PHP Web 2.0 Mashup Projects

Create practical mashups in PHP, grabbing and mixing data from Google Maps, Flickr, Amazon, YouTube, MSN Search, Yahoo!, Last.fm, and 411Sync.com

Shu-Wai Chow

PACKT PUBLISHING

BIRMINGHAM - MUMBAI

PHP Web 2.0 Mashup Projects

Create practical mashups in PHP, grabbing and mixing data from Google Maps, Flickr, Amazon, YouTube, MSN Search, Yahoo!, Last.fm, and 411Sync.com

Copyright © 2007 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, Packt Publishing, nor its dealers or distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: September, 2007

Production Reference: 1070907

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-847190-88-8

www.packtpub.com

Cover Image by Vinayak Chittar (vinayak.chittar@gmail.com)

Table of Contents

Preface

A mashup is a web page or application that combines data from two or more external online sources into an integrated experience. This book is your entryway to the world of mashups and Web 2.0. You will create PHP projects that grab data from one place on the Web, mix it up with relevant information from another place on the Web and present it in a single application. All the mashup applications used in the book are built upon free tools and are thoroughly explained. You will find all the source code used to build the mashups in the code download section on our website.

This book is a practical tutorial with five detailed and carefully explained case studies to build new and effective mashup applications.

What This Book Covers

You will learn how to write PHP code to remotely consume services like Google Maps, Flickr, Amazon, YouTube, MSN Search, Yahoo!, Last.fm, and the Internet UPC Database, not to mention the California Highway Patrol Traffic data! You will also learn about the technologies, data formats, and protocols needed to use these web services and APIs, and some of the freely-available PHP tools for working with them.

You will understand how these technologies work with each other and see how to use this information, in combination with your imagination, to build your own cutting-edge websites.

Chapter 1 provides an overview of mashups: what a mashup is, and why you would want one.

In *Chapter 2* we create a basic mashup, and go shopping. We will simply look up products on Amazon.com based on the Universal Product Code (UPC). To do this, we cover two basic web services to get our feet wet — XML-RPC and REST. The Internet UPC database is an XML-RPC-based service, while Amazon uses REST.

We will create code to call XML-RPC and REST services. Using PHP's SAX function, we create an extensible object-oriented parser for XML. The mashup covered in this chapter integrates information taken from Amazon's E-commerce Service (ECS) with the Internet UPC database.

In *Chapter 3*, we create a custom search engine using the technology of MSN, and Yahoo! The chapter starts with an introduction to SOAP, the most complex of the web service protocols. SOAP relies heavily on other standards like WSDL and XSD, which are also covered in readable detail. We take a look at a WSDL document and learn how to figure out what web services are available from it, and what types of data are passed. Using PHP 5's SoapClient extension, we then interact with SOAP servers to grab data. We then finally create our mashup, which gathers web search results sourced from Microsoft Live and Yahoo!

For the mashup in *Chapter 4*, we use the API from the video repository site YouTube, and the XML feeds from social music site Last.fm. We will take a look at three different XML-based file formats from those two sites: XSPF for song playlists, RSS for publishing frequently updated information, and YouTube's custom XML format. We will create a mashup that takes the songs in two Last.fm RSS feeds and queries YouTube to retrieve videos for those songs. Rather than creating our own XML-based parsers to parse the three formats, we have used parsers from PEAR, one for each of the three formats. Using these PEAR packages, we create an object-oriented abstraction of these formats, which can be consumed by our mashup application.

In *Chapter 5*, we screen-scrape from the California Highway Patrol website. The CHP maintains a website of traffic incidents. This site auto-refreshes every minute, ensuring the user gets live data about accidents throughout the state of California. This is very valuable if you are in front of a computer. If you are out and about running errands, it would be fairly useless. However, our mashup will use the web service from 411Sync.com to accept SMS messages from mobile users to deliver these traffic incidents to users.

We've thrown almost everything into *Chapter 6*! In this chapter, we use RDF documents, SPARQL, RAP, Google Maps, Flickr, AJAX, and JSON. We create a geographically-centric way to present pictures from Flickr on Google Maps. We see how to read RDF documents and how to extract data from them using SPARQL and RAP for RDF. This gets us the latitude and longitude of London tube stations. We display them on a Google Map, and retrieve pictures of a selected station from Flickr. Our application needs to communicate with the API servers for which we use AJAX and JSON, which is emerging as a major data format. The biggest pitfall in this AJAX application is race conditions, and we will learn various techniques to overcome these.

What You Need for This Book

To follow along with the projects and use the example code in this book, you will need a web server running PHP 5.0 or higher and Apache 1.3.

All of the examples assume you are running the web server on your local work station, and all development is done locally.

Additionally, two projects have special requirements. In Chapter 5, you will need access to a web server that can be reached externally from the Internet. In Chapter 6, you will need a MySQL server. Again, we assume you are running the MySQL server locally and it is properly configured.

To quickly install PHP, Apache, and MySQL, check out XAMPP (`http://www.apachefriends.org/en/xampp.html`). XAMPP is a one-step installer for PHP, Apache, and MySQL, among other things.

XAMPP is available for Windows, Linux, and Mac OS X. However, many standard Linux distributions already have PHP, Apache, and MySQL installed. Check your distribution's documentation on how to activate them. Mac OS X already has Apache and PHP installed by default. You can turn them on by enabling **Web Sharing** in your **Sharing Preferences**.

MySQL can be installed as a binary downloaded from MySQL.com (`http://dev.mysql.com/downloads/mysql/4.1.html`).

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

There are three styles for code. Code words in text are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code will be set as follows:

```
<?php
  $aDom = new DOMDocument();
  try {
     $aDom->loadHTMLFile('examplehtml.html');
  } catch (Exception $ex) {
     $aDom = false;
  }
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items will be made bold:

```
<param>
  <value><string>Hello, world!</string></value>
</param>
```

Any command-line input and output is written as follows:

```
Buttercup:~ root# pear list
```

`Buttercup:~ root#` is the shell prompt on the author's machine.

New terms and **important words** are introduced in a bold-type font. Words that you see on the screen, in menus or dialog boxes for example, appear in our text like this: "In the search box, enter in your keyword and the region code then press **Search**."

Important notes appear in a box like this.

Tips and tricks appear like this.

Reader Feedback

Feedback from our readers is always welcome. Let us know what you think about this book, what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply drop an email to `feedback@packtpub.com`, making sure to mention the book title in the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or email `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer Support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the Example Code for the Book

Visit `http://www.packtpub.com/support`, and select this book from the list of titles to download any example code or extra resources for this book. The files available for download will then be displayed.

The downloadable files contain instructions on how to use them.

Errata

Although we have taken every care to ensure the accuracy of our contents, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in text or code—we would be grateful if you would report this to us. By doing this you can save other readers from frustration, and help to improve subsequent versions of this book. If you find any errata, report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **Submit Errata** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata are added to the list of existing errata. The existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Questions

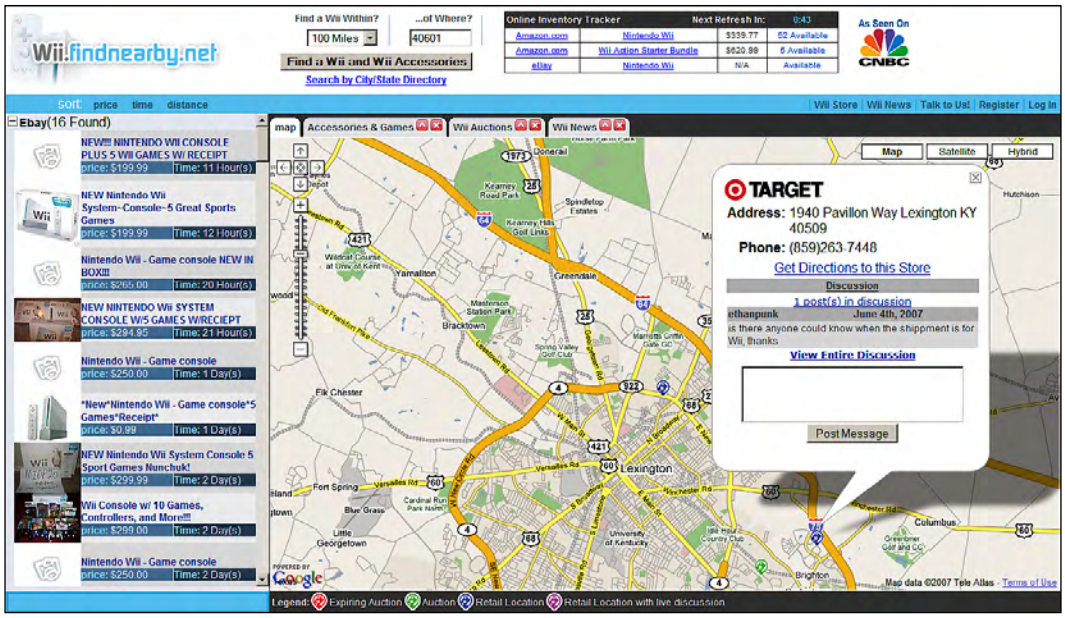
You can contact us at `questions@packtpub.com` if you are having a problem with some aspect of the book, and we will do our best to address it.

1
Introduction to Mashups

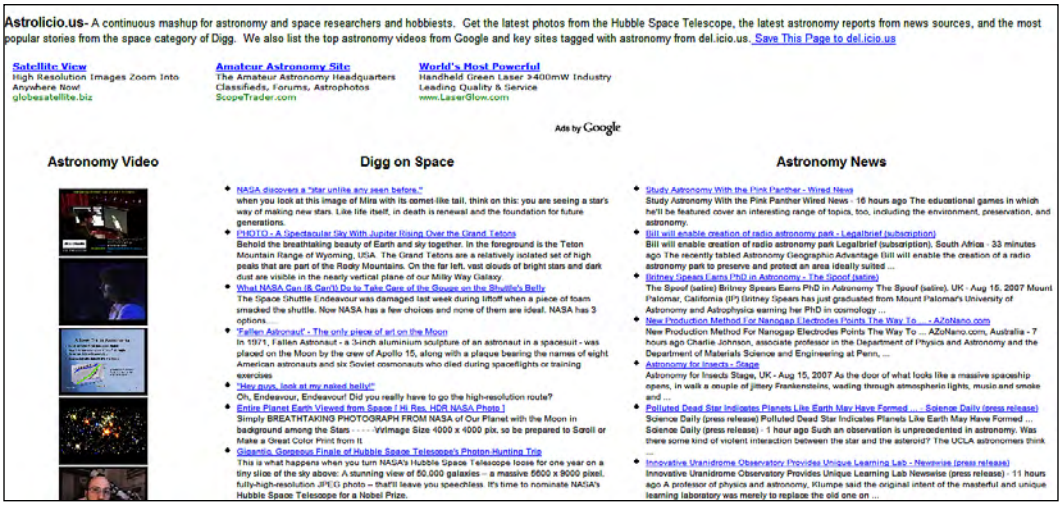
Mashups, more specifically called web application hybrids by Wikipedia, have been an exciting trend in web applications in recent years. Web mashups are exactly what they sound like—web applications that merge data from one or more sources and present them in new ways. Very often, the data owners encourage and facilitate third parties to use the data. In many cases, this facilitation is made possible by the data owners providing application programming interfaces (API) to their data. These APIs follow standard web service protocols and can be implemented quickly and easily in a variety of programming languages, including PHP. New, innovative mashups, made by individuals that combine data from traditionally unlikely pairings are popping up every day.

One example is the Wii Seeker site. When the Nintendo Wii launched in November 2006, many knew there would be shortages. The object of the Wii Seeker site is to help people find Wiis by combining expected initial shipment information to Target stores and Google Maps. A marker on a Google Map represented a Target retail store. If the user clicked on the marker they would see information about the store such as the address. They would also see the number of Wiis the store was expected to have on launch day. By representing numerical inventory data on a map, a user could see Target stores near their location and plan their store visits on launch day to maximize their chances of actually finding a Wii.

After the Nintendo Wii was launched, the site reinvented itself by adding auction information from eBay and product information from Amazon. They also added additional chain retail stores like Circuit City and Walmart. Instead of seeing Nintendo Wii inventory information on each store, the site now allows visitors to post notes for each other about the store's inventory.



Another mashup example is Astrolicio.us. This site queries data feeds from sites like Digg.com, Google News, and Google Videos and presents it to the user on one page. By combining data feeds, the site's creator has made a portal of current astronomy news for visitors.



On the homepage, the user can quickly scan items that may interest them. For news, the user is given bullet points for each news item containing the headline and a synopsis. For videos, the user is shown a thumbnail. If a user clicks on a link, they are taken to the source of the article or video. This site is clean, simple, and full of information. It is also quite easy to make using the APIs of the sources. It probably did not take the site creator more than an afternoon to go from the start of coding to launch.

Web 2.0 and Mashups

How, in just a few short years, have mashups suddenly sprung up everywhere? The story leads back to just a few years ago. After the technology industry's financial bubble collapsed in 2001, internet firms regrouped and redefined themselves. There were business lessons to be learned, technologies to be re-evaluated, and people's perceptions had changed. By the middle of the decade, many trends and differences became clear. The term "Web 2.0" started to surface, to draw separation between new sites and sites that gained popularity in the late Nineties. The term was vague and seemed suspiciously gimmicky at first. However, the differences between old and new were real. They were not just historical and chronological. Sites like Google, YouTube, and Flickr demonstrated new approaches to building a web business. These sites often had simple interfaces, fully embraced web services, and returned a lot of control to the user. Many of these sites relied solely on their users for content. In September 2005, technology publisher Tim O'Reilly wrote an article entitled *What Is Web 2.0* to succinctly declare the traits of Web 2.0 versus 1.0 sites. There were two characteristics that were direct catalysts for the growth of mashups:

- Importance of Data
- User Communities

Importance of Data

The first characteristic is the importance of data. The question of who owned data and what they choose to do with the data became a big issue. Why in the world would companies invest millions of dollars to gather their data and their database systems, but then freely give it away for others to use? The answer is by opening their systems, mashup developers help increase the reach of the data owners.

O'Reilly used the example of MapQuest to illustrate this. MapQuest was the leader in mapping in the mid to late nineties. However, their system was closed and did not allow outside parties to do anything with their data. In the early Aughts, mapping sites started to leverage this weakness. Yahoo! Maps, Microsoft Virtual Earth, and Google Maps entered the market, and each one had APIs. Despite the huge

early market lead, MapQuest quickly lost to bigger players with open data. There are many examples like this. Amazon opened up their data through the Amazon Ecommerce Service (ECS). Many mashups have used this web service to create their own store fronts. Amazon gets the sale and gives a percentage to mashup developers. This has created many more channels for Amazon to sell their goods besides `www.amazon.com`. Contrast this with a site like BarnesAndNoble.com which does not open their data. The only channel that they can sell is through the main website. Not only do they lose sales opportunities, but they lack the affiliate loyalty that Amazon has.

In our earlier examples, Wii Seeker helps the Target by funneling buyers to stores. Wii Seeker in turn, receives adverting revenue and affiliate commissions on their site. Google Videos, Google News, and Digg.com get visitors when a user clicks on a link from astrolicious.us. Astrolicious.us gets advertising revenue with very little development time invested.

User Communities

The second characteristic is that user added data is more valuable than we once thought. User product reviews on ecommerce sites are nothing new. Neither are web forums. However, it is how sites are using this information, and who owns the data, that is becoming important. Movie rental site Netflix has always allowed users to rate movies they have watched. Based on these recommendations, Netflix will suggest other movies you might like. Recently, they have added a new social networking feature called "Friends", where you can see how your friends have rated movies and what they are watching. One feature of Friends is compatibility ratings. Comparing both you and your friends' recommendations, Netflix comes up with a percentage of your shared movie tastes.

Other sites are completely dependent on user-added data. YouTube and Flickr provide video and picture hosting, respectively, for free. Their widespread adoption, though, is not simply from hosting. Before Flickr, there were many sites that hosted images for free. That was nothing new. The difference, again, is what both sites do with user-added data. Both sites provide social networking features. You can leave your ratings and comments on a hosted item and you can subscribe to a person's profile. Anytime that person uploads something, you will be notified of the new content. Both sites also allow folksonomic tagging, which basically lets uploaders describe the content with their own keywords. Visitors can use these keywords to search when they are looking for content. Tagging has proven to be an incredible aid for search algorithms.

Thus, it is these two characteristics of new sites that have allowed small web developers to appear much bigger. Backed with data from large internet presences, mashup developers create usage channels that data owners could not have foreseen, or been restricted by business rules.

How We Will Create Mashups

Technologically, the mashup phenomenon could not have happened without website owners making a clean separation between the data that is used on their sites, and the actual presentation of the data. This has always been a goal in computer application development, and therefore, it is no surprise that website and web application architecture have progressed towards this stage ever since the World Wide Web was created. This separation is quickly turning the World Wide Web into what is known as the *semantic web*—a philosophy where web content is presented not only for humans to read, but also in a way that can be easily processed by software and machines. We have moved from static pages to database-driven sites, from presentational `FONT` tags to cascading style sheets. It is perhaps inevitable that the web has become an environment that fosters mashup development.

Data sources of mashups are varied. Often, data owners provide mashup developers access to their data through official application programming interfaces. As we are talking about web applications, these APIs utilize web services, which come in a variety of protocols. Really Simple Syndication (RSS), a family of formats to present data, is another common data source that has helped spur the mashup adoption. When official methods are unavailable, developers become really creative in getting data. Screen scraping is a method that has always been around. Regardless of the method, mashups also deal with a variety of data formats. While mashups can be simple to create, a mashup developer must be flexible and well-rounded in the knowledge of their tools.

Open-source software is particularly well-suited in this mashup environment. The Apache and PHP combination makes for fast development. Being open source, developers are constantly and quickly adding new features to keep up with the web service world.

This book will take a look at how to use common data sources with PHP. Most official APIs are based on the big three web service protocols—XML-RPC, REST, and SOAP. We will of course look at these protocols. APIs and raw web service requests by hand, of course, are not the only way to retrieve data. We will look at using third-party libraries to interface with some popular sites. Feeds are also an important data source which we will use. By giving you a broad overview of the tools used in the mashup world, you should be able to start developing your own mashups quickly.

More Mashups

For more examples and inspirations, check out these popular mashups:

- Popurls (`popurls.com`)—Collects URLs from popular sites.
- Housingmaps.com (`www.housingmaps.com`)—Plots housing listings from Craigslist on to a map.
- Keegy (`us.keegy.com`)—A site that aggregates news from different sources and personalizes it for the reader.
- Alkemis (`local.alkemis.com`)—Aggregates and maps all sorts of data, for example, pictures and live web cams, in selected cities.
- Gametripping.com (`www.gametripping.com`)—A collection of satellite and Flickr photos of baseball stadiums.

2
Buy it on Amazon

Project Overview

What	Build an application that takes UPC symbols and looks them up on Amazon.com.
Protocols Used	XML-RPC, REST
Data Formats	XML-RPC, XML
Tools Featured	PHP's XML-RPC Functions and SAX Functions
APIs Used	Internet UPC Database, Amazon Web Services

We are going to start off with a relatively simple project. Our project will accept a **Universal Product Code** number from a user, look up the product information associated with the UPC number from the Internet UPC Database, and allow the user to buy the product from our site using Amazon.com. In other words, we are going to create an online store based on UPC numbers. By using Amazon.com's inventory, users can buy from Amazon.com, but they'll be able to do everything from our site alone. While such a site may not make us the next ecommerce king, it will introduce us to the two most basic web services—XML-RPC and REST. Each protocol will require us to structure our request in a certain way.

XML-RPC will return an XML document formatted to the XML-RPC specifications. REST responses are a lot more varied and in free form. They may be anything from a plain text string to huge, complex XML documents. Although most web services return a descriptive, well formed XML document, REST responses are not bound to any standard or specification. We will create utilities to process both XML-RPC and REST requests that we can use for the current and future mashups.

XML-RPC

As developers in today's world, we should be familiar with XML. On the surface, it is a group of data that is packaged and organized neatly into opening and closing tags. A deeper look tells us that this structure of XML makes it easy for machines to read and process. Thus, while XML can be hard on human eyes, we know it is designed for machine communication. However, without any sort of agreed structure of the XML document by the machines, the advantages of XML are effectively eliminated. This is where XML-RPC comes in.

RPC is an acronym for **Remote Procedure Call**; developed by **David Winer** of UserLand software in 1996. Its purpose is to allow applications, regardless of how different each program or the purpose of each program, to communicate with each other across a network in a standardized manner.

In computer terms, a procedure call is that which gets executed when the operating system communicates to the input devices about what you are doing: *Which key did you just hit on the keyboard? Where did you move the mouse? What did you just click on, with the mouse?*

XML-RPC carries this idea into the networking world (the "Remote" part of RPC) by creating a standard for one program to get information from another program across the network. Program A sends a remote procedure call to Program B. This call may include parameters that Program B needs to retrieve the data.

For example, if the query is against a list of people's name: *Do you want to retrieve a list of only those whose first name is "Peter"? Do you want to narrow your search down to a city?* The requests, and all its parameters, are formatted in a generic way that Program B understands. Regardless of the data type or size, Program B returns the answer back in a generic way that Program A understands. Program A can then do with the data whatever the user requested.

XML-RPC Structure

Two programs communicating across a network is obviously very different from an operating system talking with a mouse. An operating system has the advantage of super high speed internal buses and the ability to talk on a lower machine level. A procedure call using XML-RPC must be program neutral and friendly to the network transport protocol. It does not have the luxury of constantly polling the other machine hundreds of times per second. Thus, XML-RPC communication must accomplish its mission in the most efficient means possible in the lowest common denominator. This is accomplished by dividing calls into strictly formed XML requests and responses.

The Official Specifications

We are going to take a casual tour of an XML request and response call. For more formal details, you can read the official XML-RPC specifications at `http://www.xmlrpc.com/spec`

XML-RPC Request

XML-RPC requests function as HTTP `POST` requests. Therefore, it must have a proper HTTP `POST` header. The actual remote call and parameters, in XML format, follows the header as the body of the HTTP request.

```
POST /RPC2 HTTP/1.0
User-Agent: PHP5 XML-RPC Client (Mac OS X)
Host: betty.userland.com
Content-Type: text/xml
Content-length: 181
<?xml version="1.0"?>
<methodCall>
  <methodName>examples.getStateName</methodName>
  <params>
    <param>
      <value><int>42</int></value>
    </param>
  </params>
</methodCall>
```

The first line in the header, `POST` and the `Host` line tell us that this XML-RPC call is to a web service that sits at `betty.userland.com/RPC2`. The name of the call to be requested, in this case, `examples.getStateName`, is the first useful information in the message body. We pass an integer of `42` as the parameter to `examples.getStateName`. Let's take a look at these elements one by one:

The root element in an XML-RPC call is `methodCall`. It has one required child element, `methodName`, which specifies the name of the call to be requested. There can only be one `methodCall` per request. If parameters are passed to the call, they are encapsulated in the `params` element.

A procedure call can require an unlimited number of parameters. XML-RPC calls do not have named parameters. In other words, you do not name your parameter before assigning a value, for example:

```
// This is wrong.Parameters are not named.
<param name="myInt">
  <value><int>42</int></value>
</param>
```

Instead, for functions requiring more than one parameter, the correct parameter order is defined by the remote function. You will have to check the API's documentation for this information and make sure you order the parameters correctly.

```
//The correct way to differentiate parameters is in their order as
defined by the API
<params>
  <param><value><int>42</int></value></param>
  <param><value><int>13</int></value></param>
  <param><value><int>32</int></value></param>
</params>
```

In the request, each parameter is enclosed by a `param` element. Within each `param`, the actual parameter is wrapped up by a `value` element. Within this `value` element are the actual parameter values and data types.

XML-RPC Data Types

XML-RPC parameters can be one of the following three types. Each is represented in a different way inside the `value` element.

- **Scalars:** basic primitive data types
- **Arrays:** similar to PHP numerically indexed arrays
- **Structs:** equivalent to associative arrays in the PHP world

Right now, we are using these data types and their structural definitions as request parameters. However, the same data types are used throughout XML-RPC. The server response will give us data in the same schema.

Scalar Values

Scalar values are the common primitive data types found in most languages. Almost every scalar value must be encapsulated by an element that declares what data type that value is.

String

This is your basic text string. It corresponds to the string data type in PHP.

```
<param>
  <value><string>Hello, world!</string></value>
</param>
```

Since this is the most common data type, any values of the `value` element without data type tags inside will default to string. For example, the following is perfectly legal in XML-RPC and will default to a string:

```
<param>
   <value>Goodbye, cruel world!</value>
</param>
```

Integer

This is a four byte signed integer. It is the same as the PHP data type integer. Hence, it can take the more obvious tag of `int` or `i4`, for four byte integer.

```
<param>
  <value><int>42</int></value>
</param>
// This is the same:
<param>
  <value><i4>42</i4></value>
</param>
```

Double

Double is a double precision signed floating point number. It is the equivalent to float in PHP. Note that in PHP 4, there was also a double data type, but this has been deprecated in preference of float.

```
<param>
   <value><double>-44.352301</double></value>
</param>
```

Boolean

Boolean is your basic true or false state. It is the same as the PHP boolean. The difference is that in PHP, and many other languages for that matter, boolean can be represented by the keywords `TRUE` or `FALSE`, or a numerical setting of `1` for true and `0` for false. In XML-RPC, booleans can only be represented with 1 or 0.

```
<param>
   <value><boolean>0</boolean></value>
</param>
```

Date/Time

A date/time data type specifies a date and a time value, up to the second. It follows the format YYYYMMDDTHH:MM:SS. Year, month, day, hour, minute, and second should be apparent in that format. The "T", however, is a literal. There is no date/time data type in PHP, so you will have to represent this as a string.

```
<param>
  <value>
     <dateTime.iso8601>20060710T11:30:32</ dateTime.iso8601>
  </value>
</param>
```

In PHP 5, there is a date parameter of "c" that will return a date/time object in ISO 8601 format. It will not return the exact format that XML-RPC needs, but later we will use a function to automatically encode it into a date/time type.

Base64-Encoded Binary

To transfer binary information via XML-RPC, encode it in base64 and wrap it around `base64` tags.

```
<param>
    <value>
       <base64>Pj4UijBdhLr6IdvCc0Ad3NVP4OidTd8E1kRY5Edh</base64>
    </value>
</param>
```

There is no binary data type in PHP, but you can encode files in base64 for XML-RPC transfer by using the `file_get_contents` function.

Arrays

Numerically indexed arrays are passed as a single structure within the `value` element. Arrays are defined with a specific structure of child and grandchild elements.

```
<params>
  <param>
    <value>
      <array>
        <data>
          <value><string>One</string></value>
          <value><boolean>Monkey</boolean></value>
          <value><double>4.307400</double></value>
        </data>
```

```
        </array>
      </value>
    </param>
  </params>
```

There are two levels of children before we actually see the data. `array`, which defines the value of the parameter to be an array, and `data`, which signals the start of the data. The data is encapsulated exactly like scalar values. There is a `value` element for each item in the `array` and they may have children elements that define the data type of the value.

Arrays can be recursive. Within each `value` element, can be another `array` as long as they contain the `array/data/value` descendant sequence.

Struct

Similar to arrays, structs are the XML-RPC representation of PHP's associative arrays. Each item has a named key and a value pair. Like arrays, a single `struct` element defines a struct. Each item has one `member` element, each `member` element has a required `name` element, which names the item, and one `value` element which represents the value. Also, like arrays, the `value` element follows the definition rules of scalar values in XML-RPC.

```
<value>
  <struct>
    <member>
      <name>One</name>
      <value><string>This is a string</string></value>
    </member>
    <member>
      <name>Two</name>
      <value><boolean>1</boolean></value>
    </member>
    <member>
      <name>This is a Name</name>
      <value><double>-98.330000</double></value>
    </member>
  </struct>
</value>
```

XML-RPC requests, then, are basically HTTP `POST` actions that specify a remote method to be called with properly formatted parameters. Let's take a look at the response back from the server.

XML-RPC Response

Once we make a request, we can expect one of the two types of responses from the service. If an XML-RPC request was successful, we will receive the data we requested returned to us in a fashion defined by the XML-RPC specifications. If there was an error, a special XML-RPC fault message will be returned.

Similar to a regular web page call, a header will be returned with the results in the body.

```
HTTP/1.x 200 OK
Date: Fri, 11 Aug 2007 23:34:43 GMT
Server: Apache/1.3.33 (Darwin) PHP/5.1.4 DAV/1.0.3
X-Powered-By: PHP/5.1.4
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/xml
<?xml version="1.0" encoding="iso-8859-1"?>
<methodResponse>
  <params>
    <param>
      <value><string>A woeful jeremiad</string></value>
    </param>
  </params>
</methodResponse>
```

This is a successful XML-RPC response. It should look familiar to you. `methodResponse` is the root element that defines this as a response. Following that is a `params` child. Each value that is returned is enclosed in a `param` element. Underneath that, each value follows the rules for scalar values we saw earlier. This example shows a single string value that the service returns. However, like the request, everything under `params` can also be a multiple value return, an array or a struct in addition to single values. For example, an array returned from the service would look like this:

```
<methodResponse>
  <params><param><value>
    <array>
      <data>
        <value><string>system.multicall</string></value>
        <value><string>system.listMethods</string></value>
        <value><string>system.getCapabilities</string></value>
      </data>
    </array>
  </value></param></params>
</methodCall>
```

If the service could not fulfill your request, it will return an XML-RPC fault. Instead of a `params` element, `methodResponse` will have a `fault` element. `methodResponse` will always have either a `params` child or a `fault` child, but not both.

An XML-RPC fault is basically a struct that is returned to you. There are two named members in this struct. A `faultString` is a human readable alert of the error, and `faultCode`, which is an integer assigned by the service. Neither `faultString` or `faultCode` are defined or standardized by the XML-RPC specifications. They depend solely on the server implementation.

```
HTTP/1.x 200 OK
Date: Fri, 11 Aug 2007 23:41:18 GMT
Server: Apache/1.3.33 (Darwin) PHP/5.1.4 DAV/1.0.3
X-Powered-By: PHP/5.1.4
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/xml
<?xml version="1.0"?>
<methodResponse>
  <fault>
    <value>
      <struct>
        <member>
          <name>faultCode</name>
          <value><int>4</int></value>
        </member>
        <member>
          <name>faultString</name>
          <value><string>Too many parameters.</string></value>
        </member>
      </struct>
    </value>
  </fault>
</methodResponse>
```

Working with XML-RPC in PHP

XML-RPC is a very straightforward and effective data transport mechanism. Requests and responses are formatted in a common way. Clients and service providers can easily implement this simple protocol. PHP has a group of XML-RPC functions that are made for working with XML-RPC data.

PHP's XML-RPC functions are documented at: `http://www.php.net/manual/en/ref.xmlrpc.php`. The collection is small but invaluable. They can be grouped into functions that enable PHP applications to be XML-RPC clients, or functions that create PHP-powered XML-RPC services. These functions are available, if PHP was compiled with the `–with-xmlrpc` option.

If you check out the official PHP XML-RPC documentation, you will see they have been marked "experimental". However, they have been around since PHP 4.1 and are long considered stable and reliable for production usage.

To make your mashup, you need to know how to create an XML-RPC request, how to call the service, and how to process the return data. We will focus our attention on three PHP functions that will help us do this: `xmlrpc_encode_request` to convert our PHP variables into XML-RPC format, `xmlrpc_decode` does the reverse, and `xmlrpc_is_fault` checks to see if there was an error with our request to the server.

Making an XML-RPC Request

Our first task is to create an XML-RPC request. This process can be broken up into individual steps:

1. Serialize the PHP data into XML-RPC format
2. Create an XML-RPC request by combining HTTP headers and serialized data
3. Make the call to the service

Serializing Data with XML-RPC Encode Request

In network programming terms, "serializing" means transforming the data into a format that can be delivered across a network. In our case, we will serialize PHP variable values into the XML-RPC specifications that we discussed earlier. To do this, we will use the PHP function `xmlrpc_encode_request`. `xmlrpc_encode_request` that creates our entire XML-RPC request call starting with the `methodCall` element and drilling all the way down to the `value` elements.

This function requires two parameters. The first is the name of the remote call we wish to request. `xmlrpc_encode_request` will use this to create the `methodName` element. The second parameter is the variables you wish to pass. It will use this to create the `params` structure. For this second parameter, `xmlrpc_encode_request` will automatically detect what kind of variable was passed into it, (variable types, numerical arrays, or associative arrays), and format them into the data type schema dictated by the XML-RPC specifications.

Let's try creating some different types of XML-RPC requests using `xmlrpc_encode_request`.

Creating a Single Parameter XML-RPC Request

The simplest XML-RPC request is one single parameter. Create this simple script and put it on your web server:

```
<?php
  $singleVar = "Hello!";
  $requestMessage = xmlrpc_encode_request('theRemoteCall', $singleVar);
  echo $requestMessage;
?>
```

Hit this page with a web browser, and examine the source. Your source will look like:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<methodCall>
  <methodName>theRemoteCall</methodName>
  <params>
    <param>
      <value><string>Hello!</string></value>
    </param>
  </params>
</methodCall>
```

`xmlrpc_encode_request` has created the entire XML-RPC request for you.

If you manipulate the variable type that is passed, `xmlrpc_encode_request` is smart enough to figure out the XML-RPC data type equivalent. We can test this functionality with the following script:

```
<?php
  $singleVar = "88";
  settype($singleVar, "integer");
  $requestMessage = xmlrpc_encode_request('theRemoteCall', $singleVar);
  echo $requestMessage;
?>
```

Here, we are first assigning the value of `$singleVar` to be a string, whose value is "`88`", `settype` will cast `$singleVar` to an integer before we pass it to `xmlrpc_encode_request`. The resulting XML-RPC value returned by `xmlrpc_encode_request` will reflect this data type by wrapping it around `int` tags.

Again, if you hit this page with a web browser, you can view this returned structure:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<methodCall>
  <methodName>theRemoteCall</methodName>
  <params>
    <param>
      <value><int>88</int></value>
    </param>
  </params>
</methodCall>
```

There are two other special data type cases that we need to be careful about.

Double Data Type

The first case is the double data type. `xmlrpc_encode_request` will always pad doubles to six digits past the decimal point. For example, passing 1.6;

```
$singleVar = 1.6;
$requestMessage = xmlrpc_encode_request('theRemoteCall', $singleVar);
```

will result in the following XML-RPC snippet:

```
<value>
   <double>1.600000</double>
</value>
```

Furthermore, anything longer than six digits will be rounded. We can see how this works by passing a long number that should be rounded up, and one that should be rounded off.

