Chapter 2

Examples of Web Services

Amazon and Google

Real-world Web services will be examined in this chapter. This section is a bit more technical than other parts of the report, but nontechnical readers should not be intimidated by the programming code listed in the examples. Even nonprogrammers should be able to understand the gist of what these programs are trying to accomplish without having to have a keen grasp on the programming language. (I ask that programmers reading this report to bear with my unsophisticated programming style; more experienced programmers would write much more elegant code than my admittedly awkward efforts.)

Amazon Web Services

Amazon, one of the most popular commercial online businesses, offers a Web-service interface that provides a number of interesting features. The possibilities range from simple queries out of the Amazon catalogs to full-fledged e-commerce Web sites that operate in partnership with Amazon through the company's Amazon affiliates program.

Amazon.com has grown to be one of the most successful businesses on the Web. The company quickly established itself as a popular source for buying books on the Web and has expanded to sell many other types of products. In support of its extremely high-volume business, Amazon has developed a sophisticated, scaleable, and reliable technology infrastructure. On its own, Amazon is extremely successful, but an important part of its business strategy involves the company's efforts to go beyond its direct sales and to include affiliates and partners. For example, Amazon allows other individuals and organizations to make use of its infrastructure to sell their products and services in return for it receiving a small commission on each sale. This arrangement has proven to be beneficial for both Amazon and its affiliates. An individual or business can fairly easily set up his, her, or its own electronic storefront; this storefront features the e-commerce capabilities developed by Amazon, which would be much more difficult and expensive to create independently.

The mechanism that Amazon uses for expanding its online-business activities is based on Web services. Using Web-service technologies described in this report, Amazon Web Services (AWS) provide access to Amazon's technical infrastructure. AWS can be implemented using either SOAP or REST, but the majority of AWS implementations follow the simpler REST approach.

In order to begin using AWS, one must first set up an AWS developer account. Once registered, the user will receive a subscription ID, which will be used as a key to gain access when invoking Web services. Many of the Web services offered by Amazon can be used for free, although some require paid subscriptions or involve pay-per-use fees.

The core functionality of Amazon.com can be programmatically accessed through the Amazon E-Commerce Service (ECS). Amazon does not charge for use of ECS, though some restrictions apply regarding how information retrieved from Amazon in this way may be used and the number of transactions performed per second. (The WSDL [Web Services Description Language] that describes the ECS is available at: http://webservices.amazon.com/AWSECommerceService/AWSECommerceService.wsdl.)

Amazon offers extensive tools and documentation for building applications using AWS (http://aws.amazon.com). There are also a number of books and other resources available that provide extensive information on implementing applications based on AWS.

In the following text, I'll construct a very simple example that makes use of the Amazon Web Service (see

appendix 3). The example will explain how the AWS functions to dynamically execute a search on Amazon—with a few lines of programming—to display the results.

Amazon E-Commerce Service
http://webservices.amazon.com/AWSECommerceService/AWSECommerceService.wsdl

Amazon Web Services
http://aws.amazon.com

This, and most of the other examples in this report, will use the Perl programming language, for no other reason than it is the one with which I am most comfortable. The platform for the examples consists of the following components:

- An Intel-based server running Windows XP Professional;
- The Perl programming language. (I will use ActivePerl [available from ActiveState Software, Inc., www.activestate.com] ActiveState makes ActivePerl freely available and sells professional programming products and services);
- The SOAP::Lite Perl module; and
- The Internet Information Server (IIS) v5.0. This Web server comes as an optional component of Windows XP Professional.

A good resource for guidance in developing applications with AWS is *Amazon Hacks* by Paul Bausch. The following example is loosely based on a program, "Hack88: Program AWS with SOAP::Lite and Perl," in the book.

Building a Web-Service Client

Appendix 3 illustrates the Perl script that implements a Web-service client, a SOAP interface, which is used to send a service request to AWS, to perform an author query, parse the service response, and display selected elements on a Web page.

This script resides in a directory on the server that's configured to execute CGI-style programs. The value of the author to search is passed on as a parameter in the URL: www.family-photos.info/amazon-web-service.pl?Author=Marshall%20Breeding. When a Web browser invokes this script, it produces the Web page shown in appendix 4.

