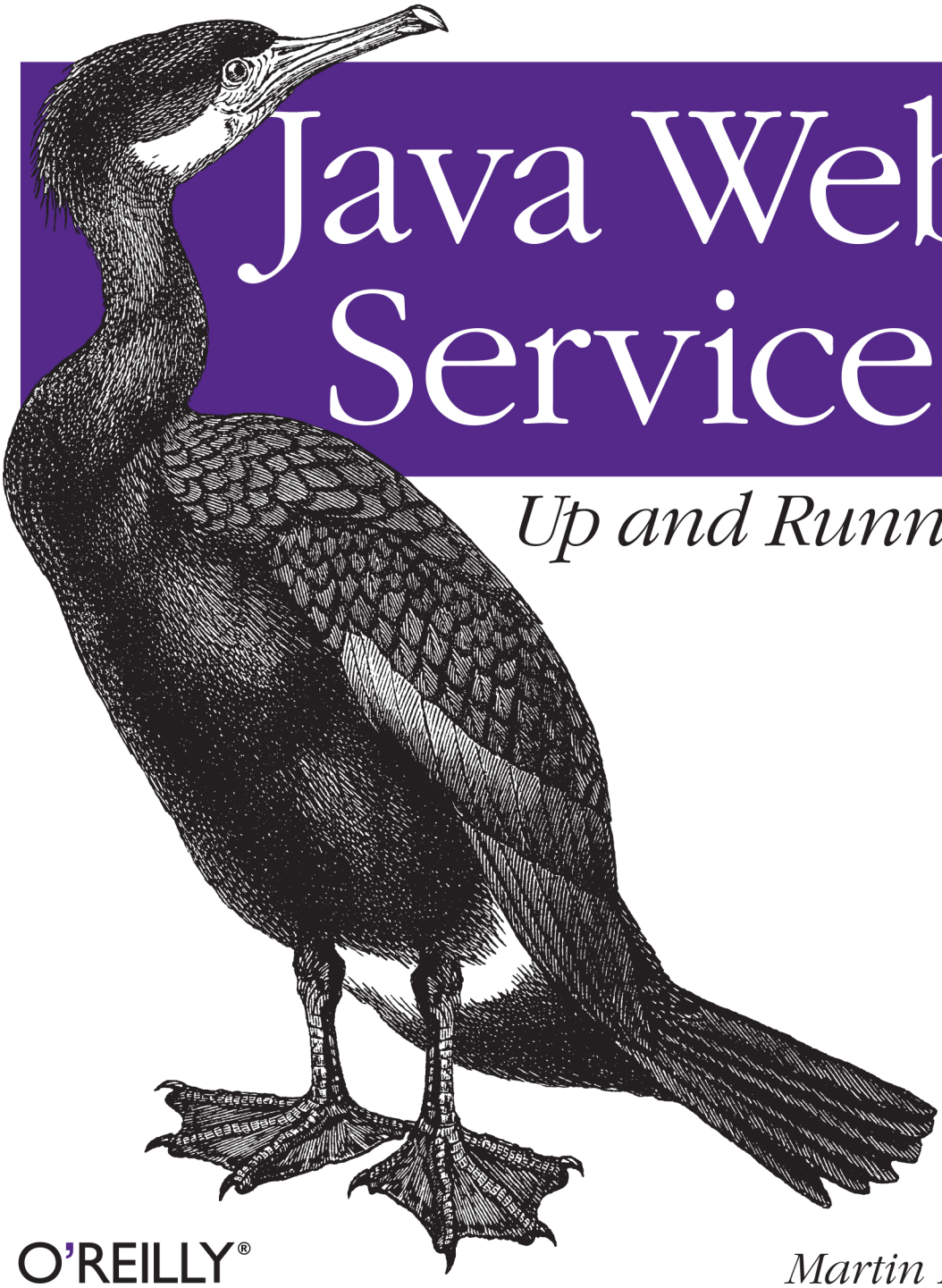
A Quick, Practical, and Thorough Introduction

2nd Edition

Java Web Services

Up and Running



O'REILLY®

Martin Kalin

Java Web Services: Up and Running

Learn how to develop REST-style and SOAP-based web services and clients with this quick and thorough introduction. This hands-on book delivers a clear, pragmatic approach to web services by providing an architectural overview, complete working code examples, and short yet precise instructions for compiling, deploying, and executing them. You'll learn how to write services from scratch and integrate existing services into your Java applications.

With greater emphasis on REST-style services, this second edition covers HttpServlet, Restlet, and JAX-RS APIs; jQuery clients against REST-style services; and JAX-WS for SOAP-based services.

- Learn differences and similarities between REST-style and SOAP-based services
- Program and deliver RESTful web services, using Java APIs and implementations
- Explore RESTful web service clients written in Java, JavaScript, and Perl
- Write SOAP-based web services with an emphasis on the application level
- Examine the handler and transport levels in SOAP-based messaging
- Learn wire-level security in HTTP(S), users/roles security, and WS-Security
- Use a Java Application Server (JAS) as an alternative to a standalone web server

Martin Kalin is a professor in the College of Computing and Digital Media at DePaul University. He has written a book on Java for programmers, and co-written a series of books on C and C++.

"Martin Kalin's in-depth guide is absolutely brimming with practical examples. A great cookbook for tasting many different Java web technologies and what they may offer you."

—Edward Yue Shung Wong
(@arkangelofkaos)

US $34.99 CAN $36.99
ISBN: 978-1-449-36511-0

Twitter: @oreillymedia
facebook.com/oreilly

O'REILLY®
oreilly.com

SECOND EDITION

Java Web Services: Up and Running

Martin Kalin

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

Java Web Services: Up and Running, Second Edition
by Martin Kalin

Printed in the United States of America.

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

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

Editor: Meghan Blanchette
Production Editor: Rachel Steely
Copyeditor: Rachel Leach
Proofreader: BIM Indexing and Proofreading Services
Indexer: Judith McConville
Cover Designer: Randy Comer
Interior Designer: David Futato
Illustrator: Rebecca Demarest

September 2013: Second Edition

Revision History for the Second Edition:

2013-08-23: First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449365110* for release details.

ISBN: 978-1-449-36511-0

[LSI]

Table of Contents

Preface

Welcome to the second edition of *Java Web Services: Up and Running*. This edition, like the first, is for programmers interested in developing web services and clients against such services. This edition, again like the first, emphasizes code. My aim is to make web services and their clients come alive through focused but realistic programming examples in Java but, of course, in other languages as well: web services are designed to be language-neutral, a point best illustrated through the interaction of services and clients written in different languages. Indeed, the client of a well-designed web service can remain agnostic about the service's implementation details, including the language in which the service is written. To ease the task of compiling and publishing services, the ZIP file with the code samples includes an Ant script that compiles, packages, and deploys web services. The major client examples include either Ant scripts for compiling and running the clients or executable JAR files with all of the dependencies included therein. The code examples are available at *https://github.com/mkalin/jwsur2*.

What's Changed in the Second Edition?

In the four years or so since the first edition, there has been continuity as well as change. Web services remain a popular and arguably even dominant approach toward *distributed software systems*—that is, systems that require the interaction of software on physically distinct devices. The Web itself is a prime example of a distributed system, and the current trend is to blur the distinction between traditional, HTML-centric *websites* and modern *web services*, which typically deliver XML or JSON payloads instead of HTML ones. Web services are an appealing way to create distributed systems because these services can piggyback on existing infrastructure such as HTTP(S) transport, web servers, database systems, modern programming languages of various stripes, widespread software libraries for JSON and XML processing, security providers, and so on. Indeed, web services are a lightweight and flexible way to integrate divergent software systems and to make the functionality of such systems readily accessible.