```
$roundedUp = 1.123456789;
$roundedOff = 1.87654321;
$requestMessage1 = xmlrpc_encode_request('theRemoteCall', $roundedUp);
$requestMessage2 = xmlrpc_encode_request('theRemoteCall',
$roundedOff);
```

These two calls will result in these two respective value elements:

```
<value>
  <double>1.123457</double>
</value>
<value>
  <double>1.876543</double>
</value>
```

Date/Time and Base64 Data Types

The second case that we need to take caution around is date/time and base64. Neither of them have a data type equivalent in PHP. If you assign them to a variable, PHP will consider that variable to be a string. And when they are passed to `xmlrpc_encode_request`, `xmlrpc_encode_request` will also consider them to be strings.

For example, if we had a special string of binary data, PHP can only treat that as a string of characters;

```
$singleVar = '8923hfjsd89783hiuyi9yhw938ihfasid9yh32iyr9wy';
$ requestMessage = xmlrpc_encode_request('theRemoteCall', $singleVar);
```

The resulting XML-RPC request will be generated. If the remote call is specifically looking for base64, the remote call will fail.

```
<value>
  <string>8923hfjsd8938ihfasid9yh32iyr9wy</string>
</value>
```

To tell PHP they are XML-RPC data types, we need to use a special function, `xmlrpc_set_type`, before we call `xmlrpc_encode_request`. `xmlrpc_set_type` takes two parameters, the first is the actual variable that you wish to change, and the second is the XML-RPC type you want to change it to.

If we took the same binary string and called `xmlrpc_set_type` to change it to base64:

```
$singleVar = '8923hfjsd89783hiuyi9yhw938ihfasid9yh32iyr9wy';
xmlrpc_set_type($singleVar, "base64");
$requestMessage = xmlrpc_encode_request('theRemoteCall', $singleVar);
```

Our resulting XML-RPC request will be what the service is expecting.

```
<value>
  <base64>ODkyM2hmanNkODkzOGloZmFzaWQ5eWgzMml5cjl3eQ==&#10;</base64>
</value>
```

The same thing is true if you had a string formatted to the correct date/time format.

```
$singleVar = '20060814T09:08:23';
xmlrpc_set_type($singleVar, "datetime");
$requestMessage = xmlrpc_encode_request('theRemoteCall', $singleVar);
```

After encoding, the value will have the correct date/time element wrapped around it in the XML-RPC request.

```
<value>
  <dateTime.iso8601>20060814T09:08:23</dateTime.iso8601>
</value>
```

`xmlrpc_set_type` was designed specifically to address the presence of base64 and date/time types in XML-RPC, and their lack in PHP. The second parameter, which determines what kind of XML-RPC data type that you wish to cast, can only take a value of `base64` or `date/time`. It is not meant to be a generic type casting function like PHP's `settype`.

The other structures that `xmlrpc_encode_request` creates, builds upon its basic handling of scalar data. This is in much the same way that XML-RPC's structs and arrays use the same basic scalar schemas at the `value` level. The main limitation is that `xmlrpc_encode_request` can accept only one parameter for its variables parameter. It works around this by accepting different array structures for the variables parameter, and outputting different XML-RPC requests depending on what it receives.

Creating a Multiple Parameter XML-RPC Request

Sometimes, you may find a remote call that requests more than one parameter in a simple list. To do this, simply create an array of the variables and pass the array as the second argument of `xmlrpc_encode_request` like:

```
<?php
  $dt = '20060814T09:08:23';
  xmlrpc_set_type($dt, «datetime»);
  $listArray = array(«One», 2.09, $dt);
  $requestMessage = xmlrpc_encode_request('theRemoteCall', $listArray);
  echo $requestMessage;
?>
```

This will create an XML-RPC request with a param element for each variable. As you can see, it is perfectly legal to mix and match data types.

```
<params>
  <param>
    <value><string>One</string></value>
  </param>
  <param>
    <value><double>2.090000</double></value>
  </param>
  <param>
    <value>
<dateTime.iso8601>20060814T09:08:23</dateTime.iso8601>
</value>
  </param>
</params>
```

Passing Arrays in XML-RPC Requests

Remember that an XML-RPC array is a serialized, numerically indexed PHP array. We need to pass it as the second argument to `xmlrpc_encode_request`. However, if we just pass an array, `xmlrpc_encode_request` will create a multiple parameter XML-RPC request, like we just demonstrated. The solution is to place the numerical array into another array and pass the new bundle as the second parameter of `xmlrpc_encode_request`:

```
<?php
  $listArray = array("One", "Monkey", 4.3074);
  $requestMessage = xmlrpc_encode_request('theRemoteCall', $listArray);
  echo $requestMessage;
?>
```

This code will create the XML-RPC array that we need. This snippet will create the example array, we saw earlier in the chapter.

This works because every parameter in an XML-RPC request is placed within its own `value` element. Technically, we passed only one parameter into `xmlrpc_encode_request`. The function recognized it as a numerical array and created the necessary structure underneath the `value` element.

Passing Struct in XML-RPC Requests

An XML-RPC array is related to a struct much like a numerically indexed PHP array is to an associative PHP array. Knowing this, you can make a logical guess that to create a struct, all we need to do is change the second parameter passed into from a numerically indexed array to an associative array.

```
$listArray = array("One" => "This is a string", "Two" => "true, "This
is a Name" => -98.33);
$requestMessage = xmlrpc_encode_request('theRemoteCall',
array($listArray));
```

In this code snippet, `$listArray` now assigns keys to the array values before being passed to `xmlrpc_encode_request`. This will create the example struct, we saw earlier in the chapter.

```
Creating the Request Call Header
```

Now that we have the body, we can create the HTTP header that the request needs. The header goes before the payload. We work backwards because one part of the header requires us to know the length of the payload.

Let's take a look at a complete XML-RPC request:

```
POST /rpc HTTP/1.0
User-Agent: XML-RPC Client
Host: www.upcdatabase.com
Content-Type: text/xml
Content-Length: 185
<?xml version="1.0" encoding="iso-8859-1"?>
<methodCall>
<methodName>lookupUPC</methodName>
<params>
  <param>
    <value><string>079400560704</string> </value>
  </param>
</params>
</methodCall>
```

Looking at this, there are several header items that can be static. Other items (that are highlighted) can be turned into variables.

1. After `POST` is the name of the script we are hitting on the server
2. The host
3. The content length
4. Finally, the payload

It would be efficient to create a function that would return a complete request to us. The variable candidates will be our parameters.

```
function createPostRequest($remoteServer, $remotePath, $requestBody) {
   $theRequest = "POST " . $remotePath . " HTTP/1.0\n" .
  "User-Agent: XML-RPC Client\n" .
  "Host: " . $remoteServer . "\n" .
  "Content-Type: text/xml\n" .
  "Content-Length: " .strlen($requestBody) . "\n" .
  "\n" . $requestBody . "\n";
  return $theRequest;
}
```

`createPostRequest` is a simple function. It takes three parameters and concatenates them with standard HTTP request headers to create a full, complete XML-RPC request. The three parameters it takes are:

- `$remoteServer` is the name and domain of the server. For example, `"www.amazon.com"`. Do not add a protocol like `http://` because that is not only redundant, but also illegal.

- `$remotePath` is the path to the XML-RPC service starting at the root of the server. This value starts with a beginning slash (/) and is the same as an absolute path in a directory structure. An example would be `"/services/xmlrpc"`.
- `$requestBody` is the part that is returned to us by `xmlrpc_encode_request`. This is the XML portion of the XML-RPC call.

This function uses `$remoteServer` and `$remotePath` to build the first line of the header and the host line. The body, `$requestBody`, is not only appended to the request, but it's size must be first analyzed by the `strlen` function for the content length part of the header.

Notice how we concatenate the headers in the function. Although the headers are broken up and formatted for readability in the header output itself, there are no spaces or tabs from the end of the lines ("\n") to the beginning of the next line. This is a strict formatting requirement of the HTTP protocol. If there is any whitespace at the beginning of the lines, the request will fail.

Calling XML-RPC Using Sockets

We now have a fully formatted XML-RPC call. All variables have been serialized. Our headers have also been created. The only thing left to do is to make the actual call to the service and send it our request.

Sockets are a clean way to call XML-RPC. A socket is basically a direct network connection to another machine. We can build a socket to the XML-RPC server and send our request directly to it. This can be accomplished by another function that builds upon our `createPostRequest` function. This function will not only send the request to the server, but it will capture the result and let us manipulate it later.

```
function send($remoteServer, $remotePort, $fullXMLRPCRequest) {
  $headers = '';
  $data = '';
  $socket = fsockopen($remoteServer, $remotePort);
  fwrite($socket, $fullXMLRPCRequest);
  while ($str = trim(fgets($socket))) {
    $headers .= $str . "\n";
  }
  while (!feof($socket)) {
    $data .= fgets($socket);
  }
  fclose($socket);
  return $data;
}
```

This function also takes the following three parameters:

- `$remoteServer` is the same exact variable you used in `createPostRequest`. Like its use in the HTTP header, do not include the protocol (`http://`).
- `$remotePort` is network port that you wish to connect through. For standard HTTP connections, this will be 80.
- `$fullXMLRPCRequest` is the complete XML-RPC request with headers and payload. This is the part that is returned from `createPostRequest.`

This function does a few things. Let's break it down:

After some initialization of variables, we use `fsockopen` to open our connection to the remote server.

```
$socket = fsockopen($remoteServer, $remotePort);
```

The call to `fwrite()` will send our XML-RPC request directly to the server through the socket.

```
fwrite($socket, $request);
```

A socket is basically a tunnel. We not only write through it, but we capture the server's response to it. After writing, the first thing that is returned is the HTTP response header. We don't need this information, so we can just read the response and dump it to a local variable that will be discarded when the function completes its execution.

```
while ($str = trim(fgets($socket))) {
  $headers .= $str . '\n';
}
```

After the headers come through, we capture what we're really after—the XML-RPC response payload.

```
while (!feof($socket)) {
  $data .= fgets($socket);
}
```

While data is coming through the socket, we capture everything that comes through and dump it into a local variable. Once the response data has stopped coming, we can get rid of our socket and return the portion of the data to the function caller.

```
fclose($socket);
return $data;
```

At this point, we have passed variables into a function we created, `createPostRequest()`, and it returned a full-fledged XML-RPC request. We took

that request and passed it to another function that we defined, `send()`. This function opens a socket connection to the remote server, writes the request, and parses the response. It returns to us this XML-RPC response. We now need to transform the XML-RPC response into PHP data we can use.

Processing an XML-RPC Response

To extract data from an XML-RPC response and transform them into PHP variables, we will use `xmlrpc_encode_request()`'s complementary function, `xmlrpc_decode()`. `xmlrpc_encode_request()` was smart enough to take variables, whether they are primitive variables, arrays, or structures, and transform them into an XML-RPC request.

`xmlrpc_decode()` will take an XML-RPC response, look for the type of data being returned for example, whether they are simple primitives, structures, or arrays, and return either a straight PHP value (if primitive) or an array of values (if they are an XML-RPC struct or array). `xmlrpc_decode()` will also process XML-RPC response errors. Using another function, `xmlrpc_is_fault()`, we can evaluate the return value to see if it is an error and handle them accordingly.

Let's create a function to demonstrate both `xmlrpc_decode()` and `xmlrpc_is_fault()`.

```
function processXMLRPCResponse($xmlrpcResponse) {
  $data = xmlrpc_decode($xmlrpcResponse);
  if (xmlrpc_is_fault($data)) {
    echo 'Error Code: ' . $data['faultCode'] . "\n";
    echo 'Error Message: ' . $data['faultString'];
  } else {
    if (is_array($data)) {
      var_dump($data);
    } else {
      echo $data;
    }
  }
}
```

This function takes an XML-RPC response as the parameter.

The first thing this function does is send the XML-RPC response through `xmlrpc_decode()`. If you were to examine the returned value, it would be either a simple value or an array. If the service returned just a single simple value, `$data` would be that value. If the return was an array, `$data` will be a numerically indexed PHP array. If the return was a struct or an XML-RPC fault, `$data` will be a PHP

associative array. The `if-else` statement and the nested `if-else` statement that follows `xmlrpc_decode()` makes this evaluation.

The first `if` statement calls `xmlrpc_is_fault()`. This function determines if the data structure is an XML-RPC fault. If it is, the return value is an associative array with two items whose indexes are `faultCode` and `faultString`, and whose values are the responses of `faultCode` and `faultString` elements.

If it is not an error, we then check to see if it's an array or a value. Here, we are simply echoing out the values. In practice, you will want to do something like pass the arrays to another function for looping or searching.

Creating an XML-RPC Parser Class

Using all three functions (`createPostRequest`, `send`, and `processXMLRPCResponse`), we have some basic tools to create an all-purpose XML-RPC parser utility class. To make it even simpler to use these functions, we can add a public facade function to call the others.

```
class XMLRPCParser {
  public function callService($remoteMethod, $parameters,
  $remoteServer, $remotePath, $port=80) {
    $requestXML = xmlrpc_encode_request($remoteMethod, $parameters);
    $fullRequest = $this->createPostRequest($remoteServer,
    $remotePath, $requestXML);
    $response = $this->send($remoteServer, $port, $fullRequest);
    return xmlrpc_decode($response);
  }
   private function createPostRequest($remoteServer, $remotePath,
   $requestBody) {
    $theRequest = "POST " . $remotePath . "HTTP/1.0\n" .
    "Host: " . $remoteServer . "\n" .
    "User-Agent: XML-RPC Client\n" .
    "Content-Type: text/xml\n" .
    "Content-Length: " .strlen($requestBody) . "\n" .
    "\n" . $requestBody . "\n";
    return $theRequest;
  }
  private function send($remoteServer, $remotePort,
  $fullXMLRPCRequest) {
    $headers = '';
    $data = '';
    $socket = fsockopen($remoteServer, $remotePort);
    fwrite($socket, $fullXMLRPCRequest);
```

```
        while ($str = trim(fgets($socket))) {
           $headers .= $str . '\n';
        }
        $data = '';
        while  (!feof($socket)) {
           $data .= fgets($socket);
        }
        fclose($socket);
        return $data;
      }
     public function processXMLRPCResponse($data) {
         if (xmlrpc_is_fault($data)) {
           echo 'Error Code: ' . $data['faultCode'] . "\n";
           echo 'Error Message: ' . $data['faultString'];
         } else {
           if (is_array($data)) {
             var_dump($data);
           } else {
             echo $data;
           }
         }
       }
   }
```

We've seen the last three functions. The first function, `callService()`, is a façade for the `xmlrpc_encode_request()`, `createPostRequest()`, and `send()` functions. Any of our applications that wants to make an XML-RPC call can just use `callService()` to handle everything. We pass it as the remote method we want to call, the parameters as an array, the remote server name, the path to the service, and, if necessary, the port number. The first line in the function takes the name of the method and the parameters and uses `xmlrpc_encode_request()` to get a XML-RPC request. We pass the request along with the server name, and the service path to `createPostRequest()`. This returns to us the XML-RPC request and a valid HTTP `POST` header. Finally, we pass that to `send()` which makes a socket connection to the server and gives it the XML-RPC request. The return value of `send()` is the XML-RPC response.

Testing Our XML-RPC Parser Class

To see this all in action, we can use it to call a few public XML-RPC services. In the example code, the file `example RPC.php` has the XML-RPC parser class plus a few lines of code at the top to call a few services:

```
$parser = new XMLRPCParser();
echo "<h3>First Example</h3>";
$returnedData = $parser->callService('latestDownloadURL',
     null,
    'www.upcdatabase.com',
    '/rpc',
    80);
$parser->processXMLRPCResponse($returnedData);
echo "<h3>Second Example</h3>";
$returnedData = $parser->callService('geocode',
  '1000 Elysian Park Ave., Los Angeles, CA',
  'geocoder.us',
  '/service/xmlrpc',
  80);
$parser->processXMLRPCResponse($returnedData);

class XMLRPCParser {
/* Rest of XMLRPCParser Class */
```

In the first line, we create the XML-RPC Parser object, which will be used twice. In the first example, we will use the Internet UPC Database XML-RPC service to call a method named `latestDownloadURL`. This returns the current URL to a zipped copy of the entire Internet UPC Database. The service is located at `http://www.upcdatabase.com/rpc`. We will use it to look up product information before we send it to Amazon, later in our mashup.

This service takes no parameters. You will run into a lot of XML-RPC methods that do not take any parameters. However, `xmlrpc_encode_request()` requires something to be passed for the parameters in the second argument. In such a case, simply pass `null` for the second argument to `xmlrpc_encode_request()` like we do here. In this example, `null` eventually gets passed to `xmlrpc_encode_request()` in the first line of the `callService()` function.

In the second example, we use the geocoder at `http://geocoder.us/service/xcmlrpc`. Geocoder takes an address and finds the longitude and latitude of the address. The remote method is called `geocode` and it takes a string of the address that you want to look up. In both examples, we pass the results of `callService()` to `processXMLRPCReponse()`, which simply displays the results from the service call.

Now we are ready to see this in action. Save this file to your web server and load the page in your browser.

You should see something like this:

In the first example, the return is just a simple string, `processXMLRPCReponse()` just outputs this on the browser. In the `geocoder.us` example, an array is returned by the response. `processXMLRPCReponse()` passes arrays through `var_dump()`, which outputs the key, value, and value data type for each item in the array.

Satisfied that this class works, we can use this class for our mashup. Remember to remove the example instantiation and usage code before actually using this class.

Using PEAR to Handle XML-RPC

Out of the box, PHP does a good job of handling XML-RPC. There are a few drawbacks with our approach. The first is that it does require PHP to have been compiled with `-with-xmlrpc` option. If not, you will have to recompile PHP. If you are in a tightly controlled corporate environment, a shared web hosting service, or using a precompiled binary, recompiling may not be an option to you.

From a development effort standpoint, our approach is a bit of a hassle. `xmlrpc_encode_request` and `xmlrpc_decode` do a lot of translation work for us, but we still need to make the HTTP headers and get down and dirty with sockets in order to make a tunnel to the server. These two tasks seem like they can be abstracted out to another function. Fortunately, PEAR has an XML-RPC extension that does both of these things for us.

PEAR, if you are not familiar with it, is an official set of third-party libraries compiled for PHP. Although, it is its own separate open-source project, the PEAR installer and package manager is included with the official PHP release. Using the PEAR installer, you can easily install, upgrade, and uninstall PEAR libraries with simple command line options.

To see what you currently have installed, go into your command line on the PHP server and type in the command to list your installed PEAR extensions:

```
Buttercup:~ root# pear list
```

Depending on how PHP was installed, you might have to be the root user to manage PEAR libraries on the machine. Another problem is that your PHP might be non-standards, and it may not have PEAR installed. If this is the case, you can download PEAR from `http://pear.php.net/package/PEAR/download`.

Assuming you have the correct permissions and PEAR is installed, you may see something like this:

```
Buttercup:~ root# pear list
Installed packages:
==================
Package              Version       State
Archive_Tar           1.1          stable
Console_Getopt        1.2          stable
…
PEAR                  1.3.6        stable
XML_Parser            1.0.1        stable
XML_RPC               1.4.0        stable
```

These are your currently installed PEAR extensions. If you see the last line, `XML_RPC`, you are in business. PEAR's XML-RPC extension is already installed and you can go ahead and use it. If an XML_RPC line is not on the output list, you can simply install the extension by typing:

```
Buttercup:~ root# pear install XML-RPC
```

Let's take a quick look at how the XML_RPC extension is used to interact with XML-RPC services. Let's create a simple script to invoke a service from the UPC Internet Database:

```
<?php
  require_once("XML/RPC.php");
  $params = array (new XML_RPC_VALUE('079400560704', 'string'));
  $msg = new XML_RPC_Message('lookupUPC', $params);
  $client = new XML_RPC_Client('/rpc', 'www.upcdatabase.com');
  $retVal = $client->send($msg);
  $valueObj = $retVal->value();
  echo $valueObj->scalarVal();
?>
```

The first line, the `require_once()` statement, will include the necessary PEAR library file into our script. Once this is included, all the XML_RPC functions in PEAR will be available to us. This statement is absolutely required to be the first line in our script. The next line:

```
$params = array (new XML_RPC_VALUE('079400560704', 'string'));
```

builds an array of parameters. There are several things going on here. The first is that we are creating a new object, `XML_RPC_Value`. In PEAR, each XML-RPC parameter is placed in its own new `XML_RPC_Value` object. The arguments are then placed in an array. XML_RPC takes this array and creates the necessary XML_RPC parameter structure in the request.

The `XML_RPC_Value` object takes two arguments. The first argument is value of the parameter. The second argument is the XML_RPC data type of parameter. While this approach explicitly requires us to specify the XML_RPC data type, it eliminates the need to call `xmlrpc_set_type()` for date/time and base64, making the conversion more consistent and the code cleaner.

```
$msg = new XML_RPC_Message('lookupUPC', $params);
```

The next line takes the name of the function we wish to call and the parameters array we just created, and places it in a new object, `XML_RPC_Message`. The return value is an object representation of everything that is needed to create an XML-RPC request message.

Finally, an `XML_RPC_Client` object is created using the `XML_RPC_Message` object and service details. This client object now has everything that we need to create an XML-RPC request. It has the server name, the path to the service, and the name of the remote call. The object has a method, `send()`, which uses all its information and makes an XML-RPC request to the service. We call `send()`, passing in the message object, which contains the parameters, to execute the remote call.

```
$retVal = $client->send($msg);
```

`send()` automatically takes the service response and places it in an object of class `XML_RPC_Response`.

`XML_RPC_Response` is a PHP class that represents the XML-RPC response. It has methods that can tell us whether the response was a fault, what the fault was, and, if it was not a fault, the actual response values. To get the response values, we use the object's `value()` method.

```
$valueObj = $retVal->value();
```

Finally, we get to the raw data. `value()` returns another `XML_RPC_Value` object that holds only the data that the remote call gave us. We call `scalarVal()` to get the actual data this holds. Note that regardless of whether the response is a single scalar value, a struct, or an array, `scalarval()` will return it. It is up to us to properly detect and handle complex return values.

In a nutshell, that is how PEAR is used to create an XML-RPC client. This has been a very brief introduction to the XML_RPC objects used. A deeper look at the objects, properties and methods will give a good idea on how to do more important things, like detecting faults and changing the port of the call. For more information on PEAR XML_RPC, you can refer to *PHP Programming with PEAR* by Stoyan Stefanov, et. al. (ISBN 1904811795) published by Packt Publishing.

REST

Since XML-RPC came along, other web services standards have arisen to fulfill needs that XML-RPC does not address. Some, like **SOAP** and its derivatives, are designed to give more power and flexibility to the developer. Others, like **RSS**, are used for a specific niche. This is not unlike other evolutionary trends in technology where standards start off simple, and for better or for worse, become more complex. Very recently, though, a new web service standard has become popular that bucks this trend. **REST**, an acronym for **Representational State Transfer**, is not a formalized standard but instead an architectural style.

In theory, REST is rich and modern. Its application is not limited to the World Wide Web. However, web browsers and web servers fit nicely in REST theory. Described by **Roy Fielding** for a doctoral dissertation, in 2000, REST attempts to describe network-based software architectures.

In practice, REST is simple and flexible. Like XML-RPC, a client makes a request to the server, but this is pretty much where the similarities end. REST requests are simply an HTTP request. They can be any of the standard five HTTP methods of `GET`, `POST`, `PUT`, `DELETE`, or `HEAD`. The latter three are rare in web application development, and are still rare in the REST world. `POST` and `GET` are prevalent. However, a lot of APIs currently will use `GET`, even if the operation is better suited for `POST`. Either way, REST requests operate very similarly to browsers hitting a web server.

Acceptable parameters are defined in the API. For REST over `GET`, the parameters would be in the URL query string. This is no different than passing URL parameters normally through `GET`. We do not have to structure our request in XML.

Likewise, the return value is completely up to the API. A lot of APIs return some sort of XML structure. Others simply return a string value without any structure. The latter is especially common in responses that return one value instead of multiple values. There are no standardized error structures returned. The API defines how it will return an error. It is completely up to us to determine how we are going to process this response.

Practitioners of REST recognize that even the simplicity of XML-RPC can be overkill in some circumstances. It is very well suited for web services that do not exchange a lot of values from the client to the server, although its flexibility does allow for complex responses. REST also realizes that from an HTTP service provider's point of view, data types are completely superficial. In HTTP, everything that gets transmitted is a string of characters. A double is only a double because of XML-RPC's `double` element tag. Likewise, booleans could easily have been represented by the words "true/false", "Y/N", or even "table/chair." It is up to the service to treat 1.974234 as a double and "Y" as true, which eliminates the need for casting on our client side.

REST and AJAX

An important reason for REST's recent rise is that it plays very well with another architectural design that has become quite popular. Asynchronous JavaScript and XML, or AJAX is based on the browser sending a bit of information to a server through JavaScript's `XMLHttpRequest` object, parsing the XML response back using JavaScript Document Object Model functions, and updating the page dynamically without reloading. XMLHttpRequest merely needs the URL of the service and a name/value string of parameter names and their values. It would be difficult and tedious to concatenate a string to make a complete XML-RPC request in JavaScript. You would have to essentially code a properly formed XML document entirely in JavaScript mixing the two languages. Concatenating a string of name/value pairs together to make a URL, which is exactly what REST needs, with is much easier. While we are concentrating on consuming REST with PHP, keep this in mind if you are coding JavaScript clients and have a choice of protocols to choose from in the API. For more information on coding AJAX applications, refer to *Ajax and PHP: Building Responsive Web Applications*, by Darie et. al (ISBN 1904811825), published by Packt Publishing..

Working with REST in PHP

Like XML-RPC, we will need to create a REST request with PHP and use PHP to process the response. Unlike XML-RPC, there are no functions created specifically for REST, but that is not necessary. As REST uses just a simple HTTP request, we can use the existing network functions to make and capture the response. The results will come back in XML. We will look at two ways of processing it and turning into PHP data that we can use.

Making a REST Request

Like we talked about earlier, REST is basically like a web page request to the web server. It does not get any more glamorous than that. We will look at a couple of ways to initiate this hit.

A GET and POST Refresher

We constructed a `POST` request earlier in order to make our XML-RPC call. Let's take a closer and quick look at the differences between a `GET` header and a `POST` header.

Here is a bare bones `GET` request:

```
GET /aService.php?One=1&Two=2+and+2%25&Three=3 HTTP/1.0
Host: localhost
User-Agent: A PHP Client
```

There are a couple of things to note here:

1. `GET` is the first command in the first line.
2. The path to the service, `/aService.php` follows.
3. The query string, `One=1&Two=2+and+2%25&Three=3`, is a set of parameters that follows the path. It is separated from the path by a question mark. Each parameter is separated by an ampersand.
4. The parameter values must be URL encoded. In this example, the value for "Two" actually reads "2 and 2%".
5. There is no body following the headers.

Now, using the same parameters, let's convert it into an example `POST` request and compare it with the `GET` request.

```
POST /aService.php HTTP/1.0
User-Agent: XML-RPC Client
Host: localhost
Content-Type: application/x-www-form-urlencoded
Content-Length: 28

One=1&Two=2+and+2%25&Three=3
```

1. `POST` is the first command in the line.
2. The path to the service, also follows this command.
3. The parameters, `One=1&Two=2+and+2%25&Three=3`, do not follow the path, instead, they make up the body of the request.
4. The values are also URL encoded.

5. `POST` requests via web browsers must also have a `Content-Type` header with a value of "`application/x-www-form-urlencoded`". In XML-RPC, as the XML-RPC request is an XML-formatted document, it was `text/xml`.
6. The `Content-Length` header must have the length of the body.

HTTP requests can have many more headers and be much more complex. However, for the purposes of making REST requests, these are the only headers that we need.

You may notice that all of the examples this far have been using HTTP 1.0. This may seem archaic since HTTP 1.1 has been around for years. The reason that we are using the older protocol is that with HTTP 1.1, servers can, and often do, respond with chunked responses when 1.1 is used. That is, to save response time, the server will start sending back the actual response without knowing what size it is, and thus, does not provide a Content-Length header. While most modern clients can handle this just fine, PHP does not. The nature of sockets makes it so that we must essentially handle the transfer chunks ourselves. While this is by no means impossible, it is quite tedious. Therefore, for this book, we'll stick with HTTP 1.0.

Using Sockets to Initiate a REST Request

Knowing the differences between `GET` and `POST`, we can make a few modifications to our previous XML-RPC request functions, which uses `POST`, to make them work with REST. Our socket connection function from before, `send()`, takes a full HTTP request, opens the socket and returns the server response. We can use this exact function to do the same thing. All we need to do is create the full HTTP request. We can modify another previous function, `createPostrequest()` to make the HTTP request. There are some key differences from things that we will have to change, though:

1. We do not have the luxury of `xmlrpc_encode_request()` to turn PHP variables into a formatted HTTP body for us.
2. The variables will have to be converted into a URL query string and, for `GET`, appended to either after the path to the service, or, in the case of `POST`, appended as the body of the request.
3. While XML-RPC requests occur over `POST`, the majority of REST requests happen with `GET`. However, some APIs may call for `POST`. We should be ready to create both.

To address the first and some of the second points, we will create a function that takes an associative PHP array, loops through it, encodes the values into a URL-friendly format, and creates a single query string from the key and values.

```
function makeParameterString($anArray) {
  $returnMe = '';
  foreach($anArray as $key => $value) {
    $returnMe .= "&" . $key . "=" . urlencode($value);
  }
  return substr($returnMe, 1);
}
```

This function assumes that the keys in the array are the names of the parameters and the values of the array are the values of the variables that you want to pass. The `foreach` loop in this function goes through the array and formats the name and value pair of each array item into a query string. Each name/value pair is separated with an ampersand, so we start off with that first. We concatenate the name of the parameter, `$key`, to the string, followed by an equal sign to assign the value, and finally the value of the parameter represented by `$value`. The value is passed through the `urlencode()` function before it is concatenated. Outside of the loop, we strip out the leading ampersand of the first name/value pair with `substr()` and return it.

Creating GET and POST Request Functions

Now that we have the parameter string, we can create the headers and make a request out of it. We'll start off with a `POST` request because it is strikingly similar to the XML-RPC version.

```
function createPostRequest($remoteServer, $remotePath, $paramString) {
  $theRequest = "POST " . $remotePath . " HTTP/1.0\n" .
  "User-Agent: XML-RPC Client\n" .
  "Host: " . $remoteServer . "\n" .
  "Content-Type: application/x-www-form-urlencoded\n" .
  "Content-Length: " .strlen($paramString) . "\n" .
  "\n" . $paramString . "\n";
  return $theRequest;
}
```

This function takes a server address, a path to the service, and a parameter string to make a full HTTP `POST` request. The only difference is that the `Content-Type` header has been changed from `test/xml`, which is what an XML-RPC request is, to `application/x-www-form-urlencoded`, which is a more generic `POST` type.

Now, we can create a function to make a `GET` request. This should be very similar to the `POST` request, but the parameters need to be moved up to the first line, the method changed to `GET`, and the `Content-Length` changed to zero, because there is no body.

```
function createGetRequest($remoteServer, $remotePath, $paramString) {
   $theRequest = "GET " . $remotePath . "?" . $paramString . "
   HTTP/1.0\n" .
  "Host: " . $remoteServer . "\n" .
  "User-Agent: XML-RPC Client\n\n";
  return $theRequest;
}
```

Looking at the first line, we see that our method now says `GET` instead of `POST`. The path is appended as normal, however, now we append a question mark to indicate that a server is going to send some query parameters. The values of the parameters are concatenated after the question mark. `Content-Type`, `Content-Length`, and the body are removed because they are not needed here.

Following functions will get a REST service response for us.

Making a REST Parser Class

Like the XML-RPC Parser class, we should create a facade function to simplify the REST call for us. For clean code organization, we'll organize all the functions into a class.

```
class RESTParser {
  public function callService($parameters, $remoteServer, $remotePath,
  $httpMethod, $port=80) {
    $paramString = $this->makeParameterString($parameters);
    switch(strtoupper($httpMethod)) {
      case "POST":
        $fullRequest = $this->createPostRequest($remoteServer,
        $remotePath, $paramString);
        break;
      case "GET":
        $fullRequest = $this->createGetRequest($remoteServer,
        $remotePath, $paramString);
        break;
      default:
        $fullRequest = $this->createGetRequest($remoteServer,
        $remotePath, $paramString);
    }
    return $this->send($remoteServer, $port, $fullRequest);
  }
  private function makeParameterString($anArray) {
    $returnMe = '';
    foreach($anArray as $key => $value) {
      $returnMe .= "&" . $key . "=" . urlencode($value);
```

```
      }
      return substr($returnMe, 1);
    }
    private function createPostRequest($remoteServer, $remotePath,
    $paramString) {
      $theRequest = "POST " . $remotePath . " HTTP/1.0\n" .
      "Host: " . $remoteServer . "\n" .
      "User-Agent: XML-RPC Client\n" .
      "Content-Type: application/x-www-form-urlencoded\n" .
      "Content-Length: " .strlen($paramString) . "\n" .
      "\n" . $paramString . "\n";
      return $theRequest;
    }
    function createGetRequest($remoteServer, $remotePath, $paramString) {
      $theRequest = "GET " . $remotePath . "?" . $paramString . "
      HTTP/1.0\n" .
      "Host: " . $remoteServer . "\n" .
      "User-Agent: XML-RPC Client\n\n";
      return $theRequest;
    }
     private function send($remoteServer, $remotePort,
     $fullXMLRPCRequest) {
      $headers = '';
      $data = '';
      $socket = fsockopen($remoteServer, $remotePort);
      fwrite($socket, $fullXMLRPCRequest);
      while ($str = trim(fgets($socket))) {
        $headers .= $str . '\n';
      }
      $data = '';
      while  (!feof($socket)) {
        $data .= fgets($socket);
      }
      fclose($socket);
      return $data;
    }
  }
```

This class uses the same functions as was described. Besides encapsulating the functions in a class, there is also the addition of a new `callService()` façade. This function is like the one in the XMLRPC Parser class with two differences. First, it does not take a remote method name as a parameter because that concept does not exist for REST. Second, the method expects the keyword of `GET` or `POST` to specify which HTTP method we want to use for the REST call.

In summary, the sequence of events that this class executes is:

1. Pass an array of parameters to `makeParameterString` to get a URL encoded query string.
2. Pass the results of that to either `createGetRequest` or `createPostRequest` depending on the request type we want. This will give us a full request with the proper headers.
3. Pass the full request, headers and all, to the `send` function from before. This will give us the REST response from the service.

Testing Our REST Parser Class

To test our REST functions, we can set up a basic REST service on our web server environment. This will demonstrate how all the functions work together. The service that we will create will basically take an incoming request, report what method was used to make the request, and report any `POST` or `GET` parameters that were sent with the request.

Create a file with the following chunk of code:

```
Method <?= $_SERVER['REQUEST_METHOD'] ?>
<p>
<b>POST Variables:</b><br />
<?php foreach ($_POST as $key => $value) { ?>
  Key: <?= $key ?> | Value: <?= $value ?><br />
<?php } ?>
</p>
<p>
<b>GET Variables:</b><br />
<?php foreach ($_GET as $key => $value) { ?>
  Key: <?= $key ?> | Value: <?= $value ?><br />
<?php } ?>
</p>
```

Name this file `RESTService.php` and place it in an area on your web server that is accessible from a web browser.

The file displays the method that was used to access the file, using the `$_SERVER` global array. It then loops through the global `$_GET` and `$_POST` arrays, and echos out any items in those arrays.

An example REST client page named `exampleREST.php` is in the examples code that will interact with this service. Using the REST Parser class we created, we put some test code at the top of the page, like we did to test the XML-RPC class. This code will instantiate the Parser and create two arrays. We are going to use the first array for the call to the service using `POST`, and the second array for the call to the service using `GET`.

