of proof that certain items are in SkatesTown's inventory. In particular, partners would like to have an e-mail record of any inventory checks they have performed.

Al Rosen got the idea to use SOAP extensibility in a way that allows the existing inventory check service implementation to be reused with no changes. SOAP inventory check requests will include a header whose element name is EMail belonging to the http://www.skatestown.com/ns/email namespace. The value of the header will be a simple string containing the e-mail address to which the inventory check confirmation should be sent.

Service Requestor View

Service requestors will have to modify their clients to build a custom SOAP envelope that includes the EMail header. Listing 3.5 shows the necessary changes. The e-mail to send confirmations to is provided in the constructor.

Listing 3.5  Updated Inventory Check Client

```java
package ch3.ex3;
import org.apache.axis.client.ServiceClient;
import org.apache.axis.message.SOAPEnvelope;
import org.apache.axis.message.SOAPHeader;
import org.apache.axis.message.RPCElement;
import org.apache.axis.message.RPCParam;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import org.w3c.dom.Document;
import org.w3c.dom.Element;

package ch3.ex3;
import org.apache.axis.client.ServiceClient;
import org.apache.axis.message.SOAPEnvelope;
import org.apache.axis.message.SOAPHeader;
import org.apache.axis.message.RPCElement;
import org.apache.axis.message.RPCParam;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import org.w3c.dom.Document;
import org.w3c.dom.Element;

```
```java
public class InventoryCheckClient {

    public InventoryCheckClient(String url, String email) {
        this.url = url;
        this.email = email;
    }

    public boolean doCheck(String sku, int quantity) throws Exception {
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        DocumentBuilder builder = factory.newDocumentBuilder();
        Document doc = builder.newDocument();
        Element emailElem = doc.createElementNS("http://www.skatestown.com/", "EMail");
        emailElem.appendChild(doc.createTextNode(email));
        SOAPEnvelope reqEnv = new SOAPEnvelope();
        reqEnv.addHeader(new SOAPHeader(emailElem));
        Object[] params = new Object[]{sku, new Integer(quantity), };
        reqEnv.addBodyElement(new RPCElement("", "doCheck", params));

        ServiceClient call = new ServiceClient(url);
        SOAPEnvelope respEnv = call.invoke(reqEnv);
    }
```
To set a header in Axis, you first need to build the DOM representation for the header. The code in the beginning of `doCheck()` does this. Then you need to manually construct the SOAP message that will be sent. This involves starting with a new `SOAPEnvelope` object, adding a SOAPHeader with the DOM element constructed earlier, and, finally, adding an `RPCElement` as the body of the message. At this point, you can use `ServiceClient.invoke()` to send the message.

When the call is made with a custom-built SOAP envelope, the return value of `invoke()` is also a `SOAPEnvelope` object. You need to pull the relevant data out of that envelope by getting the body of the response, which will be an `RPCElement`. The result of the operation will be the first `RPCParam` inside the RPC response. Knowing that `doCheck()` returns a boolean, you can get the value of the parameter and safely cast it to `Boolean`.

As you can see, the code is not trivial, but Axis does provide a number of convenience objects that make working with custom-built SOAP messages straightforward. Figure 3.5 shows a UML diagram with some of the key Axis objects related to SOAP messages.

**Figure 3.5**
Axis SOAP message objects.
Service Provider View

The situation on the side of the Axis-based service provider is a little more complicated because we can no longer use a simple JWS file for the service. JWS files are best used for simple and straightforward service implementations. Currently, it is not possible to indicate from a JWS file that a certain header (in this case the e-mail header) should be processed. Al Rosen implements three changes to enable this more sophisticated type of service:

- He moves the service implementation from the JWS file to a simple Java class.
- He writes a handler for the EMail header.
- He extends the Axis service deployment descriptor with information about the service implementation and the header handler.

Moving the service implementation is as simple as saving InventoryCheck.jws as InventoryCheck.java in /WEB-INF/classes/com/skatestown/services. No further changes to the service implementation are necessary.

Building a handler for the EMail header is relatively simple, as Listing 3.6 shows. When the handler is invoked by Axis, it needs to find the SOAP message and lookup the EMail header using its namespace and name. If the header is present in the request message, the handler sends a confirmation e-mail of the inventory check. The implementation is complex because to produce a meaningful e-mail confirmation, the handler needs to see both the request data (SKU and quantity) and the result of the inventory check. The basic process involves the following steps:

1. Get the request or the response message using getREQUESTMESSAGE() or getRESPONSEMESSAGE() on the Axis MessageContext object.
2. Get the SOAP envelope by calling getAsSOAPEnvelope().
3. Retrieve the first body of the envelope and cast it to an RPCElement because the body represents either an RPC request or an RPC response.
4. Get the parameters of the RPC element using getParams().
5. Extract parameters by their position and cast them to their appropriate type. As seen earlier in Listing 3.5, the response of an RPC is the first parameter in the response message body.

Listing 3.6  E-mail Header Handler

```java
package com.skatestown.services;

import java.util.Vector;
import org.apache.axis.*;
import org.apache.axis.axis2.*;
import org.apache.axis.axis2.message.*;
```
import org.apache.axis.handlers.BasicHandler;
import org.apache.axis.encoding.SOAPTypeMappingRegistry;
import bws.BookUtil;
import com.skatestown.backend.EmailConfirmation;

/**
* EMail header handler
*/
public class EMailHandler extends BasicHandler {

/**
* Utility method to retrieve RPC parameters
* from a SOAP message.
*/
private Object getParam(Vector params, int index) {
    return ((RPCParam)params.get(index)).getValue();
}

/**
* Looks for the EMail header and sends an email
* confirmation message based on the inventory check
* request and the result of the inventory check
*/
public void invoke(MessageContext msgContext) throws AxisFault {
    try {
        // Attempt to retrieve EMail header
        Message reqMsg = msgContext.getRequestMessage();
        SOAPEnvelope reqEnv = reqMsg.getAsSOAPEnvelope();
        SOAPHeader header = reqEnv.getHeaderByName("http://www.skatestown.com/", "EMail");

        if (header != null) {
            // Mark the header as having been processed
            header.setProcessed(true);

            // Get email address in header
            String email = (String)header.getValueAsType(SOAPTypeMappingRegistry.XSD_STRING);

            // Retrieve request parameters: SKU & quantity
            RPCElement reqRPC = (RPCElement)reqEnv.getFirstBody();

            // Other code...
        }
    }
}
It's simple code, but it does take a few lines because several layers need to be unwrapped to get to the RPC parameters. When all data has been retrieved, the handler calls the e-mail confirmation backend, which, in this example, logs e-mails “sent” to /resources/email.log.

Finally, adding deployment information about the new header handler and the inventory check service involves making a small change to the Axis Web services deployment descriptor. The book example deployment descriptor is in /resources/deploy.xml. Working with Axis deployment descriptors will be described in detail in Chapter 4.

Listing 3.7 shows the five lines of XML that need to be added. First, the e-mail handler is registered by associating a handler name with its Java class name. Following that is the
description of the inventory check service. The service options identify the Java class name for
the service and the method that implements the service functionality. The service element has
two attributes. Pivot is an Axis term that specifies the type of service. In this case, the value is
\texttt{RPCDispatcher}, which implies that \texttt{InventoryCheck} is an RPC service. The output attribute
specifies the name of a handler that will be called after the service is invoked. Because the
book examples don’t rely on an e-mail server being present, instead of sending confirmation
this class writes messages to a log file in /\texttt{resources/email.log}.

\textbf{Listing 3.7}  
\textbf{Deployment Descriptor for Inventory Check Service}

\begin{verbatim}
<!-- Chapter 3 example 3 services -->
<handler name="Email" class="com.skatestown.services.EMailHandler"/>
<service name="InventoryCheck" pivot="RPCDispatcher" response="Email">
    <option name="className" value="com.skatestown.services.InventoryCheck"/>
    <option name="methodName" value="doCheck"/>
</service>
\end{verbatim}

\textbf{Putting the Service to the Test}

With all these changes in place, we are ready to test the improved inventory check service.
There is a simple JSP test harness in ch3/ex3/index.jsp that is modeled after the JSP test
harness we used for the JWS-based inventory check service (see Figure 3.6).

\textbf{Figure 3.6}
\begin{quote}
\textit{Putting the enhanced inventory check Web service to the test.}
\end{quote}
SOAP on the Wire

With the help of TCPMon, we can see what SOAP messages are passing between the client and the Axis engine. We are only interested in seeing the request message because the response message will be identical to the one before the EMail header was added.

