REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

1 Web Services Quickstart

1.1 What Are Web Services?

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, that is, an application typically delivered over HTTP (HyperText Transport Protocol). The obvious way to publish a web service is with a web server; and a web service client needs to execute on a machine that has network access, usually through HTTP, to the web server. In slightly more technical terms, a web service is a distributed application whose components can be deployed and executed on physically distinct devices. Figure 1 depicts such a service, with *host1* as a web server machine and *host2* as a mobile device. Web services can be arbitrarily complex. 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 consume the service.

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, that is, with SOAP messages. REST-style services are hard to characterize in a sentence or two; hence, the next section goes into detail. For now, a REST-style service is one that treats HTTP not only as a transport protocol also but as a set of guidelines for structuring service requests and service responses. In a REST-style service, HTTP itself acts as an API. SOAP has standards, tool-kits, and bountiful software libraries. REST has no official standards, few tool-kits, and uneven software libraries across programming languages. The REST style can be viewed as an antidote to the creeping complexity of SOAP-based web services. This book covers SOAP-based and REST-style web services, starting with REST-style ones. This chapter ends with a sample REST-style service.

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. SOAP originally stood for Simple Object Access Protocol and then, by serendipity but not officially, might have stood for Service Oriented Architecture (SOA) Protocol. (SOA is discussed in Section 1.4.) Deconstructing SOA is non-trivial but one point is indisputable: whatever SOA may be, web services play a central role in the SOA approach to software design and development. The World Wide Web Consortium (hereafter, W3C) currently oversees the SOAP standard; and SOAP officially is no longer an acronym.

1.2 Web Service Miscellany

Except in test mode, the client of either a SOAP-based or REST-style service is rarely a web browser but 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 the key 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#, Go, JavaScript, Perl, and Ruby. 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 Perl 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). Web service clients are increasingly JavaScript programs embedded in HTML documents and executing in a browser. For a JavaScript consumer of a web service, JSON has obvious appeal because a JSON document represents a native JavaScript object. Chapter 3 focuses on 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.

- 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.

- 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 on-line ordering service to compose a service that automatically orders the appropriate products in response to inventory levels.

1.3 What Good are Web Services?

This obvious question has no simple answer. Nonetheless, the chief benefits and promises of web services are clear. 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; and 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 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.

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 also can 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 but also realistic enough to illustrate the power and flexibility that such services bring to software development. The next section clarifies the relationship between SOAP and SOA.

1.4 Web Services and Service Oriented Architecture

Web services and *service-oriented architecture* (hereafter, SOA) are related but distinct. SOA, like REST itself, is more an architectural style—indeed, a mindset—than a body of well-defined rules for the design and implementation of distributed systems; and 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.

At the implementation level, a service operation is a function call: the function takes zero or more arguments and returns a value, perhaps a list. The implementation model is thus very simple; and the simplicity of service operations promotes code reuse through the composition of new services out of existing ones and enables relatively straightforward troubleshooting because services reduce to primitive function calls. Perhaps the best way to clarify SOA is to contrast this approach to distributed systems with a preceding approach, DOA (Distributed Object Architecture). The next section goes into detail.

1.5 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 that emerged in the early 1990s. DCE includes a distributed file system (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, that is, 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.

1.5.1 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 biased towards C in practice. DCE/RPC includes utilities for generating client and server artifacts (stubs and skeletons, respectively) and 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. Example 1 is a sample IDL file.

Example 1.1 A sample IDL file 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, identified with a machine-generated UUID (Universally Unique IDentifier), declares a single function of 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.

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. 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; and *unmarshaling* references to the inverse process of generating an in-memory object from, in this example, an XML document. The O'Reilly open-wire Meerkat service is an XML-RPC application.

As an RPC technology, XML-RPC supports the request/response pattern. Here is the XML request to invoke, presumably on a remote machine, the Fibonacci function with an argument of 11. This argument is passed as a four-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>
```

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.

1.5.2 Distributed Object Architecture: A Java Example

Java RMI, including the Session and Entity EJB constructs built on it, and DotNet Remoting are examples of second-generation distributed object systems. Consider what a Java RMI client requires to invoke a method declared in a service interface such as this:

```
package doa; // distributed object architecture
import java.util.List;

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

The interface appears deceptively simple in that it declares only one method, `getBenefits`; yet the interface likewise hints at what makes a distributed-object architecture so tricky. A client against this `BenefitsService` uses a Java RMI stub, an instance of a class that implements the `BenefitsService` interface and is downloaded automatically from the server, to invoke the `getBenefits` method. Invoking the `getBenefits` method requires that the byte codes for various Java classes, standard and programmer-defined, be downloaded to the client machine. To begin, the client needs the classes `Emp`, the argument type for the `getBenefits` method, and `EmpBenefits`, the member type for the `List` that the method `getBenfits` returns. Now suppose that the class `Emp` looks like this:

```
public class Emp {
   private Department                   department;
   private List<BusinessCertification>  certifications;
   private List<ClientAccount>          accounts;
   ...
}
```

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

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

is significant, with lots of bytes required to move from the server down to the client. Anything this complicated is, of course, prone to error.