```
<?php
  $params1 = array("One" => 1, "Two" => "2 and 2 = 4");
  $params2 = array("Three" => 3, "Four" => "4 o'clock");
  $parser = new RESTParser();
  echo $parser->callService($params1, 'localhost', '/mashups/ch1/
                            RESTService.php', 'POST');
  echo "<hr>";
  echo $parser->callService($params2, 'localhost', '/mashups /ch1/
                            RESTService.php', 'GET');
  class RESTParser {
/* Rest of RESTParser Class */
```

In this code, variables ending with "1" are associated with the `POST` request and anything ending with "2" are associated with the `GET` request. For brevity, we will only take a look at the lines for one of the sets.

```
$params1 = array("One"=>1, "Two"=>"2 and 2 = 4");
```

The first thing this code does is create an array and assigns it to `$params1`.

Next, `callService()` is called, initiating the chain of events, it passes the arrays as parameters for the REST request, the name of the server, the path to the service, and the method that we wish to use. The returned data of `callService()` is whatever the REST service returns. We just echo out the return value.

Like you did for the XML-RPC parser class, save this file in an area that is servable by your web server. Hit this client page with your web browser.

You should see the following:

The horizontal rule separates the two REST requests. The top half is the result of the REST call that used `createPostRequest()`. The service looped through the `POST` variables and outputted them. There were no `GET` variables, so that section is blank. In the bottom half, the opposite happened. We used `createGetRequest()` with a completely different set of variables. The service found `GET` parameters and outputted those, but did not find any `POST` variables to output.

At last we have a tool to call and store REST services. It's time to process REST results into something that we can use and manipulate within PHP.

Processing a REST Response

If the response is one simple, unformatted text string, we can easily manipulate it using the variable that we declared and used to store the return value of `sendResult()`. However, REST APIs will rarely send us something that simple. Instead, they are likely to send us a response in XML. We will need to use PHP to load the XML file, go through it, extract the data that we need, and convert them into PHP variables.

There are a few parsers in PHP's arsenal that can be used to process raw XML. They can be classified as either SAX-based parsers or DOM-based parsers. SAX, which stands for Simple API for XML, is event-based. The parser starts at the beginning of an XML stream and executes **callback functions** when certain events occur like the start of an element, the start of an element value stream, or the start of the end of an element. Callback functions are functions that are automatically called when certain events take place.

When the parser passes any point in the XML document, it cannot revisit that point. If it wishes to do so, it would have to reload the entire document.

DOM, Document Object Model processing, works quite differently. The parser loads and stores the entire XML document in memory. It models the document as a hierarchy of elements, and because of this, DOM-based parsers are also known as tree-based parsers. We can manipulate and extract information from anywhere in the document using certain PHP functions. While this gives us more flexibility, the act of holding an XML document in memory is resource intensive. Also, in order to extract information from a DOM, PHP has to do some sort of searching within the document before anything is done with the data. In SAX, the search overhead is not there. The parser just needs to go through the document and gives the data to some other function to process. Because of these factors, DOM-based parsing is considered to be slower and more resource hungry than SAX-based parsing.

That's not to say that tree-based processing does not have its place. The ability to search a document at any time without the overhead of reloading is invaluable, and sooner or later in your career, you will need it. Which ever method you use for a project is entirely dependent on your application and how you architecture it. In future projects, we will explore PHP and DOM parsing. For now, we will use PHP's implementation of SAX in this project.

The SAX functions are documented at `http://www.php.net/manual/en/ref.xml.php`. These functions have the honor of being the first group of functions created to parse XML, and have been around since PHP 3. Enabled by default, you do not have to recompile PHP to use them, unless your build of PHP explicitly disabled them with the `-disable-xml` switch at compile time.

Basic Walkthrough with PHP and SAX

While there are a lot of functions in PHP's XML family, basic SAX processing with PHP can be done in just six steps:

1. Create a SAX parser.
2. If any parser options are necessary, set them.

3. Declare the callback function that will be called when a start of an element is encountered, and the function that will be called when an end element is encountered.
4. Declare the callback function that will be called when character data, or data in between elements, is encountered.
5. Feed the document into the parser.
6. Release the parser.

Let's take a look at this process with a very simple script. This script is named `exampleSAX.php`.

```
<?php
    $xml = "<Pet>" .
       "  <Name id='4323412'>Avi</Name>" .
       "  <Species>African Grey</Species>" .
       "  <Gender>F</Gender>" .
       "</Pet>";
    $parser = xml_parser_create();
    xml_parser_set_option($parser, XML_OPTION_CASE_FOLDING, 0);
    xml_parser_set_option($parser, XML_OPTION_TARGET_ENCODING,
                          "UTF-8");
    xml_set_element_handler($parser, "startElement", "endElement");
    xml_set_character_data_handler($parser, "characterData");
    xml_parse($parser, $xml);
    function startElement($parser, $name, $attributes) {
    echo «Start: « . $name . «<br />»;
     foreach ($attributes as $key => $value) {
       echo «Attribute: « . $key . « :: « . $value . «<br />»;
    }
}
    function endElement($parser, $name) {
      echo «End:» . $name . «<br>»;
    }
    function characterData($parser, $data) {
       echo «Data:» . $data . «<br>»;
    }
?>
```

At the beginning of this script, we set up a simulated XML document chunk and set it to the variable `$xml`. After that, we can begin the SAX parsing process.

Using the PHP's XML Functions

The very first thing we have to do is create a parser using the function `xml_parser_create()`. This function returns a SAX parser for us that we will use throughout this process.

```
$parser = xml_parser_create();
```

To set any parser options, we use `xml_parser_set_option()`.

```
xml_parser_set_option($parser, XML_OPTION_CASE_FOLDING, 0);
```

There are four options that can be changed:

Option	Value	Description
XML_OPTION_CASE_FOLDING	1 or 0	Controls whether the element name is passed to the callback function. Enabled (1) by default.
XML_OPTION_SKIP_TAGSTART	Integer	Specifies how many characters should be skipped at the beginning of a tag. Zero by default.
XML_OPTION_SKIP_WHITE	1 or 0	Controls whether data with only whitespace are skipped. Disabled (0) by default.
XML_OPTION_TARGET_ENCODING	String	Specifies which encoding is to be used. Supported are *ISO-8859-1*, *US-ASCII* and *UTF-8*. By default, it is whatever encoding was used to create the parser.

The middle two have exhibited some inconsistent behavior between PHP 4 and 5 and the UNIX and Windows version of PHP. Use them with caution.

The first, and last, however, are stable and it will not be uncommon for you to run into situations where you need them. We should take a quick look at them. We will see later how the start element and end element callback functions get passed the name of the element. We will need to test the value of the elements in the callback functions. PHP's SAX parser does **case folding** on the names of the elements when they are passed to the functions. In other words, the letter case of the element name gets passed as uppercase to the callback function. XML, however, is case sensitive. This default behavior of PHP is a little controversial, as PHP treats the element name as all uppercase when XML considers `elementName` and `ELEMENTNAME` to be different. Some regard turning off case folding in the parser to be good coding practice.

The other important option is the target encoding option. By default, the SAX parser will set the encoding to whatever the script is when `xml_create_parser()` is called. If you are processing an XML document whose encoding is different, you will need to use `XML_OPTION_TARGET_ENCODING` to switch it. Pass the encoding that you wish to set as the third parameter of `xml_parser_set_option()`.

```
xml_parser_set_option($parser, XML_OPTION_TARGET_ENCODING,
"UTF-8");
```

After we change the options, we can set the callback functions for the start and end elements. We do this with the function `xml_set_element_handler()`.

```
xml_set_element_handler($parser, "startElement", "endElement");
```

This function takes three parameters. The first is the parser we created. The second is the name of the start element callback function, and the third is the end element callback function. More notes about PHP callbacks can be found at `http://uk.php.net/callback`.

Finally, we need to set one more callback function before we start parsing. We need to specify the function that will be called when the parser encounters character data. We do this with `xml_set_character_data_handler()`.

```
xml_set_character_data_handler($parser, "characterData");
```

This function works very similar to `xml_set_element_handler`. It takes the parser as the first parameter, and the name of the callback function in the second parameter.

In our example script, we have also defined the callback functions `startElement()`, `endElement()`, and `characterData()`. We will take a look at those shortly. As long as they are defined and available in the script, you can start the parsing of the XML with `xml_parse()`.

```
xml_parse($parser, $xml);
```

Pass the parser and the XML as the parameters to `xml_parse()`.

Setting up the Callback Functions

In the callback functions we set up, they simply display the name of the element, any attributes, and any data that exists in the element. The functions get these values from parameters passed to it by PHP, when the callback functions are called. If you are familiar with object-oriented programming, think of callback functions as implementing an interface. When we create our callback function, we must declare these parameters in our method signature. Let's take a look at each callback function we have defined and how they use the individual parameters.

Our start element callback simply displays the element name and loops through any attributes the element has.

```
function startElement($parser, $name, $attributes) {
  echo "Start: " . $name . "<br />";
    foreach ($attributes as $key => $value) {
     echo "Attribute: " . $key . " :: " . $value . "<br />";
    }
}
```

PHP passes the parser, the name of the element, and the attributes as an array to the start element callback. If the element has any attributes, they are passed to the callback as an associative array in the third parameter. Here, we simply loop through the array.

The end element callback has two parameters passed to it—the parser and, again, the name of the element.

```
function endElement($parser, $name) {
   echo "End:" . $name . "<br>";
}
```

Our last callback function is the function that's called whenever the parser encounters character data, that is, data that is not an element.

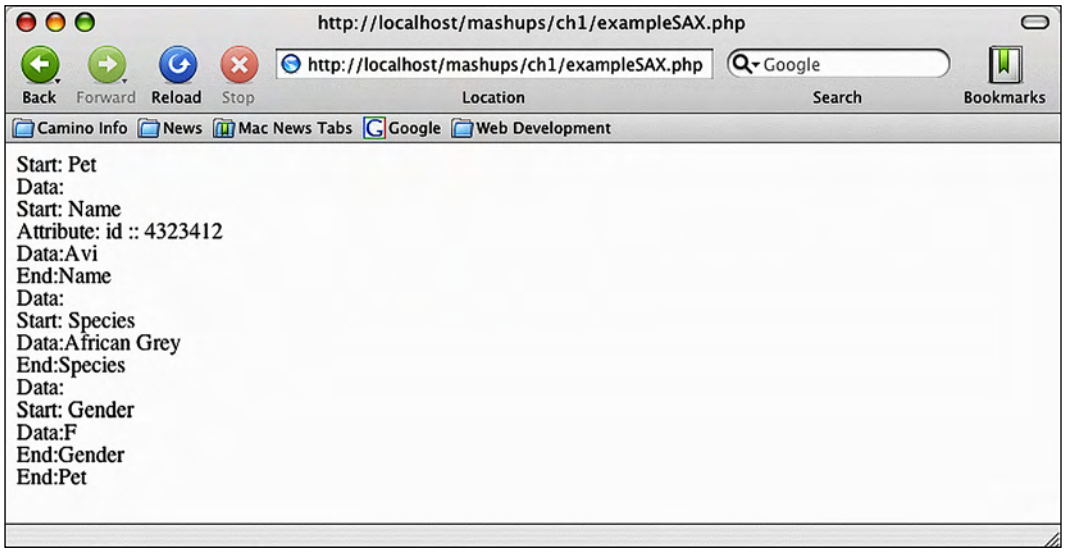
```
function characterData($parser, $data) {
   echo "Data:" . $data . "<br>";
}
```

The parser is passed as the first parameter and the value of the data is passed as the second. Again we simply echo out the data.

Seeing the Callbacks in Action

Now it's time to see the script in action and see how the callback functions are being called. Save the script in an area servable by your web server. Hit the script with a web browser.

You should see this:



As we walk through this, recollect the structure of the XML document that we passed. There is a `Pet` element, three elements (`Name`, `Species`, `Gender`) nested inside it, all of them with data.

The events and function execution are:

1. The `Pet` element is encountered, triggering `startElement()`.
2. Character data in the form of the whitespace after `Pet` executes `characterData()`.The echo statement of `characterData` is simply blank.
3. The `Name` element is encountered, triggering `startElement()`.The element has attributes, the `$attributes` value of the start element callback has an array.
4. The `foreach` loop inside `startElement()` is executed, and the value of the `Name` element's attributes are looped through and echoed.
5. The value of `Name` appears courtesy of `characterData()` since this time, the character data is not just whitespace.
6. With the closing of the `Name` element, `endElement()` is called.
7. The `startElement()`, `characterData()`, `endElement()` sequence is executed again for the `Species` and `Gender` elements.
8. Finally, `endElement()` is called when the `Pet` element is closed.

Creating a SAX Parser Class

The important thing to note about SAX is that when an opening element, character data, or a closing element is encountered, one and only one function is activated. This holds several implications:

1. When the parser goes through a document, we must store values that we care about in a more "global" variable. It cannot be stored in the callback function's local variables because they would disappear after the function is executed.
2. We must detect which "global" variable to use as storage. The only way to do this is to inspect the element name when the start element callback function is called. We can also detect element attributes when we inspect the element name.
3. As we must compare the name of the element in the start element callback, any SAX parser is highly customized to the XML document that it parses. Any changes to the XML document are likely to result in needed changes to any SAX parser we create. We should do our best to minimize the coding maintenance that is necessary because the parser and XML document is so tightly coupled.

Creating a parser class to handle our SAX processing is an easy and efficient way to handle our needs and work with SAX's event-oriented parsing method. Moreover, with PHP 5's new object-oriented features, we can design an architecture that gives us a bit of flexibility.

Instead of a single parser class, we are really going to create two classes. The first is an abstract class. The constructor will declare the parser, set the options, and execute the parsing. The start element, end element, and character data callback functions need to be customized to an individual XML document, so they will be abstract in this class.

Our second class will extend this abstract class. The second class will implement the parent's abstract classes with its own specialized code. The properties of this class are the values that we wish to extract from the XML document. This allows us to separate code that is specific to an XML document from code that can be generic. If our application needs to parse more XML documents, we do not have to rewrite the generic code. We simply need to extend the abstract class, and the loading and setup code will be there for us. While this sounds complex, we can refactor our walkthrough code from earlier.

```
abstract class SAXParser {
  private $parser;
  protected $tagName = "";
```

```
    abstract public function startElement($parser, $name, $attributes);
    abstract public function endElement($parser, $name);
    abstract public function characterData($parser, $data);
    protected parse($xml) {
      xml_parse($this->parser, $xml);
    }
    public function __construct($xml) {
      $this->parser = xml_parser_create();
      xml_set_object($this->parser, $this);
      xml_parser_set_option($this->parser, XML_OPTION_CASE_FOLDING, 0);
      xml_set_element_handler($this->parser, "startElement",
                              "endElement");
      xml_set_character_data_handler($this->parser, "characterData");
    }

  }
  class PetParser extends SAXParser {
    public $petName;
    public $petId;
    public function startElement($parser, $name, $attributes) {
      if ($name == "Name") {
        $this->tagName = $name;
        $this->petId = $attributes['id'];
      }
    }
    public function characterData($parser, $data) {
      if ($this->tagName == "Name") {
        $this->petName = $data;
      }
    }
    Public function endElement($parser, $name) {
      $this->tagName = "";
    }
  }
```

Examining the Classes

Our abstract class is called SAXParser. Any parser classes we create will implement SAXParser. SAXParser has two properties: `tagName` and `parser`. `parser` is the XML parser that PHP creates with `xml_parser_create`. By putting it as an object, we can separate out the creation and setup of the parser from the actual XML parsing. We will see that later when we look at the constructor and the `parse` abstract method. `tagName` is a container that we use to store and inspect element names. When we are

interested in an element, the start element callback will store the name of the element in this property. When the character data callback detects a value in here, we know this is an element we are interested in, and can store the value somewhere. We will see this in action when we step through the parsing process.

SAXParser then declares the start element, end element, and character data callbacks as abstract methods. As these callbacks hold logic specific to the XML document, this is where the business logic separation occurs. The parser will still call the callbacks in the same way as the example code. Therefore, like the walkthrough code, we have to define these functions with the same method signature as usual.

We create another abstract method called `parse` that will fire off the XML parsing sequence. It takes one argument, the XML, and uses the object's parser to call `xml_parse`. The last function in SAXParser is the constructor.

The constructor of this object sets up the parser. It creates the parser, puts it in the `$parser` object property, sets options, sets handlers, and parses just like in the example. The only difference is that we have to tell PHP that the parser is being used in an object. We do it with `xml_set_object`.

```
xml_set_object($this->parser, $this);
```

This function sets the object context of the parser. The first argument is the parser. The second is to which object the parser belongs. As we are creating and using the parser in the same object, pass `$this` into the second parameter.

`xml_set_object` is absolutely critical when you are creating parsers in a PHP object. Otherwise, PHP will not be able to find the callback functions that you specified, and you will get an error.

The next class is PetParser, which is the document-specific implementation of our parser. All SAXParser implementers will follow the class' general structure. The properties of this class are the values that are of interest to us in the XML document. In this example, we are only interested in the name of the pet and the ID number, so we create properties for them.

The methods in this class are the abstract methods from SAXParser. We start with the start element callback method, `startElement`. When the PHP parser goes through the document, it will call `startElement`. This method is basically a series of `if` statements that examine the name of the element, represented by `$name`, that triggered the call. If it is an element that we are interested in, we store the name of the element, in SAXParser's `tagName` property. Next, when the parser calls the character data callback, the first thing it does is check the `tagName` property. If the value is something we are interested in, we store it in the property's method that we declared. Finally, the end element callback wipes the value of `tagName`.

Let's apply our example XML document to this walkthrough.

1. When the first element, `Pet`, is encountered, `startElement()` is called. However, there is no `if` statement to catch it, so `startElement()` executes without doing anything.
2. Next, `characterData()` is called when the parser reaches the end of `<Pet>` in the XML document.
3. It examines `tagName`, which is still empty, and does nothing.
4. `endElement()` sets `tagName` equal to a blank string, even though it already is blank.
5. Next, the PHP parser reaches the `Name` element. Again, `startElement()` is triggered. This time, however, an `if` statement catches `$name`. `$name` is stored in the `tagName` property.
6. As the `Name` element has an attribute that we are interested in, `id`, we also store the value of that attribute in the class' `petId` property. If we didn't care about any attributes, we could just close off the `if` statement.
7. This time, when `characterData()` is called, it sees that there is a value in `tagName`. More specifically, the `if` statement sees that we are inside the `Name` element.
8. This is a value we are interested in, so we store it in the class' `petName` property.
9. The end element callback is then called, which wipes `tagName`. This causes the whole process to be skipped if the next element encountered is one that we are not interested in.

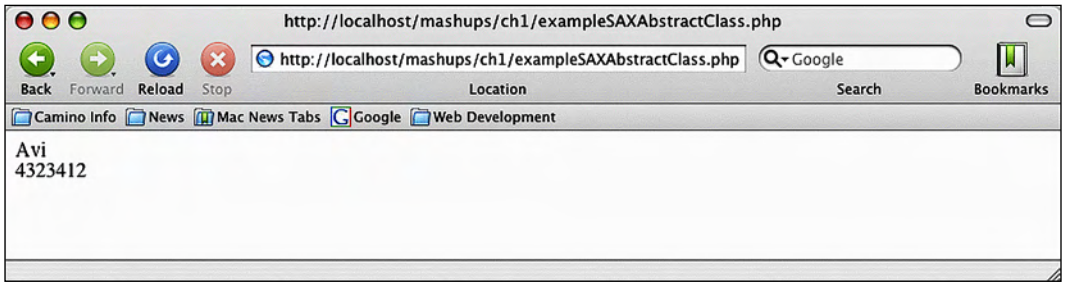
Using and Testing the Class

Using this class, we only need to do two things to extract data from an XML document. First, we feed the XML into the constructor. To actually use the data extracted, we access the properties of the implementing class. To see this in action, open the file named `exampleSAXAbstractClass.php` in the examples code. At the top of the file is some example usage code followed by the `SAXParser` and `PetParser` classes.

```
<?php
  $xml = "<Pet>" .
  "<Name id='4323412'>Avi</Name>" .
  "<Species>African Grey</Species>" .
  "<Gender>F</Gender>" .
  "</Pet>";
  $pp = new PetParser();
```

```
$pp->parse($xml);
echo $pp->petName . «<br />»;
echo $pp->petId;
abstract class SAXParser {
/* Rest of the SAXParser class and PetParser class */
```

Place the file on a servable area on your web server and pull up the page. You will see this in your web browser:



This example creates a new PetParser, the implementing class, and feeds it the example XML. As the execution code is mainly in PetParser's constructor, it does its thing and extracts the information that we're looking for, behind the scenes. We echo out that information by specifying PetParser's properties.

Using abstract classes and a little discipline, we have put an elegant implementation into a process that could have easily turned procedural. This design lets us reuse the code we need to create the parser and execute, scales easily to however many XML documents our application needs to access, and allows us to easily maintain the business logic code.

We now have a set of tools and know-how to create this mashup.

- We know how to use XML-RPC services, and created a generic utility class to make XML-RPC calls for us.
- We know how to use REST services, and created a generic utility class to make REST calls.

It's time to look at the APIs we are going to use for our mashup.

Internet UPC Database API

The Internet UPC Database is the first part of our mashup. This site, located at `http://www.upcdatabase.com`, is an interesting project. This site is a library of Universal Product Codes. Users can enter UPC numbers to see what the product

is, and they can contribute to the site by adding UPC numbers and product descriptions. There are commercial vendors that sell this information to people. However, being an open, collaborative project, the Internet UPC Database gives this information away for free. Other sites have arisen to compete with the Database, but I personally like this site because it's one of, if not the, largest and oldest sites of its kind, the content is explicitly issued under a **Creative Commons License** (Creative Commons Attribution-ShareAlike 2.5), and most importantly, the site offers a XML-RPC service to interact with its database.

The XML-RPC service is free and open. There is no need to get a developer's key or authenticate against. The API's home page is located at `http://www.upcdatabase.com/xmlrpc.asp`. To get a list of available functions, we simply request a call named `help` from the service. To use the XML-RPC class we created, we can use the following code:

```
$parser = new XMLRPCParser();
$returnedData = $parser->callService('help',
    null,
    'www.upcdatabase.com',
    '/rpc',
    80);
$parser->processXMLRPCResponse($returnedData);
```

A working version named `UPCDatabaseHelp.php` is in the examples code. You can place this file on your webserver. Hit the page to see the help file from the database. This will return a list of available calls, required parameters, and returned value from the service. This API is not very complex. There are only a handful of calls, and even a smaller subset is needed to do what we want it to do.

Help!

The Internet UPC Database's naming of `help` deviates from the norm a little bit. Instead of `help`, most XML-RPC services support a call named `system.listMethods`, which serves the same purpose. If you have trouble finding documentation for an XML-RPC service, try calling `system.listMethods` against the service.

There are two calls in this API that we should focus our attention on: `lookupEAN` and `lookupUPC`. One queries the database and returns product information based on EAN and one queries based on UPC.

Before we go further, we need to know a little bit about the "business logic" of what we are dealing with. We need to clarify the difference between a Universal Product Code, UPC, and an European Article Number, otherwise known as EAN. UPC is what most people are familiar with, especially in the United States. It is the twelve digit number underneath bar codes on products. EAN-13 is a newer version of the UPC. It has thirteen digits. EANs have been in use in Europe for years and have now been adopted world-wide. UPC is starting to disappear, but we will still encounter them in the real world, and the term serves as a generic reference for these types of numbers in the cultural lexicon. However, we will have to remember the technical difference later, when we are given a twelve-digit number by our user versus a thirteen-digit number.

Both return a struct to us, which `xmlrpc_decode` turns into an associative array. We can see what information is returned to us by calling the service with a known EAN.

```
$parser = new XMLRPCParser();
$returnedData = $parser->callService('lookupEAN',
    '0737628087501',
    'www.upcdatabase.com',
    '/rpc',
    80);
$parser->processXMLRPCResponse($returnedData);
```

Our generic echo code in `processXMLRPCResponse` will display the array contents in our browser.

Array Key	Example Value	Description
upc	"737628087501"	The UPC code of this product.
pendingUpdates	0	The number of pending updates to this product in the database, if any.
isCoupon	false (Boolean)	If this UPC/EAN is a coupon.
ean	"0737628087501"	The EAN of this product.
issuerCountryCode	"us"	The country code of the UPC/EAN
Description	"Thai Kitchen Noodle Cart Single Serve Size Thai Peanut Instant Rice Noodles & Sauce"	The product description.
found	true (Boolean)	If the entry was found in the database.
size	"2.25 oz (64 g)"	The size or weight of the product.

Array Key	Example Value	Description
message	"Database entry found"	A message response from the service.
issuerCountry	"United States"	Where this product comes from.
lastModified	"2006-04-17 21:42:26"	The last time this product was modified in the Database. A string, not an ISO8601 date.

If the UPC/EAN is valid, but not found in the Database, only two of the items are returned. `found` will be false, and `message` will be, "No database entry found". We can use either to test the results.

Note that the lookup returns both a UPC and a EAN. This is a subtle feature of the Internet UPC Database. For every search, it translates UPCs and EANs. This translation is crucial for our mashup. When we look closer at Amazon Web Services, we will see that it can grab a product directly with the EAN if you are searching any Amazon store other than Amazon US. If you are searching Amazon US, you must provide a UPC. EAN searches are not available for the US even though EANs are popping up throughout shelves in the United States. Conversely, if you wind up with a UPC number and want to search for it on Amazon UK, you must convert it to EAN.

The Internet UPC Database API is a simple API based on a simple web service protocol. Now let's take a look at a much more complex API, Amazon Web Services.

Amazon API

The Amazon Web Services (`http://www.amazon.com/aws`) is a whole family of web services for a variety of Amazon products. Among the products Amazon offers is a virtual computing service, a digital storage service, and a messaging queue service (and you thought they just sold DVDs and books…). Some of their services are fee-based like their historical pricing data, while others, like their Alexa Web Service, are free or free for a certain number of requests. Not surprisingly, Amazon has a free web service for their traditional ecommerce products like books and DVDs, called their Amazon E-Commerce Service (ECS).

Architecturally, the web services are available as either REST or SOAP. According to Amazon's Chief Web Services Evangelist, Jeff Barr, 85% of their web service developers use their REST services. Later, we will look at SOAP for another mashup, but for now, we will join that 85% and use the simpler REST service.

The ECS is what we will be looking at. Before we get started, you will need to sign up for an Access Key ID with Amazon. The ID itself is free and can be obtained by registering at `https://aws-portal.amazon.com/gp/aws/developer/registration/index.htm`. You will need to pass this key as one of your parameters of your Amazon request whenever you use any of Amazon's Web Services.

A Tour of ECS

The Amazon E-Commerce Service is really huge. Printed out, the developer documentation is over 450 pages long. ECS has a web service function, which they call **operations** (and we will keep this term in reference to them) for everything that you can do on the real Amazon.com. First and foremost, for more details on anything covered here, you can find the ECS documentation at `http://developer.amazonwebservices.com/connect/kbcategory.jspa?categoryID=5`. There is a whole collection of references, how-to guides, and best-practices papers collected there. The current and previous versions of the API documentation is located at `http://developer.amazonwebservices.com/connect/kbcategory.jspa?categoryID=19`. At the time of writing this book, the current version is dated August 30, 2007. You can quite literally build an Amazon clone using ECS. Entire books have been written about ECS, and they're quite lengthy, too. For our little mashup, we're going to focus on just a few features. However, it's worth taking a look at what other things you can do with ECS:

- **Searching For Products**
  ECS has an extensive search feature. Product searches are broken down into individual indexes. These indexes are for categories of products—books, DVDs, electronics, sporting goods, etc. There are also composite indexes that combine two or more indexes. For example, "video" is actually a combination of DVD and VHS indexes. For each search, there are a plethora of search parameters you can provide. Parameters like manufacturer, price range, and condition will allow you to fine tune a search.
- **Extensive Information About Products**
  Once you find the product you are looking for, you can retrieve an incredible amount of information about that product. Basic information about a product, like price, title, and description are just the beginning. Almost everything that you see on a product page on Amazon can be returned via ECS, for example, images, reviews, Listmania, sales rankings, and accessories. Albums can also return a list of tracks in the albums.

- **Shopping Carts Through Web Services**
  Shopping carts for ecommerce sites are not difficult to create, but they can be tedious. Luckily, ECS has a complete section to create your own Amazon shopping cart. Using ECS, you can create shopping carts, add and delete items, and pass it through to the checkout.
- **XSL Support**
  ECS's REST response is in XML format. The developer is required to properly process and present the data. This process is made easier by the support of XSLT. One of the parameters you can supply is the URL of an XSLT template. ECS will automatically apply the XSLT template to the XML results before it returns it to you.
- **Retrieve Customer Information**
  ECS has a special mode to search for customers by name or email address. No private information, like addresses or credit card information can be returned. However, you can get any public information, like reviews and wish lists.
- **Retrieve Seller Information**
  Conversely, you can get information about Amazon Marketplace Sellers or Merchants. Again, private information is not available, but you can retrieve any information that you see on Amazon.com. You can also retrieve a list of items they are currently selling.
- **Restaurant Information**
  Admittedly, this doesn't seem to fit into Amazon's traditional offerings. However, you can look up restaurant information through ECS. The number of cities is limited, but you can search by price ranges, neighborhoods, and cuisine.

Anatomy of an ECS REST Request

There are two things you need to know about making an ECS request. The first is the location of the service. The second are the parameters in each request. Each request comprises some basic global parameters enclose in brackets and parameters that are specific to the operation. We will take a look at the required global parameters in this section. Shortly, when we begin to mashup, we will see some of the operation-specific parameters in action, when we put the operations to use, and look them up in the ECS documentation.

Location of Service

The URL of the service we wish to call is entirely dependent on which Amazon location we want to use.

Amazon Locale	REST Service URL
United States	`http://webservices.amazon.com/onca/xml`
United Kingdom	`http://webservices.amazon.co.uk/onca/xml`
Germany	`http://webservices.amazon.de/onca/xml`
Japan	`http://webservices.amazon.co.jp/onca/xml`
France	`http://webservices.amazon.fr/onca/xml`
Canada	`http://webservices.amazon.ca/onca/xml`

Depending on which Amazon store you want to write your application against, these are the base URLs we will use. You will append your parameters to the end of this URL.

Each function will have its own set of parameters you need to pass. However, there are a few global parameters that apply to every ECS transaction. Some of these are required, some of these are optional. Before we muck with the innards of some ECS transactions, we should look at these global parameters:

Parameter Name	Valid Values	Description
Required Parameters		
Service	"AWSECommerceService"	Specifies that you want to use the ECS service.
SubscriptionID	Your Amazon Access Key ID	Identifies your application. This is the ID that you registered for earlier.
Operation	A valid operation name	Names the ECS operation you wish to perform.
Optional Parameters		
AssociateTag	Your Amazon Associate ID	Used to identify you for the Amazon Affiliate Program.
ResponseGroup	See Documentation	Controls what information is returned by the service. See an operation's documentation for allowed values.
Validate	"True"	For debugging purposes, used to validate if a request is valid and well structured without actually executing it.

For example, to search for MP3 Players on Amazon U.S., your REST request URL could look like this:

```
http://webservices.amazon.com/onca/xml
   ?Service=AWSECommerceService
   &SubscriptionID=(Your Key Here)
   &Operation=ItemSearch
   &AssociateTag=(Your Tag Here)
   &ResponseGroup=Large
   &searchIndex=Blended
   &Keywords=MP3%20Players
```

In this example, a search for items is triggered by the `ItemSearch` value for the `Operation` parameter. Everything else is required for all transactions, except the last two, `searchIndex` and `keywords`, which are required because of `ItemSearch`.

In its most basic form, mashups against ECS is basically a series of similarly-formatted REST requests made against ECS. The biggest challenge will be to navigate through all the available features.

Mashing Up

We now have a little more detail about how the Internet UPC Database and ECS work beyond just knowing that one is an XML-RPC service and one is a REST service. We can start mashing up.

Our mashup will use ECS and the UPC Database services to do the following:

1. A form will take the UPC, short UPC, or EAN to lookup the full UPC at Internet UPC Database.
2. Find the price and product information using Amazon's ECS. Display this on our site.
3. Offer the visitor the chance to buy this item.
4. If they wish to buy it, add the item to a shopping cart using ECS.
5. Pass the user to Amazon.com for checkout.

The following code listings are included in the example code, and, provided that you substitute your own Amazon Access Key, is a functioning version of the mashup. For the sake of brevity, I will include each entire script page and comment on them as we go.

Product Lookups

We will start with a simple form and action page. The form accepts a UPC code and has a pull-down menu. This pull-down menu contains a few categories available in Amazon.com, and just so happens that their values are SearchIndexes of Amazon ECS. This form is named `UPCForm.php` in the example code.

