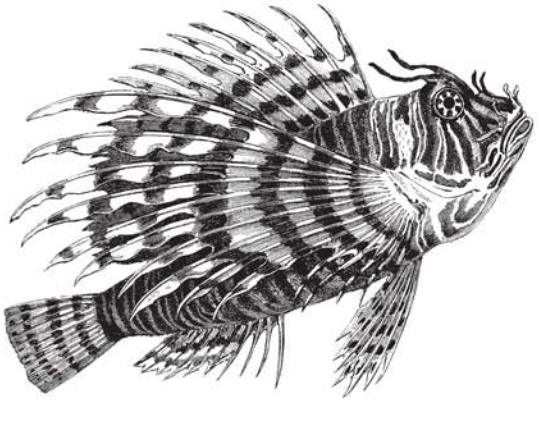
Building Web Services in Java

Java™ and SOAP



O'REILLY®

Robert Englander

O'REILLY®

Java and SOAP

Java and SOAP provides Java developers with an in-depth look at SOAP, the Simple Object Access Protocol. Of course, it covers the basics: what SOAP is, why it has soared to a spot on the Buzzwords' Top Ten list, and what its features and capabilities are. And it shows you how to work with some of the more common Java APIs in the SOAP world: Apache SOAP and GLUE.

In addition to covering fundamentals such as the structure of a SOAP message, SOAP encoding, and building simple services using RPC and messaging, *Java and SOAP* covers many topics that are essential to real-world development. Although SOAP has native support for an impressive number of object types, the nature of modern programming means that whatever SOAP gives you is not enough. When do you need to add support for your own object types, and how do you do it? How do you handle errors, and how can you add your own information to fault messages? How do you handle attachments?

In an ideal world, you could live entirely within Java, and ignore the SOAP messages being sent back and forth; you'd be able to write Java code and let the SOAP APIs work behind the scenes. However, we're not yet in that ideal world, and won't be for some time. Therefore, *Java and SOAP* pays particular attention to how SOAP messages are encoded. It doesn't just explain the document types, but shows how the documents are used in practice, as they are generated by the different APIs. If you ever have to debug interoperability problems, you'll find that this information is indispensable. Moreover, the best software is written by people who understand what's happening under the hood, and SOAP is no different. Say you need to write a custom serializer to create a SOAP representation of a structure. How do you know that your encoding is efficient? There's one definitive answer: look at the SOAP documents it produces.

In addition, *Java and SOAP* discusses interoperability between the major SOAP platforms, including Microsoft's .NET. It also covers SOAP messaging, SOAP attachments, and message routing, and provides previews of the forthcoming Axis APIs, JAX-RPC, and JAXM. If you're a Java developer who would like to start working with SOAP, this is the book you need to get going.

Robert Englander is Principal Engineer and President of MindStream Software, Inc. (*www.mindstrm.com*). He provides consulting services in software architecture, design, and development, as well as developing frameworks for use on client projects.

www.oreilly.com

US $39.95 **CAN $61.95**

ISBN-10: 0-596-00175-4

ISBN-13: 978-0-596-00175-9

9 780596 001759 53995

Java™ and SOAP

Related titles from O'Reilly

Ant: The Definitive Guide

Building Java™ Enterprise Applications

Database Programming with JDBC and Java™

Developing Java Beans™

Enterprise JavaBeans™

J2ME™ in a Nutshell

Java™ 2D Graphics

Java™ & XML

Java™ and XML Data Binding

Java™ and XSLT

Java™ Cookbook

Java™ Cryptography

Java™ Distributed Computing

Java™ Enterprise in a Nutshell

Java™ Examples in a Nutshell

Java™ Foundation Classes in a Nutshell

Java™ I/O

Java™ in a Nutshell

Java™ Internationalization

Java™ Message Service

Java™ Network Programming

Java™ Performance Tuning

Java™ Programming with Oracle SQLJ

Java™ RMI

Java™ Security

JavaServer™ Pages

JavaServer™ Pages Pocket Reference

Java™ Servlet Programming

Java™ Swing

Java™ Threads

Learning Java™

Also available

The Java™ Enterprise CD Bookshelf

Java™ and SOAP

Robert Englander

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

Java™ and SOAP
by Robert Englander

Published by O'Reilly & Associates, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly & Associates books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor:	Mike Loukides
Production Editor:	Emily Quill
Cover Designer:	Emma Colby
Interior Designer:	Melanie Wang

Printing History:

May 2002: First Edition.