1.5.3 Web Services to the Rescue

Web services simplify matters. 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, non-proprietary, 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 simple library modules can serialize and deserialize from one to the other, that is, from Java to XML or from XML to Java. 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 non-propriety protocol that has become standard, ubiquitous infrastructure.

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, gives way to relatively simple text-based representations of request and response messages.

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 takes a quick look at what REST means.

1.6 What is REST?

Roy Fielding (*http://roy.gbiv.com*) coined the acronym REST in his PhD dissertation. Chapter 5 of the dissertation lays out the guiding principles for what have come to be known as REST-style or RESTful web services. Fielding has an impressive resume. He is, among other things, a principal author of the HTTP 1.1 specification and a co-founder of the Apache Software Foundation.

REST and SOAP are quite different. SOAP is a messaging protocol in which the messages are officially XML documents, whereas REST is a style of software architecture for distributed hypermedia systems, that is, 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 our focus is 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.

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 an HTTP resource: anything that has a URI, that is, an identifier that satisfies formatting requirements. The formatting requirements are what make URIs uniform. Recall, too, that URI stands for Uniform Resource Identifier; hence, the notions of URI and resource are intertwined. In plain language, a URI names a resource and, in this way, acts as a noun.

In practical terms, a resource is an informational item that has 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 in 2013 is a resource as is the Modern Jazz Quartet. Ernie Bank's 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. For example, Ernie Bank's baseball accomplishments changed during his career with the 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 Web site (*http://www.baseball-reference.com*) 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 remains or is created on the service machine. In the usual case, 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 different terms, a RESTful client issues a request that involves a resource, for instance, a request to read the resource. If this read request succeeds, a typed representation (for instance, `text/html`) of the resource is transferred from the server that hosts the resource to the client that issued the request. The representation is a good one only if it captures the resource's state in some appropriate way.

In summary, RESTful Web services require 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, is an API (Application Programming Interface) and not simply a transport protocol. HTTP has its well-known verbs, officially known as *methods*. Table 1 lists the HTTP verbs that correspond to the CRUD (Create, Read, Update, Delete) operations so familiar throughout computing:

Table 1: 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. There are also TRACE and INFO verbs.

Figure (restful.png) is a whimsical depiction of a resource with its identifying URI together with a RESTful client and some typed representations sent as responses to HTTP requests for the resource. Each HTTP request includes a verb to indicate which CRUD operation should be performed on the resource. A good representation is precisely one that matches the requested operation and captures the resource's state in some appropriate way. For example, in this depiction a GET request could return my biography as a hacker as either an HTML document or a short video summary. The video would fail to capture the state of the resource if it depicted, say, only the major disasters in my brother's career rather than those in my own. A typical HTML representation of the resource would include hyperlinks to other resources, which in turn could be the target of HTTP requests with the appropriate CRUD verbs.

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 spirit of REST to treat GET and POST interchangeably. For example, Java `HttpServlet` instances have callback methods such as `doGet` and `doPost` that handle GET and POST requests, respectively. Each callback has the same parameter types, `HttpServletRequest` (the key/value pairs from the request) and `HttpServletResponse` (a typed response to the requester). It is not unknown for a programmer to have the two callbacks execute the same code (for instance, by having one invoke the other), thereby conflating the original HTTP distinction between *read* and *create*. A key guiding principle of the RESTful style is to respect the original meanings of the HTTP verbs. In particular, any GET request should be side-effect free (or, in jargon, idempotent) because a GET is a *read* rather than a *create*, *update*, or *delete* operation. A GET as a *read* with no side effects is called a *safe* GET.

The REST approach does not imply that either resources or the processing needed to generate adequate representations of them are simple. A REST-style Web service might be every bit as subtle and complicated as a SOAP-based service. The RESTful approach tries to simplify matters by taking what HTTP and the MIME type system already offer: built-in CRUD operations, uniformly identifiable resources, and typed representations that can capture a resource's state. REST as a design philosophy tries to isolate application complexity at the endpoints, that is, at the client and at the service. A service may require lots of logic and computation to maintain resources and to generate adequate representation of resources, for instance, large and subtly formatted XML documents; and a client may require significant XML processing to extract the desired information from the

XML representations transferred from the service to the client. Yet the RESTful approach keeps the complexity out of the transport level, as a resource representation is transferred to the client as the body of an HTTP response message. For the record, RESTful web services are Turing complete; that is, these services are equal in power to any computational system, including a system that consists of SOAP-based web services. [1]

1.6.1 Verbs and Opaque Nouns

In HTTP a URI is meant to be opaque, which means that the URI

```
http://bedrock/citizens/fred
```

has no inherent connection to the URI

```
http://bedrock/citizens
```

although Fred happens to be a citizen of Bedrock. These are simply two different, independent identifiers. Of course, a good URI designer will come up with URIs that are suggestive about what they are meant to identify. The point is that URIs have no intrinsic hierarchical structure. URIs can and should be interpreted but these interpretations are imposed on URIs, not inherent in them. Although URI syntax looks like the syntax used to navigate a hierarchical file system, this resemblance is misleading. A URI is an opaque identifier, a logically proper name that denotes exactly one resource.