Breaking it down into steps may also help nontechnical individuals understand the general components.

First, appendix 5 illustrates the code that "calls in" some subroutines, which provide some general utility functions available for setting up Web pages and user sessions on a server. One of the subroutines available, *&pagetop*, issues an HTML header and begins a Web page.

The next section shows how the code sets things up to begin using SOAP, which will issue the request and receive the response. Perl::Lite supports WSDL, so it simply points to the *AmazonWebService.wsdl* file that defines the service and lets it do all the work of informing the client about the data types, messages, methods, and bindings that comprise the service.

Appendix 6 shows how the client will send a query to AWS to search by author. In this figure, a variable—*$query*—is set up to pass the value of the query. The *diglib-common.pl* file (see appendix 3) also contains the code to parse parameters on the command line and puts them into an associative array named *fields*. In the example in appendix 6, *$fields{'Author'}* will take the value "Marshall Breeding" as specified in the *Author=Marshall Breeding* component of the URL.

The next portion of the script, shown in appendix 7, does the heavy lifting. It invokes the method *AuthorRequest* defined in the WSDL and passes on the appropriate parameters.

The AuthorRequest method in the Perl script corresponds to the way the AuthorRequest data type has been defined in the WSDL (appendix 8).

Once the service has been invoked and a response is received, that data returned can be parsed and will display the results, as implemented in the lines of Perl illustrated in appendix 9.

This simple example illustrates how one can draw information from a given resource and display that information in an interface. Although Amazon.com certainly looks much better than the simplistic interface demonstrated, the point is that the user is now able to use the content returned in whatever form his or her application might need. The script in appendix 3 builds a Web page that displays content from just the one source, but a more interesting example might involve using Web services from multiple sources. This is often called a *mashup*.

The GoogleSearch API

Google also provides a SOAP-based API for accessing its resources in a model for Web services. The Google API can be used to programmatically access several different services, including executing a search on Google and receiving the results, requesting a spelling suggestion, and fetching a cached page.

Just like Amazon, in order to use the Google Web API, you must create a Google account and receive a key that is passed with each request. Google provides information on its APIs at www.google.com/apis/index.html. The WSDL file that describes the Google Web API can be displayed with this URL: http://api.google.com/GoogleSearch.wsdl.

Google Web API
www.google.com/apis/index.html

WSDL File Describing Google Web API
http://api.google.com/GoogleSearch.wsdl

ActiveState offers the Perl module *GoogleSearch .pm*, which implements a SOAP interface to the methods available in the Google API. Appendix 10 is based on the ActiveState GoogleSearch.pm module. The GoogleSearch .pm does not use the WSDL; rather the methods and data structures known to exist in the API have been hard coded into the module, yet there are many similarities to the Amazon search examples.

When invoked, the script creates a Web page such as the one shown in appendix 11. Again, this demonstrates how a user can programmatically access a remote service using a Web service, obtain content from that service, and display the content in the way the user desires.

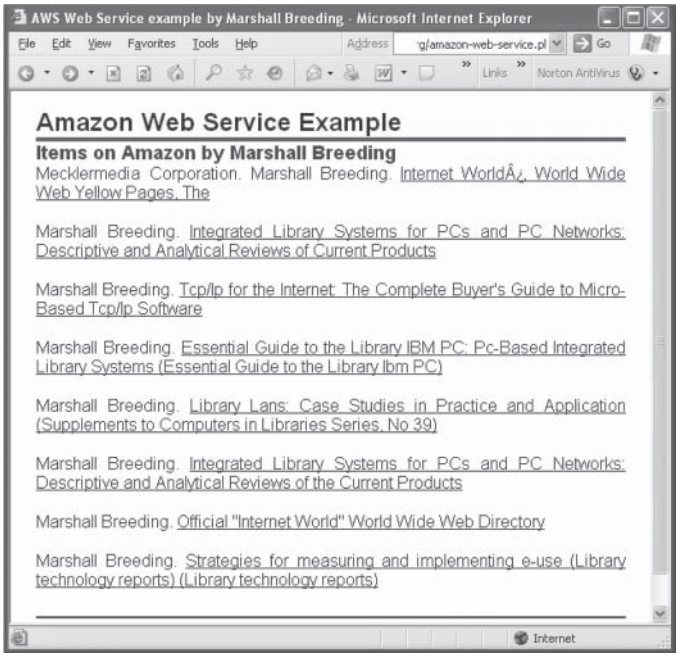
The same results can be accomplished without the GoogleSearch.pm Perl module. Instead, the user would simply use SOAP::Lite directly and link it to the GoogleSearch.wsdl. The alternate version of the script is shown in appendix 12.