Here is the SOAP request message with the EMail header present:

```xml
POST /bws/services/InventoryCheck HTTP/1.0
Content-Length: 482
Host: localhost
Content-Type: text/xml; charset=utf-8
SOAPAction: "/doCheck"

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Header>
    <e:EMail xmlns:e="http://www.skatestown.com/ns/email">
      confirm@partners.com
    </e:EMail>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <ns1:doCheck xmlns:ns1="AvailabilityCheck">
      <arg0 xsi:type="xsd:string">947-TI</arg0>
      <arg1 xsi:type="xsd:int">1</arg1>
    </ns1:doCheck>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

There are no surprises in the SOAP message. However, a couple of things have changed in the HTTP message. First, the target URL is /bws/services/InventoryCheck. This is a combination of two parts: the URL of the Axis servlet that listens for SOAP requests over HTTP (/bws/services) and the name of the service we want to invoke (InventoryCheck). Also, the SOAPAction header, which was previously empty, now contains the name of the method we want to invoke. The service name on the URL and the method name in SOAPAction are both hints to Axis about the service we want to invoke.

That’s all there is to taking advantage of SOAP custom headers. The key message is one of simple yet flexible extensibility. Remember, the inventory check service implementation did not change at all!
SOAP Intermediaries

So far, we have addressed SOAP headers as a means for vertical extensibility within SOAP messages. There is another related notion, however: horizontal extensibility. Vertical extensibility is about the ability to introduce new pieces of information within a SOAP message, and horizontal extensibility is about targeting different parts of the same SOAP message to different recipients. Horizontal extensibility is provided by SOAP intermediaries.

The Need for Intermediaries

SOAP intermediaries are applications that can process parts of a SOAP message as it travels from its origination point to its final destination point (see Figure 3.7). Intermediaries can both accept and forward SOAP messages. Three key use-cases define the need for SOAP intermediaries: crossing trust domains, ensuring scalability, and providing value-added services along the SOAP message path.

![Figure 3.7](image)

Intermediaries on the SOAP message path.

Crossing trust domains is a common issue faced while implementing security in distributed systems. Consider the relation between a corporate or departmental network and the Internet. For small organizations, it is likely that the IT department has put most computers on the network within a single trusted security domain. Employees can see their co-workers computers as well as the IT servers and they can freely exchange information between them without the need for separate logons. On the other hand, the corporate network probably treats all computers on the Internet as part of a separate security domain that is not trusted. Before an Internet request reaches the network, it needs to cross from its untrustworthy domain to the trusted domain of the network. Corporate firewalls and virtual private network (VPN) gateways are the Cerberean guards of the gates to the network’s riches. Their job is to let some requests cross the trust domain boundary and deny access to others.

Another important need for intermediaries arises because of the scalability requirements of distributed systems. A simplistic view of distributed systems could identify two types of entities: those that request some work to be done (clients) and those that do the work (servers). Clients send messages directly to the servers with which they want to communicate. Servers, in turn, get some work done and respond. In this naïve universe, there is little need for distributed computing infrastructure. Alas, you cannot use this model to build highly scalable distributed systems.
Take basic e-mail as an example—the service we’ve grown to depend on so much in the Net era. When someone@company.com sends an e-mail message to myfriend@london.co.uk, it is definitely not the case that their e-mail client locates the mail server london.co.uk and sends the message to it. Instead, the client sends the message to its e-mail server at company.com. Based on the priority of the message and how busy the mail server is, the message will leave either by itself or in a batch of other messages. Messages are often batched to improve performance. It is likely that the message will make a few hops through different nodes on the Internet before it gets to the mail server in London.

The lesson from this example is that highly scalable distributed systems (such as e-mail) require flexible buffering of messages and routing based not only on message parameters such as origin, destination, and priority but also on the state of the system measured by parameters such as the availability and load of its nodes as well as network traffic information. Intermediaries hidden from the eyes of the originators and final recipients of messages perform all this work behind the scenes.

Last but not least, you need intermediaries so that you can provide value-added services in a distributed system. The type of services can vary significantly. Here are a couple of common examples:

- Securing message exchanges, particularly when transmitting messages through untrustworthy domains, such as using HTTP/SMTP on the Internet. You could secure SOAP messages by passing them through an intermediary that first encrypts them and then digitally signs them. On the receiving side, an intermediary will perform the inverse operations—checking the digital signature and, if it is valid, decrypting the message.

- Providing message-tracing facilities. Tracing allows the recipient of messages to find out the exact path that the message went through complete with detailed timings of arrivals and departures to and from intermediaries along the way. This information is indispensable for tasks such as measuring quality of service (QoS), auditing systems, and identifying scalability bottlenecks.

**Intermediaries in SOAP**

As the previous section has shown, intermediaries are an extremely important concept in distributed systems. SOAP is specifically designed with intermediaries in mind. It has simple yet flexible facilities that address the three key aspects of an intermediary-enabled architecture:

- How do you pass information to intermediaries?
- How do you identify who should process what?
- What happens to information that is processed by intermediaries?
From the discussion of intermediaries, you can see that most of the information that intermediaries require is completely orthogonal to the information contained in SOAP message bodies. For example, whether logging of inventory check requests is enabled or not is irrelevant to the inventory check service. Therefore, only information in SOAP headers can be explicitly targeted at intermediaries. The question then becomes one of deciding how to target the recipient of a particular header. This does not mean that an intermediary cannot look at, process, or change the SOAP message body; it certainly can do that. However, SOAP itself defines no mechanism to instruct an intermediary to do that. Contrast this to a SOAP message explicitly targeting a piece of information contained in a SOAP header at an intermediary with the understanding that it must at least attempt to process it.

All header elements can optionally have the `SOAP-ENV:actor` attribute. The value of the attribute is a URI that identifies who should handle the header entry. Essentially, that URI is the “name” of the intermediary. The special value `http://schemas.xmlsoap.org/soap/actor/next` indicates that the header entry’s recipient is the next SOAP application that processes the message. This is useful for hop-by-hop processing required, for example, by message tracing. Of course, omitting the actor attribute implies that the final recipient of the SOAP message should process the header entry. The message body is intended for the final recipient of the SOAP message.

The issue of what happens to a header that is processed by an intermediary is a little trickier. The SOAP specification states, “the role of a recipient of a header element is similar to that of accepting a contract in that it cannot be extended beyond the recipient.” This means that the intermediary should remove any header targeted for it that it has processed and it is free to introduce a new header in the message that looks the same but then this constitutes a contract between the intermediary and the next application. The goal here is to reduce system complexity by requiring that contracts about the presence, absence, and content of information in SOAP messages be very narrow in scope—from the originator of that information to the first SOAP application that handles it and not beyond.

**Putting It All Together**

To get a better sense of how you might use intermediaries in the real world, let’s consider the potentially realistic albeit contrived example of SkatesTown’s overall B2B integration architecture. Please keep in mind that all XML in the example is purely fictional—currently there isn’t a standardized way to handle security and routing of SOAP messages.

SkatesTown needs to integrate various applications in several of its departments with some of its partners’ applications (see Figure 3.8). Silver Bullet Consulting started working with the purchasing department building Web services to automate business functions such as checking inventory. Following the success of this engagement, Silver Bullet Consulting has been asked
to use Web services to automate processes in other departments such as customer service. SkatesTown’s corporate IT department is demanding centralized control over the entry point of all Web service requests to the company. They also require that all SOAP messages be transmitted over HTTPS for security reasons.

![SkatesTown’s system integration architecture.](image)

At the same time, individual departments demand that their own IT units control the servers that run their own Web services. These servers have their own trust domains and are sitting deep inside the corporate network, invisible to the outside world. To address this issue, Silver Bullet Consulting develops a partner interface gateway SOAP application that acts as an intermediary between the partner applications sending SOAP messages and the department-level applications that are handling them. The gateway application is hosted on an application server that is visible to the partner applications. This server is managed by the corporate IT department. A firewall is configured to allow access to the gateway application from the partner networks only.

The gateway application has the responsibility to validate partners’ security credentials and to route messages to the appropriate departmental SOAP applications. Security information and department server locations are available from SkatesTown’s enterprise directory.
Here is an example message the gateway application might receive:

POST /bws/inventory/InventoryCheck HTTP/1.0
Host: partnergateway.skatestown.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "doCheck"

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Header>
    <td:TargetDepartment
      xmlns:td="http://www.skatestown.com/ns/partnergateway"
      SOAP-ENV:actor="urn:X-SkatesTown:PartnerGateway"
      SOAP-ENV:mustUnderstand="1">
      Purchasing
    </td:TargetDepartment>
    <ai:AuthenticationInformation
      xmlns:ai="http://www.skatestown.com/ns/security"
      SOAP-ENV:actor="urn: X-SkatesTown:PartnerGateway"
      SOAP-ENV:mustUnderstand="1">
      <username>PartnerA</username>
      <password>LongLiveSOAP</password>
    </ai:AuthenticationInformation>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <doCheck>
      <arg0 xsi:type="xsd:string">947-TI</arg0>
      <arg1 xsi:type="xsd:int">1</arg1>
    </doCheck>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

There are two header entries. The first identifies the target department as purchasing, and the second passes the authentication information of the message originator, partner A in this case. Both header entries are marked with mustUnderstand="1" because they are critical to the successful processing of the message. The partner gateway application is identified by the actor attribute as the place to process these.
After processing the message, the partner gateway application might forward the following message:

```plaintext
POST /bws/services/InventoryCheck HTTP/1.0
Host: purchasing.skatestown.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "doCheck"

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
   SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <SOAP-ENV:Header>
      <cc:ClientCredentials
         xmlns:cc="http://schemas.security.org/soap/security"
         SOAP-ENV:mustUnderstand="1">
         <ClientID>/External/Partners/PartnerA</ClientID>
      </cc:ClientCredentials>
   </SOAP-ENV:Header>
   <SOAP-ENV:Body>
      <doCheck>
         <arg0 xsi:type="xsd:string">947-TI</arg0>
         <arg1 xsi:type="xsd:int">1</arg1>
      </doCheck>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Note how the previous two header entries have disappeared. They were meant for the gateway application only. Having extracted the purchasing department’s location from the enterprise directory, the gateway application forwards the message to purchasing.skatestown.com. A new header entry is meant for the final recipient of the message. The entry specifies the security identity of the message originator as /External/Partners/PartnerA. This identity was presumably obtained from SkatesTown's security system following the successful authentication of partner A. The applications in the purchasing department will use this identity to check whether partner A is authorized to perform the operation requested in the SOAP message body.

This example scenario shows that intermediaries bring significant capabilities to SOAP-enabled applications and can be introduced and implemented at a fairly low cost. The inventory check service implementation does not need to change. The partner gateway does not need to know anything about inventory checking; it only understands the target department and authentication headers. Inventory check clients only need to add a couple of headers to the messages they are sending to fit in the new architecture.
Error Handling in SOAP

So far in our examples everything has gone according to plan. Murphy’s Law guarantees that this is not how things work out in the real world. What would happen, for example, if partner A failed to authenticate with the partner gateway application? How will this exceptional condition be communicated via SOAP? The answer lies in the semantics of the SOAP Fault element.