Java remains a major player in web services, and Java support for these services, in the form of standard and third-party software libraries and utilities, continues to improve. Yet two important and related shifts in emphasis have occurred since this book was first published:

- The consumers or clients of web services are increasingly written in JavaScript, particularly in the jQuery dialect, and these clients naturally prefer response payloads in JSON (JavaScript Object Notation) rather than in XML because a JSON document is the text representation of a native JavaScript object. A JavaScript client that receives, for example, an array of products as a JSON rather than an XML document can process the array with the usual JavaScript programming constructs. By contrast, a JavaScript client that receives an XML payload would face a challenge common across programming languages: the challenge of parsing an XML document to extract its informational content before moving on to specific application logic. Modern web services and web service frameworks acknowledge the growing popularity of JSON by treating JSON and XML formats as equals. In some frameworks, such as Rails, JSON even gets the nod over XML.
- REST-style services are increasingly popular among familiar sites such as eBay, Facebook, LinkedIn, Tumblr, and Twitter. Amazon, a web service pioneer, continues to support REST-style and SOAP-based versions of its services. The services from newer players tend to be REST-style for an obvious reason: REST-style services are relatively low fuss and their APIs are correspondingly simple. SOAP-based services still are delivered mostly over HTTP(S), although Java and DotNet continue to explore the use of other protocols, especially TCP, for transport. The first edition of this book underscored that SOAP-based services over HTTP can be seen as a special case of REST-style services; the second edition pursues the same theme.

The two changes in web services are reflected in how the second edition is organized. Chapter 1 begins with an overview of web services, including the link between such services and Service-Oriented Architecture (SOA), and the chapter includes a code-based contrast of SOA and the competing Distributed Object Architecture (DOA). The discussion then turns to REST: what the acronym means, why HTTP can be treated as an API and not just a transport, and how the RESTful mindset continues to impact the design and implementation of modern web services. The first chapter includes sample HTTP clients in Java, clients that can be targeted at either websites or web services. The first chapter ends with a RESTful service implemented as a JSP script with support from two backend POJO classes; the service is published with the Tomcat web server. The first chapter goes into the details of installing and running Tomcat; the second chapter does the same for the Jetty web server. The aforementioned Ant script is also clarified so that the sample web services can be packaged and deployed automatically.

Although this edition of the book starts with REST-style services, SOAP-based services are treated thoroughly. Chapter 4 covers SOAP-based services at the *application level*,

a level in which the SOAP remains transparent; Chapter 5 explores the *handler* and the *transport* levels at which the SOAP is exposed for inspection and manipulation. Starting with REST-style services helps to explain the advantages that come with SOAP-based services, in particular the benefit of having the XML remain mostly under the hood. Issues such as security cut across the REST/SOAP boundary, and Chapter 6 is dedicated to practical web security, from wire-level security through users/roles security up to WS-Security.

Web Service APIs and Publication Options

In the first edition, the JAX-WS APIs and their Metro implementation were dominant. In this edition, the two are important but less dominant. For REST-style services, the book has examples based on the following APIs:

`HttpServlet`
: The `HttpServlet` is well designed for REST-style services because the API is so close to the HTTP metal. Servlet instances encapsulate callbacks such as `doPost`, `doGet`, `doPut`, and `doDelete`, which cover the familiar CRUD operations: *create* (POST), *read* (GET), *update* (PUT), and *delete* (DELETE). There are symbolic versions of HTTP status codes to signal the outcome of an HTTP request, support for MIME types, utilities to access HTTP headers and bodies, and so on. JSP and other Java-based scripts execute as servlet instances and, therefore, fall under the servlet umbrella. The `HttpServlet` is grizzled but hardly obsolete. Servlets are still an excellent way to deliver REST-style services.

JAX-RS
: This is a relatively recent and increasingly popular API for delivering REST-style services. The API centers on annotations such as `@GET` and `@POST` to route HTTP requests to particular Java methods. There is likewise a convenient `@Path` annotation to identify the particular resource targeted in a request. JAX-RS can be configured to automatically generate XML and JSON responses. This API, like the Restlet API described next, has a contemporary look and feel. At the implementation level, JAX-RS represents a layering atop servlets. The same options for publishing servlet-based services are available for their JAX-RS cousins.

Restlet
: This API is similar in style to JAX-RS, although the claim is likely to upset proponents of both. The Restlet API also centers on annotations for routing HTTP requests to designated Java methods and for generating payloads. Restlet encourages interplay with other APIs. It is possible, for example, to use JAX-RS annotations in a Restlet-based service. Restlet offers an easy-to-use publisher for development and testing. Restlet services, like their JAX-RS counterparts, represent an

implementation level on top of servlets. Programmers should be able to move easily between the JAX-RS and Restlet APIs.

JAX-WS `@WebServiceProvider`
: This is a deliberately XML-centric and low-level API that could be used for either SOAP-based or REST-style services. However, JAX-WS has the `@WebService` annotation precisely for SOAP-based services; hence, the most obvious use of the `@WebServiceProvider` annotation is for XML-based REST-style services. This API is well suited for services that require granular control over XML generation and processing.

For SOAP-based services, most of the examples use the reference implementation of JAX-WS, which is Metro. However, this edition now covers Axis2 as well. Axis2 implements JAX-WS but has additional features.

The Publication Options

Each of these APIs, whether for REST-style or SOAP-based services, honors the separation-of-concerns principle with respect to publishing a web service. The web service is one concern; its publication is quite another concern. Services developed with any of these APIs can be published with a standalone web server such as Tomcat, a Java Application Server (JAS) such as GlassFish, or even with a simple command-line utility such as the standard `Endpoint` publisher. To underscore the separation-of-concerns principle and to emphasize the production-grade options, my examples are published in the following ways:

Standalone web servers
: The two obvious choices in Java are Tomcat and Jetty, although other choices are available. The aforementioned Ant script automatically compiles and packages web services, REST-style and SOAP-based alike, for publication. Although the Ant script is tailored for Tomcat publication, a generated WAR file can be deployed, as is, to Jetty, Tomcat, or one of the many JASes. Tomcat and Jetty provide the usual services such as wire-level and users/roles security, logging/debugging, and administration that one expects from a production-grade web server.

Java Application Servers
: The reference implementation is still GlassFish, which is part of the community-based Metro project. GlassFish can be used to publish either *servlet-based* services, which are the type that Tomcat and Jetty can publish, or *EJB-based* services, which are `@Stateless` Session Enterprise JavaBeans. TomEE, which is essentially Tomcat7 with OpenEJB extensions, is an emphatically lightweight publisher of both servlet-based and EJB-based services. Under TomEE, even an EJB-based service can be deployed as a standard WAR (Web ARchive) file. TomEE includes an implementation of JAX-RS.

Command-line publishers
: Examples are the standard `Endpoint` utility class and the Restlet `Component` class. These publishers are useful for development, testing, and even low-volume production.

Java in general draws strength from the many options that the language and the runtime offer; this strength carries over to web services as well. There are many ways to program web services and web service clients in Java, and there are various attractive options for publishing such services. There is no need to claim any particular way in web services as the best way. My aim is to examine and clarify the choices so that in the end, the API, implementation, and method of publication can be determined by what is best suited for the service.

Chapter-by-Chapter Overview

The second edition has seven chapters. The following list offers a summary of each chapter.