Library of Congress Cataloging-in-Publication Data

Englander, Robert.
Java and SOAP/ Robert Englander.
p. cm.
ISBN 0-596-00175-4
1. Internet programming. 2. Object oriented programming. 3. Simple object access protocol. 4. Java (Computer programming language) I. Title.

QA76.76.H94 B54 2002
005.2'76--dc21 2002023322

[M] [6/02]

Once again, for my daughter Jessica.

Table of Contents

Preface . . . . . **xi**

1. Introduction **1**
RPC and Message-Oriented Distributed Systems 2
Self-Describing Data 3
XML 4
API Specs Versus Wire-Level Specs 4
Overview of SOAP 5
SOAP Implementations 6
The Approach 8
Getting Started 8

2. The SOAP Message **9**
The HTTP Binding 9
HTTP Request 10
HTTP Response 11
The SOAP Envelope 13
The Envelope Element 16
The Header Element 17
The actor Attribute 18
The mustUnderstand Attribute 19
The encodingStyle Attribute 20
Envelope Versioning 21
The Body Element 21
SOAP Faults 22

3. SOAP Data Encoding 25
Schemas and Namespaces 25
Serialization Rules 28
Indicating Type 31
Default Values 42
The SOAP Root Attribute 43

4. RPC-Style Services 45
SOAP RPC Elements 45
A Simple Service 51
Deploying the Service 52
Writing Service Clients 62
Deploying with Request-Level Scope 71
Deploying with Session-Level Scope 73
Passing Parameters 75

5. Working with Complex Data Types 85
Passing Arrays as Parameters 85
Returning Arrays 94
Passing Custom Types as Parameters 97
Returning Custom Types 109

6. Custom Serialization 115
Custom Type Encoding 116

7. Faults and Exceptions 140
Throwing Server-Side Exceptions in Apache SOAP 140
Creating a Fault Listener in Apache SOAP 143
Throwing and Catching Exceptions in GLUE 148

8. Alternative Techniques 152
SOAP Messaging 153
Literal Encoding 163

9. SOAP Interoperability and WSDL 177
Web Services Definition Language 178
Calling a GLUE Service from an Apache SOAP Client 187
A Proxy Service Using Apache SOAP 193
Calling an Apache SOAP Service from a GLUE Client 198

Accessing .NET Services 204
Writing an Apache Axis Client 210

10. SOAP Headers 213
Apache SOAP Providers and Routers 214
Replacing the Provider and Router Classes 214
An Apache SOAP Service That Handles SOAP Headers 219

11. JAX-RPC and JAXM 226
JAX-RPC 227
Working Without Ant 228
Creating a JAX-RPC Service 229
Creating a JAX-RPC Client 235
Generating Stubs from WSDL 237
Dynamic Invocation Interface 239
JAXM, in Less Than a Nutshell 239
What Next? 240

Index 241

Preface

The Simple Object Access Protocol, or SOAP, is the latest in a long line of technologies for distributed computing. It differs from other distributed computing technologies in that it is based on XML, and also that thus far it has not attempted to redefine the computing world. Instead, the SOAP specification describes important aspects of data content and structure as they relate to familiar programming models like remote procedure calls (RPCs) and message passing systems.

These specifications live squarely in the world of XML. SOAP is not bound to a specific programming language, computing platform, or software development environment. There are SOAP implementations that provide bindings for a variety of programming languages like C#, Perl, and Java™. Without these implementations SOAP remains in the abstract: a great concept without manifestation. It is the binding to software development languages that makes SOAP come alive, and that is what this book is about. Java is a natural for XML processing, making it perfect for building SOAP services and client applications. If building SOAP-aware software in Java is what you want to do, this book is just what you need to get started.

Intended Audience

This book is for everyone interested in how to access SOAP-based web services in Java, as well as how to build SOAP-based services in Java. It's written for programmers, students, and professionals who are already familiar with Java, so it doesn't spend any time covering the basic concepts or syntax of the language. If you aren't familiar with Java, you may want to keep a copy of a Java language book, like O'Reilly's *Learning Java* or *Java in a Nutshell,* close by.