```
<html>
   <head>
      <title>UPC Lookup Form</title>
      <meta http-equiv="Content-Type" content="text/html;
                                        charset=utf-8" />
   </head>
   <body>
   <form action"UPCAction.php" method="pos">
    <p>
   Enter UPC/EAN Number Here: <input type"text" size="20"
                              name"numberSubmitted" /><br />
   Search Against:
   <select name="SearchIndex">
          <option value="Apparel">Apparel</option>
          <option value="Video">DVDs/VHS</option>
          <option value="Electronics">Electronics</option>
   </select>
   </p>
   <input type="submit" />
   </form>
   </body>
</html>
```

This form posts to a page called `UPCAction.php`.

```
<?php
  require_once('classes/XMLRPCParser.php');
  require_once('classes/RESTParser.php');
  require_once('classes/SAXParser.php');
  require_once('classes/AmazonSAXParser.php');
```

Here, we are simply including copies of the REST, SAX, and XML-RPC parsers that we created earlier and also an Amazon-specific implementation of the SAXParser class. I have organized them by placing them in a subdirectory called `classes`, but you can organize them however you see fit.

```
  $XMLRPCparser = new XMLRPCParser();
  $RESTParser = new RESTParser();
  $amazonXMLParser = new AmazonSAXParser();
  $submitted = trim($_POST['numberSubmitted']);
```

We do some object creation here. XML-RPC, REST, and Amazon XML parsers are created. The incoming UPC code is in a field named `numberSubmitted`. As we'll be checking and using this later, we set it to a local variable named `$submitted`.

```
switch ($submitted) {
  case strlen($submitted) == 12:
    $upcDbCall = "lookupUPC";
    break;
  case strlen($submitted) == 13:
    $upcDbCall = "lookupEAN";
    break;
  default:
    $upcDbCall = '';
}
```

We now set up our first call to the Internet UPC Database service. The service has two available calls. `lookupUPC` is used to look up product information based on UPC codes. `lookupEAN` is used to look up product information based on EAN. If a 12-digit code was submitted, we will assume that it was a UPC code and use `lookupUPC`. If a 13-digit code was submitted, we will assume that it was a EAN code and use `lookupEAN`. This `switch` statement detects the length of the submitted code and stores the name of the service that we will use in the variable `$upcDbCall`.

```
$returnedXMLRPC = $XMLRPCparser->callService($upcDbCall,
    $submitted,
    'www.upcdatabase.com',
    '/rpc',
    80);
```

After the setup, we call the UPC Database web service, passing the name of the call that we wish to use (`lookupEAN` or `lookupUPC`) and the submitted code. The rest of the parameters, domain, path, and port, are required values for the UPC Database service. If you recall, the `callService` method returns the values that the Database service in the form of an array. We catch this array in `$returnedXMLRPC`. On a successful lookup, one of the elements returned, regardless of whether we used `lookupEAN` or `lookupUPC`, is an element named `upc`, which is the UPC code of the item.

```
$restParams = array();
$restParams['Service'] = "AWSECommerceService";
$restParams['AWSAccessKeyId'] = 'Your Amazon Access ID Key';
$restParams['Operation'] = 'ItemLookup';
$restParams['XMLEscaping'] = 'Single';
$restParams['IdType'] = 'UPC';
$restParams['ResponseGroup'] = 'Large';
$restParams['ItemId'] = $returnedXMLRPC['upc'];
$restParams['SearchIndex'] = $_POST['SearchIndex'];
```

Next, we set up the call to Amazon ECS. The REST class we set up, takes an array and prepares a string of URL query parameters with it. These lines set the values for the ECS call. Let's take a closer look at some of these parameters.

The first important one is your Amazon Web Services Access Key ID. This is one of the global ones that need to be passed.

After that is the `Operation` parameter. The ECS API Reference's Operations section groups operations by those that are related to customer information, items, shopping carts, and seller information. In the items operations, you can either search for items given a set of criteria using the `ItemSearch` operation, or you can use a search key to look up a specific item using the `ItemLookup` operation. The latter will give you specific information about the product, like price and Amazon.com URLs. To use `ItemLookup`, though, you need a unique identifier, and a UPC code is sufficient. If you don't have an identifier, you generally would do an `ItemSearch` first, and then use that information for an `ItemLookup`. However, because we have a UPC code, we can directly call `ItemLookup`.

Now we have the operation, we can check the ECS API Reference Guide to see what other parameters we need besides the global ones. According to the API, to do an item lookup, we need to tell it what sort of unique identifier we're giving it in the `IdType` parameter. In our case, it is a UPC code. We need to pass the identifier in a parameter named `ItemId`, and we also need to tell it what `SearchIndex` to use. We found the UPC code from the call to the UPC Database service, so we use that to set `ItemId`. The `SearchIndex` was selected by the user in the form. Now this parameters array is set up, we can use it to call RESTParser's `callService()` method to execute the Amazon ECS.

```
$amazonResponse = $RESTParser->callService($restParams, 'webservices.
amazon.com', '/onca/xml', 'GET');
$amazonXMLParser->parse($amazonResponse);
```

We take the result of that and give it to our Amazon XML Parser. Remember the Amazon XML Parser is the parser specific to the XML response given by Amazon's ECS. We will take a look at how that parser is constructed, and the XML response of Amazon's `ItemLookup` after we look at this page. However, for now, as we walkthrough the rest of this page, just note the values of that XML response are stored as properties in the Amazon XML Parser class. We access these properties as we enter in the HTML section of the script.

```
?>
<html>
  <head><title>UPC Action</title></head>
  <body>
  <?php
  if (is_array($amazonXMLParser->errorMessage)) {
?>
```

After setting up the HTML `head` and `body` tags, we do a check to see if the Amazon response was an error. If it was, the `errorMessage` property of the Amazon XML Parser will be an array that holds the error messages returned by ECS, and an `is_array()` check will return true. This `if` statement essentially separates our HTML page into two sections. The top section will only execute and display if there was an ECS error, and the bottom section will only execute and display if the operation found our search product.

```
<h1>Error</h1>
There were problems with your lookup:
  <ul>
  <?php foreach ($amazonXMLParser->errorMessage as $key => $value) { ?>
    <li><?= $value ?></li>
  <?php } ?>
  </ul>
```

On an ECS error, loop through the `errorMessage` array in a `foreach` loop and display them in an HTML unordered list.

```
<?php } else { ?>
  <h1>Found!</h1>
  <img src="<?= $amazonXMLParser->imageURL ?>" />
  <p>
  Price: <?= $amazonXMLParser->formattedPrice ?>
  </p>
```

If there was not an error, the product was found. One of the values returned by ECS is a URL to the product image on Amazon. Another value returned is the Price.

```
  <a href="AmazonCartAdd.php?asin=<?= $amazonXMLParser->asin ?>">Add
  This To My Cart</a>
```

Another value returned is an Amazon Standard Identification Number. While UPC is fine for lookups, Amazon.com really likes an ASIN number for an identifier. This is Amazon's self-generated identifier, and many other ECS operations require an ASIN number. We will use this ASIN number to identify the item when we add it to the shopping cart.

```
<?php } ?>
  <a href="UPCForm.php">Search Again</a>
  </body>
  </html>
```

Finally, we close out the error check `if` statement and we close out the form. We add a link back to the form for our users' convenience.

Handling Amazon's XML Responses

The action page relied heavily on the AmazonSAXParser to parse the Amazon response. I asked you to take a leap of faith to just assume that the important response values were stored as values in the Amazon SAX parser class. Your patience and faith will be rewarded as we take a look at the Amazon ECS response and how the Amazon SAX Parser interacts with it.

An ECS Lookup Response

In the example application, the Amazon response is captured in the variable `$amazonResponse`. You can examine this variable to take a look at the entire Amazon ECS response. It is also included in the example code in the file named `AmazonLookupResponse.xml`. The response from a successful Amazon item lookup can be quite lengthy. In this section, let's take a look at some snippets from this response.

All sorts of information are returned. Everything you can see on a typical Amazon.com product page is returned in the XML response, and even a lot of information that you do not see. For example, while you only see one picture of the item on Amazon.com, there are actually three pictures of the item—small, medium, and large versions.

```
<SmallImage>
   <URL>http://ec1.images-amazon.com/images/P/B0001EMM0Q.01._
               SCTHUMBZZZ_.jpg</URL>
  <Height Units="pixels">60</Height>
  <Width Units="pixels">29</Width>
</SmallImage>
```

This is the returned chunk of XML that contains information for the small image. When we first looked at SAX, we used simple examples where the XML response was short and off the root. This response from Amazon is much more like what we will see in the real world—large with nesting. The `URL` element is what we are looking for. However, we can't grab just the `URL` element. This `URL` element is a child of `SmallImage`. However, other `URL` elements exist in the document. We need to grab this specific on under `SmallImage`.

Another possibility is that the lookup failed. In this case, we would get an error from Amazon. The elements and structure would be much different from a successful lookup, but the key to focus on is the `Errors` element.

```
<Errors>
 <Error>
    <Code>AWS.MissingParameters</Code>
    <Message>Your request is missing required parameters. Required
```

```
parameters include ItemId.</Message>
 </Error>
<Error>
<Code>AWS.MinimumParameterRequirement</Code>
<Message>Your request should have atleast 1 of the following
                  parameters: ASIN, OfferListingId.</Message>
</Error>
</Errors>
```

Each error that is triggered is a child of the `Errors` element. Within each error is a machine-friendly `Code` element and a human-readable `Message` element. This is another situation you will often encounter—child elements that may differ in number with each request. Not only are error codes a common application of this, but also search results via web services.

Let's take a look at how the Amazon SAX Parser class is structured, and how we tackle the two previous problems.

```
<?php
class AmazonSAXParser extends SAXParser {
  private $inError;
  private $inImage;
```

These two properties hold state information about where we are during the parsing process. The first asks if we are in an `Error` element, and the second if we are in a `SmallImage` element. These two properties are Booleans that are toggled during parsing, and are not accessible outside of the object.

```
  public $errorMessage;
  public $asin;
  public $imageURL;
  public $formattedPrice;
```

These next properties are the values in the XML document that we are interested in. They are the error messages, the ASIN code, the image URL, and a price which is formatted with a currency symbol and decimal points.

```
  public $cartId;
  public $hmac;
  public $purchaseURL;
```

These three are also properties that we are interested in, but they will be used later when we look at the shopping cart. I've grouped the properties in this class to keep the application simple. However, it is perfectly legal to place these two properties in its own separate class. This design would lead you down a road where each XML Response, not just each application, would have its own class. From a performance and code maintenance standpoint, though, this is certainly not a bad thing.

Now we start with the start element callback function.

```
public function startElement($parser, $name, $attributes) {
    if ($name == "Errors") {
       $this->tagName = $name;
       $this->inError = true;
       $this->errorMessage = array();
 }
```

Let's start with the error checking. When an element with the name `Errors` is encountered, we assume that it is indeed an Amazon error. We capture the name of the element, and set the `inError` flag to true. In the character data callback function, we will check this flag.

As there were one or more errors present, we initialize the `errorMessage` property as an array. An array gives us the most flexibility to store multiple element values. Back in the form action page, the `is_array` function in the `if` statement was checking to see if `errorMessage` was turned into an array here.

```
   if ($name == "Message") {
       $this->tagName = $name;
    }
    if ($name == "SmallImage") {
       $this->tagName = $name;
       $this->inImage = true;
    }
```

A similar method is used to capture the image URL. We are interested in the small image. When we enter the `SmallImage` element, we set the `inImage` flag to true.

```
    if ($name == "URL") {
       $this->tagName = $name;
    }
    if ($name == "ASIN") {
       $this->tagName = $name;
    }
    if ($name == "FormattedPrice") {
       $this->tagName = $name;
    }
    if ($name == "CartId") {
       $this->tagName = $name;
    }
    if ($name == "HMAC") {
       $this->tagName = $name;
    }
 }
```

The rest of these are elements of interest like in the SAX Parser example. We simply note that we are interested in them by storing their name.

```
public function characterData($parser, $data) {
    if ($this->inError == true && $this->tagName == "Message") {
       array_push($this->errorMessage, $data);
    }
```

We are now in the character data callback function. The first `if` statement captures an error message. The actual human readable message is stored in the `Message` element. The start element callback has already set the `inError` flag to true, and when it encountered the `Message` element, it stores its name. We now know that this is an error message. Thus, we place it in the `errorMessage` array.

```
    if ($this->tagName == "ASIN") {
       $this->asin = $data;
    }
    if ($this->inImage == true && $this->tagName == "URL") {
       $this->imageURL = $data;
    }
```

Similarly, when the small image URL is found, we check to see if we are in the `SmallImage` element, and then see if the tag name we are looking at is indeed the URL. If it is, store it in the `imageURL` property.

```
    if ($this->tagName == "FormattedPrice") {
       $this->formattedPrice = $data;
    }
    if ($this->tagName == "CartId") {
       $this->cartId = $data;
    }
    if ($this->tagName == "HMAC") {
       $this->hmac = $data;
    }
```

Store the rest of the element values we are interested in.

```
}
public function endElement($parser, $name) {
    if ($this->tagName == "Errors") {
       $inError = false;
    }
    if ($this->tagName == "SmallImage") {
       $inImage = false;
    }
```

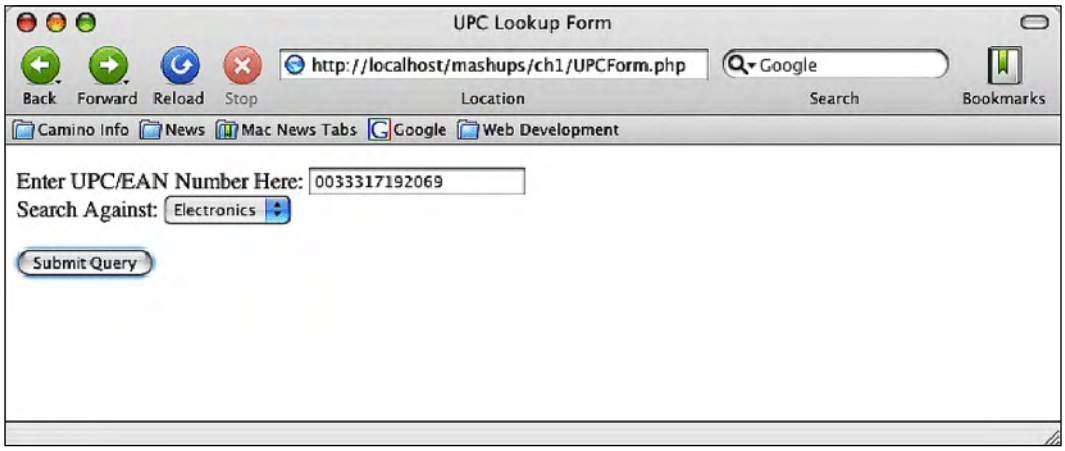
Once we encounter the end `Errors` and `SmallImage` element, we are no longer in any sort of special state. We clear these two flags here in `endElement`.

```
        $this->tagName = "";
```

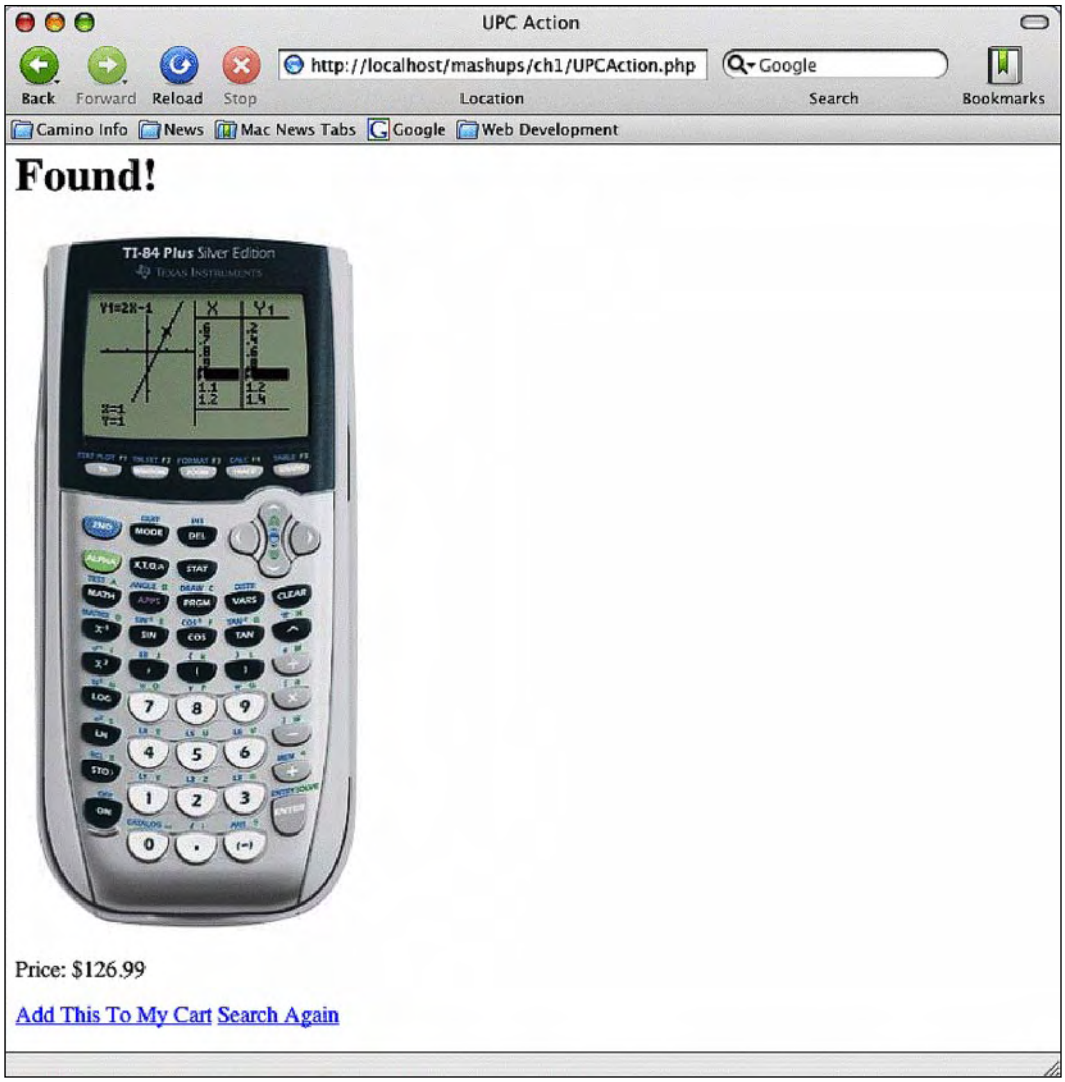
And we clear the `tagName` property to make sure the next element that we are not interested in is not captured.

```
    }
}
?>
```

Our form is ready to be used at this point. If you load `UPCForm.php` into your web browser, you can enter a UPC number and hit the submit button to see it in action.



If you enter a book's UPC number, they will probably not appear in the Internet UPC Database call results. Books usually use a special UPC number that begins with 978, followed by the ISBN number. This is a special subset of UPC and EAN numbers that the Internet UPC Database does not support.



Your Own Amazon Cart

Simply looking up the product is a good test of our mashup. We can fancy up the application a bit by allowing the user to actually purchase the item. In order to purchase an item, we need to create a shopping cart to store the items. Normally, we would have to either create our own shopping cart functionality or use a third party package that probably has more features than we need for this simple application. Luckily, Amazon ECS can manage a shopping cart for us with little coding.

There are five shopping cart operations in ECS:

- `CartAdd` to add items to the shopping cart
- `CartClear` to empty the cart
- `CartCreate` to create the cart
- `CartGet` to get the contents of the cart
- `CartModify` to change items in the cart

All of the operations are very easy to use and well documented. We'll use two of them—`CartCreate` and `CartAdd` to show a sample usage.

We will use these operations in the file named `AmazonCartAdd.php`. Using the `CartCreate` operation, this script will create the cart for us and add the item to the cart. If the user continues shopping, the script will skip the cart creation, and instead, use `CartAdd` to add the item to the existing cart. To remember the cart, ECS requires us to always pass two variables to the service. We will use sessions to hold these variables across the application.

```
<?php
   session_start();
   require_once('classes/SAXParser.php');
   require_once('classes/AmazonSAXParser.php');
   require_once('classes/RESTParser.php');
   $RESTParser = new RESTParser();
   $amazonXMLParser = new AmazonSAXParser();
   $restParams = array();
   $restParams['Service'] = “AWSECommerceService";
   $restParams['AWSAccessKeyId'] = 'Your Amazon Access ID Key';
   $restParams['XMLEscaping'] = 'Single';
   $restParams['Item.1.ASIN'] = $_GET['asin'];
   $restParams['Item.1.Quantity'] = '1';
?>
```

We begin the script just like before. We include some scripts and create some parser objects. There are a few key differences. The first is the `session_start` call to start a session. In our form action page earlier, we link to this page and pass the ASIN to this page to uniquely identify the product. We use this for the parameter `Item.1.ASIN`. This is the name of the parameter that tells ECS that our first item we are adding is an ASIN, and the number itself. Lastly, we pass another parameter, `Item.1.Quantity`, to tell ECS how many of the item we want to add.

```
<html>
<head></head>
<body>
```

```
<h1>Added</h1>
<?php
   if (!$_SESSION['cartId']) {
     $restParams['Operation'] = 'CartCreate';
     $response = $RESTParser->callService(
     $restParams, 'webservices.amazon.com', '/onca/xml', 'GET');
     $amazonXMLParser->parse($response);
     $_SESSION['cartId'] = $amazonXMLParser->cartId;
     $_SESSION['hmac'] = $amazonXMLParser->hmac;
?>
```

This first half is where the shopping cart is created. There are two variables we need to carry in the session to identify this user's cart—`cartId` and `hmac`. Both are generated by Amazon and returned when a cart is created. In this first session, we check to see if the cartID is already in session. If it is, we have stepped through this block before, and we can skip this.

Otherwise, we identify the `CartCreate` as the operation we wish to call. We then call this service. An XML response will be returned and parsed with our Amazon parser. Recall in the Amazon SAX Parser class that we were specifically looking for two elements, `CartId` and `HMAC`. These two are returned in the `CartCreate` operation's XML response. We take these returned values and store them in session.

```
<p>
   A shopping cart was created...
</p>
<p>
   The CartId is <?= $amazonXMLParser->cartId ?> <br />
   The HMAC is <?= $amazonXMLParser->hmac ?>
</p>
```

To verify this, we echo out what the parser found. This block will only be displayed to the user if a cart was indeed created.

```
<?php } else {
   $restParams['Operation'] = 'CartAdd';
   $restParams['CartId'] = $_SESSION['cartId'];
   $restParams['HMAC'] = $_SESSION['hmac'];
   $response = $RESTParser->callService(
   $restParams, 'webservices.amazon.com', '/onca/xml', 'GET');
   $amazonXMLParser->parse($response);
?>
```

The rest of this `if` block executes if a `CartId` is already in session. Here, we only need to add the cart item. We call `CartAdd` to achieve this.

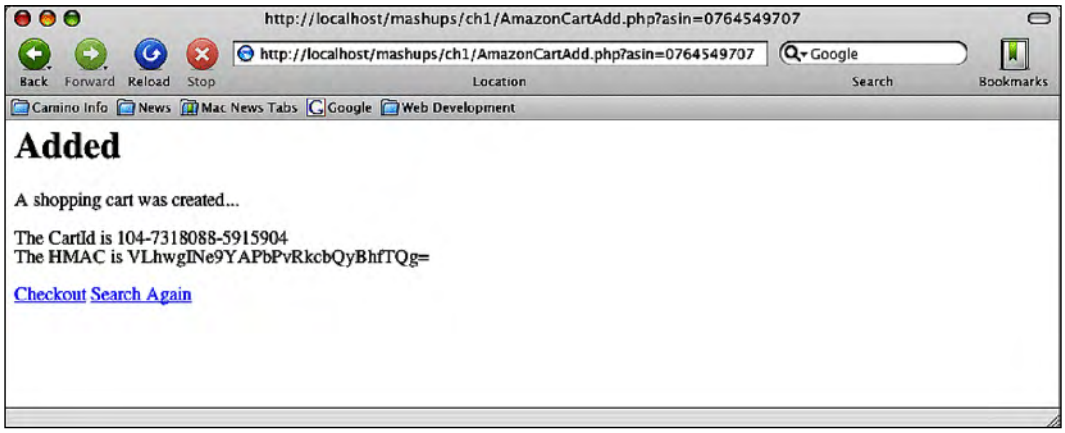
```
    <p>
       A shopping cart already exists, so one was not created...
    </p>
    <p>
       The CartId is <?=  $_SESSION['cartId'] ?> <br />
       The HMAC is <?=  $_SESSION['hmac'] ?>
    </p>
<?php } ?>
```

This section visually verifies the item add for the user.

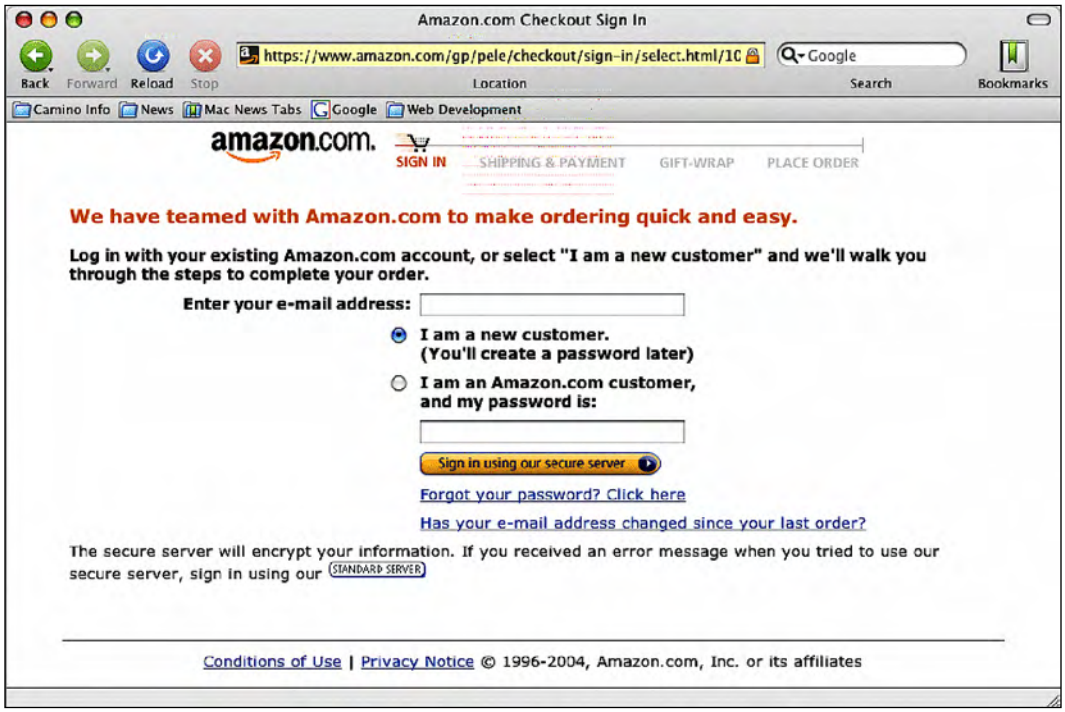
```
    <a href="<?= $amazonXMLParser->purchaseURL ?>">Checkout</a>
    <a href="UPCForm.php">Search Again</a>
    </p>
    </body>
</html>
```

Finally, the page is closed. Our SAX Parser captured an element named `PurchaseURL`. This `URL` is the checkout link for the cart. We include the link here at the bottom of the page. Clicking on the link will direct the user to the Amazon checkout process.

This is how the page will appear in our browser.



If the user clicks on the **Checkout** link, Amazon.com will take over the checkout and payment process.



Summary

We have taken a look at two web services to get our feet wet. The Internet UPC Database is an XML-RPC based service. While the ECS is available in both SOAP and REST flavors, we used the simpler and more popular REST tools. The mashup we created looked up product information from the Internet UPC Database and passed it to ECS. We found how much the product cost on Amazon.com and integrated ECS's shopping cart operations to create a shopping cart on our site. Finally, the user can purchase the items directly from Amazon.

In creating our mashup, we created a couple of object tools to call XML-RPC and REST services. REST usually returns a XML document to the caller. Using PHP's SAX function, we created an extensible object-oriented parser for XML. We will use these tools later in other projects.

XML-RPC and REST are the most simple web services. As we progress to later projects, we will see more complex protocols like SOAP.

3
Make Your Own Search Engine

Project Overview

What	Using web services from Microsoft, and Yahoo!, build your own search engine.
Protocols Used	REST, SOAP
Data Formats	XML, SOAP
Tools Featured	SoapClient
APIs Used	Microsoft Live Search API, Yahoo! Web API

At this point, we have a little bit of mashup experience and have utilized two simple, but popular, web services protocols. We're going to build on that knowledge in this chapter by creating a slightly more useful web application. We are also going to be introduced to SOAP, the third, and most complex of the currently fashionable web service protocols.

In this chapter, we are going to create our own search engine. In the past, this would require a massive amount of hardware resources, and complex search and spidering algorithms. Lucky for us, search engines like Google, Microsoft MSN, and Yahoo! have already done this for us. Even luckier for us, these sites have released web services for us to query their data centers and retrieve results. Our main advantage is that all three offer web APIs, so we can leverage the data of all three engines. Instead of just one set of results from one search engine, our application will query each engine and present the results to the user on one page. No longer will users have to visit these sites individually to search each engine.

In this project, we will use the APIs from Google, Microsoft's MSN, and Yahoo!. Yahoo! uses REST, which we already know about, to access its services. Microsoft Live Search is accessed through another web service called SOAP. Before we start building a search page, we will look at SOAP, and how PHP 5 interacts with it.

SOAP

SOAP, formerly known as Simple Object Access Protocol (until the acronym was dropped in version 1.2), came around shortly after XML-RPC was released. It was created by a group of developers with backing from Microsoft. Interestingly, the creator of XML-RPC, David Winer, was also one of the primary contributors to SOAP. Winer released XML-RPC before SOAP, when it became apparent to him that though SOAP was still a way away from being completed, there was an immediate need for some sort of web service protocol.

Like XML-RPC, SOAP is an XML-based web service protocol. SOAP, however, satisfies a lot of the shortcomings of XML-RPC: namely the lack of user-defined data types, better character set support, and rudimentary security. It is quite simply, a more powerful and flexible protocol than REST or XML-RPC. Unfortunately, sacrifices come with that power. SOAP is a much more complex and rigid protocol.

For example, even though SOAP can stand alone, it is much more useful when you use another XML-based standard, called Web Services Descriptor Language (WSDL), in conjunction with it. Therefore, in order to be proficient with SOAP, you should also be proficient with WSDL.

The most-levied criticism of SOAP is that it is overly complex. Indeed, SOAP is not simple. It is long and verbose. You need to know how namespaces work in XML. SOAP can rely heavily on other standards. This is true for most implementations of SOAP, including Microsoft Live Search, which we will be looking at. The most common external specifications used by a SOAP-based service is WSDL to describe its available services, and that, in turn, usually relies on XML Schema Data (XSD) to describe its data types. In order to "know" SOAP, it would be extremely useful to have some knowledge of WSDL and XSD. This will allow one to figure out how to use the majority of SOAP services.

We are going to take a "need to know" approach when looking at SOAP. Microsoft Live Search's SOAP API uses WSDL and XSD, so we will take a look at SOAP with the other two in mind. We will limit our discussion on how to gather information about the web service that you, as a web service consumer, would need and how to write SOAP requests using PHP 5 against it. Even though this chapter will just introduce you to the core necessities of SOAP, there is a lot of information and detail. SOAP is very meticulous and you have to keep track of a fair amount of things. Do not be discouraged, take notes if you have to, and be patient.

All three, SOAP, WSD, and XSD are maintained by the W3C. All three specifications are available for your perusal. The official SOAP specification is located at `http://www.w3.org/TR/soap/`. WSDL specification is located at `http://www.w3.org/TR/wsdl`. Finally, the recommended XSD specification can be found at `http://www.w3.org/XML/Schema`.

Web Services Descriptor Language (WSDL) With XML Schema Data (XSD)

Out of all the drawbacks of XML-RPC and REST, there is one that is prominent. Both of these protocols rely heavily on good documentation by the service provider in order to use them. Lacking this, you really do not know what operations are available to you, what parameters you need to pass in order to use them, and what you should expect to get back. Even worse, an XML-RPC or REST service may be poorly or inaccurately documented and give you inaccurate or unexpected results. SOAP addresses this by relying on another XML standard called WSDL to set the rules on which web service methods are available, how parameters should be passed, and what data type might be returned. A service's WSDL document, basically, is an XML version of the documentation. If a SOAP-based service is bound to a WSDL document, and most of them are, requests and responses must adhere to the rules set in the WSDL document, otherwise a fault will occur.

WSDL is an acronym for a technical language. When referring to a specific web service's WSDL document, people commonly refer to the document as "the WSDL" even though that is grammatically incorrect.

Being XML-based, this allows clients to automatically discover everything about the functionality of the web service. Human-readable documentation is technically not required for a SOAP service that uses a WSDL document, though it is still highly recommended. Let's take a look at the structure of a WSDL document and how we can use it to figure out what is available to us in a SOAP-based web service.

Out of all three specifications that we're going to look at in relationship to SOAP, WSDL is the most ethereal. Both supporters and detractors often call writing WSDL documents a black art. As we go through this, I will stress the main points and just briefly note other uses or exceptions.

Basic WSDL Structure

Beginning with a root `definitions` element, WSDL documents follow this basic structure:

```
<definitions>
   <types>
   …
   </types>
   <message>
   …
   </message>
   <portType>
   …
   </portType>
   <binding>
   …
   </binding>
</definitions>
```

As you can see, in addition to the `definitions` element, there are four main sections to a WSDL document: `types`, `message`, `portType`, and `binding`. Let's take a look at these in further detail.

Google used to provide a SOAP service for their web search engine. However, this service is now deprecated, and no new developer API keys are given out. This is unfortunate because the service was simple enough to learn SOAP quickly, but complex enough to get a thorough exposure to SOAP. Luckily, the service itself is still working and the WSDL is still available. As we go through WSDL elements, we will look at the Google SOAP Search WSDL and Microsoft Live Search API WSDL documents for examples. These are available at `http://api.google.com/GoogleSearch.wsdl` and `http://soap.search.msn.com/webservices.asmx?wsdl` respectively.

definitions Element

This is the root element of a WSDL document. If the WSDL relies on other specifications, their namespace declarations would be made here. Let's take a look at Google's WSDL's definition tag:

```
<definitions name="GoogleSearch"
   targetNamespace="urn:GoogleSearch"
   xmlns:typens="urn:GoogleSearch"
```

```
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns="http://schemas.xmlsoap.org/wsdl/">
```

The more common ones you'll run across are `xsd` for schema namespace, `wsdl` for the `WSDL` framework itself, and `soap` and `soapenc` for SOAP bindings. As these namespaces refer to W3C standards, you will run across them regardless of the web service implementation. Note that some searches use an equally common prefix, `xs`, for XML Schema. `tns` is another common namespace. It means "this namespace" and is a convention used to refer to the WSDL itself.

types Element

In a WSDL document, data types used by requests and responses need to be explicitly declared and defined. The textbook answer that you'll find is that the `types` element is where this is done. In theory, this is true. In practice, this is mostly true. The `types` element is used only for special data types.

To achieve platform neutrality, WSDL defaults to, and most implementations use, XSD to describe its data types. In XSD, many basic data types are already included and do not need to be declared.

Common Built-In XSD Data Types		
Time	Date	Boolean
String	Base64Binary	Float
Double	Integer	Byte

For a complete list, see the recommendation on XSD data types at `http://www.w3.org/TR/xmlschema-2/`.

If the web service utilizes nothing more than these built-in data types, there is no need to have special data types, and thus, `types` will be empty. So, the data types will just be referred to later, when we define the parameters.

There are three occasions where data types would be defined here:

1. If you want a special data type that is based on a built-in data type. Most commonly this is a built-in, whose value is restricted in some way. These are known as **simple types.**

2. If the data type is an object, it is known as a **complex type** in XSD, and must be declared.
3. An **array**, which can be described as a hybrid of the former two.

Let's take a look at some examples of what we will encounter in the `types` element.

Simple Type

Sometimes, you need to restrict or refine a value of a built-in data type. For example, in a hospital's patient database, it would be ludicrous to have the length of a field called `Age` to be more than three digits. To add such a restriction in the SOAP world, you would have to define `Age` here in the `types` section as a new type.

Simple types must be based on an existing built-in type. They cannot have children or properties like complex types. Generally, a simple type is defined with the `simpleType` element, the name as an attribute, followed by the restriction or definition. If the simple type is a restriction, the built-in data type that it is based on, is defined in the `base` attribute of the `restriction` element.

For example, a restriction for an age can look like this:

```
<xsd:simpleType name="Age">
   <xsd:restriction base="xsd:integer">
       <xsd:totalDigits value="3" />
   </xsd:restriction>
</xsd:simpleType>
```

Children elements of restriction define what is acceptable for the value. `totalDigits` is used to restrict a value based on the character length. A table of common restrictions follows:

Restriction	Use	Applicable In
enumeration	Specifies a list of acceptable values.	All except boolean
fractionDigits	Defines the number of decimal places allowed.	Integers
length	Defines the exact number of characters allowed.	Strings and all binaries
maxExclusive/ maxInclusive	Defines the maximum value allowed. If Exclusive is used, value cannot be equal to the definition. If Inclusive, can be equal to, but not greater than, this definition.	All numeric and dates
minLength/ maxLength	Defines the minimum and maximum number of characters or list items allowed.	Strings and all binaries

Restriction	Use	Applicable In
minExclusive/ minInclusive	Defines the minimum value allowed. If Exclusive is used, value cannot be equal to the definition. If Inclusive, can be equal to, but not less than, this definition.	All numeric and dates
pattern	A regular expression defining the allowed values.	All
totalDigits	Defines the maximum number of digits allowed.	Integers
whiteSpace	Defines how tabs, spaces, and line breaks are handled. Can be preserve (no changes), replace (tabs and line breaks are converted to spaces) or collapse (multiple spaces, tabs, and line breaks are converted to one space.	Strings and all binaries

A practical example of a restriction can be found in the MSN Search Web Service WSDL. Look at the section that defines `SafeSearchOptions`.