Consider the following possible reply message caused by the authentication failure:

```
HTTP/1.0 500 Internal Server Error
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <SOAP-ENV:Fault>
      <faultcode>SOAP-ENV:Client.AuthenticationFailure</faultcode>
      <faultstring>Failed to authenticate client</faultstring>
      <faultactor>urn:X-SkatesTown:PartnerGateway</faultactor>
    </SOAP-ENV:Fault>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Before we look at the XML, note that the HTTP response code is 500 Internal Server Error. This is a required response in the case of any SOAP-related error by the HTTP transport binding as presented in the SOAP specification. Other protocols will have their own way to report errors. The HTTP SOAP binding is discussed in detail in the section “SOAP Protocol Bindings.”

The body of the response contains a single Fault element in the SOAP envelope namespace. SOAP uses this mechanism to indicate that an error has occurred and to provide some diagnostic information. There are three child elements.

The faultcode element must be present in all cases. It provides information that can be used to identify the specific error that occurred. It is not meant for human consumption. The content of the element is a string prefixed by one of the four faultcode values specified by SOAP:

- VersionMismatch indicates that the namespace of the Envelope element is invalid.
- MustUnderstand indicates that a required header entry was not understood.
- Client indicates that likely cause of the error lies in the content or formatting of the SOAP message. In other words, the client should probably not re-send the message without making some changes to it.
• **Server** indicates that the message failed due to reasons other than its content or its format. This leaves the door open for the same message to perhaps succeed at a later time.

A hierarchical namespace of values can be obtained by separating fault values with the dot (.) character. In our example, `Client.AuthenticationFailure` is a more specific fault code than `Client`.

The **faultstring** element contains a human-readable message identifying the cause of the fault. It must always be present. Here we simply state that the client has failed to authenticate.

The **faultactor** element provides information about where in the message path the fault occurred. It must be present if the failure occurred somewhere other than at the final destination of the SOAP message. The content of the element is the URI of the actor where the error occurred. In our example, we identify the partner gateway application as the failure point.

What is not shown in this example is how application-specific error diagnostic information can be exchanged. SOAP provides a simple mechanism for this, as well. If the fault occurred during the processing of the message body, an optional **detail** element can be added after **faultactor**. There are no restrictions on its contents. This rule has one important exception: If the fault occurred during the processing of a header entry, a **detail** element cannot be returned. Instead, the header entry should be returned with detailed error information contained therein. This is the mechanism SOAP uses to determine whether a fault was the result of header versus body processing.

**SOAP Message Processing**

Now that we have covered headers with **mustUnderstand** behavior, intermediaries, and error handling, we can completely define the rules for SOAP message processing. Upon receiving a message, a SOAP application must:

1. Determine whether it understands the version of SOAP that the message uses by inspecting the namespace value of the SOAP **Envelope** element. If the version is unknown, it must discard the message with a **VersionMismatch** error. Otherwise, it has to move to the next step.

2. Identify all parts of the message intended for the application. Typically this is done considering the application's role in the message path (intermediary or final recipient) and the values of the actor global attribute, but other information can be taken into account as well.

3. Verify that all mandatory parts of the message identified in Step 2 are supported by the application. These include **mustUnderstand** headers and, in the case of a final recipient, the body. If any mandatory part cannot be supported, the message is discarded with a
Must understand error in the case of headers and an application-specific error in the case of bodies. Otherwise, the application will move to Step 4.

4. Process all mandatory parts identified in Step 2 plus any optional parts that it knows about.

5. If the application is not the final recipient of the message, it must remove all headers that it has processed before passing the message forward along its path.

Having covered the SOAP envelope framework, intermediaries, and error handling, it is now time to move to other areas of the SOAP specification.

**SOAP Data Encoding**

Another important area of SOAP has to do with the rules and mechanisms for encoding data in SOAP messages. So far, our Web service example, the inventory check, has dealt only with very simple datatypes: strings, integers, and booleans. All these types have direct representation in XML Schema so it was easy, through the use of the xsi:type attribute, to describe the type of data being passed in a message. What would happen if our Web services needed to exchange more complex types, such as arrays and arbitrary objects? What algorithm should be used to determine their representation in XML format? In addition, given SOAP’s extensibility requirements, how can a SOAP message specify different encoding algorithms? This section addresses such questions.

**Specifying Different Encodings**

SOAP provides an elegant mechanism for specifying the encoding rules that apply to the message as a whole or any portion of it. This is done via the encodingStyle attribute in the SOAP envelope namespace. The attribute is defined as global in the SOAP schema; it can appear with any element, allowing different encoding styles to be mixed and matched in a SOAP message. An encodingStyle attribute applies to the element it decorates and its content, excluding any children that might have their own encodingStyle attribute. Therefore, any element in a SOAP message can have either no encoding style specified or exactly one encoding style. The rules for determining the encoding style of an element are simple:

1. If an element has the encodingStyle attribute, then its encoding style is equal to the value of that attribute.

2. Otherwise, the encoding style is equal to the encoding style of the closest ancestor element that has the encodingStyle attribute...

3. ...Unless there is no such ancestor, which implies that the element has no specified encoding style.
SOAP defines one particular set of data encoding rules. They are identified by SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding" in SOAP messages. You will often see this attribute applied directly to the Envelope element in a SOAP message. There is no notion of default encoding in a SOAP message. Encoding style must be explicitly specified.

Despite the fact that the SOAP specification defines these encoding rules, it does not mandate them. SOAP implementations are free to choose their own encoding styles. There are costs and benefits to making this choice. A benefit could be that the implementations can choose a more optimized data encoding mechanism than the one defined by the SOAP specification. For example, some SOAP engines already on the market detect whether they are exchanging SOAP messages with the same type of engine and, if so, switch to a highly optimized binary data encoding format. Because this switch happens only when both ends of a communication channel agree to it, interoperability is not hindered. At the same time, however, supporting these different encodings does have an associated maintenance cost, and it is difficult for other vendors to take advantage of the benefits of an optimized data encoding.

**SOAP Data Encoding Rules**

The SOAP data encoding rules exist to provide a well-defined mapping between abstract data models (ADMs) and XML syntax. ADMs can be mapped to directed labeled graphs (DLGs)—collections of named nodes and named directed edges connecting two nodes. For Web services, ADMs typically represent programming language and database data structures. The SOAP encoding rules define algorithms for executing the following three tasks:

- Given meta-data about an ADM, construct an XML schema from it.
- Given an instance graph of the data model, we can generate XML that conforms to the schema. This is the serialization operation.
- Given XML that conforms to the schema, we can create an instance graph that conforms to the abstract data model’s schema. This is the deserialization operation. Further, if we follow serialization by deserialization, we should obtain an identical instance graph to the one we started with.

Although the purpose of the SOAP data encoding is so simple to describe, the actual rules can be somewhat complicated. This section is only meant to provide an overview of topic. Interested readers should pursue the data encoding section of the SOAP Specification.

**Basic Rules**

The SOAP encoding uses a type system based on XML Schema. *Types* are schema types. *Simple types* (often known as *scalar* types in programming languages) map to the built-in types in XML Schema. Examples include `float`, `positiveInteger`, `string`, `date`, and any
restrictions of these, such as an enumeration of RGB colors derived by restricting \texttt{xsd:string} to only "red", "green", and "blue". \textit{Compound types} \begin{itemize} \item[] \end{itemize} are composed of several parts, each of which has an associated type. The parts of a compound type are distinguished by an \textit{accessor} \begin{itemize} \item[] \end{itemize}. An accessor can use the name of a part or its position relative to other parts in the XML representation of values. \textit{Structs} \begin{itemize} \item[] \end{itemize} are compound types whose parts are distinguished only by their name. \textit{Arrays} \begin{itemize} \item[] \end{itemize} are compound types whose parts are distinguished only by their ordinal position.

\textit{Values} are instances of types, much in the same way that a string object in Java is an instance of the \texttt{java.lang.String} class. Values are represented as XML elements whose type is the value type. \textit{Simple values} are encoded as the content of elements that have a simple type. In other words, the elements that represent simple values have no child elements. \textit{Compound values} are encoded as the content of elements that have a compound type. The parts of the compound value are encoded as child elements whose names and/or positions are those of the part accessors. Note that values can never be encoded as attributes. The use of attributes is reserved for the SOAP encoding itself, as you will see a bit later.

Values whose elements appear at the top level of the serialization are considered \textit{independent}, whereas all other values are \textit{embedded} (their parent is a value element).

The following snippet shows an example XML schema fragment describing a person with a name and an address. It also shows the associated XML encoding of that schema according to the SOAP encoding rules:

```xml
<xs:element name="Person" type="Person"/>
<xs:complexType name="Person">
  <xs:sequence>
    <xs:element name="name" type="xsd:string"/>
    <xs:element name="address" type="Address"/>
  </xs:sequence>
</xs:complexType>

<xs:element name="Address" type="Address"/>
<xs:complexType name="Address">
  <xs:sequence>
    <xs:element name="street" type="xsd:string"/>
    <xs:element name="city" type="xsd:string"/>
    <xs:element name="state" type="USState"/>
  </xs:sequence>
</xs:complexType>
```

SIMPLE OBJECT ACCESS PROTOCOL (SOAP)
One thing should be apparent: The SOAP encoding rules are designed to fit well with traditional uses of XML for data-oriented applications. The example encoding has no mention of any SOAP-specific markup. This is a good thing.

**Identifying Value Types**

When full schema information is available, it is easy to associate values with their types. In some cases, however, this is hard to do. Sometimes, a schema will not be available. In these cases, Web service interaction participants should do their best to make messages as self-describing as possible by using `xsi:type` attributes to tag the type of at least all simple values. Further, they can do some guessing by inspecting the markup to determine how to deserialize the XML. Of course, this is difficult. The only other alternative is to establish agreement in the Web services industry about the encoding of certain generic abstract data types. The SOAP encoding does this for arrays.
Other times, schema information might be available, but the content model of the schema element will not allow you to sufficiently narrow the type of contained values. For example, if the schema content type is “any”, it again makes sense to use xsi:type as much as possible to specify the exact type of value that is being transferred.

The same considerations apply when you’re dealing with type inheritance, which is allowed by both XML Schema and all object-oriented programming languages. The SOAP encoding allows a sub-type to appear in any place where a super-type can appear. Without the use of xsi:type, it will be impossible to perform good deserialization of the data in a SOAP message.

Sometimes you won’t know the names of the value accessors in advance. Remember how Axis auto-generates element names for the parameters of RPC calls? Another example would be the names of values in an array—the names really don’t matter; only their position does. For these cases, xsi:type could be used together with auto-generated element names. Alternatively, the SOAP encoding defines elements with names that match the basic XML Schema types, such as SOAP-ENC:int or SOAP-ENC:string. These elements could be used directly as a way to combine name and type information in one. Of course, this pattern cannot be used for compound types.

**SOAP Arrays**

Arrays are one of the fundamental data structures in programming languages. (Can you think of a useful application that does not use arrays?) Therefore, it is no surprise that the SOAP data encoding has detailed rules for representing arrays. The key requirement is that array types must be represented by a SOAP-ENC:Array or a type derived from it. These types have the SOAP-ENC:arrayType attribute, which contains information about the type of the contained items as well as the size and number of dimensions of the array. This is one example where the SOAP encoding introduces an attribute and another reason why values in SOAP are encoded using only element content or child elements.

Table 3.1 shows several examples of possible arrayType values. The format of the attribute is simple. The first portion specifies the contained element type. This is expressed as a fully qualified XML type name (QName). Compound types can be freely used as array elements. If the contained elements are themselves arrays, the QName is followed by an indication of the array dimensions, such as [] and [,] for one- and two-dimensional arrays, respectively. The second portion of arrayType specifies the size and dimensions of the array, such as [5] or [2,3]. There is no limit to the number of array dimensions and their size. All position indexes are zero-based, and multidimensional arrays are encoded such that the rightmost position index changes the quickest.
TABLE 3.1  Example SOAP-ENC:arrayType Values

<table>
<thead>
<tr>
<th>arrayType</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsd:int[][]</td>
<td>An array of five integer arrays</td>
</tr>
<tr>
<td>xsd:int[,]</td>
<td>An array of five two-dimensional arrays of integers</td>
</tr>
<tr>
<td>xsd:string[2,3]</td>
<td>A 2x3, two-dimensional array of strings</td>
</tr>
</tbody>
</table>

If schema information is present, arrays will typically be represented as XML elements whose type is or derives from SOAP-ENC:Array. Further, the array elements will have meaningful XML element names and associated schema types. Otherwise, the array representation would most likely use the pre-defined element names associated with schema types from the SOAP encoding namespace. Here is an example:

<!-- Schema fragment for array of numbers -->
<element name="arrayOfNumbers">
   <complexType base="SOAP-ENC:Array">
      <element name="number" type="xsd:int" maxOccurs="unbounded"/>
   </complexType>
   <xsd:anyAttribute namespace="##other" processContents="strict"/>
</element>

<!-- Encoding example using the array of numbers -->
<arrayOfNumbers SOAP-ENC:arrayType="xsd:int[2]">
   <number>11</number>
   <number>22</number>
</arrayOfNumbers>

<!-- Array encoding w/o schema information -->
<SOAP-ENC:Array SOAP-ENC:arrayType="xsd:int[2]">
   <SOAP-ENC:int>11</SOAP-ENC:int>
   <SOAP-ENC:int>22</SOAP-ENC:int>
</SOAP-ENC:Array>

Referencing Data

Abstract data models allow a single value to be referred to from multiple locations. Given any particular data structure, a value that is referred to by only one accessor is considered single-reference, whereas a value that has more than one accessor referring to it is considered multi-reference. The examples shown so far have assumed single-reference values. The rules for encoding multi-reference values are relatively simple, however:
• Multi-reference values are represented as independent elements at the top of the serialization. This makes them easy to locate in the SOAP message.
• They all have an unqualified attribute named id of type ID per the XML Schema specification. The ID value provides a unique name for the value within the SOAP message.
• Each accessor to the value is an unqualified href attribute of type uri-reference per the XML Schema specification. The href values contain URI fragments pointing to the multi-reference value.

Here is an example that brings together simple and compound types, and single- and multi-reference values and arrays:

```
<!-- Person type w/ multi-ref attributes added -->
xsd:complexType name="Person">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="address" type="Address"/>
  </xsd:sequence>
  <xsd:attribute name="href" type="uriReference"/>
  <xsd:attribute name="id" type="ID"/>
  <xsd:anyAttribute namespace="##other" processContents="strict"/>
</xsd:complexType>

<!-- Address type w/ multi-ref attributes added -->
xsd:complexType name="Address">
  <xsd:sequence>
    <xsd:element name="street" type="xsd:string"/>
    <xsd:element name="city" type="xsd:string"/>
    <xsd:element name="state" type="USState"/>
  </xsd:sequence>
  <xsd:attribute name="href" type="uriReference"/>
  <xsd:attribute name="id" type="ID"/>
  <xsd:anyAttribute namespace="##other" processContents="strict"/>
</xsd:complexType>

<!-- Example array of two people sharing an address -->
<SOAP-ENC:Array SOAP-ENC:arrayType="p:Person[2]">
  <p:Person>
    <name>Bob Smith</name>
    <address href="#addr-1"/>
  </p:Person>
  <p:Person>
    <name>Joan Smith</name>
    <address href="#addr-1"/>
  </p:Person>
</SOAP-ENC:Array>
```
The schema fragments for the compound types had to be extended to support the \textit{id} and \textit{href} attributes required for multi-reference access.

\textbf{Odds and Ends}

The SOAP encoding rules offer many more details that we have glossed over in the interest of keeping this chapter focused on the core uses of SOAP. Three data encoding mechanisms are worth a brief mention:

- Null values of a specific type are represented in the traditional XML Schema manner, by tagging the value element with \texttt{xsi:null='1'}. 
- The notion of “any” type is also represented in the traditional XML Schema manner via the \texttt{xsd:ur-type} type. This type is the base for all schema datatypes and therefore any schema type can appear in its place.
- The SOAP encoding allows for the transmission of partial arrays by specifying the starting offset for elements using the \texttt{SOAP-ENC:offset} attribute. Sparse arrays are also supported by tagging array elements with the \texttt{SOAP-ENC:position} attribute. Both of these mechanisms are provided to minimize the size of the SOAP message required to transmit a certain array-based data structure.

Having covered the SOAP data encoding rules, it is now time to look at the more general problem of encoding different types of data in SOAP messages.

\textbf{Choosing a Data Encoding}

Because data encoding needs vary a lot, there are many different ways to approach the problem of representing data for Web services. To add some structure to the discussion, think of the decision space as a choice tree. A choice tree has yes/no questions at its nodes and outcomes at its leaves (see Figure 3.9).

\textbf{XML Data}

Probably the most common choice has to do with whether the data already is in (or can easily be converted to) an XML format. If you can represent the data as XML, you only need to decide how to include it in the XML instance document that will represent a message in the protocol. Ideally, you could just mix it in amidst the protocol-specific XML but under a different namespace. This approach offers several benefits. The message is easy to construct and easy to process using standard XML tools. However, there is a catch.
The problem has to do with a little-considered but very important aspect of XML: the uniqueness rule for ID attributes. The values of attributes of type ID must be unique in an XML instance so that the elements with these attributes can be conveniently referred to using attributes of type IDREF, as shown here:

```xml
<Target id="mainTarget"/>
<Reference href="#mainTarget"/>
```

The problem with including a chunk of XML inline (textually) within an XML document is that the uniqueness of IDs can be violated. For example, in the following code both message elements have the same ID. This makes the document invalid XML:

```xml
<message id="msg-1">
  A message with an attached <a href="#msg-1">message</a>.
  <attachment id="attachment-1">
    <!-- ID conflict right here -->
    <message id="msg-1">
      This is a textually included message.
    </message>
  </attachment>
</message>
```

**Figure 3.9**
Possible choice tree for data encoding.

---

**SIMPLE OBJECT ACCESS PROTOCOL (SOAP)**

---

**FIGURE 3.9**
Possible choice tree for data encoding.
And no, namespaces do not address the issue. In fact, the problems are so serious that nothing short of a change in the core XML specification and in most XML processing tools can change the status quo. Don’t wait for this to happen.

You can work around the problem two ways. If no one will ever externally reference specific IDs within the protocol message data, then your XML protocol toolset can automatically re-write the IDs and references to them as you include the XML inside the message, as follows:

```xml
<message id="msg-1">
  A message with an attached <a href="#id-9137">message</a>.
  <attachment id="attachment-1">
    <!-- ID has been changed -->
    <message id="id-9137">
      This is a textually included message.
    </message>
  </attachment>
</message>
```

This approach will give you the benefits described earlier at the cost of some extra processing and a slight deterioration in readability due to the machine-generated IDs.

If you cannot do this, however, you will have to include the XML as an opaque chunk of text inside your protocol message:

```xml
<message id="msg-1">
  A message with an attached message that we can no longer refer to directly.
  <attachment id="attachment-1">
    <!-- Message included as text -->
    &ltmessage id="id-9137"&gt;
      This is a textually included message.
    &lt;/message&gt;
  </attachment>
</message>
```

In this case, we have escaped all pointy brackets, but we also could have included the whole message in a CDATA section. The benefit of this approach is that it is easy and it works for any XML content. However, you don’t get any of the benefits of XML. You cannot validate, query, or transform the data directly, and you cannot reference pieces of it from other parts of the message.

**Binary Data**

So far, we have discussed encoding options for pre-existing XML data. However, what if you are not dealing with XML data? What if you want to transport binary data as part of your message, instead? The commonly used solution is good old base64 encoding:
On the positive side, base64 data is easy to encode and decode, and the character set of base64-encoded data is valid XML element content. On the negative side, base64 encoding takes up nearly 33% more memory than pure binary representation. If you need to move much binary data and space/time efficiency is a concern, you might have to look for alternatives. (More on this in a bit.)

You might want to consider using base64 encoding even when you want to move some plain text as part of a message, because XML’s document-centric SGML origin led to several awkward restrictions on the textual content of XML instances. For example, an XML document cannot include any control characters (ASCII codes 0 through 31) except tabs, carriage returns, and line feeds. This limitation includes both the straight occurrences of the characters and their encoded form as character references, such as \&#x04;.

Further, carriage returns are always converted to line feeds by XML processors. It is important to keep in mind that not all characters you can put in a string variable in a programming language can be represented in XML documents. If you are not careful, this situation can lead to unexpected runtime errors.

**Abstract Data Models**

If you are not dealing with plain text, XML, or binary data, you probably have some form of structured data represented via an abstract data model.

The key question when dealing with abstract data models and XML is whether the output XML format matters. For example, if you have to generate SkatesTown purchase orders, then the output format is clearly important. If, on the other hand, you just want to make an RPC call over SOAP to pass some data to a Web service, then the exact format of the XML representing your RPC parameters does not matter. All that matters is that the Web service engine can decode the XML and reconstruct a similar data structure with which to invoke the backend.

In the latter case, it is safe to use pre-built automatic “data to XML and back” encoding systems (see Figure 3.10). For example, Web service engines have data serialization/deserialization modules that support the rules of SOAP encoding. These rules are flexible enough to
represent most application-level data types. Suffice to say, in many cases you will never have to worry about the mechanics of the serialization/deserialization processes.

![Diagram of Generic XML serialization/deserialization]

**FIGURE 3.10**

*Generic XML serialization/deserialization.*

The SOAP encoding is a flexible schema model for representing data—element names in the instance document often depend on the type and format of data that is being encoded. This model allows for a link between the data and its type, which enables validation. It is one of the core reasons why XML protocols such as SOAP moved to this encoding model, as discussed earlier in the chapter when we considered the evolution of XML protocols.

In the cases where the XML output format does not matter (typically RPC scenarios), you can rely on the default rules provided by various XML data encoding systems. In many cases, however, the XML format is fixed based on the specification of a service. A SkatesTown purchase order submission service is a perfect example. From a requestor’s perspective, the input format must be a PO document and the output format must be an invoice document. Requestors are responsible for mapping whatever data structures they might be using to represent POs in their order systems to the SkatesTown PO format. Also, SkatesTown is responsible for always outputting responses in its invoice XML format.

There are two typical approaches to handling this scenario. The simplest one is to completely delegate XML processing to the application. In other words, the Web service engine is responsible only for delivering a chunk of XML to the Web service implementation. Another approach involves building and registering custom serializers/deserializers (datatype mappers) with the Web service engine. The serializers manipulate application data to produce XML. The deserializers manipulate the XML to generate application data. You can build these serializer/deserializer modules two ways: by hand, using the APIs of the Web service engine; or using a tool for mapping data to and from XML given a pre-existing schema. These tools are known as *schema compilers* (see Figure 3.11).

Schema compilers are tools that analyze XML schema and code-generate serialization and deserialization modules specific to the schema. These modules will work with data structures tuned to the schema.

Schema compilation is a difficult problem, and this is one reason there aren’t many excellent tools in this space. The Java Architecture for XML Binding (JAXB) is one of the projects that
is trying to address this problem in the context of the Java programming language
(http://java.sun.com/xml/jaxb/). Unfortunately, at the time of this writing, JAXB only
supports DTDs and does not support XML Schema. Castor
(http://castor.exolab.org/index.html) is an open-source project that is focused on the
Java-to-XML data mapping space as well. Chapter 8, “Interoperability, Tools, and Middleware
Products,” focuses on the current Web service tooling for the Java platform. It provides more
details on these and other important implementation efforts in the space.

![Diagram of Serialization/deserialization process with a schema compiler.]

**FIGURE 3.11**
Serialization/deserialization process with a schema compiler.

**Linking Data**

So far, we have only considered scenarios where the encoded data is part of the XML docu-
ment describing a protocol message. This can create some problems for including pre-existing
XML content and can waste space in the case of base64-encoded binary objects. The alterna-
tive would be keeping the data outside of the message and somehow bringing it in at the right
time. For example, an auto insurance claim might carry along several accident pictures that
come into play only when the insurance claim needs to be displayed in a browser or printed.

You can use two general mechanisms in such cases. The first comes straight out of XML 1.0.
It involves external entity references, which allow content external to an XML document to be
brought in during processing. Many people in the industry prefer pure markup and therefore
favor a second approach that uses explicit link elements that comply with the XLink specifi-
cation. Both methods could work. Both require extensions to the core Web services toolsets that
are available now. In addition, purely application-based methods are available for linking; you
could just pass a URI known to mean “get the actual content here.” However, this approach
does not scale to generic data encoding mechanisms because it requires application-level
knowledge.

External content can be kept on a separate server to be delivered on demand. It can also be
packaged together with the protocol message in a MIME envelope. The SOAP Messages with
Attachments Note to the W3C ([http://www.w3.org/TR/2000/NOTE-SOAP-attachments-20001211](http://www.w3.org/TR/2000/NOTE-SOAP-attachments-20001211)) defines a mechanism for doing this. An example SOAP message with an attachment is shown later in the chapter in the section “SOAP Protocol Bindings.”

There are many, many ways to encode data in XML, and well-designed XML protocols will let you plug any encoding style you choose. How should you make this important decision? First, of course, keep it simple. If possible, choose standards-based and well-deployed technology. Then, consider your needs and match them against some of the important facets of XML data encoding described here.

**Architecting Distributed Systems with Web Services**

Although SOAP is typically demoed with simple RPC-based Web services, such as SkatesTown’s inventory check service, the SOAP specification does not mandate any particular communication mechanism or interaction pattern between the participants of a Web-service–enabled distributed system. System designers basically have complete control over the system architecture, choice of communication protocols, message routing, intermediary configuration, and so on. The hard part about having so much flexibility is that without solid experience with distributed systems and good judgment, it is easy to make sub-optimal choices.

The most commonly asked questions about distributed systems based on Web services center around a long-running debate in distributed computing circles regarding the rules and regulations for using RPC and messaging (often identified as Message Oriented Middleware—MOM) to solve problems. Typically, the debate takes the unnecessarily polarized form of “MOM vs. RPC.” The fact of the matter is that both messaging and RPC play significant, albeit different, roles in distributed computing. Both approaches continue to be very relevant in the era of Web services.

Unfortunately, a lot of confusion exists about the meaning of the terms, the capabilities of messaging and RPC systems, and the scenarios in which they are best applied. Service-oriented architectures fundamentally can support both models. Therefore, to best take advantage of Web services, it helps to have a good understanding of both. What follows is a brief analysis of the two approaches and their relation to SOAP and Web services. Given that people are generally more familiar with RPCs, we start with a discussion of messaging in its many forms.

**Messaging**

As a model for distributed computing, *messaging* refers to a mechanism for getting systems to interact via the passing of messages. A message is a single unit of communication encapsulating some information. (A SOAP message is a great example.) This is where the differences begin. Messaging models can vary significantly based on the following criteria:
Message Participants

There are three different ways to organize messaging participants (see Figure 3.12). The simplest case is 1-to-1 (point-to-point) messaging, which involves only two systems. An example could be an e-commerce scenario where the client application submits a purchase order to a digital marketplace. In this case, the sender needs to know where to send the message.

A slightly more complicated organization is 1-to-many messaging, where the sender sends a single message but copies of it go to multiple recipients. This is often referred to publish/subscribe or topic-based messaging. The idea is that the sender is a publisher that sends a message to a "topic" and that the recipients are all the systems that have subscribed to receive notifications on this topic. E-mail distribution lists are a good example of this type of messaging. The name of the distribution list is the topic, and the subscribers are all the e-mail addresses on the list.
Finally, many-to-many messaging involves a pattern of message exchange among any number of participants. Clearly, in this case, some system in the middle (typically some type of a workflow engine supporting business processes) needs to direct message traffic. This is described by the cloud in Figure 3.12.

Interaction Patterns
There are four common messaging interaction patterns (see Figure 3.13). One-way (fire-and-forget) messaging involves the simple sending of a message from one system to another. No response is generated at the application level. Of course, depending on the transport (such as HTTP), a response might be generated at the network level. In the case of request-response messaging, a response message is generated for every request message. The response message is sent from the target of the request message to its source. Chapter 6 describes how requests and responses can be correlated and how multiple request-response pairs can be organized into logical “conversations.”

The other two interaction patterns, notification and notification-response, are mirror images of one-way and request-response. They are callback patterns. Rather than a client system pushing messages to a server system, the server system is pushing messages to the client. The stock ticker application you might have on your desktop is a perfect example of notification combined with publish-subscribe messaging. Chapter 6 gets into more detail about Web service interaction patterns.
Synchronicity
Messaging can be either synchronous or asynchronous. In synchronous messaging, a send operation does not complete until the target of the message has finished processing the message. Asynchronous messaging is harder to define. Typically, the send operation will return immediately (or very quickly), before the target has processed the message. Response messages, if any, typically arrive via callbacks.

Direct vs. Queued Messaging
The synchronicity of messaging is controlled by the presence of messaging middleware, particularly queuing systems. Direct messaging works without any middleware present. For messages to be exchanged, a direct connection between the source and the target(s) must be available. This is why it is sometimes referred to as connection-oriented messaging. You can get some amount of asynchronicity in direct messaging by using threads to manage the sending and receiving of messages.

Indirect messaging involves some type of message queuing. Queues provide message buffering and dispatch capabilities. Consider the e-mail server example from earlier in the chapter. An e-mail server is a perfect example of a message queuing system. When you send an e-mail message, your e-mail client does not contact the e-mail client of the person you are trying to reach. Instead, your e-mail client sends the message to a local e-mail server. The server saves the message in some safe place and waits for a good moment to send it out. Typically, many messages are sent at once. This is the buffering function. The dispatch function has to do with the e-mail server inspecting the target e-mail addresses and deciding where to forward the e-mail message. In some cases, an e-mail message will make several hops between e-mail servers before it arrives at the destination e-mail server where your mail client can read it. This configuration is so powerful because it works even in the cases where mail clients and even some mail servers are offline for long periods of time. A mail server will keep trying to send e-mail for several days and will store received messages potentially indefinitely.

Figure 3.14 contrasts direct messaging (the topmost configuration) with a number of possible queuing configurations. In the second and third configurations, the queuing system acts primarily as a message buffer. For example, if the receiver is not on the network, the message will still be safely stored in the queue. The last configuration is the most interesting, in that the message can be moved from the local to the remote system without either the sender or the receiver being online—the message queuing systems can do the job by themselves. In addition, the presence of more than one queuing system allows for flexible message dispatch.
Another important aspect of messaging is **quality of service** (QoS). Direct messaging exhibits the QoS parameters with which we are most familiar, such as security and transaction management. When queuing is in use, other types of QoS become available. For example, messages can be stored in the queuing server in various ways: in memory (the fastest queuing mechanism but one that does not guarantee against system failure) or in some persistent store, such as a DBMS.

Further, transactions can guarantee that the message is sent to the receiver once and only once or not at all. In the case of message delivery failure, QoS policy might dictate that a failure notification is sent to the message sender. In addition, it is common QoS policy to send acknowledgement notifications that the message has been successfully delivered to the receiver. These types of QoS considerations are very relevant to Web services. Chapter 5 looks in more detail at some QoS aspects.

**Message Format**

The last but not least important aspect of messaging is the format of message data. Most messaging systems allow the transfer of text and binary data, to enable the easy transfer of XML. Some newer messaging systems treat XML messages specially and try to use an optimized XML encoding format. There is also the notion of queues that can automatically allow only XML messages that comply with certain schema. Some platform-focused messaging systems, such as Java Messaging Service (JMS) middleware and Microsoft’s .NET messaging server,
also allow for the automatic serialization of application data (Java objects in the case of JMS and Common Language Runtime [CLR] data structures in the case of Microsoft).

### Messaging Versus RPC

If messaging is all about possibilities and variations, RPCs are much more constrained. As the name suggests, the goal of RPCs is to make the invocation of remote code seem like a local procedure call (LPC). To make an RPC call, you need the following information:

- A target to invoke
- An operation name
- Optionally, parameters to pass to the operation

Therefore, whereas messaging is primarily about data (which can be in any conceivable format), RPCs are about combining specific application-level data with remote code. This is the one fundamental difference between messaging and RPC. A nice side-effect is that programmers using RPC do not have to worry about manually performing data encoding and decoding—something that typically has to happen when using messaging systems, especially across programming languages and platforms.

Another way to state the main difference between messaging and RPC is to note that messaging deals with generic APIs such as `sendMessage()`, `getmessage()`, and `registerMessageResponseCallback()`, whereas RPCs deal with special-purpose APIs that vary based on the interface of the target that is being invoked. For example, if you are trying to invoke a remote EJB that has a `processOrder()` method, you will most likely call the `processOrder()` method of a local object that acts like a proxy to the remote EJB. Chapter 6 discusses this topic in much more detail.

Another key difference between RPCs and messaging is that RPCs are direct invocations. There is no queuing mechanism; the backend must be running and it must be directly accessible at a well-known location. This limits the dispatch capabilities of RPC middleware. MOM message dispatch can be much more flexible.

Finally, extensive use of RPCs tends to result in somewhat brittle distributed systems. Because the APIs are fine-grained, even small changes in the data being passed around can break the system. Messaging uses much rougher-grain data exchanges and is therefore more likely to sustain small changes in the data being exchanged without failure.

Apart from these key differences, RPCs and messaging have many similarities:

- RPCs can be implemented on top of a request-response messaging pattern.
- Contrary to popular belief, however, RPCs do not have to have a request-response messaging pattern. Some systems support one-way RPCs.
• In addition, RPCs do not have to be synchronous. Some systems automatically spawn threads to wait in the background for RPC responses.

• RPCs and messaging share many of the same QoS requirements such as security and transaction management.

• Direct, synchronous, 1-to-1 messaging can be simulated via a simple RPC, e.g., void sendMessage(data).

It should become clear by now that the real issue isn’t which of the two approaches to distributed computing is better (the simple interpretation of “messaging vs. RPC”) but when each approach should be used in the world of Web services. To answer this question, after we have mentioned so many possible variations of both messaging and RPC, it helps to establish some stereotypes. When working with Web services, it will generally be the case that:

• RPCs will be direct, synchronous, request-response invocations that pass encoded application-level data structures from a client to a target backend that implements the RPC functionality.

• Messages will carry XML data. The interaction pattern is most likely to be one-way or request-response. Simple architectures will use direct messaging. The organization of participants will likely be 1-to-1. More advanced architectures will be queued and therefore asynchronous.

In both cases, messages will be represented on the wire using SOAP. QoS-related information that is part of the message will be represented as message headers. A good example would be an authentication header that carries a username and password; Chapter 5 shows an example.

Table 3.2 presents a number of benefits and concerns about using messaging and RPC. Based on these and the current state-of-the-art in Web service middleware and tooling, we would recommend that you go with a simple RPC-based solution or a direct messaging solution unless disconnected operation will be of benefit, the system requires 1-to-many interactions, or synchronous operation is causing performance problems.

**Table 3.2 Pros and Cons of Messaging and RPC for Web Services**

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct messaging</td>
<td>• The basic messaging APIs are very simple.</td>
</tr>
<tr>
<td></td>
<td>• Applications must perform manual data encoding/decoding.</td>
</tr>
<tr>
<td></td>
<td>• Any data can be passed.</td>
</tr>
<tr>
<td></td>
<td>• Separates data from the code that operates on it.</td>
</tr>
</tbody>
</table>
Table 3.2  Continued

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queued messaging</td>
<td>Same as above, plus...</td>
</tr>
<tr>
<td></td>
<td>Asynchronicity spreads the load and improves performance.</td>
</tr>
<tr>
<td></td>
<td>Allows for disconnected operation.</td>
</tr>
<tr>
<td></td>
<td>Allows for 1-to-many and many-to-many interactions.</td>
</tr>
<tr>
<td>RPC</td>
<td>Local APIs match backend APIs.</td>
</tr>
<tr>
<td></td>
<td>Synchronicity makes programming easy.</td>
</tr>
<tr>
<td></td>
<td>Application data is automatically encoded/decoded.</td>
</tr>
<tr>
<td></td>
<td>Exceptions provide a good error-handling mechanism.</td>
</tr>
<tr>
<td></td>
<td>RPC products interoperate reasonably well.</td>
</tr>
<tr>
<td></td>
<td>Same as above plus...</td>
</tr>
<tr>
<td></td>
<td>Most useful forms of messaging require a queuing infrastructure.</td>
</tr>
<tr>
<td></td>
<td>Current message queuing products do not interoperate well.</td>
</tr>
<tr>
<td></td>
<td>Asynchronicity makes programming more difficult.</td>
</tr>
<tr>
<td></td>
<td>Synchronicity can cause bottlenecks.</td>
</tr>
<tr>
<td></td>
<td>Backend must be running for RPCs to succeed.</td>
</tr>
<tr>
<td></td>
<td>Only 1-to-1 interactions are supported.</td>
</tr>
</tbody>
</table>

We would expect that as messaging middleware vendors embrace Web services to a greater extent and as more Web services become increasingly used in the context of complex business process workflows, the importance of Web service messaging will grow. Broad standardization efforts such as ebXML (http://www.ebxml.org) and Java API for XML Messaging (JAXM, http://java.sun.com/xml/jaxm/index.html) will help speed up the process.

SOAP-based RPCs

So far in this chapter we have presented several examples of SOAP-based RPC without ever mentioning the details of representing RPCs in SOAP messages as described by the SOAP specification. The rules are very simple.

Recall that to invoke an RPC, you need a target URI, an operation name, some parameters, and any amount of context information (such as security context). Any such context information is modeled as SOAP headers.
SOAP’s RPC binding does not specify how the target URI is going to be provided. In other words, it leaves it up to the SOAP processor to determine how to dispatch a SOAP RPC request to a target backend. There are three common ways to do this dispatch. Two of these are HTTP-specific, and the other is based on the contents of the SOAP message:

- In the case of HTTP, the SOAP processor can dispatch based on the target URI (as in the inventory check example).
- Alternatively, it may dispatch based on the value of the `SOAPAction` HTTP header that comes as part of the HTTP request.
- Alternatively, it can use the value of the namespace URI for the first element inside the SOAP body.

Most Web services engines do not support all these dispatch mechanisms. Axis can be configured to work with any combination.

In the language of the SOAP encoding, the actual RPC invocation is modeled as a struct. The name of the struct (that is, the name of the first element inside the SOAP body) is identical to the name of the method/procedure. This is not a problem, because the character set of XML elements is a superset of the character set of valid identifier names in programming languages. Every in and in-out parameter of the RPC is modeled as an accessor with a name identical to the name of the RPC parameter and type identical to the type of the RPC parameter mapped to XML according to the rules of the active encoding style. The accessors appear in the same order, as do the parameters in the operation signature.

The RPC response is also modeled as a struct. By convention, the name of the struct is the same as the name of the operation, with `Response` appended to it. There are accessors for the operation result and all in-out and out parameters. The result is the first accessor, followed by the parameters in the order they appear in the operation signature. By convention, the result element’s name is the same as the name of the operation, with `Result` appended to it.

Java developers are not used to the concept of in-out or out parameters because, typically, in Java all objects are automatically passed by reference. When using RMI, simple objects can be passed by value, but other objects are still passed by reference. In this sense, any mutable objects (ones whose state can be modified) are automatically treated as in-out parameters.

In Web services, the situation is different. All parameters are passed by value. SOAP has no notion of passing values by reference. This design decision was made in order to keep SOAP and its data encoding simple. Passing values by reference in a distributed system requires distributed garbage collection. This not only complicates the design of the system but also imposes restrictions on some possible system architectures and interaction patterns. For example, how can you do distributed garbage collection in a queued messaging architecture when the requestor and the provider of a service can both be offline at the same time?
Therefore, for Web services, the notion of in-out and out parameters does not involve passing objects by reference and letting the target backend modify their state. Instead, copies of the data are exchanged. It is then up to the service client code to create the perception that the actual state of the object that has been passed in to the client method has been modified. Different Web service clients might have different ways to do this.

Consider the following operation signature:

```java
boolean doCheck(in String sku, in int quantity, out int numInStock)
```

Some possible SOAP RPC request and response bodies are:

**RPC request body**

```xml
<soapenv:Body>
  <doCheck>
    <sku xsi:type="xsd:string">947-TI</sku>
    <quantity xsi:type="xsd:int">1</quantity>
  </doCheck>
</soapenv:Body>
```

**RPC response body**

```xml
<soapenv:Body>
  <doCheckResponse>
    <doCheckResult xsi:type="xsd:boolean">true</doCheckResult>
    <numInStock xsi:type="xsd:int">150</numInStock>
  </doCheckResponse>
</soapenv:Body>
```

Of course, if a description of the operation is available, you can generate a schema for all the elements in the SOAP body. Doing so would eliminate the need to use `xsi:type` everywhere in the SOAP message. Chapter 6 looks in more detail at the mechanisms for doing this.

**SOAP-based Messaging**

The technical term for non-RPC SOAP messaging is *document-centric* messaging. The name comes from the fact that the data sent over SOAP is represented as an XML document embedded inside the SOAP envelope. Although the RPC binding for SOAP has a number of rules governing the representation and encoding of operation names and parameters, simple SOAP messages have absolutely no restrictions as to the information that can be stored in their bodies. In short, any XML can be included in the SOAP message. The next section of this chapter shows an example of SOAP-based messaging.

**Purchase Order Submission Web Service**

Recall that when Al Rosen of Silver Bullet Consulting was investigating SkatesTown’s e-business processes, he noticed that one area that badly needed automation was purchase
order submission. Purchase orders and invoices were being exchanged over e-mail, and they were manually input into the company’s purchase order system.

Because SkatesTown already has defined an XML schema for its purchase orders and invoices, Al thinks it makes sense to build a purchase order Web service that accepts a purchase order as an XML document and returns an XML invoice. This service would be an example of 1-to-1 direct messaging using a request-response interaction pattern.

**Purchase Order and Invoice Schemas**

The schemas for SkatesTown’s purchase orders and invoices are explained in detail in Chapter 2. Listings 3.8 and 3.9 show example XML document instances for both.

**Listing 3.8  Example SkatesTown Purchase Order**

```xml
<po xmlns="http://www.skatestown.com/ns/po"
   id="50383" submitted="2001-12-06">
  <billTo>
    <company>The Skateboard Warehouse</company>
    <street>One Warehouse Park</street>
    <street>Building 17</street>
    <city>Boston</city>
    <state>MA</state>
    <postalCode>01775</postalCode>
  </billTo>

  <shipTo>
    <company>The Skateboard Warehouse</company>
    <street>One Warehouse Park</street>
    <street>Building 17</street>
    <city>Boston</city>
    <state>MA</state>
    <postalCode>01775</postalCode>
  </shipTo>

  <order>
    <item sku="318-BP" quantity="5">
      <description>Skateboard backpack; five pockets</description>
    </item>
    <item sku="947-TI" quantity="12">
      <description>Street-style titanium skateboard.</description>
    </item>
    <item sku="008-PR" quantity="1000"/>
  </order>
</po>
```
LISTING 3.9  Example SkatesTown Invoice

```xml
<invoice inv="http://www.skatestown.com/ns/invoice"
    id="50383" submitted="2001-12-06">
    <billTo>
        <company>The Skateboard Warehouse</company>
        <street>One Warehouse Park</street>
        <street>Building 17</street>
        <city>Boston</city>
        <state>MA</state>
        <postalCode>01775</postalCode>
    </billTo>
    <shipTo>
        <company>The Skateboard Warehouse</company>
        <street>One Warehouse Park</street>
        <street>Building 17</street>
        <city>Boston</city>
        <state>MA</state>
        <postalCode>01775</postalCode>
    </shipTo>
    <order>
        <item sku="318-BP" quantity="5" unitPrice="49.95">
            <description>Skateboard backpack; five pockets</description>
        </item>
        <item sku="947-TI" quantity="12" unitPrice="129.00">
            <description>Street-style titanium skateboard.</description>
        </item>
        <item sku="008-PR" quantity="1000" unitPrice="0.00">
            <description>Promotional: SkatesTown stickers</description>
        </item>
    </order>
    <tax>89.89</tax>
    <shippingAndHandling>89.89</shippingAndHandling>
    <totalCost>1977.52</totalCost>
</invoice>
```

XML-Java Data Mapping

Unfortunately, Al Rosen finds out that the actual SkatesTown purchase order system does not know how to deal with XML. The XML capabilities were added as an extension to the system by a developer who has since left the company. To make matters worse, much of the source code pertaining to XML processing seems to have been lost during an upgrade of the source control management (SCM) system at the company.
The PO system’s APIs work in terms of a set of Java beans representing concepts such as product, purchase order, invoice, address, and so on. Figure 3.15 shows a UML diagram.

![UML diagram](image)

**Figure 3.15**
*UML model for the PO system’s data objects.*

Al knows that because he is using SOAP-based messaging, the task of mapping the purchase order XML to Java objects and the invoice Java objects back to XML is left entirely up to him. Therefore, he implements a serializer and a deserializer that know how to encode and decode objects from the `com.skatestown.data` package to and from XML. Because the schemas for purchase orders and invoices are relatively simple, he decided to do this by hand rather than to rely on available schema compiler tools; he had no experience with these. The two classes that he builds are `Serializer` and `Deserializer` in the `com.skatestown.xml` package. The combined code size is slightly over 300 lines of Java code.

Listing 3.10 shows the key purchase order deserialization methods. They use a number of simple utility methods such as `getValue()` and `getElements()` to traverse the DOM representation of a purchase order and construct a purchase order and all its contained objects. Reusable functionality, such as reading the common properties of POItem and InvoiceItem or creating addresses, is put in separate methods (`readItem()` and `createAddress()`, respectively). This pattern for XML to Java data mapping is very simple and readable yet flexible to handle a large variety of input XML formats.
Listing 3.10  Core Purchase Deserialization Methods