A Moment in Time

The SOAP specification is still evolving. This book describes SOAP according to Version 1.1 of the spec. Although the concepts and techniques covered should continue to be relevant in future SOAP releases, there will certainly be important additions to SOAP as new versions of the spec are finalized. The Java implementations we'll be looking at will continue to evolve as well. Obviously, the descriptions and examples in this book will become dated or even obsolete over time—and that time will probably be sooner rather than later, given the speed at which web services are evolving. In fact, the handwriting is already on the wall: Apache SOAP Version 2, on which many of the examples are based, is destined to be replaced by Apache SOAP 3 (also known as Axis), which is currently available in an early release and is discussed briefly in Chapter 9. Axis, in turn, is committed to supporting the JAX RPC and JAXM API specifications, which are themselves still under development. An early access release of the reference implementation for these specifications is available from Sun Microsystems (and discussed in Chapter 11); this release is more recent than the most recent release of Axis. And it would be foolish to think that the JAX Pack specifications will mark the end of the evolutionary process. However, when the inevitable happens, you'll be armed with the knowledge and understanding necessary to keep pace with the changes.

How This Book Is Organized

The chapters in this book are organized so that each one builds upon the information presented in previous chapters, so it's best if you read the chapters in order.

Chapter 1, *Introduction*
: This chapter provides an overview of SOAP, including related technologies, problem spaces, and comparisons to other solutions. It also introduces Apache SOAP and GLUE, the SOAP implementations that will be used throughout the book.

Chapter 2, *The SOAP Message*
: This chapter describes the SOAP Envelope, a structured XML document that carries the payload of a SOAP transaction between client and server. It covers all aspects of a SOAP Envelope, including Headers, SOAP Body elements, and Faults. Some details of the SOAP HTTP binding are also included.

Chapter 3, *SOAP Data Encoding*
: This chapter covers the data encoding of a SOAP transaction, including rules for encoding and serializing data elements. It starts out with a description of namespaces, and then delves into the serialization of both simple and complex data types.

Chapter 4, *RPC-Style Services*

This chapter goes deep into SOAP-based remote procedure call (RPC) style services. Extensive coverage of service methods and parameters is provided, along with the details of service deployment and activation mechanisms.

Chapter 5, *Working with Complex Data Types*

This chapter looks at the creation of services with complex method parameters and return values such as arrays and Java beans. It covers the mechanisms available for mapping these types to Java classes on both client and server systems.

Chapter 6, *Custom Serialization*

This chapter covers the use of nonstandard custom data types, picking up where Chapter 5 left off. It looks at some of the tools and APIs used to pass instances of custom data types as parameters and return values. It also details the techniques of writing Java classes for serializing and deserializing custom types.

Chapter 7, *Faults and Exceptions*

This chapter describes SOAP Faults, along with their relationship to Java exceptions. It looks at the default mechanisms provided, as well as techniques for generating and extending the contents of Faults.

Chapter 8, *Alternative Techniques*

This chapter starts out by describing the use of SOAP message-style services, an alternative to the RPC model. It also looks at passing literal XML inside of a SOAP Envelope, and finishes up with a look at SOAP Attachments.

Chapter 9, *SOAP Interoperability and WSDL*

This chapter looks at getting SOAP clients and servers, developed using different technologies, to work properly together. An introduction to the Web Services Description Language (WSDL) is provided. Examples are developed that cover clients and services built using Apache SOAP and GLUE, a sneak peek at Apache Axis, and Java clients accessing Microsoft .NET services.

Chapter 10, *SOAP Headers*

This chapter looks at the use of SOAP Headers, which provide a means to pass data between clients and services that lie outside the scope of the SOAP Body. It covers the development of an intermediary service that acts as a message router to another service. Some Java classes are developed for extending the Apache SOAP framework in order to work with SOAP Headers.

Chapter 11, *JAX-RPC and JAXM*

This chapter examines the emerging standard: the Java API for XML-based RPC (JAX-RPC). It's a look at an early release of Sun's reference implementation. This chapter covers the development of both a service and a client, and also looks at using the tools to develop code for accessing services described by WSDL. A final commentary on JAXM is also included.

Conventions Used in This Book

`Constant Width` is used for:

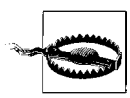
- Anything that might appear in a Java program, including keywords, operators, data types, constants, method names, variable names, class names, interface names, and Java package names.
- Command lines and options that should be typed verbatim on the screen.
- Namespaces.