```
<xsd:simpleType name="SafeSearchOptions">
   <xsd:restriction base="xsd:string">
       <xsd:enumeration value="Moderate" />
       <xsd:enumeration value="Strict" />
       <xsd:enumeration value="Off" />
   </xsd:restriction>
</xsd:simpleType>
```

In this example, the `SafeSearchOptions` data type is based on a string data type. Unlike a regular string, however, the value that `SafeSearchOptions` takes is restricted by the `restriction` element. In this case, the several enumeration elements that follow. `SafeSearchOptions` can only be what is given in this enumeration list. That is, `SafeSearchOptions` can only have a value of `"Moderate"`, `"Strict"`, or `"Off"`.

Restrictions are not the only reason to use a simple type. There can also be two other elements in place of restrictions. The first is a list. If an element is a list, it means that the value passed to it is a list of space-separated values. A list is defined with the `list` element followed by an attribute named `itemType`, which defines the allowed data type. For example, this example specifies an attribute named `listOfValues`, which comprises all integers.

```
<xsd:simpleType name="listOfValues">
    <xsd:list itemType="xsd:integer" />
</xsd:simpleType>
```

The second is a union. Unions are basically a combination of two or more restrictions. This gives you a greater ability to fine-tune the allowed value. Back to our age example, if our service was for a hospital's pediatrics ward that admits only those under 18 years old, we can restrict the value with a union.

```
<xsd:simpleType name="Age">
   <xsd:union>
       <xsd:simpleType>
          <xsd:restriction base="decimal">
                <xsd:minInclusive value="0" />
          </xsd:restriction>
       </xsd:simpleType>
       <xsd:simpleType>
          <xsd:restriction base="decimal">
                <xsd:maxExclusive value="18" />
          </xsd:restriction>
       </xsd:simpleType>
   </xsd:union>
</xsd:simpleType>
```

Finally, it is important to note that while simple types are, especially in the case of WSDLs, used mainly in the definition of elements, they can be used anywhere that requires the definition of a number. For example, you may sometimes see an attribute being defined and a simple type structure being used to restrict the value.

Complex Type

Generically, a complex type is anything that can have multiple elements or attributes. This is opposed to a simple type, which can have only one element. A complex type is represented by the element `complexType` in the WSDL. The most common use for complex types is as a carrier for objects in SOAP transactions. In other words, to pass an object to a SOAP service, it needs to be serialized into an XSD complex type in the message.

The purpose of a `complexType` element is to explicitly define what other data types make up the complex type. Let's take a look at a piece of Google's WSDL for an example:

```
<xsd:complexType name="ResultElement">
   <xsd:all>
       <xsd:element name="summary" type="xsd:string"/>
       <xsd:element name="URL" type="xsd:string"/>
       <xsd:element name="snippet" type="xsd:string"/>
       <xsd:element name="title" type="xsd:string"/>
       <xsd:element name="cachedSize" type="xsd:string"/>
```

```
        <xsd:element name=
                  "relatedInformationPresent" type="xsd:boolean"/>
        <xsd:element name="hostName" type="xsd:string"/>
        <xsd:element name=
                  "directoryCategory" type="typens:DirectoryCategory"/>
        <xsd:element name="directoryTitle" type="xsd:string"/>
     </xsd:all>
  </xsd:complexType>
```

First thing to notice is how the `xsd:` namespace is used throughout `types`. This denotes that these elements and attributes are part of the XSD specification.

In this example, a data type called `ResultElement` is defined. We don't exactly know what it is used for right now, but we know that it exists. An `element` tag denotes complex type's equivalent to an object property. The first property of it is `summary`, and the `type` attribute tells us that it is a string, as are most properties of `ResultElement`. One exception is `relatedInformationPresent`, which is a Boolean. Another exception is `directoryCategory`. This has a data type of `DirectoryCategory`. The namespace used in the type attribute is `typens`. This tells us that it is not an XSD data type. To find out what it is, we'll have to look for the namespace declaration that declared `typens`.

Namespace definitions are usually at the top root element tag. Looking there, we find our namespace is indeed defined:

```
<definitions name="GoogleSearch"
             targetNamespace="urn:GoogleSearch"
             xmlns:typens="urn:GoogleSearch"
```

The name of the root element is `GoogleSearch`, and that is the target of `typens`. The value of the declaration is this document itself. Therefore, `DirectoryCategory` must be defined elsewhere in this document.

Looking further down the WSDL, we find the definition:

```
<xsd:complexType name="DirectoryCategory">
    <xsd:all>
       <xsd:element name="fullViewableName" type="xsd:string"/>
       <xsd:element name="specialEncoding" type="xsd:string"/>
    </xsd:all>
</xsd:complexType>
```

`DirectoryCategory` appears to be another complex type. Two string elements comprise the object.

The point of looking at this is to understand that basically anything can be a property of the main class. Most properties of objects will be XSD built in data types, but it is perfectly legal to hold other complex types as properties.

Arrays

The last common data type that you'll encounter is an array. Arrays in WSDL are a little unusual. Up to this point, WSDL has been using XSD to define data types. XSD is primarily used to define a document structure, unlike WSDL, which is used to define a network transport payload.

In the latter, passing arrays is crucial, while in the former, it is less so. Not entirely surprising then, that in XSD, declaring an array is not a straightforward and easy thing to do.

To keep the writing of WSDL as simple as possible, WSDL drops the use of XSD when declaring arrays, and instead, uses SOAP's array structures to define its own arrays. An array is declared in WSDL by creating a complex type and restricting it (using the same `restriction` element found in simple types) to the SOAP array data type. After that, the data type of each element of the array is declared.

In this example, an array of integers is declared.

```
<xsd:complexType name="ArrayOfInteger">
   <xsd:complexContent>
      <xsd:restriction base="soapenc:Array">
         <xsd:attribute ref="soapenc:arrayType" wsdl:
                              arrayType="integer[]"/>
      </xsd:restriction>
   </xsd:complexContent>
</xsd:complexType>
```

The declaration begins with the `complexType` tag. We name the array with the name attribute. WSDL convention states that the name of arrays should be in the format of `"ArrayOfxxxx"` where `xxxx` is the type of items in the array, be they one of the built-in data types or other specialized types defined here in the `types` element.

Remember, in types, we are merely defining the available data types in the web service. We are not tying them to any operation just yet. If the web service has more than one operation that uses arrays of integers, the operations definitions later will just reuse this one array. Therefore, it is perfectly legal to have a generic name like `ArrayOfInteger` or `ArrayOfDate`, etc.

A tag named `complexContent` is the first and only child. Then we begin with the restriction. This is where the divergence from XSD takes place. First, note the `base` attribute. Unlike the previous restriction tags we've seen, this one does not have the `xsd:` namespace. Instead, as WSDL uses SOAP encoding for arrays, the `base` attribute now uses the `soapenc:` namespace followed by the SOAP structure named `Array`.

Next is an XSD attribute to specify the encoding and a WSDL attribute named `arrayType` that defines the data type. The data type is followed by open and close brackets. This example uses integers. An array of strings would have `string[]`, and an array of objects would have the name of the complex type followed by the brackets.

The data types are probably the most important things you will need to understand from the WSDL apart from the actual operations themselves, which are defined later. This section lets us know what data format we need to pass into operation parameters and what we can expect back.

This has been just an overview of how simple type elements are structured. For a reference of every built-in data type and restrictions supported, see the official XSD data type recommendation at `http://www.w3.org/TR/xmlschema-2/.`

message Element

This is the second of the four children of the `definitions` root element. In this element, we gather up the data types and bundle them together to prepare them for use later in `portType`, where we actually define the available service operations. `Message` is merely a layer of abstraction between the data types and `portType`. Think of the items in `types` as data types and `message` assigns these data types to parameters. Later, we'll assign these parameters to actual operations in `portType`.

The format of messages depends on what is used in the SOAP binding, later on in the `binding` element. The binding also affects how we create the SOAP message later when we are actually writing the SOAP message body. We will see this in more detail later, however, for now, know that the `style` attribute in the SOAP binding element can be either `rpc` or `Document`.

RPC Binding

If the binding is `rpc`, which stands for Remote Procedure Call, like in XML-RPC, messages are merely an element named `message`, each with one or more `part` elements. Each part is the name of an item in `types`.

The Google WSDL provides us with another good example:

```
<message name="doGoogleSearch">
   <part name="key"    type="xsd:string"/>
   <part name="q" type="xsd:string"/>
…
   <part name="oe" type="xsd:string"/>
</message>
```

The WSDL defines a message called `doGoogleSearch`. It's made up of approximately 10 part elements. These part elements take two attributes `name`, which defines the parameter name, and `type`, which is the data type. In `doGoogleSearch`, a part named `key` is a string. Another named `q` is also a string, and so forth.

```
<message name="doGoogleSearchResponse">
   <part name="return"  type="typens:GoogleSearchResult"/>
</message>
```

Another message named `doGoogleSearchResponse` is defined. The parameter returned is named `return` and it is a `GoogleSearchResult` object. In `doGoogleSearch`, the data types were regular XSD built-in types, they are not mentioned in `types`. However, `GoogleSearchResult` is obviously not an XSD standard data type, so we would have to look back in `types` to get the object definition.

Document Binding

Another common value for the `binding` elment is `document`. By fortunate chance, the Microsoft Live Search API gives us an example of document binding. We can use the messages in this WSDL to see the difference:

```
<wsdl:message name="SearchMessage">
    <wsdl:part name="parameters" element="tns:Search" />
</wsdl:message>
```

Here, the message element is the same as the `rpc` version. The `part` element and `name` parameters are also present. However, instead of a `type` attribute, there is an `element` attribute. This is the operation name that we will call later on when we write our SOAP request.

How do we know what parameters `Search` needs? We look back up to the types section. In here, we find a data type of element that has the same name as the element in `SearchMessage`:

```
<xsd:element name="Search">
    <xsd:complexType>
       <xsd:sequence>
          <xsd:element minOccurs="1" maxOccurs="1"
```

```
                name="Request" type="tns:SearchRequest" />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
```

In it, there are some data rules, but this element points to another data type called `SearchRequest`. We look again in the data types for `SearchRequest`:

```
    <xsd:complexType name="SearchRequest">
        <xsd:sequence>
            <xsd:element minOccurs="1" maxOccurs="1"
                name="AppID" type="xsd:string" />
            <xsd:element minOccurs="1" maxOccurs="1"
                name="Query" type="xsd:string" />
            <xsd:element minOccurs="1" maxOccurs="1"
                name="CultureInfo" type="xsd:string" />
                    …
        </xsd:sequence>
    </xsd:complexType>
```

Finally, we see that `SearchRequest` is a complex type, and we see which elements make up this object, and hence, which parameters the Search operation will need.

This type of document hopping is all too common in the WSDL world. Fortunately, the basic elements and theory of WSDL have a bit of logic and common sense behind them. It may take a bit of diligence, but you can eventually find the operation name and parameters in a SOAP-based service.

portType Element

Finally, we get to the definition of the actual web service operations. This is done in the `portType` element. Think of operations as the actual functions available in a web service. A `portType` element is the parent element for a group of operations. `portType` is merely a way to categorize sets of operations. For example, a web service may have one group of operations used solely by partner sites and another group of services used only be customers. The group of operations used by partners may be under a single `portType` element named `PartnerOperations` while operations used by customers are grouped under another `portType` named `CustomerOperations`. Most simple web services, though, will just have one `portType` element. This is true of the Google and Microsoft Live Search APIs we will look at.

Each operation is defined with the `operation` element. Each operation can be either of the following four types:

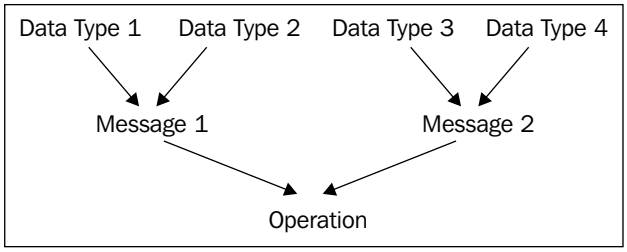
1. **One-way**: The client sends an input message to the server.
2. **Request-Response**: The client sends an input message to the server. The server responds with an output message.
3. **Solicit-Response**: The server sends an output message to the client. The client responds with an input message.
4. **Notification**: The server sends an output message to the client.

Out of the four, the vast majority used in web services is the Request-Response method. Even if the web service has an operation that just takes an input to manipulate data on the server, best practices state that a service should send a success or failure response message back to the client. This operation type defines the necessary children elements for the `operation` element. For Request-Reponses, both an `input` element specifying the message to be used must be defined, and an `output` element specifying the associated message needs to be included. Let's take another look at the Google SOAP Search API WSDL.

```
<portType name="GoogleSearchPort">
    <operation name="doGetCachedPage">
       <input message="typens:doGetCachedPage"/>
       <output message="typens:doGetCachedPageResponse"/>
    </operation>
   ...
</portType>
```

The first operation defined is `doGetCachedPage`. It takes an input message of `doGetCachedPage` and returns a `doGetCachedPageResponse` message as the output. This tells us that to do a `doGetCachedPage` operation against the Google SOAP Search service, we need to pass whatever parameters are specified in the `doGetCachedPage` message. If successful, we will get data back that is in the form defined in the `doGetCachedPageResponse` message.

Think of the relationship as "types make up messages, and messages make up operations".



binding Element

The last element in a WSDL we should note is the binding element. In WSDL, these are extensions to SOAP. This element ties the operations defined in portType to specific SOAP actions. Unless you are making your own SOAP client or writing a WSDL document, you do not need too much detail about this. However, it is nice to know what exactly is going on here.

The binding from WSDL portType to SOAP actions is declared in the type attribute of the binding element. The value of this should point back to a name of a portType element.

```
<binding name="GoogleSearchBinding" type="typens:GoogleSearchPort">
    <soap:binding style="rpc" transport=
                            "http://schemas.xmlsoap.org/soap/http"/>
    <operation name="doGetCachedPage">
      <soap:operation soapAction="urn:GoogleSearchAction"/>
      <input>
        <soap:body use="encoded" namespace="urn:GoogleSearch"
           encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
      <output>
        <soap:body use="encoded" namespace="urn:GoogleSearch"
           encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </output>
```

The children elements are basically metadata details for the SOAP transactions. We talked about the `soap:binding` element when we described messages. The style attribute here is where rpc or document is set.

The important thing you should pay attention to are the `operation` elements. These elements expose the portType operations to SOAP by mapping them directly to a SOAP operation. If the portType operations are not listed here, they will not be available to the caller. Underneath this, can be body elements. These elements provide SOAP-specific metadata to the operation.

The `use` attribute here is either `encoded` or `literal`. In combination with the binding's style attribute, this determines how the messages section and SOAP body elements are created.

The only thing you may run across in the course of making a mashup, especially with proprietary data, is the use of SOAP headers. SOAP headers often contain information about the transaction itself. One of the common uses for headers is authentication data. You may have to supply some credentials to the service before it fulfills your request. In the WSDL, a header requirement is defined in the `input` element here in the bindings. Like the messages definition, it will state the data type required with the `part` attribute.

```
<input>
     <soap:header message="tns:submitPassword" part=
            "xsd:string" use="literal"/>
<soap:body
```

Then, you can trace back up the WSDL to the message section to find out the exact element name you need to pass in the SOAP header.

```
<message name="submitPassword">
<part name="password_header" element="Password" />
```

The element attribute is the key. It states the name of the element you need to pass in the SOAP header. In this example, this service expects a SOAP header with an element of `Password` that is a string to be passed with the SOAP message.

We won't run across headers in the Google and Microsoft Live Search APIs, but be aware of them if you do run across one in the WSDL.

service Element

Lastly, we come to the `service` element. This element gives us the specific location of where the SOAP action point is for each port. Each port will have its own element here, followed by the SOAP address tag that points back to the service. Google's service element looks like this:

```
<service name="GoogleSearchService">
    <port name="GoogleSearchPort" binding="typens:
    GoogleSearchBinding">
       <soap:address location="http://api.google.com/search/beta2"/>
</port>
</service>
```

The main thing this tells us is that all operations that happen in `GoogleSearchPort` occur at the URL `http://api.google.com/search/beta2`. We may need to use this information later if we hit a service directly.

The SOAP Message

Being able to decipher a WSDL gives us the rules that we need to write a SOAP message to a service. We can now call operations against a service, pass parameters that the service needs in the data type that it expects, and prepare for a response from the server.

The structure of a SOAP message is fairly straightforward. Further, SOAP utilizes the same structure for request and responses. An element named `Envelope` is the root element for the whole message. Within that, an element named `Header` holds routing data for the message. A `Body` element holds the message content. This may be the parameters we pass when we make a request, or the service results when we get a response. Finally, a `Fault` element gives information on any errors that occurred during execution. Structurally, a SOAP message looks like this:

```
<?xml version="1.0"?>
<soap:Envelope
   xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
   soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
   <soap:Header>
       ...
    </soap:Header>
    <soap:Body>
   ...
        <soap:Fault>
...
       </soap:Fault>
     </soap:Body>
</soap:Envelope>
```

Again, we will not take too detailed a look into this. Luckily for us, PHP's SoapClient, which we will investigate later, hides a lot of SOAP details for us. However, knowing how a SOAP request is structured is essential to using the SOAP client and troubleshooting.

Envelope

The `Envelope` element identifies the XML document as a SOAP message. There are two things it must do. First, and absolutely essential, is to declare the SOAP namespace. Second, the encoding style must be set in the document. Both are done through attributes of the `Envelope` element.

```
<soap:Envelope
   xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
   soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
```

If there are other namespaces that the SOAP messages uses, they must be declared here also. For example, a proper request to Google SOAP Search API would need to declare XSD data types when we define parameters. In order to do this, we would declare the XSD data types in a namespace here in the Envelope element.

```
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:ns1="urn:GoogleSearch"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
   SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
```

Header

If the service requires any headers, they should be added in the SOAP header element. If headers are required, they will be noted in the documentation, or explicitly stated in the WSDL. SOAP headers are specific to the web service implementation. Therefore, each one needs to be namespace qualified.

In the previous password example, the service is expecting an element named `password` that is a string to be passed in the `Header`. We place this in the SOAP header like so:

```
<soapenv:Header>
   <mysoap:password xmlns:mysoap="http://yourserviceURL">
password
</xmlns:mysoap>
</soapenv:Header>
```

If the service does not require headers, we can omit the `Header` element. In the SOAP specifications, this element is optional.

Body

Finally, we get to the body of the SOAP message. Here is where we pass the request to the server and all the required parameters. What is included in this section and how it's structured is dictated entirely by the web service. Most notably, we refer back to the SOAP binding element in the WSDL for the exact structure of the body.

In most public web service cases, the schema is rather simple. The name of the operation you wish to call is a child element of the `Body` tag. Underneath that, the parameters are nested as elements.

In the most basic form, a body element will look like this:

```
<soapenv:Body>
    <nameOfOperationToBeCalled>
       <parameterOne>Parameter One</parameterOne>
       <parameterTwo>23.39</parameterTwo>
       <parameterThree>true</parameterThree>
    </nameOfOperationToBeCalled>
</soapenv:Body>
```

Let's see how the SOAP binding affects the creation of the messages.

RPC Binding

RPC binding needs the data types of the parameter passed in each parameter. We can look at how a request to the Google SOAP Search API is made:

```
<soap-env:Body>
   <ns1:doGoogleSearch>
       <key xsi:type="xsd:string">Your Google License Key</key>
       <q xsi:type="xsd:string">Orange Tabbies</q>
       <start xsi:type="xsd:int">0</start>
       <maxResults xsi:type="xsd:int">10</maxResults>
        …
   </ns1:doGoogleSearch>
</soap-env:Body>
```

The data type is declared in the `type` attribute. In Google's implementation, it calls upon the XSD and XSDI standards for the data types. These namespaces were declared back in the `envelope` element.

Document Binding

In document binding, the data types are not required to be part of the body. We can look at how to make a call against the Microsoft Live Search service to see how that works.

```
<soap-env:Body>
<ns1:Search>
    <ns1:Request>
       <ns1:AppID>Your MSN Search API Key</ns1:AppID>
       <ns1:Query>Orange Tabbies</ns1:Query>
       <ns1:CultureInfo>en-US</ns1:CultureInfo>
       …
    </ns1:Request>
</ns1:Search>
</soap-env:Body>
```

This looks very similar to the RPC binding version.`ns1` is the Microsoft schema namespace, so that is required for all elements. Like RPC, the operation name serves as the parent element within `Body`. Each parameter name is a child element with the parameter value set as the value of the element. This time, though, there are no data types attributes.

Fault

Error reporting is standardized in SOAP through the `Fault` element. This element is passed by the server if there was a problem with the request. It always appears as a child of the `Body` element. Obviously, we won't have to deal with writing a fault when consuming web services. However, knowing about what comes back when something goes wrong is crucial to troubleshooting.

There are four children of `Fault`:

Child Element	Description
faultcode	One of four codes to identify the fault.
faultstring	A descriptive explanation of what went wrong.
faultactor	Where the fault occurred.
detail	A container for any application-specific error information about the body element.

In SOAP 1.1, there are four possible fault codes:

faultcode	Description
VersionMismatch	Problem with the SOAP namespace declaration in the Envelope element.
MustUnderstand	A MustUnderstand attribute in the Header was not understood.
Client	The error is related to the client-end. The request was misformed.
Server	There was a server processing error. The request could not continue.

Now that we have some working knowledge of SOAP, WSDL, and XSD, we can put this to use to start writing PHP code.

PHP's SoapClient

Knowing the intricacies of SOAP, WSDL, and XSD is very helpful. However, coding every little detail is a headache. For requests, we'd have to extensively manipulate and parse an XML document. Quite frankly, SOAP comes with a lot of overhead. We'd have to create the `Envelope` and `Header` by hand, and manually create the message body. When data is returned, we'd have to create our own parser to loop through an XML-based SOAP document. We also haven't even mentioned creating our own sockets to talk to the server. That's a lot of things that can go wrong. Fortunately, PHP 5 has a great interface for talking with SOAP. Much of the dirty details are hidden away and completed by the client. We don't have to manually touch the request and response at all. In fact, we really don't need to do anything in XML. Further, the ugly networking connections are executed for us automatically.

PHP 5's SOAP interface is a built-in extension called SOAP. This extension actually comprises six classes. SoapClient is the class we will use to make SOAP calls. We will also take a look at SoapFault to handle errors. SoapServer is used for PHP SOAP providers. We won't be using this for mashups, but if you wish to offer SOAP services in the future, this is the class you want to use. The three other classes, SoapHeader, SoapParam, and SoapVar, are used primarily by SoapClient and SoapServer. Unless you are mucking in the really low levels of SOAP and networking, you will not fiddle with these.

The documentation to the SOAP extension is located at `http://php.net/manual/ref.soap.php`.

The steps for using SoapClient to make a SOAP request is very simple and straightforward. Assuming you know something about the service you are hitting, the process is, at the most, four steps:

1. If parameters are required, place them into an array. Subobjects and subarrays will have to be nested.
2. Instantiate the SoapClient object. SoapClient supports two modes of operation—WSDL and Non-WSDL. We will discuss the difference between these two modes soon, but for now, know that the way the client is instantiated decides on which mode is used.
3. Make the call to the service using the methods available to us in SoapClient.
4. Handle the request or error. This involves capturing the results and manipulating them as necessary.

We will walk through these four steps to create a working SOAP client. XMethods (`www.xmethods.com`) is a site that lists freely available web services to test against. One operation takes in a URL and returns back an array of hyperlinks used on the site. As we create our client, we will use this service to test.

Creating Parameters

As we discussed, parameters are dictated by the WSDL. With a lot of web services, there will be some form of documentation about available operations and required parameters. However, in this, and many other cases, there is no documentation. We will have to work through a WSDL.

The WSDL is located at `http://webservices.daelab.net/temperatureconversions/TemperatureConversions.wso?WSDL`. A good strategy is to start in the binding element and work your way up the chain to get the operation name, then the required parameter names.

Looking at the binding, there appears to be an operation that will be helpful to us:

```
<binding name="TemperatureConversionsSoapBinding"
    type="tns:TemperatureConversionsSoapType">
        <soap:binding style="document" transport=
                              "http://schemas.xmlsoap.org/soap/http"/>
        <operation name="CelciusToFahrenheit">
```

The operation name, `CelciusToFahrenheit` appears to be what we want.

Looking further up the WSDL, we see this chunk:

```
<portType name="TemperatureConversionsSoapType">
    <operation name="CelciusToFahrenheit">
      <documentation>Converts a Celcius Temperature to a Fahrenheit
                                                  value</documentation>
      <input message="tns:CelciusToFahrenheitSoapRequest"/
```

The documentation element confirms this is what we are looking for. The last line in this chunk is the input tag. The message attribute points to a message named `CelciusToFahrenheitSoapRequest`. Let's take a look at this message:

```
<message name="CelciusToFahrenheitSoapRequest">
    <part name="parameters" element="tns:CelciusToFahrenheit"/>
</message>
```

This points us towards the `types` element. We'll look for a data type named `CelciusToFahrenheit` in there.

```
<xs:element name="CelciusToFahrenheit">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="nCelcius" type="xs:decimal"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

Finally, we reach the exact name and type of the parameter we need to pass. We need to pass a decimal named `nCelcius` to an operation named `CelciusToFahrenheit`, in order to get the data we want.

This parameter requirement is quite simple, but common. All parameters that need to be passed by SoapClient need to be put into an associative array.

Create a PHP file on your web server that can be served. To create the parameters for this request, create an array with one element in it. The key is `nCelcius` while the value is the temperature you wish to convert.

```
$params = array('nCelcius' => 37);
```

Instantiate the SoapClient

The SoapClient is an object and all calls are made through methods to this object, so we need to first instantiate it. Before we do this, we need to decide if we are going to use WSDL mode or non-WSDL mode for our SoapClient.

In WSDL mode, SoapClient gets the WSDL file from the remote server and automatically figures out which operation is available on the service, which parameters are needed, and how they should be constructed. There are also a few methods available to us in WSDL mode that allow us to query the service. For example, `__getFunctions` returns a list of SOAP operations available for the service, and `__getTypes` returns a list of SOAP data types used.

In non-WSDL mode, we specify the actual location of the web service, not the WSDL, in arguments when we instantiate. We also specify the namespace of the document at creation time. We have to be a bit more careful later when we create the actual request.

Given that WSDL will do a lot of things automatically, why would anyone ever use non-WSDL? Performance is one reason.

In non-WSDL, you do not have to make the request for a WSDL document and you do not have the parsing overhead. Control is another reason. In non-WSDL mode, for some advanced operations, you may be required to handle more minute tasks. Some people like this.

Non-WSDL mode also gives you access to a couple of instantiation options that are not available otherwise. The main reason, though, is that you simply might not have any choice. The WSDL file might not be publicly available to you. To make your life easier, I recommend using WSDL mode whenever possible. However, working in non-WSDL mode is certainly acceptable and not uncommon in practice. We will cover how to make calls using both, and later, in the actual mashup, we will use non-WSDL for one of our calls.

Instantiating in WSDL Mode

The decision on whether a SoapClient is WSDL or non-WSDL is dependent on how you initially instantiate SoapClient. Let's take a look at the constructor for the class:

```
class SoapClient {
__construct ( mixed wsdl [, array options] )
}
```

The first parameter passed to it is a WSDL. This usually is a simple URI to the WSDL file, although the mixed type allows you to pass a WSDL file stored in a string into this parameter. The second parameter is an optional array of options.

If you want to go into WSDL mode, simply pass a WSDL into the first argument.

```
$client = new SoapClient(
'http://webservices.daelab.net/temperatureconversions/
TemperatureConversions.wso?WSDL'
);
```

All further operations with this object will be in WSDL mode.

Instantiating in Non-WSDL Mode

Creating a non-WSDL client is a tad bit more complicated. First, the `wsdl` parameter is irrelevant. You must pass a null value to this option. This also means that your client won't automatically know where the service action point is. You will have to tell it manually. To do this, set the location and the target namespace of the service in the options array when you instantiate. In other words, this optional parameter no longer is optional. This is done with the location and options elements of the array, respectively.

```
$client = new SoapClient(
   null,
   array(.
      'location' => 'http://webservices.daelab.net/
      temperatureconversions/TemperatureConversions.wso',
      'uri' => 'http://webservices.daelab.net/temperature'
   )
);
```

Where did we get the location value of the URL? This is grabbed from the `service` element. It is the child address `element` for the port. The `uri` value is also taken from the WSDL, but at the very top. It is the `targetNamespace` value of the definitions element. In this case, they are the same value, but in many implementations they are not.

Using Other Options

You may want to take a look at the options array documentation, although most of the options are not needed, especially in WSDL mode, in the majority of basic calls. They are all listed in the SoapClient constructor page at `http://php.net/manual/function.soap-soapclient-construct.php`. One particular option, `trace`, is quite useful. If set to true, this will allow us to examine the actual SOAP envelope later on, and do general debugging. Still, it is nice to know about the others, especially if you run across a weird communication issue with the SOAP service. The solution might be a setting in the options array.

Either of these options will give us a new SoapClient object named `$client`. We can now use the methods of the object.

Making the Call and Using SoapClient Methods

Before we make the SOAP call, let's take a look at some of the methods available to us. If you are using WSDL mode, two methods will be especially useful, `__getFunctions` and `__getTypes`.

Remember that PHP has already queried the WSDL when we created the SoapClient object. It doesn't just make a mental note of the WSDL. It reads it and makes our life easier by translating the operations into available methods within the SoapClient object itself. (We'll see how this affects the way we make SOAP calls in a little bit.) In doing so, it obviously holds a list of available functions and data types.