Chapter 1, Web Services Quickstart
: This chapter begins the code-driven tour of web services with an overview of the differences—and the similarities—between REST and SOAP. Why are web services of any use? This question is addressed with examples: one example focuses on using web services to automate access to the data and functionality available on the Web; the other example focuses on web services as a way to integrate diverse software systems. The theme of *interoperability* is pursued throughout the book with examples. Chapter 1 includes a short history of web services, with emphasis on how the SOA approach to distributed systems differs significantly from the DOA approach that predates yet continues to compete with web services. The chapter then focuses on how HTTP itself is at the center of the RESTful way to web-based, distributed software systems. XML and JSON are introduced as document-exchange formats of special interest in RESTful services. The chapter includes code examples: a pair of Java HTTP clients used to illustrate key features of HTTP; and a first RESTful service, which consists of a JSP script and two backend POJO classes. The *curl* utility is used to make sample client calls, including failed ones, against the first service. The chapter covers practical matters such as installing the Tomcat web server and using the provided Ant script to compile, package, and deploy a web service.

Chapter 2, RESTful Web Services: The Service Side
: This chapter introduces various APIs and implementations available for programming and delivering RESTful web services in Java. The `HttpServlet`, JAX-RS, Restlet, and JAX-WS `@WebServiceProvider` APIs are explored through full code examples. The chapter clarifies various ways of generating XML and JSON payloads, using both standard Java classes and different third-party ones. The code examples

adhere to RESTful principles such as honoring the intended meaning of each CRUD verb; using intuitive URIs to name resources; relying upon MIME data types to describe resource representations; and taking full advantage of those HTTP status codes that report on the outcome of an HTTP request against a RESTful service. Chapter 2, along with later chapters, looks at options for publishing RESTful services. The options include standalone web servers such as Tomcat and Jetty together with command-line publishers such as `Endpoint`, `HttpServer`, and Restlet `Component`. The chapter goes into the technical details of multithreading and thread synchronization in services deployed with a web server such as Tomcat or Jetty. The installation and management of Jetty are also covered. Chapter 2 also takes a first look at the powerful JAX-B (Java API for XML-Binding) and JAX-P (Java API for XML-Processing) utilities, which are especially important in the Chapter 3 coverage of the client side in RESTful services.

Chapter 3, RESTful Web Services: The Client Side

This chapter shifts focus from the service to the client side of RESTful services. There are sample clients written with the weathered but still trusty `URLConnection` class and also clients written using REST-specific APIs. (JAX-RS, Restlet, and JAX-WS provide both service-side and client-side APIs.) As evidence of interoperability, the chapter offers jQuery and Perl clients against Java services and Java clients against commercial services whose implementation language is officially unknown. The code samples explore various possibilities for dealing with XML and JSON payloads, in particular the standard JAX-B packages and third-party contributions such as `XStream`. These utilities are especially useful in transforming XML documents into native Java objects, which obviates the need for explicit parsing. Most RESTful services now furnish a grammar, in the form of an XML Schema or equivalent, for the service; core Java has utilities such as *xjc* that convert an XML Schema into Java classes. Chapter 3 has clients against real-world RESTful services at Twitter, Amazon, and the Chicago Transit Authority. This chapter pays special attention to the growing importance of JavaScript clients, which are highlighted in several examples using jQuery. Finally, the chapter shows how distinct web services can be orchestrated to form a single, composite service.

Chapter 4, SOAP-Based Web Services

This chapter turns from REST-style to SOAP-based services, in particular to the JAX-WS API and its central annotation `@WebService`. The chapter opens by converting a REST-style service from earlier chapters to a SOAP-based service. The emphasis in this chapter is on the application level, a level in which the XML in SOAP-based messaging remains transparent. Indeed, a chief attraction of SOAP-based services is that neither the services nor their clients require any attention to the underlying XML: service operations are, in Java, `public` methods preferably annotated with `@WebMethod`, and remote clients invoke the operations straightforwardly. The data types of arguments and return values include all of the primitive

types and their wrappers, the `String` and `Calendar` types, various other standard types, arrays of any acceptable type, and programmer-defined classes whose properties reduce ultimately to any of these. The chapter explains in detail the programmer-friendly *wsimport* utility, which generates client-side support code from the web service contract, the WSDL (Web Service Description Language) document. The structure, purpose, and various uses of the WSDL are clarified through coding examples. The role of XML Schema or equivalent in a WSDL document is given particular emphasis. The chapter includes two Java clients against the Amazon E-Commerce service together with C# and Perl clients against a Java service. These examples underscore that SOAP-based services, like their REST-style cousins, are language-neutral. The clients against the SOAP-based version of the Amazon E-Commerce service introduce but do not explore the handler level of SOAP-based services; these examples also provide a first look at security issues.

Chapter 5, SOAP Handlers and Faults

This chapter examines the handler and transport levels in SOAP messaging, levels at which the XML in a SOAP message comes to the fore for inspection and manipulation. The chapter begins with a look at the SOAP message architecture, which distinguishes among a message *sender*; an *intermediary* that should confine its activity to the SOAP header rather than the SOAP body or attachments in a message; and an ultimate *receiver*, which should have access to the entire SOAP message. The distinct parts of SOAP messages, the raw XML and any attachments, are accessible to SOAP handlers, which come in two flavors: *message handlers* have access to the entire SOAP message (header, body, and attachments), whereas *logical handlers* have access only to the payload in the body. WS-Security and related extensions of SOAP beyond the *basic profile* may use such access to inject or inspect security elements in SOAP headers. Handlers are akin to `Filter` instances in websites, although handlers are inherently bidirectional and can occur on either the client or the service side. The chapter covers both individual handlers and handler chains, in this case a chain consisting of a message and a logical handler. The handler chain example mimics the user authentication at work in Amazon's web services. The chapter likewise examines the related topic of SOAP *faults*, special error messages that can be generated at either the application or the handler level. The chapter also looks at how SOAP messages can transport arbitrary binary data as attachments. A final topic is the transport level, usually HTTP; this level is especially useful in the users/roles security examined in Chapter 6.

Chapter 6, Web Services Security

This chapter covers security, a topic that cuts across the REST/SOAP boundary, at three levels: wire-level security of the type that HTTPS provides, users/roles security, and WS-Security in SOAP-based messaging. The chapter begins with wire-level security and its constituent services of *peer authentication*, *message confidentiality*, and *message integrity*. The clarification of these terms requires, in

turn, an examination of concepts such as *symmetric* and *asymmetric* encryption/decryption, *public key security*, *cryptographic hash function*, and *cryptographic suite*. HTTPS is examined in detail through coding examples: two HTTPS clients against the Google site and a lightweight HTTPS server built with the `HttpsServer` class that comes with the core Java JDK. There is also a sample HTTPS client against a RESTful service. These and other coding examples clarify additional security artifacts such as the *keystore* and the *truststore*, *digital certificates*, and *certificate authority*. There is a section to explain how a web server such as Tomcat can be set up to handle and even to enforce HTTPS connections. Chapter 6 also covers *container-managed* users/roles security, again with coding examples that involve a production-grade web server such as Tomcat. The two-phased user authentication and roles authorization process is studied in detail and includes examples of HTTP BASIC and DIGEST authentication. The chapter ends with a code example focused on WS-Security and its end-to-end approach to security.

Chapter 7, Web Services and Java Application Servers

This chapter introduces the Java Application Server as an alternative to the standalone web server for deploying REST-style and SOAP-based services. The JAS as a service publisher also brings the option of deploying a web service as an EJB, in particular a web service as a `@Stateless` Session EJB. The chapter begins with a list of the popular JASes and then clarifies the core components and attendant functionalities of a JAS. There is a review of how sample services from previous chapters —from `HttpServlet` examples through JAX-RS, Restlet, `@WebServiceProvider`, and `@WebService` examples—can be ported to a JAS. In all cases, almost no change is required. Chapter 7 also motivates the option of a JAS, in particular the benefits that come with deploying a service in the thread-safe EJB container. A typical JAS is not only a publisher but also a development, testing, and management environment. This chapter has a further coding example of how web services and websites can interact, and the excellent JPA (Java Persistence API) is introduced with two coding examples that persist data in different databases: HSQLDB and Java Derby. The chapter introduces two JASes through code examples: GlassFish, which is the reference implementation, and TomEE, which is an emphatically lightweight option among JASes. Various sidebars explain installation and management details. The chapter ends with a review and a recommendation that programmers embrace the many excellent choices of API and implementation that Java offers in the area of web services.