1.7 Review of HTTP Requests and Responses

The next section has a REST-style sample service whose URL is

```
http://localhost:8080/cliches/
```

If this URL were typed into a browser's window, the browser would generate an HTTP request similar to

```
GET /cliches/ HTTP/1.1
User-Agent: Mozilla/5.0 (X11; Linux x86_64) Chrome/24.0.1312.56
Host: localhost:8080
Accept: text/html
```

The browsers parses the entered URL into these parts, with clarifications below:

- `GET /cliches/ HTTP/1.1`

  This is the HTTP request *start line*:

  - GET is the HTTP method (verb)
  - `/cliches/` is the URI (resource's name)
  - `HTTP/1.1` is the HTTP version that the requester is using

- `User-Agent:  Mozilla/5.0 (X11; Linux x86_64) Chrome/24.0`

  Chrome is the browser used in this request and Mozilla/5.0 specifies a browser-compatibility type. The `User-Agent` information also includes the operating system in use, 64-bit Linux. Of interest here is that term *user agent* captures the intended meaning: it is the application (agent) that a user employs to make a request.

- `Host:  localhost:8080`

  In `localhost:8080`, the network address of the machine that hosts the resource is to the left of the colon; and the port number, in this case `8080`, is to the right. In this example, the network address is `localhost` and its dotted-decimal equivalent 127.0.0.1. Because the network address is `localhost`, the web server and the requesting application are on the same machine, which is convenient during development. In a production environment, the web server might have a network address such as `dcequip.cti.depaul.edu`. Port numbers range from 0 to roughly 65,000, with port numbers from 0 through 1023 typically reserved for standard applications such as web servers (port 80 for HTTP and 443 for HTTPS), SMTP (email, port 25), SSH (secure shell, port 22), and so on. For convenience, Tomcat and Jetty use port 8080 by default but the number can be changed (for example, to the standard HTTP port number 80).

[1]For a thorough coverage of REST-style web services, see Richardson and Ruby's book *RESTful Web Services* (O'Reilly, 2007).

- `Accept: text/html`

  This is the MIME type (`text`) and subtype (`html`), which the browser is ready to accept. The application running on web server may not honor the requested type and respond instead with, for example, `text/plain` or `text/xml`.

The key/value pairs such as

```
Accept: text/html
```

make up the HTTP request headers. These pairs may occur in any order and only the

```
Host: <network address>
```

pair is mandatory under HTTP 1.1. In an HTTP header element, a colon separates the key from the value.

Two newlines terminate the headers section. A GET request has no body; hence, a GET request consists only of the start line and the headers. A POST request always has a body, which may be empty. In a POST request, two newlines also mark the end of the headers.

Because a GET request has no body, such a request often includes, in the URI, a query string that consists of key/value pairs. For example, this GET request

```
http://.../products?id=27&category=boots
```

includes a query-string with two key/value pairs: `id` is the first key and `27` is the value; `category` is the second key and `boots` is the value. The query string thus provides a way for a body-less GET request to include information within the request. The query string data are encapsulated in the HTTP request headers. POST requests always have a body, which is usually non-empty. The body of a POST request holds key/value pairs as well.

If all goes well, sending an HTTP request to the URL

```
http://localhost:8080/cliches/sayings.jsp
```

leads to an HTTP response, which is similar to

Example 1.2 HTTP XML response from the *sayings* RESTful service

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Set-Cookie: JSESSIONID=35B1E3AA21EB7242FD2FC50044D2166A; Path=/cliches/;
Content-Type: text/html;charset=ISO-8859-1
Transfer-Encoding: chunked
Date: Tue, 29 Jan 2013 17:40:15 GMT

<?xml version="1.0" encoding="UTF-8"?>
<java version="1.7.0" class="java.beans.XMLDecoder">
 <array class="cliches.Prediction" length="32">
  <void index="0">
   <object class="cliches.Prediction">
    <void property="what">
     <string>
        Managed holistic contingency will grow killer action-items.
     </string>
    </void>
    <void property="who">
     <string>
        Cornelius Tillman
     </string>
    </void>
   </object>
  </void>
  ...
```

The start line