To access these lists, use `__getFunctions` against the SoapClient to return an array of all SOAP operations, and their parameters, available in the WSDL. `__getTypes` returns an array of all data types used in the WSDL. These are extremely useful if documentation is not available.

We can see this in action against the URL Extraction service:

```
<?php
  $client = new SoapClient(
  'http://webservices.daelab.net/temperatureconversions/
                        TemperatureConversions.wso?WSDL'
  );
  ?>
  <h1>Types:</h1>
  <?= var_dump($client->__getTypes()); ?>
  <h1>Functions:</h1>
<?= var_dump($client->getFunctions()); ?>
```

This code is in the examples as `SoapClientTest.php`. Running the page will give us this output in the page source:

```
<h1>Types:</h1>
array(8) {
  [0]=>
  string(49) "struct CelciusToFahrenheit {
 decimal nCelcius;
}"
  [1]=>
  string(74) "struct CelciusToFahrenheitResponse {
 decimal CelciusToFahrenheitResult;
}"
  [2]=>
  string(52) "struct FahrenheitToCelcius {
 decimal nFahrenheit;
}"
…
<h1>Functions:</h1>
array(4) {
  [0]=>
  string(80) "CelciusToFahrenheitResponse CelciusToFahrenheit(CelciusT
                                          oFahrenheit $parameters)"
  [1]=>
  string(80) "FahrenheitToCelciusResponse FahrenheitToCelcius(Fahrenhe
                                          itToCelcius $parameters)"
  [2]=>
```

```
    string(77) "WindChillInCelciusResponse WindChillInCelcius(
                                    WindChillInCelcius $parameters)"
    [3]=>
    string(86) "WindChillInFahrenheitResponse WindChillInFahrenheit(
                                    WindChillInFahrenheit $parameters)"
}
```

The format of the method signatures is very C-style. It uses object hinting for inputs and returns. The very first function listed is `CelciusToFahrenheit`. It takes in an `CelciusToFahrenheit` object.

Looking at the types, we see that `CelciusToFahrenheit` is a struct of only one element, a decimal called `nCelcius`. Like we found in the WSDL, this tells us we need to pass a parameter called `nCelcius` to the `CelciusToFahrenheit` operation.

The return is an `CelciusToFahrenheitResponse` object. Looking at the types, we trace that object back to a struct named `CelciusToFahrenheitResponse`. That's a struct that gives us this definition:

```
string(74) "struct CelciusToFahrenheitResponse {
 decimal CelciusToFahrenheitResult;
```

Finally, we see that `CelciusToFahrenheit` is basically a decimal named `CelciusToFahrenheitResult`. You can see if there are more details in the WSDL. Alternatively, as you already know which parameters are required, you can just make the SOAP call and use SoapClient to examine the results.

How you make the WSDL call depends on whether you are in WSDL mode or Non-WSDL mode. This is the only grammatical area, besides access to certain functions, which varies depending on WSDL or Non-WSDL.

Calling SOAP Operations in WSDL Mode

In WSDL mode, PHP has internalized the operations and made them an extension of the SoapClient object. Their use is extraordinarily easy. You treat these operations as if they were methods of the SoapClient object.

```
$params = array('nCelcius' => 37);
$client = new SoapClient(
'http://webservices.daelab.net/temperatureconversions/
TemperatureConversions.wso?WSDL',
array('trace' => true)
);
$result = $client->CelciusToFahrenheit($params);
```

Calling SOAP Operations in Non-WSDL Mode

In Non-WSDL mode, SoapClient has no idea what methods are available. To handle this, SoapClient holds an operation calling method called `__soapCall`. This method's sole existence is to launch SOAP calls against a service, and it takes two parameters. The first is the name of the SOAP operation. The second is an array of the SOAP parameters you wish to pass to the service. Rewriting the above call in Non-WSDL mode, we get this:

```
$params = array('nCelcius' => 37);
$client = new SoapClient(
null,
array(
'location' => 'http://webservices.daelab.net/temperatureconversions/
                    TemperatureConversions.wso',
    'uri' => 'http://webservices.daelab.net/temperature
     )
);
$result = $client->__soapCall('CelciusToFahrenheit', $params);
```

Remember we need the `location` and `uri` options when instantiating SoapClient to go to non-WSDL mode, and we pass null to the first parameter. After that, we can use `__soapCall` to make the call to the operation.

Either of these two methods will place the SOAP response into a variable named `$result`. Now, we can use this to see what the server returned to us.

Handling the SOAP Response

Before we start displaying the result, we need to see if it was even successful.

Handling SOAP Errors with SoapFault

SoapClient leverages the new try/catch blocks in PHP 5 to handle errors. If something went wrong, either from our end or if the server returned a fault, SoapClient throws a SoapFault object. We need to catch this and handle the error gracefully. In the following example, I purposely changed the method name to `GetURLs`. Because this method doesn't exist, it will fail.

```
try {
    $result = $client-> ConvertTemp($params);
    }
catch (SoapFault $e) {
   echo "Error!<br>";
```

```
    echo "faultcode: " . $e->faultcode . «<br>»;
    echo «faultstring: « .$e->faultstring. «<br>»;
    echo «detail: « .$e->detail;
}
```

The best way to do this is to wrap the call to the operation in the try block. If that fails, the catch block will catch the SoapFault and place it in an object, in this case, `$e`. SoapFault's documentation is at `http://php.net/manual/function.soap-soapfault-construct.php`. It has no methods besides the constructor. It does have several parameters. These parameters directly correlate to the children element of SOAP's `Faults` element we talked about earlier. Therefore, using the SoapFault object, we can examine which SOAP message the server sent us.

Running the above block will generate this message on the browser:

```
Error!
faultcode: Client
faultstring: Function ("ConvertTemp") is not a valid method for this
service
detail:
```

Of course, displaying the error is not your only option. Your business rules may dictate that your application does something else behind the scenes on an error.

Handling Successful Results

Hopefully, though, our query was successful. If this is the case, SoapClient will take the incoming response SOAP message, extract the data elements, place it in a PHP data structure, and return it to the variable that was assigned to the SOAP call.

Unfortunately, this is both a blessing and a curse. The blessing is that once you understand the data structure created by that particular SOAP response, it is very easy to manipulate and very powerful. The curse is getting to the point of understanding the data structure.

With XML, it is very easy to understand elements and child elements. Even long XML documents are easy to read because of the nice structure. PHP has to shoe horn this into a system that includes objects, primitive data types, arrays of primitive data types, arrays of objects, arrays of objects that hold other arrays, and so forth. Needless to say, it can get quite messy. While SoapClient does a superb job of mapping SOAP data types to PHP data types, it takes a little bit of patience for a human to figure out what decisions SoapClient made.

Fortunately, we have some clues to help us. The first clue is documentation. If you don't live in a perfect world, though, you can turn to the trusty WSDL document. Looking back at the `portType` for this operation, we see that there is an output message.

```
<output message="tns:CelciusToFahrenheitSoapResponse"/>
```

We can then do the upward dance again back to the `message` and `types` elements.

```
<message name="CelciusToFahrenheitSoapResponse">
    <part name="parameters" element=
                                    "tns:CelciusToFahrenheitResponse"/>
  </message>
…
<xs:element name="CelciusToFahrenheitResponse">
    <xs:complexType>
        <xs:sequence>
            <xs:element name=
                        "CelciusToFahrenheitResult" type="xs:decimal"/>
        </xs:sequence>
     </xs:complexType>
</xs:element>
```

This tells us we should expect an object named `CelciusToFahrenheitResponse`. Within that is a decimal named `CelciusToFahrenheitResult`.

Our next clue is the PHP variable that holds the response. In our previous examples, we stored this in a variable named `$results`. We can pass this through `var_dump` to examine how SoapClient mapped the data.

Passing it through `var_dump` will yield this result:

```
object(stdClass)#2 (1) {
    ["CelciusToFahrenheitResult"]=>string(4) "98.6"
}
```

Working our way from the top line down, we see the first line tells us that this whole variable is an object. The first property is named `CelciusToFahrenheitResult`, and it is a string. Even though the WSDL says this should be a decimal, SoapClient will turn this into a string because PHP is typeless. To access the Fahrenheit value, we simply access the object property:

```
$result->CelciusToFahrenheitResult ->
```

We can now take everything and put it all together. In this example, let's convert 37 degrees Celcius to Fahrenheit. Our script will execute the SOAP call, check for an error, and output the result.

```
<?php

$params = array('nCelcius' => 37);

$client = new SoapClient(
'http://webservices.daelab.net/temperatureconversions/
TemperatureConversions.wso?WSDL',
array('trace' => true)
);

try {
    $result = $client->CelciusToFahrenheit($params);
} catch (SoapFault $e) {
    echo "Error!<br>";
    echo "faultcode: " . $e->faultcode . "<br>";
    echo "faultstring: " .$e->faultstring. "<br>";
    echo "detail: " .$e->detail;
}
?>

<html>.
<head><title>Soap Test</title></head>
<body>
<?php if ($result) { ?>
    Fahrenheit: <?= $result->CelciusToFahrenheitResult ?>
<?php } else { ?>
    Problem with conversion.
<?php } ?>

</body>
</html>
```

This script is named `SoapClientTest2.php`. Running it on our web browser will show us this:

Now we have all the background knowledge we need to make SOAP calls and use the Microsoft Live Search service. Combine this with what we already know about REST, and we can start to create our mashup. Before we start writing code, as every web service implementation is different, we need to take a look at each service to see what we're dealing with.

Microsoft Live Search Web Service

The Microsoft Live Search Web Service is an easy to use SOAP service. These are a few key highlights:

- MSN Search Web Service only has one operation, which is named Search. A cached version of the page is available, as part of the Search result.
- MSN Search has a search request limit of 20,000 requests per day.
- There are 250 maximum results returned with each search.

The home page for MSN Search Web Service is at `http://msdn.microsoft.com/live/msnsearch/default.aspx`. Extensive documentation, sample code (in .NET), and developer forums are available. You will also need a license key, which they call an **AppID**, for Live Search. You can get that at `http://search.msn.com/developer`.

Using Search

Microsoft Live Search Web Service organizes all of its related operations and parameters into objects. This is a bit of a challenge for us because the parameters argument for SoapClient's SOAP calls must be an array. SoapClient then serializes it into a format the service likes. We will have to construct an array that is more than two dimensions to accommodate this. However, we must be careful when constructing this initial array.

On the surface, Search requires only one parameter—a `SearchRequest` object. Translated into PHP array terms, this is already one array level.

This object has several properties:

Property	Description	Default if Optional
AppID	Your generated AppID.	*Required*
Query	The query term you are searching for.	*Required*
CultureInfo	Language and locale information. See the Microsoft Live Search Web Service reference for a complete list of codes.	*Required*
SafeSearch	A setting level for filtering adult content. Value can be Strict, Moderate, or Off.	Moderate
Flags	Controls whether query words are marked when they are returned. Value can be either None or MarkQueryWords.	None
Location	Controls location data, if applicable, for the search. See the Microsoft Live Search Web Service reference for applicability.	*Optional, No default*
Requests	An array of SourceRequest objects giving you further control of how the results are returned.	*Required*

The SourceRequest properties are also well documented:

Property	Description	Default if Optional
Source	An enumerated list of possible source searches. See the Microsoft Live Search Web Service reference for a complete list, but this value should be set to Web.	*Optional, No default*
Offset	With zero being the most relevant, this is the first result you want to be returned back. If for some reason you only want moderately relevant responses, pick a higher number.	0
Count	Specifies the number of results to return.	10
ResultFields	An enumerated list of fields to return in the search. See the Microsoft Live Search Web Service reference for a complete list. To see all, set this value to All.	All

The Requests property is an array of objects, so we need to set two array levels to that. A complete parameter for a Microsoft Live Search, then looks as follows:

```
$params = array('Request' => array(
                'AppID' => 'Your Microsoft Live Search AppID',
                'Query' => 'Orange Tabby Kittens',
```

```
                'CultureInfo' => 'en-US',
                'SafeSearch' => 'Strict',
                'Flags' => '',
                'Location' => '',
                'Requests' => array(
                      'SourceRequest' => array(
                            'Source' => 'Web',
                            'Offset' => 0,
                            'Count' => 50,
                            'ResultFields' => 'All')
                                  )
                               )
            );
```

Again, all fields are required, but they can be blank. The `Requests` nesting makes it a little confusing to construct, but SoapClient handles the request admirably. After this is setup, we can go ahead and pass it to SoapClient and make the call to Search. This time, I will demonstrate the use of WSDL mode. We will wrap the call in a try-catch block to check for errors.

```
$client = new SoapClient(
   'http://soap.search.msn.com/webservices.asmx?wsdl',
    array('trace' => true);
                         );
try{
     $result = $client->Search($params);
   }
  catch(SoapFault $e) {
   echo "Error!<br>";
   echo "faultcode: " . $e->faultcode . «<br>»;
   echo «faultstring: « .$e->faultstring. «<br>»;
   echo «detail: « .$e->detail;
                      }
```

When it comes back, we can break down the `$result` variable to see how we're going to parse the response. Looking at `var_dump` results and documentation, we see the `$result` is an object of the `Response` class. There is a property that is an object named `Responses`. `Responses` has a property named `SourceResponse`. `SourceResponse` holds information about the query results. The actual query results are in an array named `Results`, which is one of the properties of `SourceResponse`. The elements of the `Results` array are objects named Result. The properties of the `Result` object are the Title, Description, Actual URL, Display URL, and a URL to a cached version.

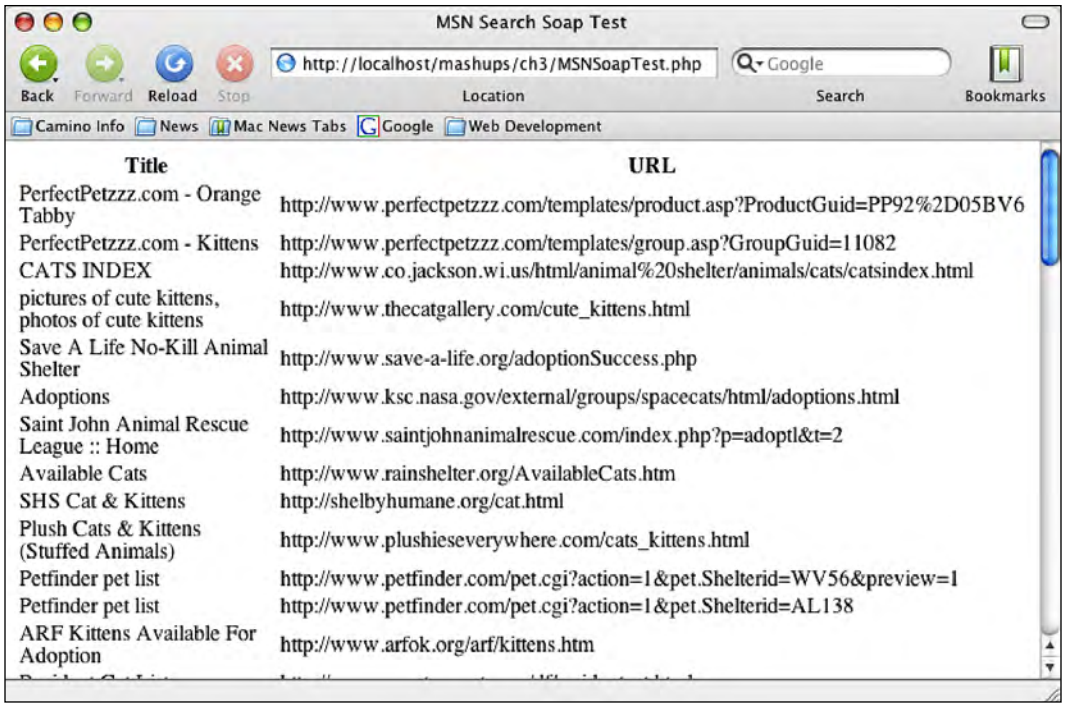
This, then, is the chain to the `Results` object:

```
$result->Response->Responses->SourceResponse->Results->Result
```

The example file named `MSNSoapTest.php` demonstrates this. This script executes a SOAP-based query against the Microsoft Live Search Web Service and parses the results with the Microsoft Live Search `response` structure.

```
<?php if ($result) { ?>
<table>
<tr>
<th>Title</th>
<th>URL</th>
</tr>
    <?php foreach ($result->Response->Responses->SourceResponse
                    ->Results->Result as $key => $value) { ?>
       <tr>
       <td><?= $value->Title ?></td>
       <td><?= $value->Url ?></td>
       </tr>
    <?php } ?>
</table>
<?php } ?>
```

And this will appear in our browser like so:



Title	URL
PerfectPetzzz.com - Orange Tabby	http://www.perfectpetzzz.com/templates/product.asp?ProductGuid=PP92%2D05BV6
PerfectPetzzz.com - Kittens	http://www.perfectpetzzz.com/templates/group.asp?GroupGuid=11082
CATS INDEX	http://www.co.jackson.wi.us/html/animal%20shelter/animals/cats/catsindex.html
pictures of cute kittens, photos of cute kittens	http://www.thecatgallery.com/cute_kittens.html
Save A Life No-Kill Animal Shelter	http://www.save-a-life.org/adoptionSuccess.php
Adoptions	http://www.ksc.nasa.gov/external/groups/spacecats/html/adoptions.html
Saint John Animal Rescue League :: Home	http://www.saintjohnanimalrescue.com/index.php?p=adoptl&t=2
Available Cats	http://www.rainshelter.org/AvailableCats.htm
SHS Cat & Kittens	http://shelbyhumane.org/cat.html
Plush Cats & Kittens (Stuffed Animals)	http://www.plushieseverywhere.com/cats_kittens.html
Petfinder pet list	http://www.petfinder.com/pet.cgi?action=1&pet.Shelterid=WV56&preview=1
Petfinder pet list	http://www.petfinder.com/pet.cgi?action=1&pet.Shelterid=AL138
ARF Kittens Available For Adoption	http://www.arfok.org/arf/kittens.htm

Yahoo! Search Web Service

The second search service we will look at is the Yahoo! Search Web Service. Unlike the other Microsoft Live Search, this service is REST based. We will utilize what we learned in the previous chapter to access this service. Still, we need to examine this web service to see what parameters we need to pass, and what we can expect back.

The home page of this web service is located at `http://developer.yahoo.com/search/`. Just like the other two, the service is supported by excellent documentation, a forum community, and sample codes. Out of all three services, Yahoo!'s sample code is far more extensive and provides examples in PHP, Perl, Python, Java, JavaScript, and Flash. Again, like all three, you will have to download an identifier key. You can get this key at `http://api.search.yahoo.com/webservices/register_application`.

Yahoo! Search Web Service is actually a whole family of web search services. Among the other services available are audio search, image search, and news search. The Web Search interface is the more traditional World Wide Web search engine, and that is what we will be concentrating on.

Yahoo! Search Web Service offers some key differences from Microsoft Live Search:

1. The REST based service is a blessing in that it is quite easy to use, and integrates well with more technologies including client side ones like Flash and JavaScript. It's also a curse because we won't have the advantages of things like WSDL.

2. Yahoo! Search Web Service gives you 5,000 queries per 24-hour period.

Using Web Search

The location of the web service is at `http://search.yahooapis.com/WebSearchService/V1/webSearch`. There are a number of parameters that we will have to include:

Parameter Name	Description
appid	Your application ID.
query	The query term you are searching for.
region	The region that you are searching against.
type	The type of search you wish to return. "all" for the whole query submitted, "any" returns results all results with any of the search terms. Phrase considers the query to be a phrase and will treat it as such.

Parameter Name	Description
results	Maximum number of results to return. Maximum value is 100.
start	Number of result you wish to start at.
format	The kind of file to look for.
adult_ok	Flag to indicate whether you wish to filter adult material.
similar_ok	Flag to indicate whether you wish to include duplicate results.
language	The language of the response.
country	The country you wish to restrict the search to.
site	The domain you wish to restrict the search to.
subscription	Special container for premium, fee-based content.
license	The Creative Commons license you wish to apply to the search results.
output	The format to return the search. Can be XML (default), JSON, or serialized PHP.
callback	JavaScript callback function to call.

Of particular importance is the `Output` flag. Yahoo! Web Services can pass the result back as serialized PHP, so we can just pass the return value to `unserialize()`. We do not have to pass the results through a SAX parser.

Because this service is REST, we have a bit more flexibility for our parameters. If we want to accept a default value, we can simply omit it from the parameters. A simple parameters array for these values may look like this:

```
$params = array('appid' => 'Your Yahoo! App ID',
                'query' => 'Orange Tabby Kittens',
                'region' => 'us',
                'type' => 'all',
                'results' => 50,
                'start' => 0,
                'format' => 'any',
                'adult_ok' => 1,
                'similiar_ok' => 0,
                'output' => 'php'
               );
```

We are going to leverage the serialized PHP functionality by setting the `output` parameter to `php`.

To make the call, we will use the RESTParser class we created in the last chapter, instantiate the class, and make the REST call.

```
require_once('classes/RESTParser.php');
$parser = new RESTParser();

$t = $parser->callService($params, 'search.yahooapis.com', '/
WebSearchService/V1/webSearch', 'GET');
$result = unserialize($t);
```

Note the last line. We pass the result of the REST call through `unserialize()`. Now the result is easily manipulated through PHP.

The PHP structure of the returned value is different than that of Microsoft Live Search. The response does not use objects, mainly because that concept does not exist in REST, but is common in SOAP. Instead, Yahoo! uses arrays extensively.

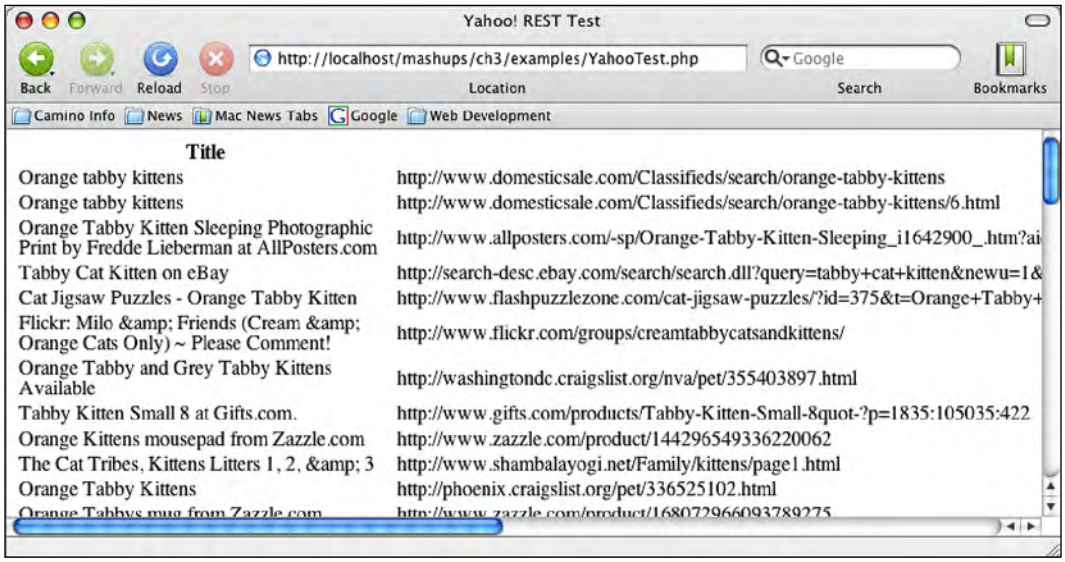
To access the values of the service return, we must go through `ResultSet`, which is an array of `Result` objects, then we must loop through `Result` itself because that is an array that holds the returned objects. For example, to go from the top level to `Result` objects, we would use this chain:

```
result['ResultSet']['Result']
```

For each individual `Result` object returned, we would have to access those array elements. We can demonstrate using the sample `YahooTest.php` page:

```
<?php if ($result) { ?>
<table>
<tr>
<th>Title</th>
<th>URL</th>
</tr>
   <?php foreach ($result['ResultSet']['Result'] as $key => $value) {
?>
       <tr>
       <td><?= $value['Title'] ?></td>
       <td><?= $value['Url'] ?></td>
       </tr>
   <?php } ?>
</table>
<?php } ?>
```

Note how we access the title and URL as array elements, not objects. Running this through the browser will give us the results from Yahoo!



Mashing Up

Now is the time to put it all together. We've looked at SOAP, which allowed us to access the Microsoft Live Search Web Service. We then looked at each individual service and saw how they accept and return data. Let's now put it all together under one common interface.

Let's start with a front-end form to accept queries.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en_US" xml:lang="en_
US">
 <head>
  <title>Mashups Chapter 2</title>
 </head>
 <body>
 <form action="action.php" method="post">
   <input type="text" size="20" maxlength="40" "name="query"  />
   <input type="submit"
 </form>
 </body>
</html>
```

This will post the query to an action page. For our action page, we will just take the example code we have written so far and consolidate it. The only changes we will make are to a few variable names that ensure a particular object is unique. At the very beginning, we also include the required RESTParser class for Yahoo! Web Services.

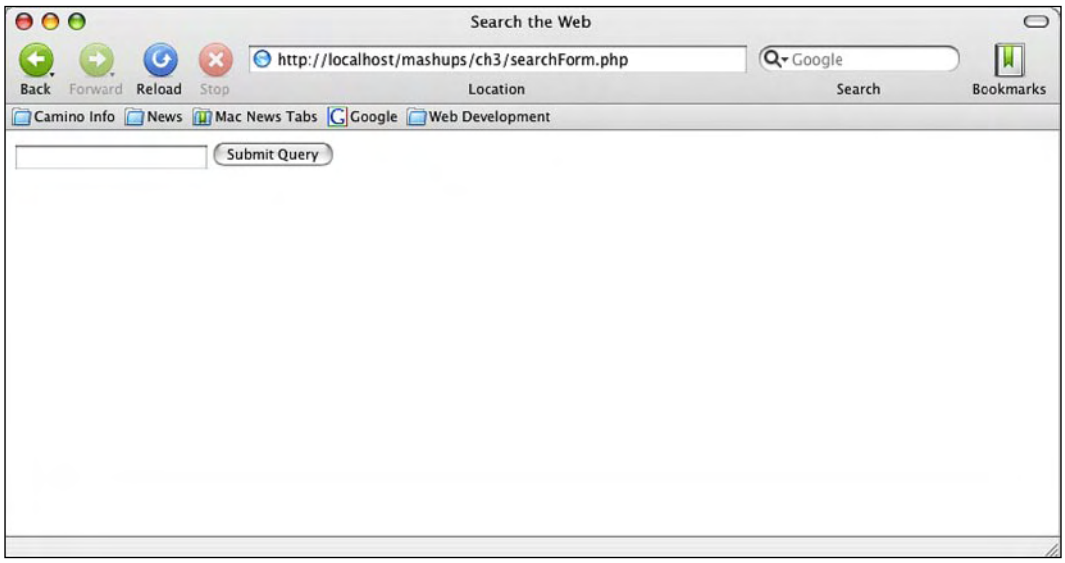
```
<?php
$q = $_POST['query'];
 require_once('classes/RESTParser.php');
/******
 * Begin MSN
 */
$params = array('Request' => array(
                'AppID' => 'Your Microsoft Live Search API Key',
                'Query' => $_POST['query'],
                'CultureInfo' => 'en-US',
                'SafeSearch' => 'Strict',
                'Flags' => '',
                'Location' => '',
                'Requests' => array(
                            'SourceRequest' => array(
                            'Source' => 'Web',
                            'Offset' => 0,
                            'Count' => 10,
                            'ResultFields' => 'All')
                                  )
                                  )
            );

   $msnClient = new SoapClient("http://soap.search.msn.com/
                                    webservices.asmx?wsdl");
   try {
       $msnResults = $msnClient->Search($params);
       }
   catch(SoapFault $msnError) {
       echo "Error!<br>";
       echo "faultcode: " . $msnError->faultcode . "<br>";
       echo "faultstring: " .$msnError->faultstring. "<br>";
       echo "detail: " . $msnError->detail;
                              }
 /****
  * End MSN
  */
```

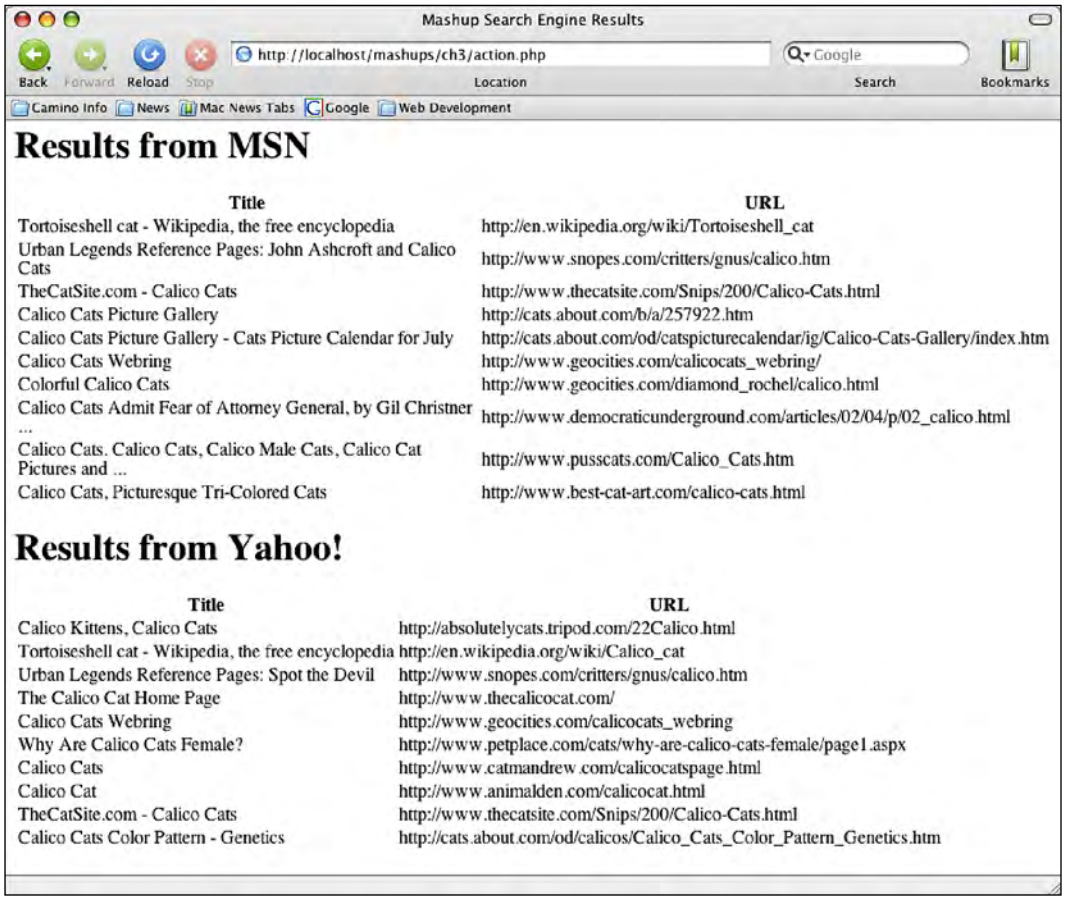
```
 /*****
  * Begin Yahoo!
  */
$params = array('appid' => 'Your Yahoo! App ID',
                'query' => $_POST['query'],
                'region' => 'us',
                'type' => 'all',
                'results' => 10,
                'start' => 0,
                'format' => 'any',
                'adult_ok' => 1,
                'similiar_ok' => 0,
                'output' => 'php'
                );
$parser = new RESTParser();
$t = $parser->callService($params, 'search.yahooapis.com', '/
WebSearchService/V1/webSearch', 'GET');
$yahooResults = unserialize($t);
 /****
  * End Yahoo!
  */
?>
<html>
<head><title>Mashup Search Engine Results?</title></head>
<body>
<?php if ($msnResults) { ?>
<h1>Results from MSN</h1>
<table>
<tr>
<th>Title</th>
<th>URL</th>
</tr>
   <?php foreach ($msnResults->Response->Responses->SourceResponse-
>Results->Result as $key => $value) { ?>
       <tr>
       <td><?= $value->Title ?></td>
       <td><?= $value->Url ?></td>
       </tr>
<?php } ?>
</table>
<?php } ?>
<?php if ($yahooResults) { ?>
<h1>Results from Yahoo!</h1>
```

```
<table>
<tr>
<th>Title</th>
<th>URL</th>
</tr>
   <?php foreach ($yahooResults['ResultSet']['Result'] as $key =>
$value) { ?>
       <tr>
       <td><?= $value['Title'] ?></td>
       <td><?= $value['Url'] ?></td>
       </tr>
   <?php } ?>
</table>
<?php } ?>
</body>
</html>
```

Our mashup is ready to be used. Hitting `searchForm.php` will give us a form to enter in a query.



Our action page will then query both web services and display the results for the user.



Results from MSN

Title	URL
Tortoiseshell cat - Wikipedia, the free encyclopedia	http://en.wikipedia.org/wiki/Tortoiseshell_cat
Urban Legends Reference Pages: John Ashcroft and Calico Cats	http://www.snopes.com/critters/gnus/calico.htm
TheCatSite.com - Calico Cats	http://www.thecatsite.com/Snips/200/Calico-Cats.html
Calico Cats Picture Gallery	http://cats.about.com/b/a/257922.htm
Calico Cats Picture Gallery - Cats Picture Calendar for July	http://cats.about.com/od/catspicturecalendar/ig/Calico-Cats-Gallery/index.htm
Calico Cats Webring	http://www.geocities.com/calicocats_webring/
Colorful Calico Cats	http://www.geocities.com/diamond_rochel/calico.html
Calico Cats Admit Fear of Attorney General, by Gil Christner ...	http://www.democraticunderground.com/articles/02/04/p/02_calico.html
Calico Cats. Calico Cats, Calico Male Cats, Calico Cat Pictures and ...	http://www.pusscats.com/Calico_Cats.htm
Calico Cats, Picturesque Tri-Colored Cats	http://www.best-cat-art.com/calico-cats.html

Results from Yahoo!

Title	URL
Calico Kittens, Calico Cats	http://absolutelycats.tripod.com/22Calico.html
Tortoiseshell cat - Wikipedia, the free encyclopedia	http://en.wikipedia.org/wiki/Calico_cat
Urban Legends Reference Pages: Spot the Devil	http://www.snopes.com/critters/gnus/calico.htm
The Calico Cat Home Page	http://www.thecalicocat.com/
Calico Cats Webring	http://www.geocities.com/calicocats_webring
Why Are Calico Cats Female?	http://www.petplace.com/cats/why-are-calico-cats-female/page1.aspx
Calico Cats	http://www.catmandrew.com/calicocatspage.html
Calico Cat	http://www.animalden.com/calicocat.html
TheCatSite.com - Calico Cats	http://www.thecatsite.com/Snips/200/Calico-Cats.html
Calico Cats Color Pattern - Genetics	http://cats.about.com/od/calicos/Calico_Cats_Color_Pattern_Genetics.htm

Summary

In this chapter, we were introduced to SOAP, the most complex of the web service protocols so far. SOAP relies heavily on other standards like WSDL and XSD. We took a look at a WSDL document and learned how to figure out what web services are available from it, and what types of data are passed. Using PHP 5's SoapClient extension, we then interacted with SOAP servers to grab data. This culminated into the creation of our mashup, which gathered web search results from MSN, and Yahoo!. This mashup was not just about SOAP, though. Using web services from MSN, and Yahoo!, we have now been exposed to three very big players in the web service game. If we use web services from them in the future, we'll know what type of documentation and support we can expect from each one.

4
Your Own Video Jukebox

Project Overview

What	Mashup the web APIs from Last.fm and YouTube to create a video jukebox of songs
Protocols Used	REST (XML-RPC available)
Data Formats	XML, XPSF, RSS
Tools Featured	PEAR
APIs Used	Last.fm and YouTube

Now that we've had some experience using web services, it's time to fine tune their use. XML-RPC, REST, and SOAP will be frequent companions when you use web services and create mashups. You will encounter a lot of different data formats, and interesting ways in which the PHP community has dealt with these formats. This is especially true because REST has become so popular. In REST, with no formalized response format, you will encounter return formats that vary from plain text to ad-hoc XML to XML-based standards.

The rest of our projects will focus on exposing us to some new formats, and we will look at how to handle them through PHP. We will begin with a project to create our own personalized video jukebox. This mashup will pull music lists feeds from the social music site, Last.fm. We will parse out artist names and song titles from these feeds and use that information to search videos on YouTube, a user-contributed video site, using the YouTube web service. By basing the song selections on ever-changing feeds, our jukebox selection will not be static, and will change as our music taste evolves. As YouTube is a user-contributed site, we will see many interesting interpretations of our music, too. This jukebox will be personalized, dynamic, and quite interesting.

Both Last.fm and YouTube's APIs offer their web services through REST, and YouTube additionally offers an XML-RPC interface. Like with previous APIs, XML is returned with each service call. Last.fm returns either plain text, an XML playlist format called XSPF (XML Shareable Playlist Format), or RSS (Really Simple Syndication). In the case of YouTube, the service returns a proprietary format. Previously, we wrote our own SAX-based XML parser to extract XML data. In this chapter, we will take a look at how PEAR, the PHP Extension and Application Repository, can do the XSPF parsing work for us on this project and might help in other projects.

Let's take a look at the various data formats we will be using, and then the web services themselves.

XSPF

One of XML's original goals was to allow industries to create their own markup languages to exchange data. Because anyone can create their own elements and schemas, as long as people agreed on a format, XML can be used as the universal data transmission language for that industry. One of the earliest XML-based languages was ChemXML, a language used to transmit data within the chemical industry. Since then, many others have popped up.

XSPF was a complete grassroots project to create an open, non-proprietary music playlist format based on XML. Historically, playlists for software media players and music devices were designed to be used only on the machine or device, and schemas were designed by the vendor themselves. XSPF's goal was to create a format that could be used in software, devices, and across networks.

XSPF is a very simple format, and is easy to understand. The project home page is at `http://www.xspf.org`. There, you will find a quick start guide which outlines a simple playlist as well as the official specifications at `http://www.xspf.org/specs`. Basically, a typical playlist has the following structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<playlist version="1" xmlns="http://xspf.org/ns/0/">
   <title>Shu Chow's Playlist</title>
   <date>2006-11-24T12:01:21Z</data>
   <trackList>
      <track>
         <title>Pure</title>
         <creator>Lightning Seeds</creator>
         <location>
            file:///Users/schow/Music/Pure.mp3
```

```
            </location>
         </track>
        <track>
            <title>Roadrunner</title>
            <creator>The Modern Lovers</creator>
            <location>
               file:///Users/schow/Music/Roadrunner.mp3
            </location>
         </track>
         <track>
            <title>The Bells</title>
            <creator>April Smith</creator>
            <location>
              file:///Users/schow/Music/The_Bells.mp3
            </location>
         </track>
      </trackList>
    </playlist>
```

`playlist` is the parent element for the whole document. It requires one child element, `trackList`, but there can be several child elements that are the metadata for the playlist itself. In this example, the playlist has a title specified in the `title` element, and the creation date is specified in the `date` element. Underneath `trackList` are the individual tracks that make up the playlist. Each track is encapsulated by the `track` element. Information about the track, including the location of its file, is encapsulated in elements underneath `track`. In our example, each track has a title, an artist name, and a local file location. The official specifications allow for more track information elements such as track length and album information.