Tools and IDEs

Java programmers have a wide choice of productivity tools. Among the build tools are Ant, Maven, and Meister; among the many testing tools are EasyMock, JMockit, JUnit, Mockito, and TestNG. Java likewise offers choices among IDEs, including Eclipse,

IntelliJ IDEA, and NetBeans. In a production environment, tools and IDEs are the way to go as they hide the grimy details that slow the journey from initial design through deployment and maintenance. In a learning environment, a build tool makes sense because it facilitates experimentation. My Ant script is meant to serve this purpose: it allows a web service, with all of the dependencies, to be compiled, built, and deployed with a single command:

```
% ant deploy -Dwar.name=myFirstService
```

Chapter 1 goes into the setup details, which are minimal.

With respect to IDEs, this second edition is, like the first, neutral. The very grimy details that are an obstacle in a production environment are critical in a learning environment. For that reason, my code examples include all of the `import` statements so that dependencies are clear. Package/directory structure is explained whenever third-party libraries are used. The code listings usually have numbered statements and expressions for ease of reference and explanation. The examples themselves are designed to highlight the challenges inherent in any serious programming, but these examples also focus on patterned approaches to meeting the challenges. Web services and their clients are, in the end, code—and this book focuses on code.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, filenames, file extensions, and emphasis.

`Constant width`
: Used for program listings as well as within paragraphs to refer to program elements such as variable or method names, data types, environment variables, statements, and keywords.

Sidebars
: The book uses sidebars (see "This Is a Sidebar" on page xv) to focus on particular topics. Sidebars often contain practical information about installing and running applications such as a standalone web server or a Java Application Server.

This Is a Sidebar

A topic of special interest.

Using Code Examples

This book is here to help you get your job done. In general, if this book includes code examples, you may use the code in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Java Web Services: Up and Running*, Second Edition, by Martin Kalin. Copyright 2013 Martin Kalin, 978-1-449-36511-0."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

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

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://oreil.ly/Java_web_services*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

Edward Yue Shung Wong and Ken Yu were generous enough to review this book and offer many insightful suggestions for its improvement. They made the book better than it otherwise would have been. I thank them heartily for the time and effort that they invested in this project. The remaining shortcomings are mine alone, of course. Meghan Blanchette, my editor, has provided invaluable support, and the book would not be without her help. My thanks go as well to the many behind-the-scenes people at O'Reilly Media who worked on this project.

This edition, like the first, is dedicated to Janet.

CHAPTER 1
Web Services Quickstart

Although the term *web service* has various, imprecise, and evolving meanings, a working definition should be enough for the upcoming code example, which consists of a service and a client, also known as a consumer or requester. As the name suggests, a web service is a kind of webified application—an application typically delivered over HTTP (HyperText Transport Protocol). HTTPS (HTTP Secure) adds a security layer to HTTP; hence, a service delivered over HTTPS likewise counts as a web service. Until the main topic of interest is web service security, HTTP should be understood to include HTTPS.

Amazon, a pioneer in web services, is well known for its various websites, among which is the E-Commerce site for shopping. Amazon has other popular websites as well. Of interest here is that the data and functionality available at Amazon websites are likewise available as Amazon web services. For example, someone can use a browser to shop interactively at the Amazon E-Commerce site, but this person also could write a program, as later examples show, to do the shopping through the corresponding Amazon E-Commerce web service. Amazon is particularly good at pairing off its websites with web services.

Web services can be programmed in a variety of languages, old and new. The obvious way to publish a web service is with a web server; a web service client needs to execute on a machine that has network access, usually over HTTP, to the web server. In more technical terms, a web service is a distributed software system whose components can be deployed and executed on physically distinct devices. Consider, for example, a web server *host1* that hosts a web service and a mobile device *host2* that hosts an application issuing requests against the service on *host1* (see Figure 1-1). Web services may be more architecturally complicated than this, of course; for one thing, a service may have many clients issuing requests against it, and the service itself may be composed of other services. For instance, a stock-picking web service might consist of several code components, each hosted on a separate commercial-grade web server, and any mix of PCs, handhelds, and other networked devices might host programs that consume the service. Although

the building blocks of web services are relatively simple, the web services themselves can be arbitrarily complex.

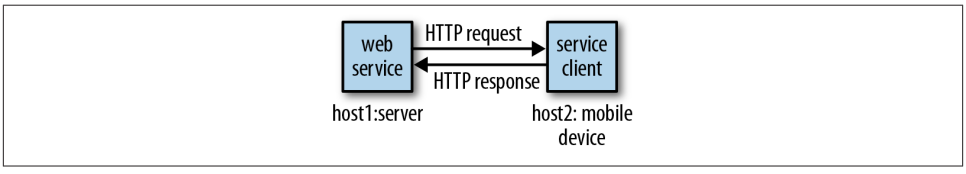


Figure 1-1. A web service and one of its clients

An HTTP request goes, by definition, from client to server, and an HTTP response goes, also by definition, from server to client. For web services over HTTP, the HTTP messages are the infrastructure, and these HTTP messages can be combined into basic conversational patterns that characterize a web service. For example, if the web service conversation starts with an HTTP request that expects an HTTP response, the pattern is the familiar *request/response* conversation. By contrast, if the conversation starts with an HTTP message from the server, a message that expects a message from the client in return, then the pattern is *solicit/response*. Richer conversational patterns can be composed out of such basic two-message patterns. Indeed, these two-message patterns are composed of even more primitive ones: a message from client to server without a response is a pattern known as *one-way,* and the reverse pattern, from server to client without a client response, is known as *notification*. Web services tend to be simple in structure. The four conversational patterns enumerated just now cover most modern web services, and request/response is the pattern that still dominates.

Web services come in two popular flavors: SOAP-based and REST-style. SOAP is an XML dialect with a grammar that specifies the structure that a document must have in order to count as SOAP. In a typical SOAP-based service, the client sends SOAP messages to the service and the service responds in kind, with SOAP messages. REST-style services are hard to characterize in a sentence or two, but with respect to pattern, these services tend to be request/response; the same holds for SOAP-based services. For now, a REST-style service is one that treats HTTP not only as transport infrastructure but also as a set of guidelines for designing service requests and service responses. In a REST-style service, HTTP itself can be seen as an API. SOAP has standards, toolkits, and bountiful software libraries. REST has no official standards, comparatively few toolkits, and uneven software libraries among programming languages. Yet there is growing support for REST-style services across programming languages; hence, it seems only a matter of time until toolkits and libraries for REST-style services mature.

From a historical perspective, the RESTful approach to web services can be viewed as an antidote to the creeping complexity of SOAP-based web services. SOAP-based services are designed to be transport-neutral; as a result, SOAP messaging may seem overly complicated if the transport is, in fact, HTTP. This book covers SOAP-based and

REST-style web services, starting with REST-style ones. This chapter ends with a sample REST-style service and sample client calls against the service. At present, the distinction between the two flavors of web service is not sharp, because a SOAP-based service delivered over HTTP can be seen as a special case of a REST-style service; HTTP remains the dominant transport for SOAP-based services.

SOAP originally stood for Simple Object Access Protocol and then, by serendipity but never officially, might have stood for Service-Oriented Architecture (SOA) Protocol. (SOA is discussed in the section "Web Services and Service-Oriented Architecture" on page 7.) The World Wide Web Consortium (hereafter, W3C) currently oversees SOAP, and SOAP is officially no longer an acronym.