Italic is used for:

- Pathnames, filenames, and Internet addresses, such as domain names and URLs. Italics is also used for executable files.

Making fine distinctions in a book like this is generally a losing battle. But I have tried to distinguish between namespaces (constant width) and URLs (italic), even though they look identical. Likewise, I've tried to distinguish between Java methods (constant width and ending in a pair of parentheses) and the methods exported by the SOAP service (constant width, no parentheses).

This icon signifies a note relating to the nearby text.



This icon signifies a warning relating to the nearby text.

How to Contact Us

I've certainly tried to be accurate in my descriptions and examples, but errors and omissions will inevitably exist. If you find mistakes, or you think I've left out important details, or you'd like to contact me for some other reason related to this work, you can contact me directly at:

rob@mindstrm.com

Alternately, address comments and questions concerning this book to the publisher:

O'Reilly & Associates, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (in the United States or Canada)
(707) 829-0515 (international/local)
(707) 829-0104 (fax)

There is a web page for this book, which lists errata, examples, or any additional information. You can access this page at:

http://www.oreilly.com/catalog/javasoap

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about books, conferences, Resource Centers, and the O'Reilly Network, see the O'Reilly web site at:

http://www.oreilly.com

Retrieving Examples Online

The code for the examples throughout this book is available online at:

http://www.mindstrm.com/javasoap

Acknowledgments

My good friend Rinaldo DiGiorgio continues, to this day, to keep me interested in Java and its related technologies. I don't think anyone has been a greater influence on my Java work than he has. Thanks, Rinaldo, for keeping me on the right path.

Many thanks go to David Askey and Anne Thomas Manes for reviewing the book and providing valuable feedback. They managed to find errors and offer advice that makes this a better book than it would have been without their help. Thanks to Lorraine Pecorelli for reading every chapter and making sure the words made sense. My deepest appreciation goes to Mike Loukides, the editor of this book. There were many obstacles to getting this project finished, and Mike's commitment and loyalty was key to turning the effort into a book. A thank you also is due to the O'Reilly design and production crew.

And finally, thanks to my family, Jessica and Carolyn, for their support. I'm not going to thank my friends this time—they were no help at all!

CHAPTER 1
Introduction

In the history of software development, new approaches frequently bring discarded ideas back into the mainstream of common practice. Each time an idea is revisited, prior successes and failures become invaluable aides in improving the concept and making its implementation better, or at least more usable. Now I'm not saying that we keep reinventing the wheel; rather, we keep going back and improving the wheel. And doing so can often be the catalyst for new ideas and new technologies that were not possible with the old wheel.

We've seen centralized computing with mainframes and their associated terminals come back disguised as application servers and thin clients. We've seen the concept of P-Code return in the form of interpreted languages like Java and Visual Basic. The universe of software development seems to expand and contract like, well, the cosmic Universe. If you wait around long enough, you may just be able to use the work you're doing today at some time in the future.

The point is, really, that a good idea is a good idea, regardless of whether it's a new idea. Timeliness is what matters most. So it goes for the world of distributed computing. The concept isn't new, but it gets revisited constantly. Pervasive infrastructure and technologies like the Internet, web browsers, and their associated protocols have allowed us to go back and advance the state of distributed computing. The evolution's latest craze is web services.

Web services are basically server functions that have published the interface mechanisms needed to access their capabilities. They're being implemented in a wide variety of technologies, but have a very important thing in common: they are providers of computational services that can be accessed using a standardized protocol. For instance, you might find a stock quote service that can return current stock market pricing and trading information based on a company's stock symbol. This is a very specific function, and that's the essence of web services. They do not provide the breadth of capability found in application servers—they provide small, focused capabilities that are likely to prove useful when combined with other services. You can

imagine an online trading application that makes use of web services ranging from stock quotes to trade execution to banking transactions. The vision of web services is that it will ultimately be possible to create complex applications on the fly—or at least, with minimal development time—by combining bits and pieces of data and services that are distributed across the Web. Sun's slogan used to be "The network is the computer," and that vision is certainly coming to fruition.