```java
protected void readDocument(BusinessDocument doc, Element elem)
{
    doc.setId(Integer.parseInt(elem.getAttribute( "id" )));
    doc.setDate(elem.getAttribute("submitted"));
    doc.setBillTo(createAddress(getElement(elem, "billTo")));  
    doc.setShipTo(createAddress(getElement(elem, "shipTo")));  
}

protected void readItem(POItem item, Element elem)
{
    item.setSKU(elem.getAttribute("sku"));
    item.setQuantity(Integer.parseInt(elem.getAttribute("quantity")));
    item.setDescription( getValue( elem, "description" ) );
}

protected Address createAddress(Element elem)
{
    Address addr  = new Address();
    addr.setName( getValue( elem, "name" ) );  
    addr.setCompany( getValue( elem, "company" ) );
    addr.setStreet( getValues( elem, "street" ) );
    addr.setCity( getValue( elem, "city" ) );
    addr.setState( getValue( elem, "state" ) );
    addr.setPostalCode( getValue( elem, "postalCode" ) );
    addr.setCountry( getValue( elem, "country" ) );
    return addr;
}

protected PO _createPO(Element elem)
{
    PO po = new PO();
    readDocument(po, elem);
    Element[] orderItems = getElements(elem, "item" );
    POItem[] items = new POItem[orderItems.length];
    for (int i = 0 ; i < items.length; ++i)
    {
        POItem item = new POItem();
        readItem(item, elem);
        items[i] = item;
    }
}
```
Listing 3.11 shows the key invoice serialization methods. In this case, they traverse the Java data structures describing an invoice and use utility methods such as `addChild()` to construct a DOM tree representing an invoice document. Again, shared functionality such as serializing an address is separated in methods that are called from multiple locations.

### Listing 3.11  Core Invoice Serialization Methods

```java
protected void writeDocument(BusinessDocument bdoc, Element elem) {
    elem.setAttribute("id", ""+bdoc.getId());
    elem.setAttribute("submitted", bdoc.getDate());
    writeAddress(bdoc.getBillTo(), addChild(elem, "billTo"));
    writeAddress(bdoc.getShipTo(), addChild(elem, "shipTo"));
}

protected void writeAddress(Address addr, Element elem) {
    addChild(elem, "name", addr.getName());
    addChild(elem, "company", addr.getCompany());
    addChildren(elem, "street", addr.getStreet());
    addChild(elem, "city", addr.getCity());
    addChild(elem, "state", addr.getState());
    addChild(elem, "postalCode", addr.getPostalCode());
    addChild(elem, "country", addr.getCountry());
}

protected void writePOItem(POItem item, Element elem) {
    elem.setAttribute("sku", item.getSKU());
    elem.setAttribute("quantity", ""+item.getQuantity());
    addChild(elem, "description", item.getDescription());
}

protected void writeInvoiceItem(InvoiceItem item, Element elem) {
    writePOItem(item, elem);
    elem.setAttribute("unitPrice", nf.format(item.getUnitPrice()));
}
```
protected void writeInvoice(Invoice invoice, Element elem)
{
    writeDocument(invoice, elem);
    Element order = addChild(elem, "order");
    InvoiceItem[] items = invoice.getItems();
    for (int i = 0; i < items.length; ++i)
    {
        writeInvoiceItem(items[i], addChild(order, "item"));
    }
    addChild(elem, "tax", nf.format(invoice.getTax()));
    addChild(elem, "shippingAndHandling",
        nf.format(invoice.getShippingAndHandling()));
    addChild(elem, "totalCost", nf.format(invoice.getTotalCost()));
}

Service Requestor View
The PO Web service client implementation follows the same pattern as the invoice checker
clients (see Listing 3.12). The goal of its API is to hide the details of Axis-specific APIs from
the service requestor. Therefore, the invoke() method takes an InputStream for the purchase
order XML and returns the generated invoice as a string. Alternatively, the invoke() method
might have been written to take in and return DOM documents.

Listing 3.12  PO Submission Web Service Client
package ch3.ex4;

import java.io.*;
import org.apache.axis.encoding.SerializationContext;
import org.apache.axis.message.SOAPEnvelope;
import org.apache.axis.message.SOAPBodyElement;
import org.apache.axis.client.ServiceClient;
import org.apache.axis.Message;
import org.apache.axis.MessageContext;

/**
 * Purchase order submission client
 */
public class POSubmissionClient
{
Listing 3.12  Continued

* Target service URL
  */
private String url;

/**
 * Create a client with a target URL
 */
public POSubmissionClient(String targetUrl)
{
  url = targetUrl;
}

/**
 * Invoke the PO submission web service
 *
 * @param po Purchase order document
 * @return Invoice document
 * @exception Exception I/O error or Axis error
 */
public String invoke(InputStream po) throws Exception
{
  // Send the message
  ServiceClient client = new ServiceClient(url);
  client.setRequestMessage(new Message(po, true));
  client.invoke();

  // Retrieve the response body
  MessageContext ctx = client.getMessageContext();
  Message outMsg = ctx.getResponseMessage();
  SOAPEnvelope envelope = outMsg.getAsSOAPEnvelope();
  SOAPBodyElement body = envelope.getFirstBody();

  // Get the XML from the body
  StringWriter w = new StringWriter();
  SerializationContext sc = new SerializationContext(w, ctx);
  body.output(sc);
  return w.toString();
}

Sending the request message is simple. We have to create a ServiceClient from the target URL and set its request message to a message constructed from the purchase order input stream. The second parameter to the Message constructor, the boolean true, is an indication
that the input stream represents the message body as opposed to the whole message. Calling `invoke()` sends the message to the Web service.

The second part of the method has to do with retrieving the body of the response message. This code should be familiar from the implementation of the E-mail header handler.

Finally, we use an Axis serialization context to write the XML in the response body into a `StringWriter`. We could have easily gotten the body as a DOM element by calling `getAsDOM()` method. The trouble is, there is no standard way in DOM Level 2 to convert a DOM element into a string! Java API for XML Processing (JAXP) defines such a mechanism in its transformation API (javax.xml.transform package), but the method is fairly cumbersome. It is easiest to use an Axis `SerializationContext` object.

### Service Provider View

The implementation of the purchase order submission service is very simple (see Listing 3.13). Because this is not an RPC-based service, the input and output are both XML documents (represented via DOM `Document` objects). The input document is deserialized to produce a purchase order object. It is passed to the actual PO processing backend. Its implementation is not shown here because it has nothing to do with Web services. It looks up item prices by their SKU, calculates totals based on item quantities, and adds tax and shipping and handling. The resulting invoice object is serialized to produce the result of the purchase order submission service.

#### Listing 3.13 Purchase Order Submission Web Service

```java
package com.skatestown.services;

import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.apache.axis.MessageContext;
import com.skatestown.backend.*;
import com.skatestown.data.*;
import com.skatestown.xml.*;
import bws.BookUtil;

/**
 * Purchase order submission service
 */
public class POProcess
{
    /**
     */
```
Finally, adding deployment information about the new service involves making a small change to the Axis Web services deployment descriptor (see Listing 3.14). Again, Chapter 4 will go into the details of Axis deployment descriptors.

### Listing 3.14 Deployment Descriptor for Inventory Check Service

```xml
<!-- Chapter 3 example 4 services -->
<service name="POSubmission" pivot="MsgDispatcher">
  <option name="className" value="com.skatestown.services.POSubmission"/>
  <option name="methodName" value="doSubmission"/>
</service>
```

### Putting the Service to the Test

A simple JSP test harness in ch3/ex4/index.jsp (see Figure 3.16) tests the purchase order submission service. By default, it loads /resources/samplePO.xml, but you can modify the purchase order on the page and see how the invoice you get back changes.
**FIGURE 3.16**
*Putting the PO submission Web service to the test.*

### SOAP on the Wire

With the help of TCPMon, we can see what SOAP messages are passing between the client and the Axis engine:

```xml
POST /bws/services/POSubmission HTTP/1.0
Host: localhost
Content-Length: 1169
Content-Type: text/xml; charset=utf-8
SOAPAction: ""

<?xml version="1.0" encoding="UTF-8"?><soap:Envelope
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soap:Body>
    <po xmlns="http://www.skatestown.com/ns/po" id="50383" submitted="2001-12-06">
      <billingTo>
        <company>The Skateboard Warehouse</company>
        <street>One Warehouse Park</street>
        <city>Boston</city>
        <state>MA</state>
        <postalCode>02175</postalCode>
      </billingTo>
      <shipTo>
        <company>The Skateboard Warehouse</company>
        <street>One Warehouse Park</street>
      </shipTo>
    </po>
  </soap:Body>
</soap:Envelope>
```

---

**SIMPLE OBJECT ACCESS PROTOCOL (SOAP)**

**Example 3.4**

*Purchase Order Submission*

This example implements a web form driven for SkateTown’s purchase order submission client. You can modify the purchase order on the form if you wish (the default one is from Chapter 2).

```xml
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Body>
    <po xmlns="http://www.skatestown.com/ns/po"
        id="50383" submitted="2001-12-06">
      ...
    </po>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```
The target URL is /bws/services/POSubmission. The response message simply carries an invoice inside it, much in the same way that the request message carries a purchase order. As a result, there is no need to show it here.

That’s all there is to taking advantage of SOAP-based messaging. Axis makes it very easy to define and invoke services that consume and produce arbitrary XML messages.

Figure 3.17 shows one way to think about the interaction of abstraction layers in SOAP messaging. It is modeled after Figure 3.3 earlier in the chapter but includes the additional role of a service developer. As before, the only “real” on-the-wire communication happens between the HTTP client and the Web server that dispatches a service request to Axis.

The abstractions at this level are HTTP packets. At the Axis level, the abstractions are SOAP messages with some additional context. For example, on the provider side, the target service is determined by the target URL of the HTTP packet. This piece of context information is “attached” to the actual SOAP message by the Axis servlet that listens for HTTP-based Web service requests. The job of a service-level developer is to create an abstraction layer that maps Java APIs to and from SOAP messages. During SOAP messaging, a little more work needs to happen at this level than when doing RPCs. The reason is that data must be manually encoded and decoded by both the Web service client and the Web service backend. Finally, at the top of the stack on both the requestor and provider sides sits the application developer who is happily insulated from the fact that Web services are being used and that Axis is the Web service engine. The application developer needs only to understand the concepts pertaining to his application domain—in this case, purchase orders and invoices.
SOAP Protocol Bindings

So far in this chapter, we have only shown SOAP being transmitted over HTTP. SOAP, however, is transport-independent and can be bound to any protocol type. This section looks at some of the issues involved in building Web services and transporting SOAP messages over various protocols.

General Considerations

The key issue in deciding how to bind SOAP to a particular protocol has to do with identifying how the requirements for a Web service (RPC or not, interaction pattern, synchronicity, and so on) map to the capabilities of the underlying transport protocol. In particular, the task at hand is to determine how much of the total information needed to successfully execute the Web service needs to go in the SOAP message versus somewhere else.

As Figure 3.18 shows with an HTTP example, many protocols have a packaging notion. If SOAP is to be transmitted over such protocols, a distinction needs to be made between physical (transport-level) and logical (SOAP) messages. Context information can be passed in both.

In the case of HTTP, context information is passed via the target URI and the SOAPAction header. Security information might come as HTTP username and password headers. In the case of SOAP, context information is passed as SOAP headers.