Web Service Miscellany

Except in test mode, the client of either a SOAP-based or REST-style service is rarely a web browser but, rather, usually an application without a graphical user interface. The client may be written in any language with the appropriate support libraries. Indeed, a major appeal of web services is language transparency: the service and its clients need not be written in the same language. Language transparency is a key contributor to web service interoperability—that is, the ability of web services and their consumers to interact seamlessly despite differences in programming languages, support libraries, operating systems, and hardware platforms. To underscore this appeal, my examples use a mix of languages besides Java, among them C#, JavaScript, and Perl. My sample clients in Java consume services written in languages other than Java; indeed, sometimes in languages unknown.

There is no magic in language transparency, of course. If a web service written in Java can have a Python or a Ruby consumer, there must be an intermediary layer that handles the differences in data types between the service and the client languages. XML technologies, which support structured document interchange and processing, act as one such intermediary level. Another intermediary level is JSON (JavaScript Object Notation). XML and JSON are both data-interchange formats, but JSON clearly has the upper hand with data receivers written in JavaScript because a JSON document is the text representation of a native JavaScript object. Web service clients are increasingly JavaScript programs embedded in HTML documents and executing in a browser; such clients process JSON with less fuss than they do XML. Even among non-JavaScript clients, JSON has gained in popularity; for one thing, JSON is more readable than XML because JSON has relatively less markup. Chapter 2 illustrates various ways in which REST-style services can generate XML and JSON payloads; Chapter 3 focuses on consuming XML and JSON payloads from RESTful web services. In SOAP-based services, XML remains the dominant format, although the DotNet framework is especially good at giving JSON equal status.

Several features distinguish web services from other distributed software systems. Here are three:

Open infrastructure
: Web services are deployed using industry-standard, vendor-independent protocols and languages such as HTTP, XML, and JSON, all of which are ubiquitous and well understood. Web services can piggyback on networking, data formatting, security, and other infrastructures already in place, which lowers entry costs and promotes interoperability among services. Organizations that publish websites with production-grade web servers such as Apache2, IIS, and Nginx can publish web services with these very web servers. Firewalls and other security mechanisms that defend websites thereby defend web services as well.

Platform and language transparency
: Web services and their clients can interoperate even if written in different programming languages. Languages such as C, C#, Go, Java, JavaScript, Perl, Python, Ruby, and others provide libraries, utilities, and even frameworks in support of web services. Web services can be published and consumed on various hardware platforms and under different operating systems. Web services are an excellent way to integrate diverse software systems while allowing the programmer to work in the programmer's language of choice. The web service approach to software development is not to rewrite but, rather, to integrate.

Modular design
: Web services are meant to be modular in design so that new services can be composed out of existing ones. Imagine, for example, an inventory-tracking service integrated with an online ordering service to compose a service that automatically orders the appropriate products in response to inventory levels. Web services are the small software parts out of which arbitrarily large systems can be built. A guiding principle in web service design is to begin with very simple service operations, essentially uncomplicated functions, and then group these operations into services, which in turn can be orchestrated to work with other services, and so on indefinitely.

What Good Are Web Services?

This obvious question has no simple answer, but the benefits and promises of web services can be clarified with examples. The first example underscores how the distinction between *websites* and *web services* continues to blur: the data and functionality available at one can be available at the other. (One web framework that emphasizes the blurring is Rails, which is discussed in more detail later.) The second example focuses on how web services can be used to integrate diverse software systems and to make legacy systems more widely accessible.

A visit to a website such as the Amazon E-Commerce site is often interactive: a shopper uses a browser to search the Amazon site for desired items, places some of these in a shopping cart, checks out the cart, finalizes the order with a credit card or the equivalent, and receives a confirmation page and usually an email. Pioneers in web services, such as Amazon, expose the information and functionality of websites through web services as well. Searching and shopping against Amazon, as code examples in later chapters illustrate, are tasks that are automated readily because Amazon makes a point of coordinating its websites with its web services. In any case, here is a sketch of how a search-and-shop experience might be automated.

1. A shopper has a database table or even a simple text file, *wishList.txt*, that contains items of interest such as books, movies, or any other search-and-shop category that Amazon supports.
2. The database table or text file, which acts as a wish list with constraints, provides pertinent information such as the ISBN of a desired book, the maximum price the shopper is willing to pay, the number of items to order, and so on.
3. The shopper programs a client, in whatever language the shopper prefers, that reads the database table or text file, opens a connection to Amazon, searches Amazon for wishlist items, checks whether the items are available under the constraints in the wishlist, and orders the items that meet the constraints.
4. The client program checks an email account for the confirming email; if all goes well, the client places confirmation information in a data store such as a database table or another simple text file.

An interactive shopping experience thus gives way to an automated one. Of course, some shoppers derive as much pleasure from the activity as from the outcome. The point is not that shopping should be automated but, rather, that web services open up this possibility for many tasks, shopping included. At one time, *HTML screen scraping* was a popular way to have applications other than browsers hit a website, download HTML documents, and then parse the HTML for its informational content. As more sites follow the Amazon practice of exposing the same or, at least, nearly the same data and functionality as both websites and web services, this screen scraping becomes increasingly unnecessary. Later chapters illustrate, with code examples, the close relationship between websites and web services.

The second example of what makes web services attractive focuses on a major challenge in modern software development: systems integration. Modern software systems are written in a variety of languages—a variety that seems likely to increase. These software systems will continue to be hosted on a variety of platforms. Institutions large and small have significant investment in legacy software systems whose functionality is useful and perhaps mission critical; few of these institutions have the will and the resources, human or financial, to rewrite their legacy systems. How are such disparate software systems

to interact? That these systems must interact is taken for granted nowadays; it is a rare software system that gets to run in splendid isolation.

A challenge, then, is to have a software system interoperate with others, which may reside on different hosts under different operating systems and may be written in different languages. Interoperability is not just a long-term challenge but also a current requirement of production software. Web services provide a relatively simple answer to question of how diverse software systems, written in many languages and executing on various platforms under different operating systems, can interoperate. In short, web services are an excellent way to integrate software systems.

Web services address the problem of interoperability directly because such services are, first and foremost, language- and platform-neutral. If a legacy COBOL system is exposed through a web service, the system is thereby interoperable with service clients written in other currently more widely used languages. Exposing a legacy COBOL system as a web service should be significantly less expensive than, say, rewriting the system from scratch. Legacy database systems are an obvious source of data and functionality, and these systems, too, can be made readily accessible, beyond the local machine that hosts the database, through web services.

In the past, data sources for applications were usually *data stores* such as relational database management systems (RDBMS) or even local filesystems. Nowadays web services also serve as data sources, at least as intermediate ones that are backed up ultimately with persistent data stores. Indeed, web services integrate readily with RDBMS and other data storage systems as frontends that are easier conversational partners than the data storage systems themselves—because web services, at least well-designed ones, have APIs that publish their functionality in high-level, language-neutral, and platform-independent terms. A web service thus can be viewed as a uniform access mechanism for divergent data stores. A web service can act as the frontend of a database system, a frontend that exposes, through a published API, the data and the functionality of the database system (see Figure 1-2).

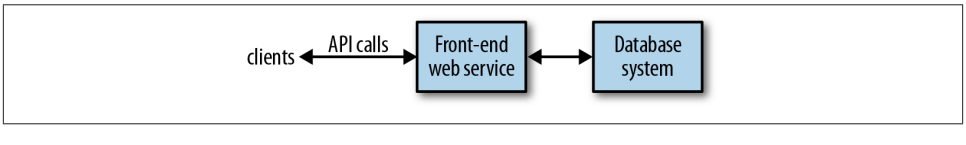


Figure 1-2. A web service as the frontend of a data store

Web services are inherently distributed systems that communicate mostly over HTTP but can communicate over other popular transports as well. The communication payloads of web services are typically structured text, usually XML or JSON documents, which can be inspected, transformed, persisted, and otherwise processed with widely and even freely available tools. When efficiency demands it, however, web services can

also deliver compact binary payloads. Finally, web services are a work in progress with real-world distributed systems as their test bed. For all of these reasons, web services are an essential tool in any modern programmer's toolbox.