```
HTTP/1.1 200 OK
```

begins with the HTTP version in use on the server. Next comes the HTTP status code (**SC** for short) as a number, 200, and in English, OK. Status codes in the 200-range signal success. Five header elements follow, including the name of the web server that sends the response and the content type of the response. Note that the response type is given as `text/html` rather than as what it actually is: `text/xml`. The reason is that my code, which generates the response, does not bother to set the content type; hence, the Apache-Coyote web server assumes the default type of `text/html`. Two newline characters again separate the headers from the HTTP body, which can be empty. In this case, the body is an XML document that lists corporate predictions together with their predictors.

1.8 HTTP as an API

HTTP can be viewed as an API. Among frameworks for developing web sites and RESTful web services, Rails has pioneered this view of HTTP, which deliberately blurs the distinction between web sites that deliver HTML and web services that deliver XML or JSON. In a well-designed Rails application, a GET request against a URI such as */products* is equivalent to the same request for */products.html*; and an HTML list of products is returned in response. A GET request against */products.json* or */products.xml* would return the same list but in JSON or XML, respectively. Rails as a scheme of URI naming patterns and the HTTP verbs that highlight the elegant yet practical use of HTTP as an API. Below is a summary of the Rails approach. In a URI, a term such as *:id*, which begins with a colon character, indicates a placeholder or parameter, in this case a placeholder whose intended value is a numerical identifier such as 27.

Table 2: Rails routing idioms

HTTP Verb	**URI**	**Meaning**
GET	/products	Read all products
POST	/products	Create new product from information in POST body
GET	/products/new	Read form to create new product
GET	/products/:id/edit	Read form to edit existing product
GET	/products/:id	Read a single product
PUT	/products/:id	Update product with information in POST body
DELETE	/products:id	Delete the specified product

These verb/URI pairs are terse, precise, and uniform in style. The pairs illustrate that RESTful conventions can yield simple, clear expressions about which operation should be performed on which resource. The POST and PUT verbs are used in requests that have an HTTP body; hence, the request data are in the HTTP message body. The GET and DELETE verbs are used in requests that have no body; hence, the request data are sent as query-string key/value pairs.

The decision about whether to be RESTful in a particular application depends, as always, on practical matters that will come to the fore throughout this book. The current section has looked at REST from on high; it is now time to descend into details through a code example.

1.9 A First RESTful Example

As befits a first example, the implementation is simple but sufficient to highlight key aspects of a RESTful web service. The implementation consists of a JSP (Java Server Pages) script and a back-end JavaBean that the JSP script accesses to get data. The data are sage corporate predictions. Here is a sample:

```
Decentralized 24/7 hub will target robust web-readiness.
Synergistic disintermediate policy will expedite back-end experiences.
Universal fault-tolerant architecture will synthesize bleeding-edge channels.
```

Each prediction has an associated human predictor. The RESTful resource is thus a list of predictor names (*e.g.*, Hollis McCullough) and their predictions (Hollis is responsible for the third prediction shown above). The resource name or URI is `/cliches/`; and the only allowable HTTP verb is GET, which corresponds to read among the CRUD operations. If the HTTP request is correct, the RESTful service returns an XML representation of the predictor/prediction list; otherwise, the service returns the appropriate HTTP status code, *e.g.*, 404 for "Not Found", if the URI is incorrect, or 405 for "Method Not Allowed", if the verb is not GET. Figure 1.5 shows a slice of the XML payload returned upon a successful request.

Example 1.3 The XML response from the sayings service

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.7.0" class="java.beans.XMLDecoder">
 <array class="cliches.Prediction" length="32">
  <void index="0">
   <object class="cliches.Prediction">
    <void property="what">
     <string>
       Managed holistic contingency will grow killer action-items.
     </string>
    </void>
    <void property="who">
     <string>Cornelius Tillman</string>
    </void>
   </object>
  </void>
  ...
  <void index="30">
   <object class="cliches.Prediction">
    <void property="what">
     <string>
       Balanced clear-thinking utilisation will expedite collaborative initiatives.
     </string>
    </void>
    <void property="who">
     <string>Deven Blanda</string>
    </void>
   </object>
  </void>
  <void index="31">
   <object class="cliches.Prediction">
    <void property="what">
     <string>
       Versatile tangible application will maximize rich e-business.
     </string>
    </void>
    <void property="who">
     <string>Hiram Gulgowski</string>
    </void>
   </object>
  </void>
 </array>
</java>
```

1.9.1 How the *sayings* Web Service Works

The JSP script (see Example 4) first checks the request's HTTP method and, if this is GET, returns an XML representation of the predictor/prediction list. If the verb is not GET, the script returns an error message together with the HTTP status code. The relevant code is:

```
String verb = request.getMethod();
if (!verb.equalsIgnoreCase("GET")) {
  response.sendError(response.SC_METHOD_NOT_ALLOWED, "GET requests only are allowed.");
}
```

JSP scripts have implicit object references such as `request`, `response`, and `out`; each of these is a field or a parameter in the servlet code into which the web server, in this case Tomcat, translates the JSP script. Accordingly, the JSP script can make the same calls as an `HttpServlet`.