Here are the `playlist` child elements summarized:

Playlist Child Element	Required?	Description
trackList	Yes	The parent of individual track elements. This is the only required child element of a playlist. Can be empty if the playlist has no songs.
title	No	A human readable title of the XSPF playlist.
creator	No	The name of the playlist creator.
annotation	No	Comments on the playlist.
info	No	A URL to a page containing more information about the playlist.
location	No	The URL to the playlist itself.

Playlist Child Element	Required?	Description
identifier	No	The unique ID for the playlist. Must be a legal Uniform Resource Name (URN).
image	No	A URL to an image representing the playlist.
date	No	The creation (not the last modified!) date of the playlist. Must be in XML schema dateTime format. For example, "2004-02-27T03:30:00".
license	No	If the playlist is under a license, the license is specified with this element.
attribution	No	If the playlist is modified from another source, the attribution element gives credit back to the original source, if necessary.
link	No	Allows non-XSPF resources to be included in the playlist.
meta	No	Allows non-XSPF metadata to be included in the playlist.
extension	No	Allows non-XSPF XML extensions to be included in the playlist.

A `trackList` element has an unlimited number of track elements to represent each track. `track` is the only allowed child of `trackList`. `track's` child elements give us information about each track. The following table summarizes the children of `track`:

Track Child Element	Required?	Description
location	No	The URL to the audio file of the track.
identifier	No	The canonical ID for the playlist. Must be a legal URN.
title	No	A human readable title of the track. Usually, the song's name.
creator	No	The name of the track creator. Usually, the song's artist.
annotation	No	Comments on the track.
info	No	A URL to a page containing more information about the track.
image	No	A URL to an image representing the track.
album	No	The name of the album that the track belongs to.
trackNum	No	The ordinal number position of the track in the album.

Track Child Element	Required?	Description
duration	No	The time to play the track in milliseconds.
link	No	Allows non-XSPF resources to be included in the track.
meta	No	Allows non-XSPF metadata to be included in the track.
extension	No	Allows non-XSPF XML extensions to be included in the track.

Note that XSPF is very simple and track oriented. It was not designed to be a repository or database for songs. There are not a lot of options to manipulate the list. XSPF is merely a shareable playlist format, and nothing more.

RSS

The simplest answer to, "What is RSS?", is that it's an XML file used to publish frequently updated information, like news items, blogs entries, or links to podcast episodes. News sites like Slashdot.org and the New York Times provide their news items in RSS format. As new news items are published, they are added to the RSS feed. Being XML-based, third-party aggregator software makes reading news items easy. With one piece of software, I can tell it to grab feeds from various sources and read the news items in one location. Web applications can also read and parse RSS files. By offering an RSS feed for my blog, another site can grab the feed and keep track of my daily life. This is one way by which a small site can provide rudimentary web services with minimal investment.

The more honest answer is that it is a group of XML standards (used to publish frequently updated information like news items or blogs) that may have little compatibility with each other. Each version release also has a tale of conflict and strife behind it. We won't dwell on the politicking of RSS. We'll just look at the outcomes. The RSS world now has three main flavors:

- The RSS 1.0 branch includes versions 0.90, 1.0, and 1.1. It's goal is to be extensible and flexible. The downside to the goals is that it is a complex standard.
- The RSS 2.0 branch includes versions 0.91, 0.92, and 2.0.x. Its goal is to be simple and easy to use. The drawback to this branch is that it may not be powerful enough for complex sites and feeds.

There are some basic skeletal similarities between the two formats. After the XML root element, metadata about the feed itself is provided in a top section. After the metadata, one or more items follow. These items can be news stories, blog entries, or podcasts episodes. These items are the meat of an RSS feed.

The following is an example RSS 1.1 file from XML.com:

```
<Channel xmlns="http://purl.org/net/rss1.1#"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   rdf:about="http://www.xml.com/xml/news.rss">

   <title>XML.com</title>
   <link>http://xml.com/pub</link>
   <description>
       XML.com features a rich mix of information and services for the
       XML community.
   </description>
   <image rdf:parseType="Resource">
      <title>XML.com</title>
      <url>http://xml.com/universal/images/xml_tiny.gif</url>
   </image>

   <items rdf:parseType="Collection">
      <item rdf:about=
                  "http://www.xml.com/pub/a/2005/01/05/restful.html">
      <title>
          The Restful Web: Amazon's Simple Queue Service
      </title>
      <link>
          http://www.xml.com/pub/a/2005/01/05/restful.html
      </link>
      <description>
           In Joe Gregorio's latest Restful Web column, he explains
          that Amazon's Simple Queue Service, a web service offering a
          queue for reliable storage of transient
           messages, isn't as RESTful as it claims.
         </description>
      </item>

      <item rdf:about=
                   "http://www.xml.com/pub/a/2005/01/05/tr-xml.html">
      <title>
          Transforming XML: Extending XSLT with EXSLT
      </title>
         <link>
         http://www.xml.com/pub/a/2005/01/05/tr-xml.html
```

```
        </link>
           <description>
            In this month's Transforming XML column, Bob DuCharme
            reports happily that the promise of XSLT extensibility via
            EXSLT has become a reality.
           </description>
        </item>
     </items>
  </Channel>
```

The root element of an RSS file is an element named `Channel`. Immediately, after the root element are elements that describe the publisher and the feed. The `title`, `link`, `description`, and `image` elements give us more information about the feed.

The actual content is nested in the `items` element. Even if there are no items in the feed, the `items` element is required, but will be empty. Usage of these elements can be summarized as follows:

Channel Child Element	Required?	Description
title	Yes	A human readable title of the channel.
link	Yes	A URL to the feed.
description	Yes	A human readable description of the feed.
items	Yes	A parent element to wrap around `item` elements.
image	No	A section to house information about an official image for the feed.
others	No	Any other elements not in the RSS namespace can be optionally included here. The namespace must have been declared earlier, and the child elements must be prefixed.

If used, the `image` element needs its own child elements to hold information about the feed image. A `title` element is required and while optional, a `link` element to the actual URL of the image would be extremely useful.

Each news blog, or podcast entry is represented by an `item` element. In this RSS file, each item has a title, link, and a description, each, represented by the respective element. This file has two items in it before the `items` and `Channel` elements are closed off.

Note the use of the `rdf` namespace. The RSS 1.0 branch uses Rich Description Framework (RDF) extensively. RDF is an XML framework to make documents not only machine-friendly, but also human-friendly by organizing topics together. To consume RSS 1.0, we do not need to know too much about RDF. However, we will be taking a little closer look at it on a future project. For now, the key concept to take away is that individual items are represented by the item element in the RSS 1.0 branch.

The `item` element's children are summarized as follows:

Item Child Element	Required?	Description
title	Yes	A human readable title of the item.
link	Yes	A URL to the item.
description	Yes	A human readable description of the item.
others	No	Any other elements not in the RSS namespace can be optionally included here. The namespace must have been declared earlier, and the child elements must be prefixed.

Looking at a RSS 2.0 feed from IBM DeveloperWorks, we can see a lot of similarities:

```
<rss version="2.0">
<channel>
<title>developerWorks : Linux : Technical library</title>
<link>http://www.ibm.com/developerworks/index.html</link>
<description>
   The latest content from IBM developerWorks
</description>
<pubDate>Wed, 10 Jan 2007 01:03:11 EST</pubDate>
<language>en-us</language>
<copyright>Copyright 2004 IBM Corporation.</copyright>
<image>
   <title>IBM developerWorks</title>
   <url>
      http://www-106.ibm.com/developerworks/i/dwlogo-small.gif
   </url>
   <link>
      http://www.ibm.com/developerworks/index.html
   </link>
</image>
<item>
   <title>
```

```
        <![CDATA[
        Whistle while you work to run commands on your computer
        ]]>
      </title>
      <description>
        <![CDATA[
        Use Linux or Microsoft Windows, the open source sndpeek
        program, and a simple Perl script to read specific
        sequences of tonal events -- literally whistling,
        humming, or singing to your computer -- and run commands
        based on those tones. Give your computer a short low
        whistle to check your e-mail or unlock your your
        screensaver with the    opening bars of Beethoven's Fifth
        Symphony. Whistle while you work for higher efficiency
        ]]>
      </description>
      <link>
        <![CDATA[
        http://www.ibm.com/developerworks/library/os-
        whistle/index.html?ca=drs-
        ]]>
      </link>
      <category>Articles</category>
  </item>
  <item>
      <title>
        <![CDATA[
        Programming high-performance applications on the Cell BE
        processor, Part 1: An introduction to Linux on the
        PLAYSTATION 3
        ]]>
      </title>
      <description>
        <![CDATA[
        The Sony PLAYSTATION 3 (PS3) is the easiest and cheapest
        way for programmers to get their hands on the new Cell
        Broadband Engine (Cell BE) processor and take it for a
        drive. Discover what the fuss is all about, how to
        install Linux on the PS3, and how to get started
        developing for the Cell BE processor on the PS3.
        ]]>
      </description>
      <link>
        <![CDATA[
        http://www.ibm.com/developerworks/linux/library/pa-
        linuxps3-1/index.html?ca=drs-
        ]]>
```

```
    </link>
    <category>Articles</category>
</item>
</channel>
</rss>
```

Like the 1.1 feed, 2.0 starts with information about the feed and the publisher, followed by news items.

There are some key differences between 1.1 and 2.0:

- The `rss` element is the root element for 2.0.
- A `channel` element follows `rss` and encompasses all other elements.
- Each item is represented by an `item` element, but items do not have a parent element that groups like all together (like `items` in 1.1 does).

While structurally the feeds are similar, the tags and nesting are different enough to cause problems, especially when you consider that `\programs` are the primary consumers of RSS feeds.

RSS 2.0 also has many more standard tags available in its namespace. RSS 2.0 recognized there is a lot of common information for channels and items that people would like to include beyond just titles, links, and descriptions. Things like a publication date, languages, and categories are used frequently. In RSS 1.1, one would have to use another standard and pull it into the document through namespacing. This is not only added overhead, but creates many ways of putting something as simple as a publication date into the feed.

Channel Child Element	Required?	Description
title	Yes	A human readable title of the feed.
link	Yes	A URL to the feed.
description	Yes	A human readable description of the feed.
category	No	One or more categories for the feed. There is no set standard for the available values of this element.
cloud	No	A cloud is a centralized server that holds information about a group of RSS feeds. This element will hold information about a remote procedure to call on the cloud server when the feed is updated. Attributes for this are the domain, port, path, procedure, and protocol. The remote procedure can be either XML-RPC or SOAP.

Channel Child Element	Required?	Description
copyright	No	If the content is copyrighted, the copyright is placed into this element.
docs	No	A URL to the documentation for the feed.
generator	No	The name of the program used to generate the feed.
image	No	A section to house information about an official image for the feed.
language	No	The language in which the feed is written.
lastBuildDate	No	The last modification date of the feed.
managingEditor	No	The email address for the person responsible for the content of the feed.
pubDate	No	The last publication date of the feed.
rating	No	The PICS (Platform for Internet Content Selection) rating of the feed. See `http://www.w3.org/PICS/` for more information about PICS.
skipDays	No	Days in which RSS aggregators should not read this feed.
skipHours	No	Hours in which RSS aggregators should not read this feed.
textInput	No	The name of a text input field to be displayed with this field.
ttl	No	TTL stands for, "Time To Live". It is the number of minutes the feed should stay in a client's cache.
webMaster	No	An email address of the person responsible for the technical aspects of this feed.

Likewise, RSS 2.0 has many more available child elements for `item` elements.

Item Child Element	Required?	Description
title	Yes	A human readable title of the item.
link	Yes	A URL to the item.
description	Yes	A human readable description of the item.
author	No	The email address of the author of this item.
category	No	One or more categories for the item. There is no set standard for the available values of this element.

Item Child Element	Required?	Description
comments	No	URL to a page of reader comments of the item.
enclosure	No	Allows a media file to be included for the item. The URL to the media file is included in a required attribute named `url.length`, in bytes, of the media file and `type`, the MIME type of the file, are also required. The most common use of this tag is to specify an MP3 in podcasts.
guid	No	A unique identifier for the item.
pubDate	No	The last publication date of the item.
source	No	A third-party source for the item. Used in citation sources for an item.

Atom Syndication Format

There is a new, third syndication feed called Atom. Atom attempts to bridge the extensibility and simplicity goals of both RSS branches. Structurally, Atom feeds share a similar model to RSS—metadata followed by entries. However, the element names are quite different. We won't be using Atom in this project, but you should be aware of it. Although it is the least mature of the formats and the market share is relatively smaller, it is the only format supported by an industry standards body (specifically, the Internet Engineering Task Force) and is already being used by powerhouses like Google News.

YouTube Overview

YouTube almost needs no introduction these days. The site is as ubiquitous as many of the previous sites we discussed—Amazon, Google, Yahoo!, and MSN. Links to its videos have been passed around email accounts. In case you have lead a very sheltered existence, we will take a brief look at what YouTube does, some features, and its Web API.

In a nutshell, YouTube is a site that allows users to share homemade videos with the public via the Internet. Users upload any video they wish (with much respect to copyright laws) and other users may view and comment on them. The latter has led YouTube to become a strong social networking site in addition to just sharing videos.

Some of the available features of YouTube include:

- **Video Tagging**
  YouTube relies on its user community to describe the videos in its repository. This is done by allowing users to associate descriptive words and short phrases with a particular video. This process is known as **tagging**. For example, if I am watching a clip of a live performance by the Brooklyn, New York-based band, They Might Be Giants, YouTube allows me to tag the clip with, "They Might Be Giants", "Brooklyn", "alternative".
- **Video Search**
  YouTube has a robust search engine that queries the tags placed on videos. By searching on tags, a video's description is democratized. If YouTube's search engine queried only the description given to it by the submitter, that person has a large influence on how that video is returned in search results. By also querying tags, the search engine can find videos the community thinks should be returned. In our example, with "They Might Be Giants", the clip submitter may not have noted that the band is based in Brooklyn. However, because I tagged the clip with "Brooklyn," any Brooklynite looking for local bands may also discover They Might Be Giants.
- **Submitter Subscriptions**
  YouTube keeps track of a user's video submissions. It is then natural to let other users subscribe to another user's submissions. Through their antics and jeremiads, a few people have gained quite a following on YouTube, and found their fifteen minutes of fame.
- **Community**
  YouTube builds to the social aspect of its site by allowing users to comment on videos.
- **Embedding on Other Sites**
  YouTube allows users to embed a video on their own web site. YouTube, then, essentially acts a video hosting service. In addition, an external site can also just display a quick thumbnail of a video with a link to the video and all its information and comments on YouTube.

The social networking aspect of YouTube makes for a good web service subject, and the site has taken advantage of that.

YouTube Developer API

Like some of the other web services we've encountered, YouTube's API requires a developer ID that needs to be passed to the server when calling services. You can sign up for a developer ID at `http://www.youtube.com/signup?next=my_profile_dev`. After you have an ID, you can dig into the documentation. The documentation can be found at `http://www.youtube.com/dev_docs`.

The available methods fall into two categories.
A method can either be related to user information, or related to video viewing. YouTube uses a quasi-Java, dot notation to name their web service methods. For example, the method to get a user's profile is named `youtube.users.get_profile`, and the method to get a list of featured YouTube videos is named `youtube.videos.list_featured`.

Each method can be called either using REST or XML-RPC. A REST request takes the method name as a parameter along with any other parameters needed. The format used is `http://www.youtube.com/api2_rest?method=METHOD_NAME&parameter1=VALUE1&parameter2=VALUE2`. For example, let's look at `get_profile`, which gets a user's profile. The documentation for this method (located at `http://youtube.com/dev_api_ref?m=youtube.users.getprofile`) says this method requires three parameters:

- `method` is the method name itself. For `get_profile`, the value for method is the method's formal name, `youtube.users.getprofile`. This parameter is only needed for REST requests, which we are using in this example.
- `dev_id`, which is your developer ID.
- `user`, the profile name of the user whom you want to retrieve.

The REST request for `get_profile` would be `http://www.youtube.com/api2_rest?method=youtube.users.getprofile&dev_id=YOUR_DEVELOPER_ID&user=PROFILE_NAME`.

The same method names are used when interfacing with XML-RPC. To use XML-RPC, the request is a standard XML-RPC call that takes only one `param` element, a `struct`, and each YouTube method parameter is sent as a value of the `struct`. The name of the method is not passed into the `struct`. Instead, follow XML-RPC standards by putting the name of the method in the `methodName` element of the call. Under XML-RPC, the call to `get_profile` would look like this:

```
<?xml version="1.0" ?>
<methodCall>
   <methodName>youtube.users.get_profile</methodName>
   <params>
```

```
        <param>
           <value>
              <struct>
                 <member>
                    <name>dev_id</name>
                    <value>
                       <string>YOUR DEVELOPER ID</string>
                    </value>
                 </member>
                 <member>
                    <name>user</name>
                    <value>
                       <string>PROFILE NAME</string>
                    </value>
                 </member>
              </struct>
          </value>
        </param>
     </params>
  </methodCall>
```

The server will always return an XML response. Each method's response is detailed in the documentation. If REST is used, the response is just the XML in string form. If XML-RPC is used, that same documented XML string is escaped and encased in an XML-RPC response wrapper.

We can see how this works with `get_profile`. Consult the documentation to see how the response is structured. According to the documentation, this is how the response will be returned if we are using REST:

```
<user_profile>
   <first_name>Shu</first_name>
   <last_name>Chow</last_name>
   <about_me>I pound on a keyboard for a living</about_me>
   …
   <friend_count>3</friend_count>
   <favorite_video_count>7</favorite_video_count>
   <currently_on>false</currently_on>
</user_profile>
```

If we made the call with XML-RPC, the XML structure would be the same, but this time, the XML is a string within a `methodResponse` element:

```
<?xml version='1.0' ?>
<methodResonse>
   <params>
```

```
        <param>
           <value><string>
              <user_profile>
                 <first_name>Shu</first_name>
                 <last_name>Chow</last_name>
                 <about_me>
                      I pound on a keyboard for a living
                  </about_me>
                    …
                 <friend_count>3</friend_count>
                 <favorite_video_count>7</favorite_video_count>
                 <currently_on>false</currently_on>
              </user_profile>
           </string></value>
        </param>
     </params>
  </methodResponse>
```

As you can see, the YouTube API is simple, consistent, and well-documented.

Last.fm Overview

Last.fm (`http://www.last.fm`) describes itself as a social music networking site. It is an interesting way to find new music based on user contributions. Users register for free at the site. You then download a small client program to run on your machine. This program monitors what music you are playing through software media players such as iTunes and WinAmp. You can also tag the songs through the program. The program uploads the song information, including tags, back to the Last.fm server. Based on the artist, song, genre, and what other people have tagged songs, Last.fm builds a music profile for you. It uses this profile to build a custom streaming radio station for you that you can also listen to through the client program, and recommends other artists and songs that are similar to the music that you like.

Vigilant users may, understandably, bristle at installing a desktop program that sends their music listening habits off to a server somewhere. However, Last.fm does a good job of protecting your privacy. First off, the client program is open source. It does not contain any malware that would compromise your privacy or your computer's security, and the code is opened up for peer review. Second, their privacy policy explicitly states they do not share your personal information to outside parties. Finally, the registration process does not ask for information such as address or names, although you can enter your zip or postal code to see music events in your area. The music profile that it generates is tied to your account and profile you create. Any private information regarding you, the individual, is not required.

As we walk through the project and the examples, I will be demonstrating things using my own personal account. This will guarantee that the examples indeed work. To truly personalize this project as your own, you should consider creating an account and uploading some track data through the Last.fm client program.

Audioscrobbler Web Services

Audioscrobbler Web Services API is the web service that allows you to access data displayed on Last.fm. Audioscrobbler's home is at `http://www.audioscrobbler.net/data/webservices`. This web service is basically a collection of RSS feeds that fall within several categories:

- **User Profile Data**: The largest collection of feeds, this category provides data about a certain user. For example, there are feeds that list the top artists they've listened to, and the top tracks.
- **Artist Data**: This category aggregates data about all artists in the Last.fm database. By providing an artist name, you can get things such as their most listened to tracks on Last.fm, their largest fans, the users that have listened to them the most, etc.
- **Album Data**: This category holds information about albums. Currently, there is only one feed associated with this category. By providing a title name to the Info feed, you can get a track listing for the album.
- **Track Data**: Information about specific tracks is given here.
- **Tag Data**: Like YouTube, Last.fm allows users to tag songs with their own descriptions.
- **Group Data**: Like other social networking sites, Last.fm provides groups users can create and join. Information about what members are listening to in these groups is available in the Group Data feeds.
- **Forum Data**: Last.fm has a community forum. The posts are offered in an RSS feed.
- **LiveJournal Protocol**: Audioscrobbler can interact with the LiveJournal web service through this web service.

Looking at the Audioscrobbler Web Services API home page, we see that each feed can be in up to four formats—plain text, generic XML, XSPF, or RSS. Often, the available formats are dictated by their purpose. For example, it would not make sense to output user information in XSPF, a format made just for representing song playlists.

The web service is not documented well, but is very simple, and documentation is by example. On the API home page, the format links to each feed is an example of how to use the feed. Feeds pertaining to member information use the user "RJ", who is one of the original founders of Last.fm. Feeds pertaining to artist information, like top listeners, use an assortment of artists. To use any of the feeds, take the URL of the playlist and format you want, and replace it with the user or artist.

Let's play with some examples. The feeds are easy to use and experiment with, because the Audioscrobbler feeds are just URLs and no developer ID is necessary to use them. The API home page shows the top tracks played by a user are available in plain text, Last.fm's own XML format, and as an XSPF playlist. The URL on that page for RJ's top tracks in XSPF is `http://ws.audioscrobbler.com/1.0/user/RJ/toptracks.xspf`.

If I want to see my top tracks, I can just replace RJ with my own Last.fm user name: `http://ws.audioscrobbler.com/1.0/user/ShuTheMoody/toptracks.xspf`.

The artist feeds uses Metallica as the default example. To see a list of top Last.fm fans of Metallica, use the Top Fans feed on the API home page: `http://ws.audioscrobbler.com/1.0/artist/Metallica/fans.xml`. To see another artist's top fans, just change the `Metallica` section of the URL: `http://ws.audioscrobbler.com/1.0/artist/Donnas/fans.xml`.

Parsing With PEAR

If we were to start mashing up right now, between XSPF, YouTube's XML response, and RSS, we would have to create three different parsers to handle all three response formats. We would have to comb through the documentation and create flexible parsers for all three formats. If the XML response for any of these formats changes, we would also be responsible for changing our parser code. This isn't a difficult task, but we should be aware that someone else has already done the work for us. Someone else has already dissected the XML code. To save time, we can leverage this work for our mashup.

We used PEAR, earlier in Chapter 1 to help with XML-RPC parsing. For this project, we will once again use PEAR to save us the trouble of writing parsers for the three XML formats we will encounter.

For this project, we will take a look at three packages for our mashup. File_XSPF is a package for extracting and setting up XSPF playlists. Services_YouTube is a Web Services package that was created specifically for handling the YouTube API for us. Finally, XML_RSS is a package for working with RSS feeds.

For this project, it works out well that there are three specific packages that fits our XML and RSS formats. If you need to work with an XML format that does not have a specific PEAR package, you can use the XML_Unserializer package. This package will take a XML and return it as a string.

Is PEAR Right For You?

Before we start installing PEAR packages, we should take a look if it is even feasible to use them for a project. PEAR packages are installed with a command line package manager that is included with every core installation of PHP. In order for you to install PEAR packages, you need to have administrative access to the server. If you are in a shared hosting environment and your hosting company is stingy, or if you are in a strict corporate environment where getting a server change is more hassle than it is worth, PEAR installation may not be allowed. You could get around this by downloading the PEAR files and installing them in your web documents directory. However, you will then have to manage package dependencies and package updates by yourself. This hassle may be more trouble than it's worth, and you may be better off writing your own code to handle the functionality.

On the other hand, PEAR packages are often a great time saver. The purpose of the packages is to either simplify tedious tasks, or interface with complex systems. The PEAR developer has done the difficult work for you already. Moreover, as they are written in PHP and not C, like a PHP extension would be, a competent PHP developer should be able to read the code for documentation if it is lacking. Finally, one key benefit of many packages, including the ones we will be looking at, is that they are object-oriented representations of whatever they are interfacing. Values can be extracted by simply calling an object's properties, and complex connections can be ignited by a simple function call. This helps keep our code cleaner and modular. Whether the benefits of PEAR outweigh the potential obstacles depends on your specific situation.

Package Installation and Usage

Just like when we installed the XML-RPC package, we will use the `install` binary to install our three packages. If you recall, installing a package, simply type `install` into the command line followed by the name of the package. In this case, though, we need to set a few more flags to force the installer to grab dependencies and code in beta status.

To install File_XSPF, switch to the root user of the machine and use this command:

```
[Blossom:~] shuchow# /usr/local/php5/bin/pear install –f --
alldeps File_XSPF
```

This command will download the package. The `-alldeps` flag tells PEAR to also check for required dependencies and install them if necessary. The progress and outcome of the downloads will be reported.

Do a similar command for Services_YouTube:

```
[Blossom:~] shuchow# /usr/local/php5/bin/pear install -f --alldeps
Services_YouTube
```

Usually, you will not need the `-f` flag. By default, PEAR downloads the latest stable release of a package. The `-f` flag, force, forces PEAR to download the most current version, regardless of its release state. As of this writing, File_XSPF and Services_YouTube do not have stable releases, only beta and alpha respectively. Therefore, we must use `-f` to grab and install this package. Otherwise, PEAR will complain that the latest version is not available. If the package you want to download is in release state, you will not need the `-f` flag.

This is the case of XML_RSS, which has a stable version available.

```
[Blossom:~] shuchow# /usr/local/php5/bin/pear install --alldeps XML_RSS
```

After this, sending a `list-all` command to PEAR will show the three new packages along with the packages you had before.

PEAR packages are basically self-contained PHP files that PEAR installs into your PHP includes directory. The `includes` directory is a directive in your `php.ini` file. Navigate to this directory to see the PEAR packages' source files. To use a PEAR package, you will need to include the package's source file in the top of your code. Consult the package's documentation on how to include the main package file. For example, File_XSPF is activated by including a file named `XSPF.php`. PEAR places `XSPF.php` in a directory named `File`, and that directory is inside your `includes` directory.

```
<?php
   require_once 'File/XSPF.php';
   //File_XSPF is now available.
```

File_XSPF

The documentation to the latest version of XSPF is located at `http://pear.php.net/package/File_XSPF/docs/latest/File_XSPF/File_XSPF.html`.

The package is simple to use. The heart of the package is an object called XSPF. You instantiate and use this object to interact with a playlist. It has methods to retrieve and modify values from a playlist, as well as utility methods to load a playlist into memory, write a playlist from memory to a file, and convert an XSPF file to other formats.

Getting information from a playlist consists of two straightforward steps. First, the location of the XSPF file is passed to the XSPF object's `parse` method. This loads the file into memory. After the file is loaded, you can use the object's various getter methods to extract values from the list. Most of the XSPF getter methods are related to getting metadata about the playlist itself. To get information about the tracks in the playlist, use the `getTracks` method. This method will return an array of XSPF_Track objects. Each track in the playlist is represented as an XSPF_Track object in this array. You can then use the XSPF_Track object's methods to grab information about the individual tracks.

We can grab a playlist from Last.fm to illustrate how this works. The web service has a playlist of a member's most played songs. Named Top Tracks, the playlist is located at `http://ws.audioscrobbler.com/1.0/user/USERNAME/toptracks.xspf`, where `USERNAME` is the name of the Last.fm user that you want to query.

This page is named `XSPFPEARTest.php` in the examples. It uses File_XSPF to display my top tracks playlist from Last.fm.

```
<?php
   require_once 'File/XSPF.php';
   $xspfObj =& new File_XSPF();
   //Load the playlist into the XSPF object.
   $xspfObj->parseFile('http://ws.audioscrobbler.com/1.0/user/
                              ShuTheMoody/toptracks.xspf');
   //Get all tracks in the playlist.
   $tracks = $xspfObj->getTracks();
?>
```

This first section creates the XSPF object and loads the playlist. First, we bring in the File_XSPF package into the script. Then, we instantiate the object. The `parseFile` method is used to load an XSPF file list across a network. This ties the playlist to the XSPF object. We then use the `getTracks` method to transform the songs on the playlist into XSPF_Track objects.