The examples that follow, in this and later chapters, are simple enough to isolate critical features of web services such as security but also realistic enough to illustrate the power and flexibility that such services bring to software development. The main service examples have an accompanying Ant script to compile and then publish the web service on a production-grade web server such as Tomcat or Jetty; many of the Java clients against web services are packaged as executable JAR files in order to reduce hassle.

As noted earlier, web services come in different flavors: SOAP-based and REST-style. SOAP and SOA, though related, remain distinct. The next section goes into detail about the relationship between SOA and REST-style and SOAP-based web services.

Web Services and Service-Oriented Architecture

Web services and SOA are related but distinct. SOA, like REST, is more an architectural style—indeed, a mindset—than a body of precisely defined rules for the design and implementation of distributed systems; web services are a natural, important way to provide the services at the core of any SOA system. A fundamental idea in SOA is that an application results from integrating network-accessible services, which are interoperable because each has an interface that clearly defines the operations encapsulated in the service. Per operation, the interface specifies the number and type of each argument passed to the service operation together with the number and type of values returned from each service operation. The very point of a service interface is to publish the invocation syntax of each operation encapsulated in the service. One attraction of the SOA approach is that the ultimate building blocks of even large, complicated systems are structurally simple components; this simplicity at the base level makes it relatively easy to test, debug, deploy, extend, and otherwise maintain a software system.

In an SOA system, services as building block components may be characterized as *unassociated* and *loosely coupled*. Consider, for example, two primitive services, S_1 and S_2, in an SOA application. The two services are unassociated in that neither S_1 nor S_2 depends on the other: S_1 is not required to use S_2 or vice versa. The services are mutually independent but can be used together or orchestrated as parts of a larger software system. Following the same theme, components such as S_1 and S_2 are loosely coupled in that neither needs to know anything about the internal structure of the other in order for both of these services to work together as parts of a larger distributed system. A persistent theme in the many discussions of SOA is the modularity of SOA-based systems.

At the implementation level, a service operation is a function call: the function takes zero or more arguments and returns zero or more values. Although functions in many

languages such as C and even Java technically return, at most, only a single value and therefore must resort to aggregate data structures such as a lists to return multiple values, newer languages such as Go have uncomplicated syntax for functions to return arbitrarily many values including, of course, none. This fact underscores the inherent richness and flexibility of the function as a system building block. Programmers fluent in virtually any language are thereby knowledgeable about the syntax and semantics of functions.

In an SOA system, a very simple service may consist of a single function. The implementation model is thus uncomplicated and familiar to programmers, and the simplicity of service operations promotes code reuse through the composition of new services out of existing ones. This ground-level simplicity also enables relatively straightforward troubleshooting because services reduce to primitive function calls. An SOA system can be quite complicated, of course, but the complication arises from the composition and not from the simple services into which the system ultimately decomposes.

Web services are well suited as components in an SOA system. Following best practices, a web service should consist of operations, each of which is implemented as a stateless function call: the call is *stateless* in that the return value(s) depend only on the arguments passed to the call. In an object-oriented language such as a Java, a well-designed web service is a class that has instance methods as service operations but no instance fields that impact the value returned from a particular method. In practice, *statelessness* is easier said than done, as the many examples in this book illustrate. In the context of SOA, it is common to distinguish between *providers* and *consumers* of web services: the provider furnishes the service's functionality, and the consumer is a client that issues requests against the service's operations. The provider/consumer pair is commonly used to describe web services and their clients, respectively.

Perhaps the best way to clarify SOA in the concrete is to contrast this approach to distributed systems with a quite different approach: DOA (Distributed Object Architecture). Web services came to fore as a reaction against the complexity of DOA systems. The next section provides a short history of web services, with emphasis on the kinds of software challenges that web services are meant to address.

A Very Short History of Web Services

Web services evolved from the RPC (Remote Procedure Call) mechanism in DCE (Distributed Computing Environment), a framework for software development from the early 1990s. DCE includes a distributed filesystem (DCE/DFS) and a Kerberos-based authentication system. Although DCE has its origins in the Unix world, Microsoft quickly did its own implementation known as MSRPC, which in turn served as the infrastructure for interprocess communication in Windows. Microsoft's COM/OLE (Common Object Model/Object Linking and Embedding) technologies and services were built on a DCE/RPC foundation. There is irony here. DCE designed RPC as a way

to do distributed computing (i.e., computing across distinct physical devices), and Microsoft cleverly adapted RPC to support interprocess communication, in the form of COM infrastructure, on a single device—a PC running Windows.

The first-generation frameworks for distributed object systems, CORBA (Common Object Request Broker Architecture) and Microsoft's DCOM (Distributed COM), are anchored in the DCE/RPC procedural framework. Java RMI (Remote Method Invocation) also derives from DCE/RPC, and the method calls in Java EE (Enterprise Edition), specifically in Session and Entity EJBs (Enterprise Java Bean), are Java RMI calls. Java EE (formerly J2EE) and Microsoft's DotNet are second-generation frameworks for distributed object systems, and these frameworks, like CORBA and DCOM before them, trace their ancestry back to DCE/RPC. By the way, DCE/RPC is not dead. Various popular system utilities (for instance, the Samba file and print service for Windows clients) use DCE/RPC.

From DCE/RPC to XML-RPC

DCE/RPC has the familiar client/server architecture in which a client invokes a procedure that executes on the server. Arguments can be passed from the client to the server and return values can be passed from the server to the client. The framework is platform- and language- neutral in principle, although strongly tilted toward C in practice. DCE/RPC includes utilities for generating client and server artifacts (stubs and skeletons, respectively). DCE/RPC also provides software libraries that hide the transport details. Of interest now is the IDL (Interface Definition Language) document that acts as the service contract and is an input to utilities that generate artifacts in support of the DCE/RPC calls. An IDL document can be short and to the point (see Example 1-1).

Example 1-1. A sample IDL document that declares the `echo` function

```
/* echo.idl */
[uuid(2d6ead46-05e3-11ca-7dd1-426909beabcd), version(1.0)]
interface echo {
    const long int ECHO_SIZE = 512;
    void echo(
        [in]             handle_t h,
        [in,  string]    idl_char from_client[ ],
        [out, string]    idl_char from_server[ECHO_SIZE]
    );
}
```

The IDL interface named `echo`, identified with a machine-generated UUID (Universally Unique IDentifier), declares a single function with the same name, `echo`. The names are arbitrary and need not be the same. The `echo` function expects three arguments, two of which are `in` parameters (that is, inputs into the remote procedure) and one of which is an `out` parameter (that is, an output from the remote procedure). The first argument, of built-in type `handle_t`, is required and points to an RPC data structure. The function

`echo` could but does not return a value, because the echoed string is returned instead as an `out` parameter. The IDL specifies the invocation syntax for the `echo` function, which is the one and only operation in the service. Except for annotations in square brackets to the left of the three `echo` parameters, the syntax of the IDL is essentially C syntax. The IDL document is a precursor of the WSDL (Web Service Description Language) document that provides a formal specification of a web service and its operations. The WSDL document is discussed at length in Chapter 4 on SOAP-based services.