Example 1.4 The JSP script `sayings.jsp`.

```
<!-- Connect to the back-end Predictions POJO and set the ServletContext. -->
<jsp:useBean id    = "preds"
             type  = "cliches.Predictions"
             class = "cliches.Predictions">

  <% // Check the HTTP verb: if it's anything but GET, return 405 (Method Not Allowed).
     String verb = request.getMethod();

     if (!verb.equalsIgnoreCase("GET")) {
       response.sendError(response.SC_METHOD_NOT_ALLOWED, "GET requests only are allowed.") ↩
           ;
     }
     // If it's a GET request, return the predictions.
     else {
       // Object reference application has the value
       // pageContext.getServletContext()
       preds.setServletContext(application);
       out.println(preds.getPredictions());
     }
  %>
</jsp:useBean>
```

On a successful request, the JSP script returns a list of predictions and their predictors, a list available from the back-end JavaBean `cliches.Predictions`. The JSP code is straightforward:

```
out.println(preds.getPredictions());
```

The object reference `out`, available in every JSP script, refers to an output stream through which the JSP script can communicate with the client. In this example, the object reference `preds` refers to the back-end JavaBean that maintains the collection of predictions; and the `getPredictions` method converts the Java list of `Predictions` into an XML document.

The back-end code consists of two classes, `Prediction` (see Example 5) and `Predictions` (see Example 6). The `Prediction` class is quite simple, consisting of two properties: `who` is the person making the prediction and `what` is the prediction. The `Predictions` class does the grunt work. For example, its `populate` method reads the prediction data from a text file, *predictions.db*, encapsulated in the deployed WAR file; and the `toXML` method serializes a Java `List<Prediction>` into an XML document, which in turn is sent back to the client.

On a successful request, the JSP script invokes the back-end bean method `setServletContext` (the implicit object reference is `application`) because the back-end bean needs access to the servlet context in order to read data from a text file embedded in the deployed WAR file. The `ServletContext` is a data structure through which a servlet/JSP script interacts explicitly with the servlet container. The call to the `setServletContext` method sets up the subsequent call to the `getPredictions` method, which returns the XML representation shown in Example 3. Here is the `getPredictions` method without the comments:

```
public String getPredictions() {
   if (null == getServletContext()) return null;
   if (null == predictions) populate();
   return toXML();
}
```

The `predictions` reference is to the `Map` in which `Prediction` references are values. If the servlet context has not been set, then there is no point in continuing because the `populate` method requires the servlet context (the reference is `sctx` in the code) in order to access the data:

```
private void populate() {
   String filename = "/WEB-INF/data/predictions.db";
   InputStream in = sctx.getResourceAsStream(filename);
   ...
}
```

If the `predictions` reference is `null`, then populating the `Map` must occur. The `Map`, in turn, contains references to `Prediction` objects built from data in the *predictions.db* file. Finally, the `toXML` method serializes the Java predictions into XML using an `XMLEncoder`:

```
private String toXML() {
   String xml = null;
   try {
      ByteArrayOutputStream out = new ByteArrayOutputStream();
      XMLEncoder encoder = new XMLEncoder(out);
      encoder.writeObject(predictions); // serialize to XML
      encoder.close();
      xml = out.toString(); // stringify
   }
   catch(Exception e) { }
   return xml;
}
```

The XML document from the `toXML` method becomes the body of the HTTP response to the client.

Example 1.5 The `cliches.Prediction` class

```
package cliches;

import java.io.Serializable;

// An array of Predictions is to be serialized
// into an XML document, which is returned to
// the consumer on a request.
public class Prediction implements Serializable {
    private String who;   // person
    private String what;  // his/her prediction

    public Prediction() { }

    public void setWho(String who) {
        this.who = who;
    }
    public String getWho() {
        return this.who;
    }

    public void setWhat(String what) {
        this.what = what;
    }
    public String getWhat() {
        return this.what;
    }
}
```