Appendix 3:
SOAP Interface Example

```
require “diglib-common.pl”;
&pagetop;
print “<h1>Amazon Web Service Example</h1>\n”;
# Amazon developer’s token
my $dev_token=’xxxxx put your Amazon developer token here xxxxx’;

#Location of the Amazon WSDL file
my $amazon_wdsl = “http://soap.amazon.com/schemas2/AmazonWebServices.wsdl”;

use strict;
use SOAP::Lite;

my $query = “Marshall Breeding”;

#Construct a new SOAP::Lite instance
my $search = SOAP::Lite->service(“$amazon_wdsl”);

# Execute Amazon Web Service
my $results = $search ->
  AuthorSearchRequest(SOAP::Data->name(“AuthorSearchRequest”)
    ->type(“AuthorRequest”)
    ->value(\SOAP::Data->value(
    SOAP::Data->name(“author” => $query),
    SOAP::Data->name(“page” => “1”),
    SOAP::Data->name(“mode” => “books”),
    SOAP::Data->name(“type” => “lite”),
    SOAP::Data->name(“devtag” => $dev_token)
    ))
  );
print “<h2>Items on Amazon by Marshall Breeding</h2>\n”;
# Display results
foreach my $result (@{$results->{Details}}){
  #print each entry of the result set
  print “<p>\n”;
  foreach my $auth (@{$result->{Authors}}) {
    print $auth,”. “;
  }
```

```
  print "<a href=\"",$result->{Url},"\">" if (length($result->{Url}) > 0);
  print "$result->{ProductName}";
  print "</a>" if (length($result->{Url}) > 0);
  print "\Publisher: ",$result->{Publisher},"\n" if (length($result->{Publisher}) > 0);
  print "\nPrice on Amazon: ",$result->{OurPrice},"\n" if (length($result->{OurPrice}) > 0);
  print "</p>\n";
}
&pagebottom;
```

Appendix 4:
AWS Web Service Example



Appendix 5:
Code That "Calls in" Subroutine for HTML Header and Beginning of Web Page

```
require "diglib-common.pl";
&pagetop;
print "<h1>Amazon Web Service Example</h1>\n";
```

Appendix 6:
Script Utilized for Client to Send Query to AWS to Search by Author

```
my $amazon_wdsl = "http://soap.amazon.com/schemas2/AmazonWebServices.wsdl";
use strict;
use SOAP::Lite;

my $query = $fields{'Author'};

#Construct a new SOAP::Lite instance
my $search = SOAP::Lite->service("$amazon_wdsl");
```

Appendix 7:
Script That Invokes AuthorRequest, Defined in WSDL

```
# Execute Amazon Web Service
my $results = $search ->
  AuthorSearchRequest(SOAP::Data->name("AuthorSearchRequest")
    ->type("AuthorRequest")
    ->value(\SOAP::Data->value(
    SOAP::Data->name("author" => $query),
    SOAP::Data->name("page" => "1"),
    SOAP::Data->name("mode" => "books"),
    SOAP::Data->name("type" => "lite"),
    SOAP::Data->name("devtag" => $dev_token)
    ))
  );
```