There is a Microsoft twist to the IDL story as well. An ActiveX control under Windows is a DLL (Dynamic Link Library) with an embedded *typelib*, which in turn is a compiled IDL file. For example, suppose that a calendar ActiveX control is plugged into a browser. The browser can read the *typelib*, which contains the invocation syntax for each operation (e.g., displaying the next month) in the control. An ActiveX control is thus a chunk of software that embeds its own interface. This is yet another inspired local use of a technology designed for distributed computing.

In the late 1990s, Dave Winer of UserLand Software developed XML-RPC, a technology innovation that has as good a claim as any to mark the birth of web services. XML-RPC is a very lightweight RPC system with support for elementary data types (basically, the built-in C types together with a `boolean` and a `datetime` type) and a few simple commands. The original specification is about seven pages in length. The two key features are the use of XML marshaling/unmarshaling to achieve language neutrality and reliance on HTTP (and, later, SMTP) for transport. The term *marshaling* refers to the conversion of an in-memory object (for instance, an `Employee` object in Java) to some other format (for instance, an XML document); *unmarshaling* refers to the inverse process of generating an in-memory object from, in this example, an XML document. The marshal/unmarshal distinction is somewhere between close to and identical with the serialize/deserialize distinction. My habit is to use the distinctions interchangeably. In any case, the O'Reilly open-wire Meerkat service and the WordPress publishing platform are based on XML-RPC.

Two key differences separate XML-RPC, on the one side, from DCE/RPC and its offshoots, on the other side:

- XML-RPC payloads are text, whereas DCE/RPC payloads are binary. Text is relatively easy to inspect and process with standard, readily available tools such as editors and parsers.
- XML-RPC transport uses HTTP rather than a proprietary system. To support XML-RPC, a programming language requires only a standard HTTP library together with libraries to generate, parse, transform, and otherwise process XML.

As an RPC technology, XML-RPC supports the request/response pattern. Here is the XML request to invoke, on a remote machine, the Fibonacci function with an argument of 11. This argument is passed as a 4-byte integer, as the XML start tag `<i4>` indicates:

```
<?xml version="1.0">
<methodCall>
   <methodName>fib<methodName>
   <params>
     <param><value><i4>11</i4></value></param>
   </params>
</methodCall>
```

The integer 11 occurs in the XML-RPC message as text. An XML-RPC library on the receiving end needs to extract 11 as text and then convert the text into a 4-byte integer in the receiving language such as Go or Java. Even this short example illustrates the idea of having XML—in particular, data types expressed in XML—serve as the leveling mechanism between two different languages involved in an XML-RPC exchange.

XML-RPC is deliberately low fuss and lightweight. SOAP, an XML dialect derived straight from XML-RPC, is considerably heavier in weight. From inception, XML-RPC faced competition from second-generation DOA systems such as Java EE (J2EE) and AspNet. The next section considers the challenges inherent in DOA systems. These challenges sustained and eventually intensified interest in lighter-weight approaches to distributed computing—modern web services.

Distributed Object Architecture: A Java Example

What advantages do web services have over DOA technologies such as Java RMI? This section addresses the question with an example. Java RMI (including the Session and Entity EJB constructs built on Java RMI) and DotNet Remoting are examples of second-generation distributed object systems. Consider what a Java RMI client requires in order to invoke a method declared in a service interface such as this:

```
import java.util.List;
public interface BenefitsService extends java.rmi.Remote {
   public List<Benefit> getBenefits(Emp emp) throws RemoteException;
}
```

The interface appears deceptively simple in that it declares only the method named `getBenefits`, yet the interface likewise hints at what makes a Distributed Object Architecture so tricky. A client against this `BenefitsService` requires a Java RMI stub, an instance of a class that implements the `BenefitsService` interface. The stub is downloaded automatically from the server to the client as part of the Java RMI setup (see Figure 1-3).

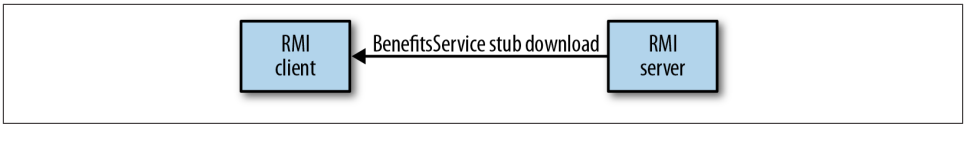


Figure 1-3. Downloading a stub in Java RMI

Once the stub setup is done, the `getBenefits` method is executed as a stub method; that is, the stub acts as the client-side object making a remote method call through one of stub's encapsulated methods. The call thus has the following syntax:

```
Emp fred = new Emp();
//...
List<Benefit> benefits = rmiStub.getBenefits(fred); // rmiStub = reference
```

Invoking the `getBenefits` method requires that the byte codes for various Java classes, standard and programmer-defined, be available on the client machine. To begin, the client needs the class `Emp`, the argument type for the `getBenefits` method, and the class `Benefit`, the member type for the `List` that the method `getBenefits` returns. Suppose that the class `Emp` begins like this:

```
public class Emp {
   private Department                  department;
   private List<BusinessCertification> certifications;
   private List<ClientAccount>         accounts;
   private Map<String, Contact>        contacts;
   ...
}
```

The standard Java types such as `List` and `Map` are already available on the client side because the client is, by assumption, a Java application. The challenge involves the additional, programmer-defined types such as `Department`, `BusinessCertification`, `ClientAccount`, and `Contact` that are needed to support the client-side invocation of a remotely executed method. The setup on the client side to enable a remote call such as:

```
Emp fred = new Emp();
// set properties, etc.
List<EmpBenefits> fredBenefits = rmiStub.getBenefits(fred);
```

is significant, with lots and lots of bytes required to move from the server down to the client just for the setup. Anything this complicated is, of course, prone to problems such as versioning issues and outright errors in the remote method calls.

Java RMI uses proprietary marshaling/unmarshaling and proprietary transport, and DotNet does the same. There are third-party libraries for interoperability between the two frameworks. Yet a Java RMI service can be expected to have mostly Java clients, and a DotNet Remoting service can be expected to have mostly DotNet clients. Web services represent a move toward standardization, simplicity, and interoperability.

Web Services to the Rescue

Web services simplify matters in distributed computing. For one thing, the client and service typically exchange XML or equivalent documents, that is, *text*. If needed, non-text bytes can be exchanged instead, but the preferred payloads are text. The exchanged text can be inspected, validated, transformed, persisted, and otherwise processed using

readily available, nonproprietary, and often free tools. Each side, client and service, simply needs a local software library that binds language-specific types such as the Java `String` to XML Schema or comparable types, in this case `xsd:string`. (In the qualified name `xsd:string`, `xsd` is a namespace abbreviation and `string` is a local name. Of interest here is that `xsd:string` is an XML type rather than a Java type.) Given these Java/XML bindings, relatively uncomplicated library modules can convert from one to the other—from Java to XML or from XML to Java (see Figure 1-4).

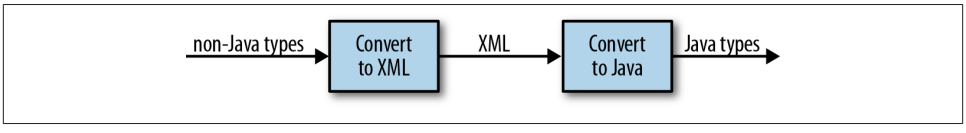


Figure 1-4. Java/XML conversions

Processing on the client side, as on the service side, requires only locally available libraries and utilities. The complexities, therefore, can be isolated at the endpoints—the service and the client applications together with their supporting libraries—and need not seep into the exchanged messages. Finally, web services are available over HTTP, a nonpropriety protocol that has become standard, ubiquitous infrastructure; HTTP in particular comes with a security extension, HTTPS, that provides multifaceted security services.