![Figure 3.18](Logical versus physical messages.)
Sometimes, SOAP messages have to be passed over protocols whose physical messages do not have any mechanism for storing context. Consider pure sockets-based exchanges. By default, in these cases the physical and the logical message are one and the same. In these cases, you have four options for passing context information:

- By convention, as in, “When listening on port 12345, I know that I have to invoke service X.”
- By entirely using SOAP’s header-based extensibility mechanism to pass all context information.
- By custom-building a very light physical protocol under SOAP messages, as in, “The first CRLF delimited line of message will be the target URI; the rest will be the SOAP message.”
- By using a lightweight protocol that can be layered on top of the physical protocol and can be used to move SOAP messages. Examples of such protocols are Simple MIME Exchange Protocol (SMXP) or Blocks Extensible Exchange Protocol (BEEP).

As in most cases in the software industry, reinventing the wheel is a bad idea. Therefore, the second and fourth approaches listed here typically make the most sense. The first approach is not extensible and can leave you in a tight spot if requirements change. The third approach smells of reinventing the wheel. The cost of going with the second approach is that you have to make sure that all clients interacting with your Web service will be able to support the necessary extensions. The cost of going with the fourth approach is that it might require additional infrastructure for both requestors and providers.

Another consideration that comes into play is the interaction pattern supported by the transport protocol. For example, HTTP is a request-response protocol. It makes RPCs and request-response messaging interactions very simple. For other protocols, you might have to explicitly manage the association of requests and responses. As we mentioned in the previous section, Chapter 6 discusses this topic in more detail.

Contrary to popular belief, Web services do not have to involve stateless interactions. For example, Web services could be designed in a session-oriented manner. This is probably not the best design for a high-volume Web service, but it could work fine in many cases. HTTP sessions can be leveraged to provide context information related to the session. Otherwise, you will have to use a session ID of some kind, much in the same way a message conversation ID is used.

Finally, when choosing transport protocols for Web services, think carefully about external requirements. You may discover important factors entirely outside the needs of the Web service engine. For example, when considering Web services over sockets as a higher-performance alternative to Web services over HTTP (requests and responses don’t have to go through the Web server), you might want to consider the following factors:
• If services have to be available over a public unsecured network, is it an acceptable risk to open a hole through the firewall for Web service traffic?

• Can clients support SSL to ensure the privacy of messages? Surprisingly, some clients can speak HTTPS but not straight SSL.

• What are the back-end load balancing and failover requirements? Straight sockets-based communication requires sticky load balancing. You establish a session with one server and you have to keep using this server. This approach potentially compromises scalability and failover, unless steps are taken to build request redirection and session persistence and failover capabilities into the system.

As with most things in the software industry, there is no single correct approach and no single right answer. Investigate your requirements carefully and do not be easily tempted by seemingly exciting, out-of-the-ordinary solutions. The rest of this section provides some more details about how certain protocols can be used with SOAP.

**HTTP/S**

This chapter has shown many examples of SOAP over HTTP. The SOAP specification defines a binding of SOAP over HTTP with the following set of rules:

• The MIME media type of both HTTP requests and responses (defined in the Content-Type HTTP header) must be text/xml.

• Requests must come as HTTP POST operations.

• The SOAPAction header is reserved as a hint to the SOAP processor as to which Web service is being invoked. The value of the header can be any URI; it is implementation-specific.

• Successful SOAP message processing must return an HTTP error code in the 200 range. Typically, this is 200 OK.

• In the case of an error processing the SOAP message, the HTTP response code must be 500 Internal Server Error and it must include a SOAP message with a Fault element describing the error.

In addition to these simple rules, the SOAP specification defines how SOAP messages can be exchanged over HTTP using the HTTP Extension Framework (RFC 2774, http://www.norms.org/ietf/rfc/rfc2774.txt), but this information is not very relevant to us.

In short, HTTP is the most commonly used mechanism for exchanging SOAP messages. It is aided by the industry’s experience building relatively secure, scalable, reliable networks to handle HTTP traffic and by the fact that traditional Web applications and application servers primarily use HTTP. HTTP is not perfect, but we are very good at working around its limitations.
For secure message exchanges, you can use HTTPS instead of HTTP. The most common extension on top of what the SOAP specification describes is the use of HTTP usernames and passwords to authenticate Web service clients. Combined with HTTPS, this approach offers a good-enough level of security for most e-commerce scenarios. Chapter 5 discusses the role of HTTPS in Web services.

SOAP Messages with Attachments

SOAP messages will often have attachments of various types. The prototypical example is an insurance claim form in XML format that has an accident picture associated with it and/or a scanned copy of the signed accident report form. The SOAP Messages with Attachments specification defines a simple mechanism for encoding a SOAP message in a MIME multipart structure and associating this message with any number of parts (attachments) in that structure. These attachments can be in their native format, which is typically binary.

Without going into too many details, the SOAP message becomes the root of the multipart/related MIME structure. The message refers to attachments using a URI with the cid: prefix, which stands for “content ID” and uniquely identifies the parts of the MIME structure. Here is how this is done. Note that some long lines (such as the Content-Type header) have been broken in two for better readability:

```
MIME-Version: 1.0
Content-Type: Multipart/Related; boundary=MIME_boundary; type=text/xml;
             start="<claim061400a.xml@claiming-it.com>"
Content-Description: This is the optional message description.

--MIME_boundary
Content-Type: text/xml; charset=UTF-8
Content-Transfer-Encoding: 8bit
Content-ID: <claim061400a.xml@claiming-it.com>

<?xml version='1.0' ?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
><SOAP-ENV:Body>
  ..
  <theSignedForm href="cid:claim061400a.tiff@claiming-it.com"/>
  ..
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

--MIME_boundary
Content-Type: image/tiff
```
One excellent thing about encapsulating SOAP messages in a MIME structure is that the packaging is independent of an actual transport protocol. In a sense, the MIME package is another logical message on top of the SOAP message. This type of MIME structure can then be bound to any number of other protocols. The specification defines a binding to HTTP, an example of which is shown here:

POST /insuranceClaims HTTP/1.1
Host: www.risky-stuff.com
Content-Type: Multipart/Related; boundary=MIME_boundary; type=text/xml;
        start="<claim061400a.xml@claiming-it.com>"
Content-Length: XXXX
SOAPAction: http://schemas.risky-stuff.com/Auto-Claim

SOAP over SMTP

E-mail is pervasive on the Internet. The important e-mail-related protocols are Simple Mail Transfer Protocol (SMTP), Post Office Protocol (POP), and Internet Message Access Protocol (IMAP). E-mail is a great way to exchange SOAP messages when synchronicity is not required because:

- E-mail messages can easily carry SOAP messages.
- E-mail messages have extensible headers that can be used to transmit context information outside the SOAP message body.
- Both sending and receiving of e-mail messages can be configured to require authentication. Further, using S/MIME with e-mail provides additional security for a range of applications.
- E-mail can support one-to-one and one-to-many participant configurations.
- E-mail messaging is buffered and queued with reliable dispatch that automatically includes multiple retries and failed delivery notification.
- The Internet e-mail server infrastructure is highly scalable.

Together, these factors make e-mail a very suitable alternative to HTTP for asynchronous Web service messaging applications.
Other Protocols

Despite its low-tech nature, FTP can be very useful for simple one-way messaging using Web services. Access to FTP servers can be authenticated. Further, roles-based restrictions can be applied to particular directories on the FTP server. When using FTP, SOAP messages are mapped onto the files that are being transferred. Typically, the file names indicate the target of the SOAP message.

In addition, with companies such as Microsoft backing SMXP for their Hailstorm initiatives, the protocol is emerging as a potential candidate to layer on top of straight socket-based communications for transmission of SOAP messages.

Finally, sophisticated messaging infrastructures such as IBM’s MQSeries, Microsoft’s Message Queue (MSMQ), and the Java Messaging Service (JMS) are well-suited for the transport of SOAP messages. Chapter 5 shows an example of SOAP messaging using JMS.

The key constraint limiting the wide deployment of SOAP bindings to protocols other than HTTP and e-mail is the requirement of Web service interoperability. HTTP and e-mail are so pervasive that they are likely to remain the preferred choices for SOAP message transport for the foreseeable future.

Summary

This chapter addressed the fourth level of the Web services interoperability stack—XML messaging. It focused on explaining some of the core features of XML protocols and SOAP 1.1 as the de facto standard for Web service messaging and invocation. The goal was to give you a solid understanding of SOAP and a first-hand experience building and consuming Web services using the Apache Axis engine. To this end, we covered, in some detail:

- The evolution of XML protocols from first-generation technologies based on pure XML 1.0 (WDDX and XML-RPC) to XML Schema and Namespace powered second-generation protocols, of which SOAP is a prime example. The chapter also discussed the motivation and history behind SOAP’s creation.
- The simple yet flexible design of the SOAP envelope framework, including versioning and vertical extensibility using SOAP headers. In SOAP, all context information orthogonal to what is in the SOAP body is carried via headers. SOAP’s envelope framework allows you to design higher-level protocols on top of SOAP in a decentralized manner.
- SOAP intermediaries as the key innovation enabling horizontal extensibility. Because of intermediaries, Web services can be organized into very flexible system and network architectures and value-added services can be provided on top of basic Web service messaging.
• SOAP error handling using SOAP faults. Any robust messaging protocol needs a well-designed exception-handling model. With their ability to communicate error information targeted at both software and humans, as well as clearly identifying the source of the error condition, SOAP faults make it possible to integrate SOAP as part of robust, mission-critical systems.

• Encoding data using SOAP. The chapter covered both SOAP’s abstract data model encoding and a number of other heuristics for determining an appropriate data representation model for SOAP messages.

• Using SOAP for both messaging and RPC applications. By design, SOAP is independent of all traditional aspects of messaging: participant organization, interaction pattern, synchronicity, and so on. As a result, SOAP can be used for just about any distributed system. This chapter provided some guidelines that help narrow the space of what is possible to the space of what makes sense in the real-world solutions.

• Using SOAP over multiple protocols. The SOAP specification mentions an HTTP binding for SOAP, but Web services can be meaningfully bound to many other packaging and protocol schemes: MIME packages to support attachments, SMTP for scalable asynchronous messaging without the need for special middleware, and many others.

During the course of the chapter, we developed two meaningful e-commerce Web services for SkatesTown: an inventory check RPC service (with or without e-mail confirmations) and a purchase order submission messaging service. Our implementation on both the server and the client used design best practices for separating data and business logic from the details of SOAP and XML processing.

The Road Ahead

This chapter focused on the de facto standard protocol for Web service invocation as of the time of this writing—SOAP 1.1. (SOAP 1.2 is still in early draft stage.) However, many more pieces to the puzzle are required to bring meaningful Web services-enabled business solutions online. The rest of the book will complete the Web services puzzle. Chapter 5 focuses on building secure, robust, scalable enterprise-grade Web services. Chapter 6 introduces the concept of service descriptions and the Web Services Description Language (WSDL). Chapter 7 discusses service registries and the Universal Description, Discovery and Integration (UDDI) effort. Chapter 8 reviews the state of the currently available Web services tooling. Chapter 9 looks at the exciting world of Web service futures. This said, the next chapter offers a short detour for those that are truly excited about building and consuming extensible, high-performance Web services—it is about building Web services using the advanced features of Apache Axis.
Resources