Appendix 8:
AuthorRequest Method in Perl Script Corresponds to AuthorRequest Data Type (in WSDL)

```
<xsd:complexType name="AuthorRequest">
   <xsd:all>
     <xsd:element name="author" type="xsd:string"/>
     <xsd:element name="page" type="xsd:string"/>
     <xsd:element name="mode" type="xsd:string"/>
     <xsd:element name="tag" type="xsd:string"/>
     <xsd:element name="type" type="xsd:string"/>
     <xsd:element name="devtag" type="xsd:string"/>
     <xsd:element name="sort" type="xsd:string" minOccurs="0"/>
     <xsd:element name="variations" type="xsd:string" minOccurs="0"/>
     <xsd:element name="locale" type="xsd:string" minOccurs="0"/>
   </xsd:all>
</xsd:complexType>
```

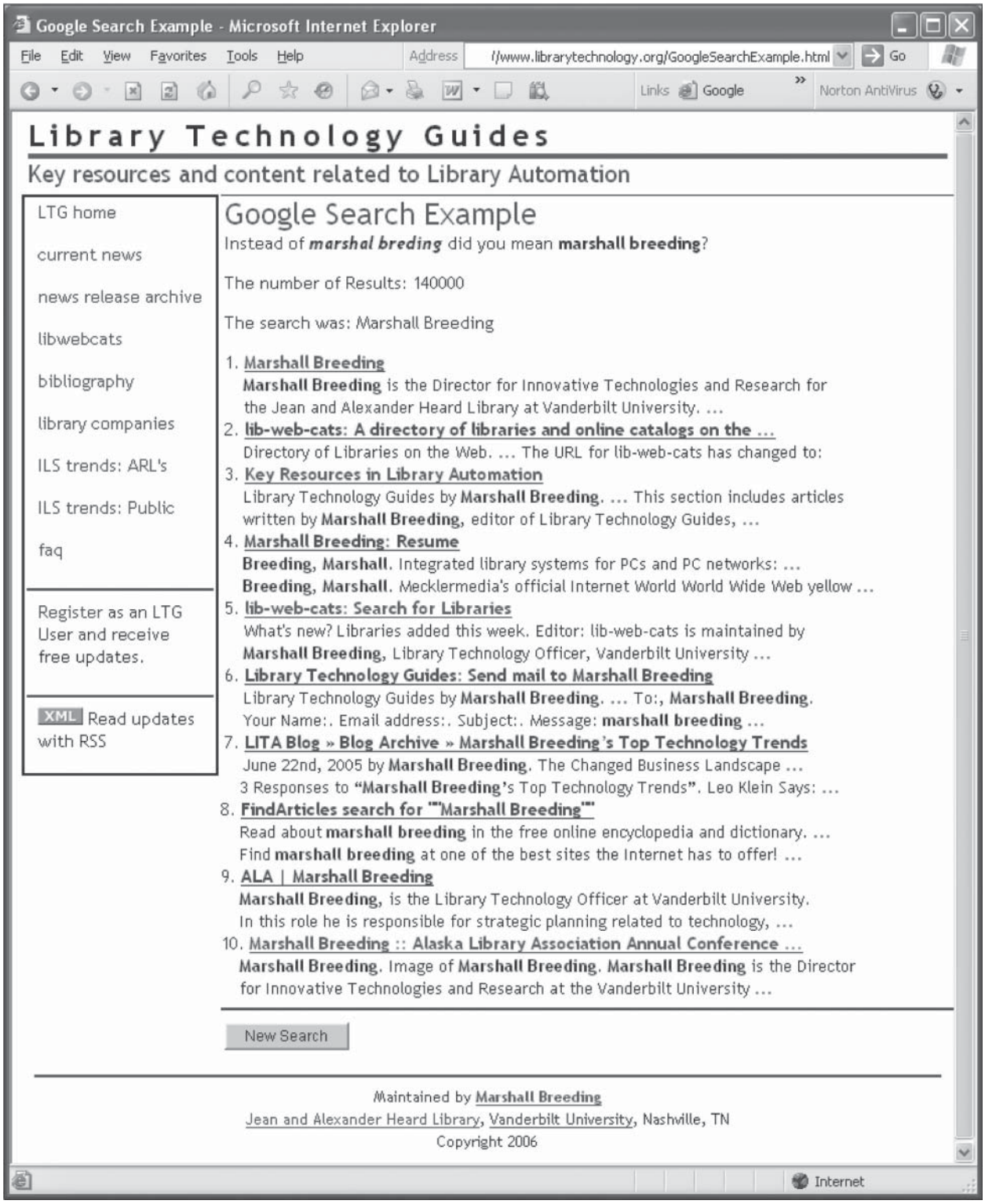
Appendix 9:
Display Results

```
print "<h2>Items on Amazon by Marshall Breeding</h2>\n";
# Display results
foreach my $result (@{$results->{Details}}){
  #print each entry of the result set
  print "<p>\n";
  foreach my $auth (@{$result->{Authors}}) {
    print $auth,". ";
  }
  print "<a href=\"",$result->{Url},"\">" if (length($result->{Url}) > 0);
  print "$result->{ProductName}";
  print "</a>" if (length($result->{Url}) > 0);
  print "\Publisher: ",$result->{Publisher},"\n" if (length($result->{Publisher}) > 0);
  print "\nPrice on Amazon: ",$result->{OurPrice},"\n" if (length($result->{OurPrice}) > 0);
  print "</p>\n";
}
&pagebottom;
```