In a web service, the requesting client and the service need not be coded in the same language or even in the same style of language. Clients and services can be implemented in object-oriented, procedural, functional, and other language styles. The languages on either end may be statically typed (for instance, Java and Go) or dynamically typed (for example, JavaScript and Ruby). The complexities of stubs and skeletons, the serializing and deserializing of objects encoded in some proprietary format, give way to relatively simple text-based representations of messages exchanged over standard transports such as HTTP. The messages themselves are neutral; they have no bias toward a particular language or even family of languages.

The first code example in this chapter, and all of the code examples in Chapter 2 and Chapter 3, involve REST-style services. Accordingly, the next section looks at what REST means and why the REST-style service has become so popular. From a historical perspective, REST-style services can be viewed as a reaction to the growing complexity of SOAP-based ones.

What Is REST?

Roy Fielding coined the acronym REST in his PhD dissertation. Chapter 5 of Fielding's dissertation lays out the guiding principles for what have come to be known as REST-style or RESTful web services. Fielding has an impressive résumé. He is, among other

things, a principal author of the HTTP 1.1 specification and a cofounder of the Apache Software Foundation.

REST and SOAP are quite different. SOAP is a messaging protocol in which the messages are XML documents, whereas REST is a style of software architecture for distributed hypermedia systems, or systems in which text, graphics, audio, and other media are stored across a network and interconnected through hyperlinks. The World Wide Web is the obvious example of such a system. As the focus here is on *web* services, the World Wide Web is the distributed hypermedia system of interest. In the Web, HTTP is both a transport protocol and a messaging system because HTTP requests and responses are messages. The payloads of HTTP messages can be typed using the MIME (Multipurpose Internet Mail Extension) type system. MIME has types such as `text/html`, `application/octet-stream`, and `audio/mpeg3`. HTTP also provides response status codes to inform the requester about whether a request succeeded and, if not, why. Table 1-1 lists some common status codes.

Table 1-1. Sample HTTP status codes and their meanings

Status code	In English	Meaning
200	OK	Request OK
303	See Other	Redirect
400	Bad Request	Request malformed
401	Unauthorized	Authentication error
403	Forbidden	Request refused
404	Not Found	Resource not found
405	Method Not Allowed	Method not supported
415	Unsupported Media Type	Content type not recognized
500	Internal Server Error	Request processing failed

REST stands for REpresentational State Transfer, which requires clarification because the central abstraction in REST—the resource—does not occur in the acronym. A *resource* in the RESTful sense is something that is accessible through HTTP because this thing has a name—URI (Uniform Resource Identifier). A URI has two subtypes: the familiar URL, which specifies a *location*, and the URN, which is a symbolic name but not a location. URIs are *uniform* because they must be structured in a certain way; there is a syntax for URIs. In summary, a URI is a standardized name for a resource and, in this sense, a URI acts as noun.

In practical terms, a resource is a web-accessible, informational item that may have hyperlinks to it. Hyperlinks use URIs to do the linking. Examples of resources are plentiful but likewise misleading in suggesting that resources must have something in common other than identifiability through URIs. The gross national product of Lithuania is a resource, as is the Modern Jazz Quartet. Ernie Banks' baseball accomplishments

count as a resource, as does the maximum flow algorithm. The concept of a resource is remarkably broad but, at the same time, impressively simple and precise.

As web-based informational items, resources are pointless unless they have at least one representation. In the Web, representations are MIME typed. The most common type of resource representation is probably still `text/html`, but nowadays resources tend to have multiple representations. For example, there are various interlinked HTML pages that represent the Modern Jazz Quartet but there are also audio and audiovisual representations of this resource.

Resources have state. Ernie Banks' baseball accomplishments changed during his career with the dismal Chicago Cubs from 1953 through 1971 and culminated in his 1977 induction into the Baseball Hall of Fame. A useful representation must capture a resource's state. For example, the current HTML pages on Ernie at the Baseball Reference website need to represent all of his major league accomplishments, from his rookie year in 1953 through his induction into the Hall of Fame.

A RESTful request targets a resource, but the resource itself typically is created on the service machine and remains there. A resource may be persisted in a data store such as a database system. Some mix of humans and applications may maintain the state of the resource. In the usual case of web service access to a resource, the requester receives a representation of the resource if the request succeeds. It is the representation that transfers from the service machine to the requester machine. In a REST-style web service, a client does two things in an HTTP request:

- Names the targeted resource by giving its URI, typically as part of a URL.
- Specifies a *verb* (HTTP method), which indicates what the client wishes to do; for example, *read* an existing resource, *create* a new resource from scratch, *edit* an existing resource, or *delete* an existing resource.

One of the basic cases is a *read* request. If a *read* request succeeds, a typed representation (for instance, `text/html`) of the resource is transferred from the server that hosts and maintains the resource to the client that issues the request. The client is an arbitrary application written in some language with support for REST-style requests. The representation returned from the service is a good one only if it captures the resource's state in some appropriate way. Figure 1-5 depicts a resource with its identifying URI together with a RESTful client and some typed representations sent back to the client in response to client requests.

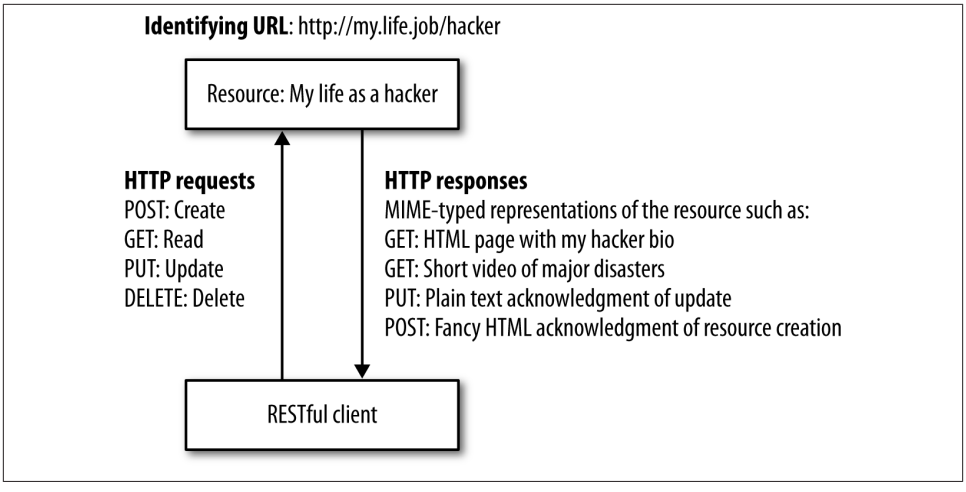


Figure 1-5. A small slice of a RESTful system

In summary, RESTful web services involve not just resources to represent but also client-invoked operations on such resources. At the core of the RESTful approach is the insight that HTTP, despite the occurrence of Transport in its name, acts as an API and not simply as a transport protocol. HTTP has its well-known verbs, officially known as *methods*. Table 1-2 lists the HTTP verbs that correspond to the CRUD (*Create*, *Read*, *Update*, *Delete*) operations so familiar throughout computing.

Table 1-2. HTTP verbs and their CRUD operations

HTTP verb	CRUD operation
POST	Create
GET	Read
PUT	Update
DELETE	Delete

Although HTTP is not case sensitive, the HTTP verbs are traditionally written in uppercase. There are additional verbs. For example, the verb HEAD is a variation on GET that requests only the HTTP headers that would be sent to fulfill a GET request.

HTTP also has standard response codes such as 404 to signal that the requested resource could not be found and 200 to signal that the request was handled successfully. In short, HTTP provides request verbs and MIME types for client requests and status codes (and MIME types) for service responses.:

Modern browsers generate only GET and POST requests. If a user enters a URL into the browser's input window, the browser generates a GET request. A browser ordinarily generates a POST request for an HTML form with a *submit* button. It goes against the