RPC and Message-Oriented Distributed Systems

Distributed systems exist, for the most part, as loosely coupled entities that communicate with each other to accomplish some task. One of the most common models used in distributed software is the remote procedure call (RPC). One reason for the popularity of RPC systems is that they closely resemble the function/method call syntax and semantics that we as programmers are so familiar with. Technologies like Java RMI, Microsoft's COM, and CORBA all use this kind of model. Of course, you have to jump through many hoops before making the ever-familiar method call to a remote system, but even with all that it still feels remarkably like making a local method call. Often, once the method call returns we don't care how it happened.* Much of the work in providing that abstraction to programmers at the API level is what makes up the majority of the distributed systems implementations.

Another popular model for distributed computing is message passing. Unlike the RPC model, messaging does not emulate the syntax of programming language function calls. Instead, structured data messages are passed between parties. These messages can serve to decouple the nodes of the distributed system somewhat, and message-based systems often prove to be more flexible than RPC-based systems. However, that flexibility can sometimes be just enough rope for programmers to hang themselves.

It seems that a reasonable, and powerful, compromise might be to combine these two models. Can you imagine a system that provides for the familiarity and ease of use of RPC-style invocations, along with some of the flexibility of message-type systems? It seems to me that it would require the definition of a data format that could describe itself, while at the same time conforming to a standard set of rules governing that very description. Hmmm…

* I'm not suggesting that you turn a blind eye to the fact that you're making calls to remote systems. Imagine, for instance, the ramifications of iterating over a remote array of ten thousand objects by using an array accessor method that goes out to the remote system for every array element. Not exactly efficient!

Self-Describing Data

In programming parlance, the term *self-describing data* is itself self-describing. Put another way, if the question is, "What is self-describing data?" then the answer is, "Data that describes itself." Not a very useful definition. But that's the result of designing a flexible data format to be used by many, many people.

Let's look at a very simple example. Let's say we were designing a message-style distributed system for delivering stock quotes. We could design the response message format to be something like this:

- The first 10 characters contain the stock symbol, right-padded with spaces
- The next 10 characters represent the last price, right-padded with spaces
- The next 10 characters represent the volume, right-padded with spaces
- The next 20 characters represent the timestamp of the quote
- The next 10 characters represent the bid price
- The next 10 characters represent the ask price

You get the point. This is a fixed format message. It doesn't describe itself; rather it is described by the spec provided in the bullet list above. There's no flexibility here. And sometimes there's no need for any flexibility—it's not a one-size-fits-all scenario. But wouldn't you agree that there is some room for improvement? Let's consider the possibility that the values for last price, bid price, and ask price could be formatted in two different ways. The first format uses standard decimal notation, for example, 25.5. The other format uses fractions, so the same value would be encoded as 25 1/2. A self-describing format would have a provision for indicating which format is used for each of the price fields.

Now take this same concept and apply it to the overall structure of the message, as well as to its constituent parts. This gives you the flexibility to fully describe the contents of a message. For example, you could make some of the fields in the stock quote response optional. Maybe you've requested the quote after the market has closed, and maybe that renders the values for bid and ask prices useless. Then why return them at all? A self-describing format could specify its content, thereby having no need to return useless data.

In order for self-describing data to be truly useful, everyone has to use the same language for describing the data. I don't mean a programming language; I mean the language for expressing the description. What we need is a new way of describing and formatting these messages; a new grammar, so to speak. This new grammar would dictate the standard rules governing the format of these messages. In fact, this is where we really see the value of self-describing data: not so much to eliminate the problem of returning useless data, but to get everyone talking the same language. If I

write a stock application that uses 11 characters to represent the stock symbol, it won't be able to talk to your stock server that uses 10 characters. But if we're exchanging self-describing data, the problem partially disappears: each piece of data says what it is in some standard way, so there's a much better chance that software coming from different people and organizations will be able to communicate.

XML

We've all been staring a form of self-describing data in the face every time we use a web browser. HTML is a good example of a standard data format that is quite flexible due to its provision for self-describing elements. For example, the color and font to be applied to a particular section of text are described right along with the text itself. This kind of self-describing data is commonly referred to as a *markup language*. The content is "marked-up" with instructions for its own presentation. This is very nice, and it obviously has gained an incredible level of industry acceptance. But HTML is not flexible enough to accommodate content that was not anticipated by its designers. That's not a knock on HTML; it's just the truth. HTML is not extensible.