Example 1.6 The `cliches.Predictions` class

```
package cliches;

import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.ByteArrayOutputStream;
import java.beans.XMLEncoder; // simple and effective
import javax.servlet.ServletContext;

public class Predictions {
    private int n = 32;
    private Prediction[ ] predictions;
    private ServletContext sctx;

    public Predictions() { }

    //** properties

    // The ServletContext is required to read the data from
    // a text file packaged inside the WAR file
    public void setServletContext(ServletContext sctx) {
        this.sctx = sctx;
    }
    public ServletContext getServletContext() { return this.sctx; }

    // getPredictions returns an XML representation of
    // the Predictions array
    public void setPredictions(String ps) { } // no-op
    public String getPredictions() {
        // Has the ServletContext been set?
        if (null == getServletContext())
            return null;

        // Have the data been read already?
        if (null == predictions)
            populate();

        // Convert the Predictions array into an XML document
        return toXML();
    }

    //** utilities
    private void populate() {
        String filename = "/WEB-INF/data/predictions.db";
        InputStream in = sctx.getResourceAsStream(filename);

        // Read the data into the array of Predictions.
        if (in != null) {
            try {
                InputStreamReader isr = new InputStreamReader(in);
                BufferedReader reader = new BufferedReader(isr);

                predictions = new Prediction[n];
                int i = 0;
                String record = null;
                while ((record = reader.readLine()) != null) {
                    String[] parts = record.split("!");
                    Prediction p = new Prediction();
                    p.setWho(parts[0]);
                    p.setWhat(parts[1]);

                    predictions[i++] = p;
                }
            }
            catch (IOException e) { }
        }
    }
```

Although the XML from the *sayings* service is generated using the standard `XMLEncoder` class, Java does provide other ways to generate XML—but none quite as simple as `XMLEncoder`. The `Prediction` objects must be serializable in order to be encoded as XML using the `XMLEncoder`; hence, the `Prediction` class implements the empty (or *marker*) `Serializable` interface and defines the *get/set* methods for the properties `who` (the predictor) and `what` (the prediction).

The predictions service can be deployed under the Tomcat web server using a provided Ant script (with `%` as the command-line prompt):

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

The first sidebar elaborates on the Apache Tomcat server and explains how to install and use this server. The second sidebar clarifies the Ant script, which is packaged with the book's code examples. The deployed WAR file *cliches.war* includes a standard web deployment document, *web.xml*, so that the URI */cliches/sayings.jsp* can be shortened to */cliches/*.

Installing and using the Tomcat web server

Apache Tomcat (*http://tomcat.apache.org/*) is a commercial-grade yet lightweight web server implemented in Java. Tomcat has various subsystems for administration, security, logging, and trouble-shooting but its central subsystem is Catalina, a container that executes servlets, including JSP and other scripts (*e.g.*, JSF scripts) that Tomcat automatically translates into servlets. Tomcat also includes a web console, tutorials, and sample code. This note focuses on installing Tomcat and on basic post-installation tasks such as starting and stopping the web server. The current version is 7.x, which requires Java SE 6 or higher. Earlier Tomcat versions are still available.

There are different ways to download Tomcat, including as a ZIP. Tomcat can be installed in any directory, for example, in */usr/local/tomcat* on a Unixy system or in *D:\tomcatWS* on a Windows system. For convenience, let *TOMCAT_HOME* be the install directory. The directory *TOMCAT_HOME/bin* has startup and showndown scripts for Unixy and Windows systems. For instance, the startup script is *startup.sh* for Unix and *startup.bat* for Windows. Tomcat is written in Java but does not ship with the Java runtime; instead, Tomcat uses the Java runtime on the host system. To that end, Tomcat requires that the environment variable *JAVA_HOME* be set to the Java install directory (*e.g.*, to */usr/local/java7*, *D:\java7*, and the like). In summary, the key commands (with comments introduced with two semicolons) are (with `%` as the command-line prompt):

```
% startup.sh   ;; or startup.bat on Windows to start Tomcat
% shutdown.sh  ;; or shutdown.bat on Windows to stop Tomcat
```

The commands can be given at a command-line prompt. On startup, a message similar to

```
Using CATALINA_BASE:   /home/mkalin/tomcat7
Using CATALINA_HOME:   /home/mkalin/tomcat7
Using CATALINA_TMPDIR: /home/mkalin/tomcat7/temp
Using JRE_HOME:        /usr/local/java
Using CLASSPATH:       /home/mkalin/tomcat7/bin/bootstrap.jar
```

should appear.

Under *TOMCAT_HOME* there is directory named *logs*, which contains various log files, and several other directories, some of which will be clarified later. A important directory for now is *TOMCAT_HOME/webapps*, which holds JAR files with a *.war* extension (hence the name WAR file). Subdirectories under *TOMCAT_HOME/webapps* can be added as needed. Deploying a web service under Tomcat is the same as deploying a web site: a WAR file containing the site or the service is copied to *TOMCAT_HOME/webapps*; and a web site or web service is undeployed by removing its WAR file.

Tomcat maintains various log files in *TOMCAT_HOME/logs*, one of which is especially convenient for *ad hoc* debugging. Tomcat automatically redirects output to `System.err` to *TOMCAT_HOME/logs/catalina.out*. Accordingly, if a servlet executes

```
System.err.println("Goodbye, cruel world!");
```

the farewell message would appear in the *catalina.out* log file.

Apache Tomcat is not the only game in town. There is the related TomEE web server, basically Tomcat with support for Java EE beyond servlets. Another popular Java-centric web server is Jetty (*http://jetty.codehaus.org*). My first example uses Tomcat but later examples use Jetty as well; and the next chapter explains how to install and run Jetty.