Appendix 10:
Script Based on ActiveState GoogleSearch.pm Module

```
require (“interface.pl”);

&pagetop(“Google Search Example”); # begins HTML page
print “<h2> Google Search Example </h2>\n”;

use GoogleSearch;

# if key is not provided, GoogleSearchService looks in $ENV{HOME},
# or in the location of GoogleSearchService.pm for googlekey.txt
my $key = ‘xxxxx put your Google key here xxxxxx’;
my $google = new GoogleSearch();
my $return;


# example of using spelling suggestion API
my $test = ‘marshal breding’;
$return = $google->doSpellingSuggestion($test)->result();
print “<p>Instead of <strong><em>$test</em></strong> did you mean <strong>$return</strong>?</p>\n”;

# example of a search
my $search = $fields{‘query’};
$return = $google->doGoogleSearch(
  query => $search,
  start => 0,
  maxResults => 10,
  restrict => ‘xml’,
)->result();

my $i = 0;
#my $resultdata;
#$resultdata->{‘GoogleSearchResults’};
print “<p>The number of Results: $return->{‘estimatedTotalResultsCount’}</p>\n”;
print “<p>The search was: $return->{‘searchQuery’}</p>\n”;

foreach my $entry (@{$return->{‘resultElements’}}) {
  $i++;
  print “<p style=\”margin-bottom: 0\”>$i. \n “;

  print “<a href=\”$entry->{URL}\”><strong>$entry->{title}</strong></a></p>\n”;
  print “<p class=\”details\”>$entry->{snippet}</p>\n”;
}
&pagebottom; # ends HTML page
```

Appendix 11:
Web Page Displayed by the Embedded Google Search Client



Appendix 12:
SOAP::Lite Client Alternate Script That Yields Web Page Shown in Appendix 11

```
require (“interface.pl”);
use SOAP::Lite;

&pagetop(“Google Search Example”); # begins HTML page
print “<h2> Google Search Example </h2>\n”;


my $key = ‘xxxxx put your Google key here xxxxxx’;
my $q=”Marshall Breeding”;
my $googleSearch = SOAP::Lite -> service(“http://api.google.com/GoogleSearch.wsdl”);

# example of using spelling suggestion API
my $check = ‘marshal breding’;
my $suggestion = $googleSearch->doSpellingSuggestion($key, $check);
print “<p>Instead of <strong><em>$check</em></strong> did you mean <strong>$suggestion</strong>?</p>\n”;

my $results = $googleSearch -> doGoogleSearch($key, $q, 0, 10, “false”, “”, “false”, “”, “latin1”, “latin1”);
print “About $results->{‘estimatedTotalResultsCount’} results.\n”;
my $i = 0;
foreach my $result (@{$results->{resultElements}}) {
  $i++;
  print “<p style=\”margin-bottom: 0\”>$i. \n “;
  print “<a href=\””,$result->{URL},”\”>”,$result->{title},”</a>.”;
  $summary = $result->{snippet};
  print “<p>”,$summary,”\n”;
  print “</p>\n”;
}
&pagebottom;
```