The Extensible Markup Language (XML) is just what we're looking for. XML is a hierarchical, tag-based language much like HTML. The important difference for us is that it is fully extensible. It allows us to describe content that is specific to our own applications in a standard way, without the designers of the language having anticipated that content. For example, XML would allow me to create content to represent the stock quote response message from the previous section. It defines the rules that I must follow in order to accomplish that task, without dictating a specific format.

I'm not going to spend any time covering XML itself. If you're not familiar with XML, you may want to remedy that before you begin reading Chapter 2. Many books have been published on XML and related topics. A good starting point for general information is *XML in a Nutshell*, by Elliotte Rusty Harold and W. Scott Means (O'Reilly). For Java developers, *Java and XML* by Brett McLaughlin (O'Reilly) should be of particular interest.

API Specs Versus Wire-Level Specs

Java programmers are used to dealing with API-level specifications, where classes, interfaces, methods, and so on are clearly defined for the purpose of addressing a specific need. These specifications are designed to be independent of any specific implementation, focusing instead on the abstractions that must be implemented.

Consider the Java Message Service (JMS) specification. It fully describes the API that Java applications can use to access the features of message-oriented middleware (MOM) products. The motivation for a standard API is simple: if MOM vendors adopt the API, it becomes that much easier for programmers to work with the various

product offerings. In theory, you could swap one JMS implementation for another without impacting the rest of your code. In practice, it means that product vendors might be somewhat handcuffed, unable to provide alternative APIs that leverage features and capabilities of their own products without sacrificing compliance. Nevertheless, API specifications have been around a long time, and they do achieve most of what they're intended to do.

However, the API specification approach, by itself, leaves a gaping hole in an extremely important area of software development: interoperability. You can't develop an application using Vendor A's JMS-compliant API to communicate with Vendor B's JMS-compliant server. The specification deals with only the API, not the format of the data being communicated between the parties. This seems to suggest that interoperability is not as important as commonality of programming syntax. Yes, it's a trade-off, but it's not always the right one.

A wire-level specification, on the other hand, deals exclusively in the content and format of the data being transmitted between parties: the data that's "on the wire." Instead of concentrating on APIs, it devotes itself to the representation used for distributed computing interactions. So you can pretty much guarantee that if you work with implementations from more than one vendor, the APIs will not be the same. However, you have a decent chance of getting distributed systems that were built using products from multiple vendors to work together. If you're doing any work in the area of web services, the wire-level specification approach is your ally; the API specification approach won't get you very far.

Overview of SOAP

One of the more recent forays into the world of distributed computing resulted in a wire-level specification called the Simple Object Access Protocol, or SOAP. The protocol is relatively lightweight, is based on XML, and is designed for the exchange of information in a distributed computing environment. There is no concept of a central server in SOAP; all nodes can be considered equals, or even peers.

The protocol is made up of a number of distinct parts. The first is the *envelope*, used to describe the content of a message and some clues on how to go about processing it. The second part consists of the rules for encoding instances of custom data types. This is one of the most critical parts of SOAP: its extensibility. The last part describes the application of the envelope and the data encoding rules for representing RPC calls and responses, including the use of HTTP as the underlying transport.

RPC-style distributed computing is the most common Java usage of SOAP today, but it is not the only one. Any transport can be used to carry SOAP messages, even though HTTP is the only one described in the specification. There has been significant discussion of using SMTP, BEEP, JXTA, and other protocols for carrying SOAP messages.

Using SOAP with Java

SOAP differs from RMI, CORBA, and COM in that it concentrates on the content, effectively decoupling itself from both implementation and underlying transport. An interesting concept for a Java book, wouldn't you say? After all, implementation and transport are likely to be built using Java. Yet SOAP in no way addresses Java or any other implementation strategy.

The reality is that SOAP is an enabler, incapable of existing on its own beyond the abstraction of the specification. To benefit from SOAP, or any other protocol, there have to be real implementations. Java is a great technology for implementing SOAP, and for building web services and applications that use SOAP as the "on the wire" data format.