An Ant script to automate Tomcat deployment

The first sample web service is deployed to a web server such as Tomcat. The ZIP file with my code examples includes an Ant script to ease the task of deployment. The Ant utility, written in Java, is available on all platforms. My script requires Ant 1.6 or greater.

Consider a web service that includes a JSP script, a back-end JavaBean, the Tomcat deployment file *web.xml*, and a JAR file that holds a JSON library. These artifacts reside in any directory on the local file system, hereafter the *current working directory* or *cwd* for short. The Ant file *build.xml* is in the *cwd*. Under the *cwd* is a subdirectory named *src*

```
cwd: build.xml
 |
src
```

that holds the JSP script, the JAR file, and the deployment file:

```
cwd: build.xml
 |
src: products.jsp, json.jar, web.xml
```

Suppose that the back-end JavaBean has the fully qualified name `acme.Products`. The layout is now

```
cwd: build.xml
 |
src: products.jsp, json.jar, web.xml
 |
acme: Products.java
```

Finally, assume that the *src* directory also holds the data file *new_products.db*. From the *cwd* command-line, the command

```
% ant -Dwar.name=products.war deploy
```

does the following:

- Creates the directory *service1/build*, which holds copies of files in directory *src* and any subdirectories.
- Compiles any *.java* files, in this case *acme.org.Products.java*.
- Builds the WAR file, whose major contents are:

  ```
  WEB-INF/web.xml
  WEB-INF/classes/acme/org/Products.class
  WEB-INF/data/new_products.db
  WEB-INF/lib/json.jar
  acme/org/Products.java
  products.jsp
  ```

Any *.xml* file winds up in *WEB-INF*; any *.jar* file winds up in *WEB-INF/lib*; and any *.db* file winds up in *WEB-INF/data*. JSP files such as *products.jsp* are at the WAR file's top level.

- Copies the WAR file to *TOMCAT_HOME/webapps* and thereby deploys the web service.
- Leaves a copy of the WAR file in the *cwd*.

The Ant file *build.xml* has extensive documentation and explains, in particular, what needs to be done to customize this file for your environment. Although the Ant script is targeted at Tomcat, the WAR files that it produces can be deployed to Jetty as well. Chapter 2 goes into the details of installing and running Jetty.

1.9.2 A Client against the *sayings* Web Service

Later examples introduce RESTful clients in Java and other languages; but, for now, either a browser or a utility such as *curl* is good enough. (The *curl* utility is available on Unixy systems and a port for Windows can be found at from *http://curl.haxx.se/download.h*
On a successful *curl* request to the service

```
% curl -v http://localhost:8080/cliches/
```

the response includes not only the XML shown earlier in Example 3 but also a trace (thanks to the *-v* flag) of the HTTP request and response messages. The HTTP request is

```
GET /cliches/ HTTP/1.1
User-Agent: curl/7.19.7
Host: localhost:8080
Accept: */*
```

and the HTTP response start line and headers are

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Set-Cookie: JSESSIONID=96C78773C190884EDE76C714728164EC; Path=/test1/;
Content-Type: text/html;charset=ISO-8859-1
Transfer-Encoding: chunked
```

Recall that an HTTP GET message has no body; hence, the entire message is the start line and the headers. The response shows the session identifier (a 128-bit statistically unique number, in hex, that Tomcat generates) in the header. In the JSP script, the session identifier could be disabled as it is not needed; but, for now, the goal is brevity and simplicity.

If a POST request were sent to the RESTful predictions service

```
% curl --request POST --data "foo=bar" http://localhost:8080/cliches/
```

the request message header becomes

```
POST /cliches/ HTTP/1.1
User-Agent: curl/7.19.7
Host: localhost:8080
Accept: */*
Content-Length: 7
Content-Type: application/x-www-form-urlencoded
```

and the response header is

```
HTTP/1.1 405 Method Not Allowed
Server: Apache-Coyote/1.1
Set-Cookie: JSESSIONID=34A013CDC5A9F9F8995A28E30CF31332; Path=/test1/;
Content-Type: text/html;charset=ISO-8859-1
Content-Length: 1037
```

The error message