SOAP Implementations

As I write this book, there are dozens of SOAP implementations, and new ones emerge all the time. Some are implemented in Java, some aren't. Some are free, some aren't. And inevitably some are good, and some aren't. It would be impractical to do a side-by-side comparison of every available implementation or even to give equal coverage to them all. On the other hand, it wouldn't be wise to focus on a single implementation, since that would present a bias that I don't intend. A reasonable compromise, and the one I've elected to use, is to select two interesting Java SOAP implementations and use them both extensively throughout the book. This gives you an opportunity to see different APIs and programming strategies. In Chapters 9 and 11, I'll break this rule and look briefly at a couple of other important SOAP technologies.

Apache SOAP

The Apache Software Foundation has an ongoing project known as Apache SOAP. This is a Java implementation of the SOAP specification that can be hosted by servers that support Java servlets. The examples in this book are based on Apache SOAP Version 2.2, which is available at *http://xml.apache.org/soap/index.html*. Four very important factors led me to choose this implementation: it supports a good deal of the specification, it has a reasonably large user base, the source code is available, and it's free.

Although Apache SOAP can be hosted by a variety of server technologies, I've chosen Apache Tomcat (Version 3.3 and Version 4), available at *http://jakarta.apache.org/tomcat/index.html*. The reasons are not particularly tied to SOAP, but it does work well in Tomcat. The use of Tomcat has no real impact on the examples in the book, so you can feel free to select some other server technology if you like.

Apache SOAP was developed at a time when there were no standards for a SOAP API. The SOAP specification doesn't address language bindings, so the implementors

were forced to come up with their own APIs. More recently, the Java community has been working on standards for SOAP-based messaging and RPC APIs, known as JAXM and JAX-RPC, respectively. We'll take a look at these APIs in Chapter 11; they will be implemented by Axis, which is the next-generation Apache SOAP implementation. In an ideal world, JAXM and JAX-RPC would have been completed in time for me to give them the coverage they deserve, since they will almost certainly replace the APIs developed for Apache SOAP 2.2. In reality, though, the standard APIs are just solidifying now, and should be in final form just in time to make me write a second edition earlier than I'd like. The bulk of this book will focus on technology that you can use to write production code now. Once you have the concepts down, moving to a different API will not be a challenge.

GLUE

GLUE is an implementation of SOAP developed by a company called The Mind Electric (*www.themindelectric.com*). It's developed completely in Java, and can be hosted by a variety of servers that support servlets or can be run standalone using its own HTTP server. The GLUE examples in the book are based on GLUE Version 2, available at *http://www.themindelectric.com/products/glue/glue.html*. I chose GLUE for four reasons as well: it uses a very programmer-friendly approach, its APIs are quite different from those found in Apache SOAP, it relies on the Web Services Description Language (WSDL), and it's free for most uses.*

The GLUE APIs are also proprietary. I'd expect that a future version of GLUE would adopt standards like JAXM and JAX-RPC if the user community demanded it, but that, of course, remains to be seen.

Others

In Chapter 9 we'll work a little with some other technologies. Microsoft's .NET is a major player in the area of SOAP-based web services, so we'll look at what it takes to write Java applications that use SOAP to communicate with .NET services.

The Apache Software Foundation is currently working on a next-generation SOAP implementation called Axis. Although it's still a bit early to cover Axis in any detail, there's no doubt that it will some day replace or subsume Apache SOAP Version 2.X. We'll take a peek at writing a simple Axis application using the current release. (And, as I've already said, Chapter 11 will look at the JAXM and JAX-RPC proposed standards, which Axis will eventually implement.)

* Please refer to the GLUE license agreement.

The Approach

This book certainly does not cover every aspect of the SOAP technologies. My goal is to give you a good understanding of the major aspects of SOAP in the context of Java software development. You'll find that many of the examples are presented not only in Java source code, but also in the SOAP XML that is generated through the execution of the Java code. This will give you a sense of what the various APIs are actually accomplishing. Learning SOAP this way will allow you to go beyond the scope of this book with confidence, exploring the features and capabilities of the implementations I have covered as well those I have not.

No Security

One particular area is not covered in this book: security. How can you talk about a distributed computing technology without talking about security? The answer is actually quite simple: the SOAP specification does not deal with security. The current implementations rely on the security features of the hosting technology. Be it SSL, basic HTTP authentication, proxy authentication, or some other mechanism, all security is a function of the hosting technology and not part of SOAP itself. It's expected that either a future version of the SOAP spec or a separate SOAP Security spec will address that issue, but for now you'll have to rely on whatever your hosting technology supports.

The current spec does, however, mention the use of SOAP headers in security schemes, and you will find that some SOAP implementations follow that lead. Nonetheless, the mechanisms are likely to be specific to each implementation until a standard is adopted.

Getting Started

If you plan to work with the examples, you'll need to install all of the necessary software components, including the JDK, the SOAP implementations described earlier, and a number of other supporting technologies. All of these packages are reasonably well documented, so you should have no trouble getting them installed properly.

The chapters in this book are arranged so that each one builds on concepts of the preceding chapters. I suggest that you follow along in order, but of course that's entirely up to you. If you are already comfortable with XML or (even better) with some aspects of web services, you may find that you can jump around a bit.

CHAPTER 2

The SOAP Message

All SOAP messages are packaged in an XML document called an *envelope*, which is a structured container that holds one SOAP message. The metaphor is appropriate because you stuff everything you need to perform an operation into an envelope and send it to a recipient, who opens the envelope and reconstructs the original contents so that it can perform the operation you requested. The contents of the SOAP envelope conform to the SOAP specification,* allowing the sender and the recipient to exchange messages in a language-neutral way: for example, the sender can be written in Python and the recipient can be written in Java or C#. Neither side cares how the other side is implemented because they agree on how to interpret the envelope. In this chapter we'll get inside the SOAP envelope.

The HTTP Binding

The SOAP specification requires a SOAP implementation to support the transporting of SOAP XML payloads using HTTP, so it's no coincidence that the existing SOAP implementations all support HTTP. Of course, implementations are free to support any other transports as well, even though the spec doesn't describe them. There's nothing whatsoever about the SOAP payload that prohibits transporting messages over transports like SMTP, FTP, JMS, or even proprietary schemes; in fact, alternative transports are frequently discussed, and a few have been implemented. Nevertheless, since HTTP is the most prevalent SOAP transport to date, that's where we'll concentrate. Once you have a grasp of how SOAP binds to HTTP, you should be able to easily migrate your understanding to other transport mechanisms.

SOAP can certainly be used for request/response message exchanges like RPC, as well as inherently one-way exchanges like SMTP. The majority of Java-based SOAP implementations to date have implemented RPC-style messages, so that's where

* The spec can be found at *http://www.w3.org/TR/SOAP*. The SOAP 1.1 specification is not a W3C standard, but the SOAP 1.2 spec currently under development will be.

we'll spend most of our time; HTTP is a natural for an RPC-style exchange because it allows the request and response to occur as integral parts of a single transaction. However, one-way messaging shouldn't be overlooked, and nothing about HTTP prevents such an exchange. We'll look at one-way messaging in Chapter 8.

HTTP Request

The first SOAP message we'll look at is an RPC request. Although it's rather simple, it contains all of the elements required for a fully compliant SOAP message using an HTTP transport. The XML payload of the message is contained in an HTTP POST request. Take a quick look, but don't get too caught up in figuring out the details just yet. The following message asks the server to return the current temperature in degrees Celsius at the server's location:

```
POST /LocalWeather HTTP/1.0
Host: www.mindstrm.com
Content-Type: text/xml; charset="utf-8"
Content-Length: 328
SOAPAction: "WeatherStation"

<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
    <m:GetCurrentTemperature xmlns:m="WeatherStation">
         <m:scale>Celsius</m:scale>
    </m:GetCurrentTemperature>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The SOAP HTTP request uses the HTTP POST method. Although a SOAP payload could be transported using some other method such as an HTTP GET, the HTTP binding defined in the SOAP specification requires the use of the POST method. The POST also specifies the name of the service being accessed. In the example, we're sending the data to `/LocalWeather` at the host specified later in the HTTP header. This tells the server how to route the request within its own processing space. Finally, our example indicates that we're using HTTP Version 1.0, although SOAP doesn't require a particular version of HTTP.

The `Host:` header field specifies the address of the server to which we're sending this request, *www.mindstrm.com*. The next header field, `Content-Type:`, tells the server that we're sending data using the `text/xml` media type. All SOAP messages must be sent using `text/xml`. The content type in the example also specifies that the data is encoded using the UTF-8 character set. The SOAP standard doesn't require any particular encoding. `Content-Length:` tells the server the character count of the POSTed SOAP XML payload data to follow.