```
GET requests only are allowed
```

is in an HTML document that makes up the response message's body. Tomcat generates an HTML response because my code does not but could stipulate a format other than the Tomcat default format, HTML.

This first example illustrates how a JSP script is readily adapted to support web services in addition to web sites. The next section goes into more detail on servlets and JSP scripts. In summary, the predictions web service, which is implemented as the JSP script *sayings.jsp* and the two back-end Java classes +Prediction' and *Predictions*, highlights key aspects of a REST:

- The service provides access to resource under a standard name, the URI `/cliches/sayings.jsp` or, in abbreviation, `/cliches/`.

- The service filters access on the HTTP request verb. In this example, only GET requests are successful; any other type of request generates a *bad method* error.
- The service responds with an XML payload, which the consumer now must process in some appropriate way. This first example merely displays the XML without any further processing.

1.10 Why Use Servlets and JSP Scripts to Implement RESTful Web Services?

Chapter 2 explores various ways in which to implement and publish RESTful services. The current chapter introduces a tried-and-true way to do RESTful services in Java: the service is implemented as a servlet and published with a lightweight, Java-based web server such as Apache Tomcat.

An `HttpServlet` is a natural, convenient way to implement RESTful web services for two main reasons. First, such servlets are close to the HTTP metal. For example, an `HttpServlet` provides methods such as `doGet`, `doPost`, `doPut`, and `doDelete` that match up with the HTTP verbs aligned with the CRUD operations. These servlet methods execute as call-backs that the servlet container, explained shortly, invokes as needed. The `HttpServlet` class also provides symbolic constants for HTTP status codes, for example, `SC_NOT_FOUND` for status code 404 and `SC_METHOD_NOT_ALLOWED` for status code 405. Each of the `HttpServlet` *do*-methods take the same two arguments: an `HttpServletRequest` and an `HttpServletResponse`. The servlet request contains, as key/value pairs, all of the appropriate information encapsulated in the HTTP request. The `HttpServletRequest` map is easy to read and, if needed, to update and to forward. The `HttpServletResponse` has methods to adjust the HTTP response message as needed; and this class encapsulates an output stream to communicate back to the client.

A second major advantage of servlets is that they execute in a servlet container, middleware that mediates between the application code of the servlet and the web server that provides the usual types of support: wire-level security in the form of HTTPS transport, user authentication and authorization, logging and troubleshooting support, server configuration, local or remote database access, naming services, application deployment and administration, and so on. In the Tomcat web server, the servlet container is named Catalina. Because the servlet container is such an integral part of a Java-based web server, it is common to conflate the container name (Catalina) and the server name (Tomcat), a practice followed here. In any case, a Java-centric web server such as Tomcat is the natural way to publish real-world web services, including RESTful ones. Figure x.x depicts a servlet container with several instances of executing servlets, each awaiting client requests.

Here is a short, more technical review of servlets with emphasis on their use to deliver RESTful services. The class `HttpServlet` extends the class `GenericServlet`, which in turn implements the `Servlet` interface. All three are in the package `javax.servlet` which is not included in core Java. The `Servlet` interface declares five methods, the most important of which is the service method that a web container invokes on every request to a servlet. The service method has a `ServletRequest` and a `ServletResponse` parameter. The request is a map that contains the request information from a client and the response provides a network connection back to the client. The `GenericServlet` class implements the `Service` methods in a transport-neutral fashion, whereas its `HttpServlet` subclass implements these methods in an HTTP-specific way. Accordingly, the service parameters in the `HttpServlet` have the types `HttpServletRequest` and `HttpServletResponse`. The `HttpServlet` also provides request filtering that naturally supports a REST-style service: the service method dispatches a incoming GET request to the method `doGet` , an incoming POST request to the method `doPost`, and so on.

In the `HttpServlet` class, the *do*-methods are defined as no-ops (that is, as methods with empty bodies) that can be overridden as needed in a programmer-derived subclass. For example, if the class `MyServlet` extends `HttpServlet` and overrides `doGet` but not `doPost` , then `doPost` remains a no-op in `MyServlet` instances.

JSP scripts are an arbitrary mix of HTML and code. In the case of web services, these scripts would consist predominantly and, for the most part, exclusively of code. The advantage of a JSP script over an `HttpServlet` is that the programmer does not need to compile a JSP script. The web container assumes this responsibility. A JSP script is deployed as a text file but executes as a servlet because the web container automatically translates the script into an `HttpServlet` before loading one or more instances of the resulting servlet into the container. For short examples and for the kind of experimentation typical of code development, JSP scripts are attractive. For deployment to production, the straight Java code of a servlet would be best practice. My examples use a mix of JSP scripts and servlets.

1.11 What is Next?

RESTful services are rich enough to warrant two chapters on the basics. Accordingly, the next chapter focuses on the service side by exploring options for implementing and publishing RESTful services. The options include

- explicit servlets and JSP scripts published with a web server such as Tomcat.
- JAX-WS `WebServiceProvider` instances published with a web server such as Tomcat or the handy Java `Endpoint` publishing class.
- JAX-RS annotated resources published with a web server such as Tomcat or a lightweight container such as Grizzly.

Chapter 3 then changes the focus to the client or consumer side. The chapter includes client code against commercial RESTful services such as Amazon, Twitter, and Tumblr together with a discussion about how the JAX-B (Java API for XML-Binding) packages can be put to good use by the hiding the XML in the consumption of RESTful services.