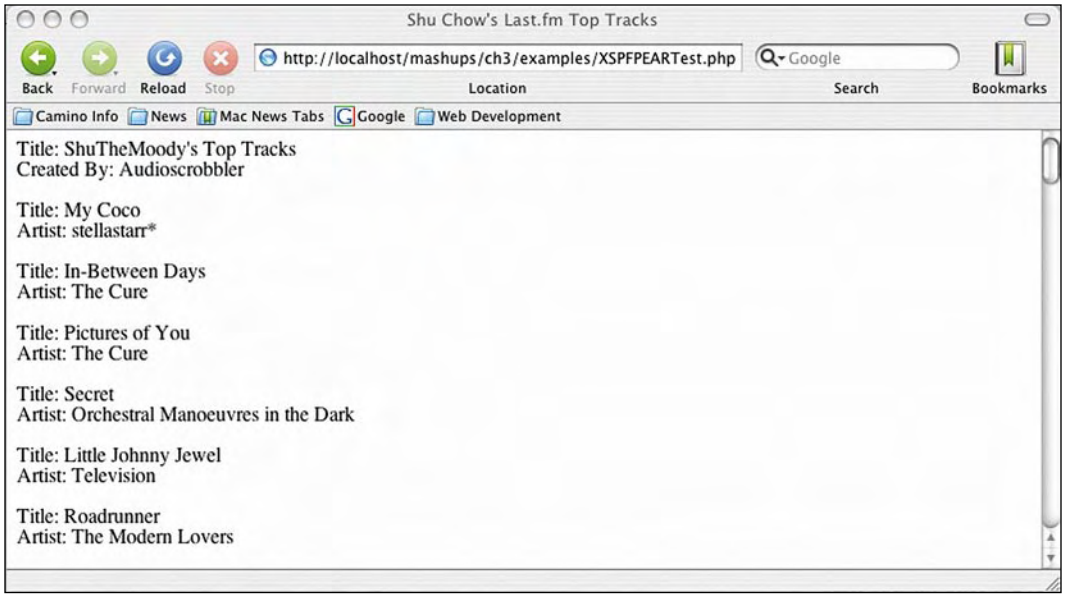
```
<html>
<head>
   <title>Shu Chow's Last.fm Top Tracks</title>
</head>
<body>
   Title: <?= $xspfObj->getTitle() ?><br />
   Created By: <?= $xspfObj->getCreator() ?>
```

Next, we prepare to display the playlist. Before we do that, we extract some information about the playlist. The XSPF object's `getTitle` method returns the XSPF file's `title` element. `getCreator` returns the `creator` element of the file.

```
<?php foreach ($tracks as $track) { ?>
   <p>
      Title: <?= $track->getTitle() ?><br />
      Artist: <?= $track->getCreator() ?><br />
   </p>
<?php } ?>
</body>
</html>
```

Finally, we loop through the tracks array. We assign the array's elements, which are XSPF_Track objects, into the `$track` variable. XSPF_Track also has `getTitle` and `getCreator` methods. Unlike XSPF's methods of the same names, `getTitle` returns the title of the track, and `getCreator` returns the track's artist.

Running this file in your web browser will return a list populated with data from Last.fm.



Services_YouTube

Services_YouTube works in a manner very similar to File_XSPF. Like File_XSPF, it is an object-oriented abstraction layer on top of a more complicated system. In this case, the system is the YouTube API.

Using Services_YouTube is a lot like using File_XSPF. Include the package in your code, instantiate a Services_YouTube object, and use this object's methods to interact with the service. The official documentation for the latest release of Services_YouTube is located at `http://pear.php.net/package/Services_YouTube/docs/latest/`. The package also contains online working examples at `http://pear.php.net/manual/en/package.webservices.services-youtube.php`.

Many of the methods deal with getting members' information like their profile and videos they've uploaded. A smaller, but very important subset is used to query YouTube for videos. We will use this subset in our mashup. To get a list of videos that have been tagged with a specific tag, use the object's `listByTag` method.

`listByTag` will query the YouTube service and store the XML response in memory. It is does not return an array of video objects we can directly manage, but with one additional function call, we can achieve this. From there, we can loop through an array of videos similar to what we did for XSPF tracks.

The example file `YouTubePearTest.php` illustrates this process.

```
<?php
  require_once 'Services/YouTube.php';
  $dev_id = 'Your YouTube DeveloperID';
  $tag = 'Social Distortion';
  $youtube = new Services_YouTube($dev_id, array('usesCache' => true));
  $videos = $youtube->listByTag($tag);
?>
```

First, we load the Services_YouTube file into our script. As YouTube's web service requires a Developer ID, we store that information into a local variable. After that, we place the tag we want to search for in another local variable named `$tag`. In this example, we are going to check out which videos YouTube has for the one of the greatest bands of all time, Social Distortion. Service_YouTube's constructor takes this Developer ID and uses it whenever it queries the YouTube web service. The constructor can take an array of options as a parameter. One of the options is to use a local cache of the queries. It is considered good practice to use a cache, as to not slam the YouTube server and run up your requests quota.

Another option is to specify either REST or XML-RPC as the protocol via the `driver` key in the options array. By default, Services_YouTube uses REST. Unless you have a burning requirement to use XML-RPC, you can leave it as is.

Once instantiated, you can call `listByTag` to get the response from YouTube. `listByTag` takes only one parameter—the tag of our desire.

Services_YouTube now has the results from YouTube. We can begin the display of the results.

```
<html>
<head>
   <title>Social Distortion Videos</title>
</head>
<body>
   <h1>YouTube Query Results for Social Distortion</h1>
```

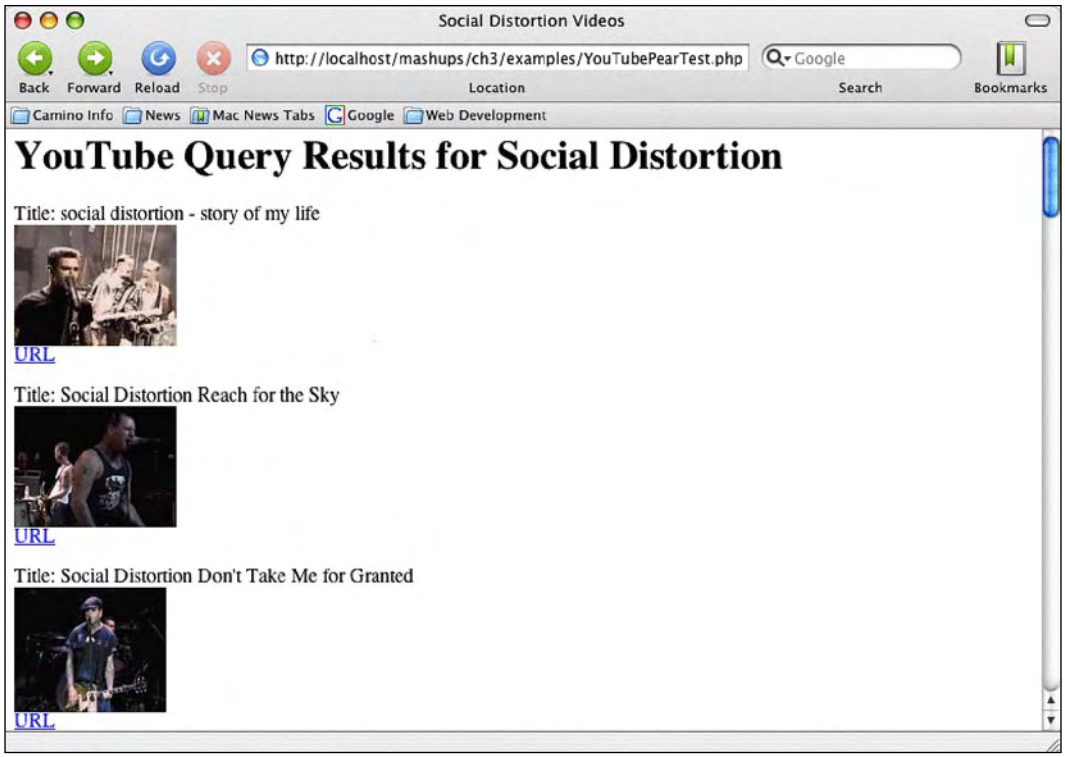
Next, we will loop through the videos. In order to get an array of video objects, we first need to parse the XML response. We do that using Services_YouTube's xpath method, which will use the powerful XPATH query language to go through the XML and convert it into PHP objects. We pass the XPATH query into the method, which will give us an array of useful objects. We will take a closer look at XPATH and XPATH queries later in another project. For now, trust that the query `//video` will return an array of `video` objects that we can examine.

Within the loop, we display each video's title, a thumbnail image of the video, and a hyperlink to the video itself.

```
<?php foreach ($videos->xpath('//video') as $i => $video) { ?>
<p>
   Title: <?= $video->title ?><br />
   <img src='<?= $video->thumbnail_url ?>' alt='<?= $video->title ?>'
/><br />
   <a href='<?= $video->url ?>'>URL</a>
</p>
<?php } ?>
</body>
</html>
```

Running this query in our web browser will give us a results page of videos that match the search term we submitted.

Note that before running `YouTubePearTest.php` file, you will have to install `CURL`.



XML_RSS

Like the other PEAR extensions, XML_RSS changes something very complex, RSS, into something very simple and easy to use, PHP objects. The complete documentation for this package is at `http://pear.php.net/package/XML_RSS/docs/XML_RSS`.

There is a small difference to the basic philosophy of XML_RSS compared to Services_YouTube and File_XSPF. The latter two packages take information from whatever we're interested in, and place them into PHP object properties.

For example, File_XSPF takes track names into a `Track` object, and you use a `getTitle()` getter method to get the title of the track. In Services_YouTube, it's the same principle, but the properties are public, and so there are no getter methods. You access the video's properties directly in the `video` object.

In XML_RSS, the values we're interested in are stored in associative arrays. The available methods in this package get the arrays, then you manipulate them directly. It's a small difference, but you should be aware of it in case you want to look at the code. It also means that you will have to check the documentation of the package to see which array keys are available to you.

Let's take a look at how this works in an example. The file is named `RSSPEARTest.php` in the example code. One of Audioscrobbler's feeds gives us an RSS file of songs that a user recently played. The feed isn't always populated because after a few hours, songs that are played aren't considered recent. In other words, songs will eventually drop off the feed simply because they are too old. Therefore, it's best to use this feed on a heavy user of Last.fm.

RJ is a good example to use. He seems to always be listening to something. We'll grab his feed from Audioscrobbler:

```
<?php
   include ("XML/RSS.php");
   $rss =& new XML_RSS("http://ws.audioscrobbler.com/1.0/user/RJ/
                                recenttracks.rss");
   $rss->parse();
```

We start off by including the module and creating an XML_RSS object. XML_RSS is where all of the array get methods reside, and is the heart of this package. It's constructor method takes one variable—the path to the RSS file. At instantiation, the package loads the RSS file into memory.

`parse()` is the method that actually does the RSS parsing. After this, the get methods will return data about the feed. Needless to say, `parse()` must be called before you do anything constructive with the file.

```
   $channelInfo = $rss->getChannelInfo();
?>
```

The package's `getChannelInfo()` method returns an array that holds information about the metadata, the channel, of the file. This array holds the `title`, `description`, and `link` elements of the RSS file. Each of these elements is stored in the array with the same key name as the element.

```
<?= "<?xml version=\"1.0\" encoding=\"UTF-8\" ?>" ?>
```

The data that comes back will be UTF-8 encoded. Therefore, we need to force the page into UTF-8 encoding mode. This line outputs the XML declaration into the top of the web page in order to insure proper rendering. Putting a regular `<?xml` declaration will trigger the PHP engine to parse the declaration. However, PHP will not recognize the code and halt the page with an error.

```
<html>
   <head>
   <title><?= $channelInfo['title'] ?></title>
   </head>
   <body>
      <h1><?= $channelInfo['description'] ?></h1>
```

Here we begin the actual output of the page. We start by using the array returned from `getChannelInfo()` to output the `title` and `description` elements of the feed.

```
         <ol>
           <?php foreach ($rss->getItems() as $item { ?>
              <li>
               <?= $item['title'] ?>:
               <a href="<?= $item ['link'] ?>"><?= $item ['link'] ?></a>
              </li>
           <?php } ?>
        </ol>
```

Next, we start outputting the items in the RSS file. We use `getItems()` to grab information about the items in the RSS. The return is an array that we loop through with a `foreach` statement. Here, we are extracting the item's `title` and `link` elements. We show the title, and then create a hyperlink to the song's page on Last.fm. The `description` and `pubDate` elements in the RSS are also available to us in `getItems`'s returned array.

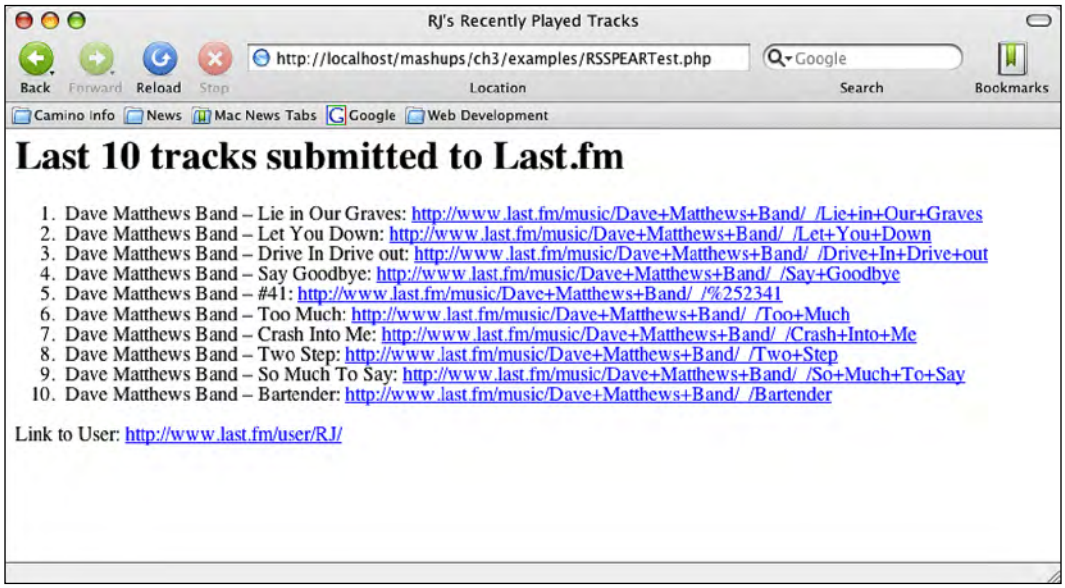
```
        Link to User:
        <a href="<?= $channelInfo['link'] ?>"><?=
                                            $channelInfo['link'] ?></a>
   </body>
</html>
```

Finally, we use the channel's `link` property to create a hyperlink to the user's Last.fm page before we close off the page's `body` and `html` tags.

Using More Elements

In this example, the available elements in the channel and item arrays are a bit limited. `getChannelInfo()` returns an array that only has the title, description, and link properties. The array from `getItems()` only has title, description, link, and pubDate properties. This is because we are using the latest release version of XML_RSS. At the time of writing this book, it is version 0.9.2. The later versions of XML_RSS, currently in beta, handle many more elements. Elements in RSS 2.0 like `category` and `authors` are available. To upgrade to a beta version of XML_RSS, use the command PEAR upgrade `-f XML_RSS` in the command line. The `-f` flag is the same flag we used to force the beta and alpha installations of Service_YouTube and File_XSPF. Alternatively, you can install the beta version of XML_RSS at the beginning using the same `-f` flag.

If we run this page on our web browser, we can see the successful results of our hit.



At this point, we know how to use the Audioscrobbler feeds to get information. The majority of the feeds are either XSPF or RSS format. We know generally how the YouTube API works. Most importantly, we know how to use the respective PEAR packages to extract information from each web service. It's time to start coding our application.

Mashing Up

If you haven't already, you should, at the very least, create a YouTube account and sign up for a developer key. You should also create a Last.fm account, install the client software, and start listening to some music on your computer. This will personalize the video jukebox to your music tastes. All examples here will assume that you are using your own YouTube key. I will use my own Last.fm account for the examples. As the feeds are open and free, you can use the same feeds if you choose not to create a Last.fm account.

Mashup Architecture

There are obviously many ways in which we can set up our application. However, we're going to keep functionality fairly simple.

The interface will be a framed web page. The top pane is the navigation pane. It will be for the song selection. The bottom section is the content pane and will display and play the video.

In the navigation pane, we will create a select menu with all of our songs. The value, and label, for each option will be the artist name followed by a dash, followed by the name of the song (For example, "April Smith—Bright White Jackets"). Providing both pieces of information will help YouTube narrow down the selection.

When the user selects a song and pushes a "Go" button, the application will load the content page into the content pane. This form will pass the artist and song information to the content page via a `GET` parameter. The content page will use this `GET` parameter to query YouTube. The page will pull up the first, most relevant result from its list of videos and display it.

Main Page

The main page is named `jukebox.html` in the example code. This is our frameset page. It will be quite simple. All it will do is define the frameset that we will use.

```
<html>
<head>
<title>My Video Jukebox</title>
</head>
   <frameset rows="10%,90%">
      <frame src="navigation.php" name="Navigation" />
      <frame src="" name="Content" />
   </frameset>
</html>
```

This code defines our page. It is two frame rows. The navigation section, named `Navigation`, is 10% of the height, and the content, named `Content`, is the remaining 90%. When first loaded, the mashup will load the list of songs in the navigation page and nothing else.

Navigation Page

The navigation page is named `navigation.php`. This page will use File_XSPF and XML_RSS to load the songs from Last.fm's Top Tracks and Recent Tracks feeds for the user, and merge them together to create a select menu.

```
<?php
  require_once ('File/XSPF.php');
  require_once ('XML/RSS.php');
```

In the beginning, the File_XSPF and XML_RSS PEAR packages are loaded.

```
  $songsArray = array();
```

An array to hold all the songs is initialized. We need this because we are dealing with two feeds—Top Tracks and Recent Tracks. Also, both feeds return different things. File_XSPF returns an array of song objects. RSS_XML returns an associative array of values where the property names are the key in the array.

```
  //Top Tracks
  $xspfObj =& new File_XSPF();
  $xspfObj->parseFile('http://ws.audioscrobbler.com/1.0/user/
                              ShuTheMoody/toptracks.xspf');
  $topTracks = $xspfObj->getTracks();
```

First, we create an array of song objects from the Top Tracks XSPF feed.

```
//Recent Tracks
$rss =& new XML_RSS('http://ws.audioscrobbler.com/1.0/user/
                            ShuTheMoody/recenttracks.rss');
$rss->parse();
$recentTracks = $rss->getItems();
```

We extract the second, associative array from XML_RSS:

```
foreach ($topTracks as $trackObj) {
   $songsArray[] =
             $trackObj->getCreator() . " - " . $trackObj->getTitle();
                                  }
```

We use the `getCreator()` and `getTitle()` methods on the Top Track objects to create a string in the Artist—Song format, and we place it in `$songsArray`.

```
foreach ($recentTracks as $tracksArray) {
   $tempSong =
            htmlentities($tracksArray['title'], ENT_COMPAT, 'UTF-8');
    $songsArray[] = str_replace('&ndash;', "-", $tempSong);
                                        }
```

By default, the "Artist—Song" format is what the RSS feed returns. Therefore, we can just extract it and place it into the `$songsArray`. However, we have to do a little massaging first. As the data is coming from an XML-based file, certain characters are encoded. This includes the dash mark inbetween the artist and song. The first line uses the PHP `htmlentities()` function and converts the value from XML encoding to the equivalent HTML entity. In this case, the dash in the RSS file becomes "`&ndash;`" This new value is placed inside a variable named `$tempSong`. The next line replaces `&ndash;` with a regular dash character, and pushes it into `$songsArray`.

```
$songsArray = array_unique($songsArray);
sort($songsArray);
```

Finally, we make the list presentable. `array_unique()` will eliminate duplicate songs between the Top Tracks feed and the Recent Tracks feed. `sort()` will sort the list for us alphabetically. This ends the preliminary required PHP code. We can start with the HTML next:.

```
?>
<?= '<?xml version="1.0" encoding="UTF-8" ?>' ?>
<html>
    <head><title>Selections</title></head>
<body>
    <form method="GET" action="content.php" target="Content">
```

```
<select name="query">
<?php foreach ($songsArray as $key => $value) { ?>
     <option value="<?= $value ?>"><?= $value ?></option>
<?php } ?>
</select>
<input type="submit" value="Go" />
</form>
</html>
```

Again, before we start with the actual HTML, we use a PHP echo to declare the encoding on this page. In the HTML, we create the form. The form will use a `GET` method to pass the selected option value to the `content.php` page targeted for the `Content` frame. We use `$songsArray`, which we previously populated to create the select menu.

Finally, an input button is added to trigger the load. The `form` and `html` tags are then closed.

Content Page

Our content page, named `content.php`, is loaded when the form on the navigation page is submitted. The form sends the artist and song title to `content.php` via a query parameter named `query`. `content.php` will have to take this parameter, pass it to YouTube's Web API, and display the results.

```
<?php

  require_once ('Services/YouTube.php');

  //YouTube parameters
  $devId = 'YOU OWN YOUTUBE DEVELOPER ID';
  $tag = $_GET['query'];

  //YouTube Result Parameters
  $videosArray = array();
  $firstVideo = null;
```

We start off with some basic initialization. We pull in the Services_YouTube package, set up the `$dev_id` variable, which holds our YouTube Developer ID, and set up a variable named `$tag` which holds the `GET` parameter received when the page is called.

Next, an array is set up to hold the video return results. A variable named `$firstVideo` is declared. A query can, and often does, return multiple results, with the most relevant result returned at the top. `$firstVideo` will hold this most relevant hit.

```
$youtube = new Services_YouTube($devID, array('usesCache' => true));
$videos = $youtube->listByTag($tag);

$videosArray = $videos->xpath('//video');
$firstVideo = $videosArray[0];
```

The first three lines of this block operate just like the Services_YouTube example: a Services_YouTube object is declared, the service is called, and the results are queried through XPATH.

We now have an array of all results. As we're only interested in the most relevant, we capture the one at index position 0 and place it into the `$firstVideo` array. Our setup PHP code is now completed, and we can begin the HTML.

```
?>
<?= '<?xml version="1.0" encoding="UTF-8" ?>' ?>
<html>
<head><title>Content</title></head>
<body>
<h1>YouTube Query Results for <?= $_GET['query'] ?></h1>

<?php if ($firstVideo) { ?>
```

In the HTML, we encounter a PHP `if` statement. This `if` statement checks to see if the `$firstVideo` array actually has anything in it. We check to see if the object exists.

```
Title: <?= $firstVideo->title ?><br />
<object width="425" height="350">
<param name="movie" value="http://www.youtube.com/v/<?=
                                  $firstVideo->id ?>"></param>
<param name="wmode" value="transparent"></param>
<embed src="http://www.youtube.com/v/<?= $firstVideo->id ?>"
type="application/x-shockwave-flash" wmode="transparent"
width="425" height="350"></embed>
</object>
<p>
<a href="<?= $firstVideo->url ?>"><?=
                           $firstVideo->title ?>'s YouTube Page</a>
</p>
```

If the object exists, we use it to create the display page. The video object has three properties that we use in this section:

- `title` is the YouTube title.
- `id` is the YouTube unique identifier.
- `url` is the URL to the YouTube page.

The `id` is particularly important. Each video's page on YouTube has sample code you can use to copy and paste into your own web page. This will embed the video into the page and play it within the page. The sample code involves an `object`, two `param` tags, and an `embed` tag. We have copied this code into our application. However, we substitute the hard-coded example ids with our object's `id` property.

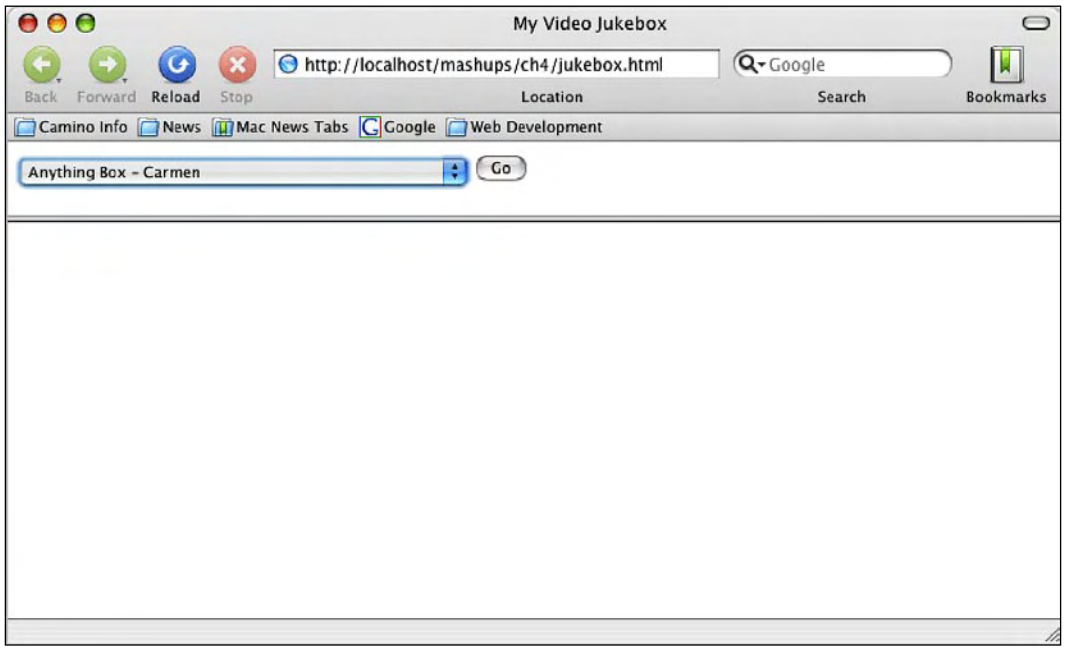
```
<?php } else { ?>
   No Results Found
<?php } ?>
</body>
</html>
```

We add an `else` block to display a, "No Results Found", message if YouTube returns nothing from the query. Finally, we close the `body` and `html` tags.

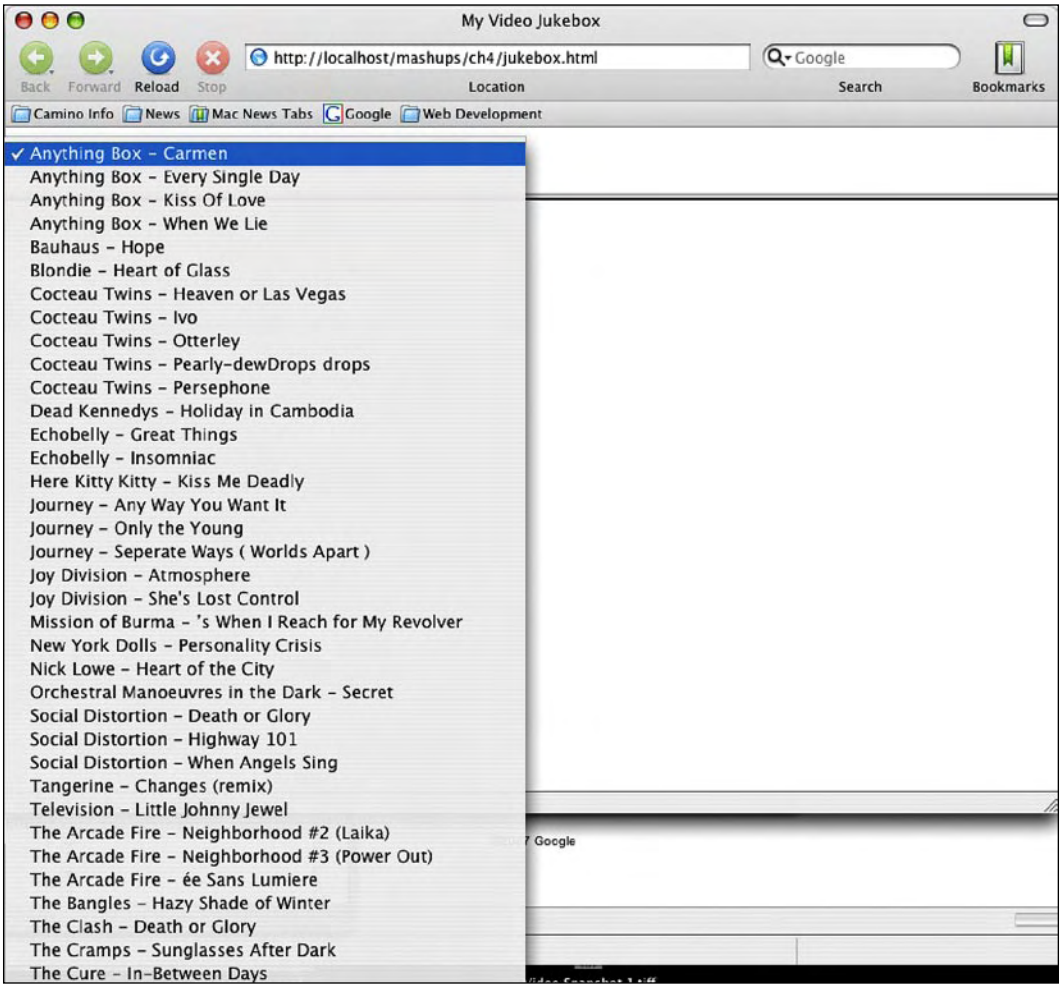
Place these three files into the same, web-server accessible directory on a web server. You can start using the mashup by launching `jukebox.php` in your web browser.

Using the Mashup

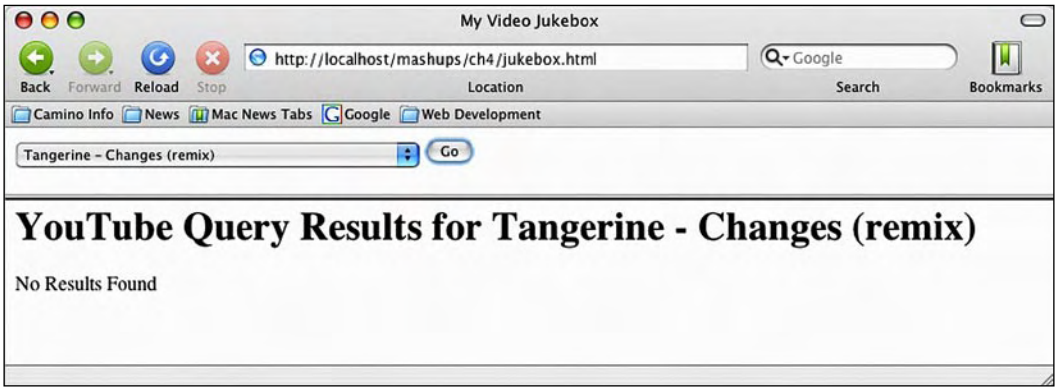
Using the mashup is quite simple. Load `jukebox.html` into your web browser.



When you load `jukebox.php`, you will see a page similar to the one shown on the previous page. `navigation.php` will load, and the select menu will be created from our Last.fm feeds.

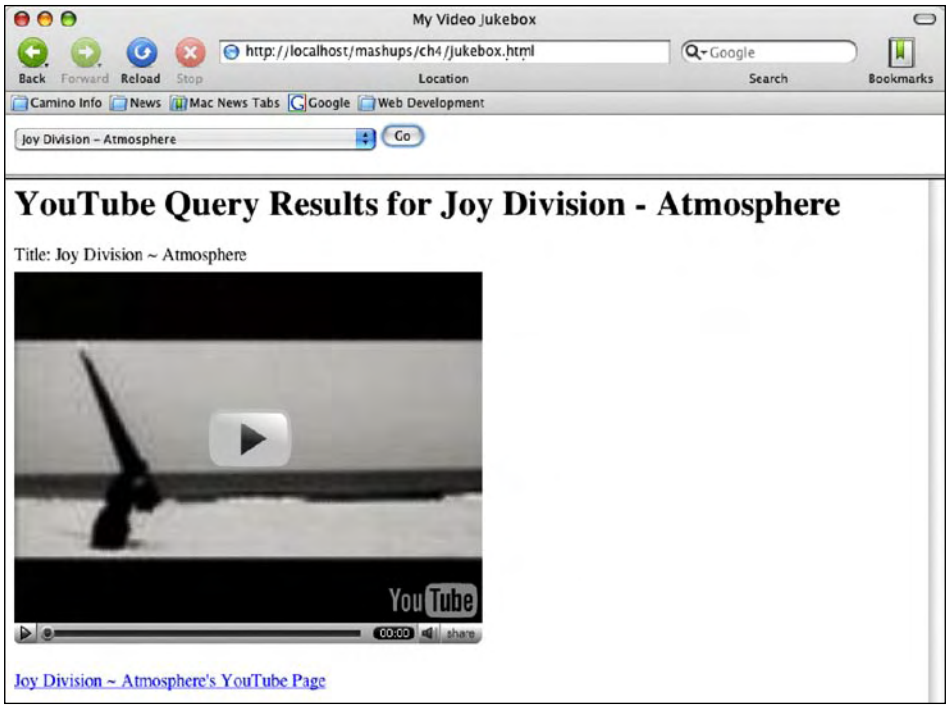


Click on the menu. You will see a list of the songs from the two feeds. Select one and press the **Go** button.



My first selection, a song by Tangerine, yielded no results. We see the error message in the content pane. This isn't surprising because they are a small, up and coming local band based in Pittsburgh, Pennsylvania. Once they become more popular, their legions of fans will grow and hopefully someone will put something up on YouTube.

Let's select a more well-known band and song.



Our next selection, Atmosphere by Joy Division, was more successful. In this case, someone upload Atmosphere's music video. Pressing on the Play button will start playing the video without leaving our mashup.

The great thing about this mashup is the user-driven nature of YouTube. You can't be sure of what will be returned In Atmosphere's case, we saw the official music video that someone uploaded. In many other cases, we get to see rare live performances that someone recorded with a camcorder and uploaded. Sometimes we'll see some creative minds who made their own music video and set it to the music that we queried. Other times, the video may have nothing to do with the song at all, except that it plays in the background. For example, when I submitted The Clash's classic Death or Glory, the only video that was returned was a tribute slideshow of, of all people, Harry Potter star Daniel Radcliffe that someone created.

Summary

In this mashup, we used two different web APIs—one from video repository site YouTube, and the XML feeds from social music site Last.fm. We took a look at three different XML-based file formats from those two sites: XSPF for song playlists, RSS for publishing frequently updated information, and YouTube's custom XML format. We created a mashup that took the songs in two Last.fm feeds and queried YouTube to retrieve videos based on the song.

If we were to create our own XML-based parsers to parse the three formats, this would have taken much more time than it actually did. We found that the PHP Extension and Application Repository, PEAR, already had parsers we could use; one for each of the three formats. Using these PEAR packages, we were able to create an object-oriented abstraction of these formats, which allowed us to easily finish our application.

5
Traffic Incidents via SMS

Project Overview

What	Deliver content from the California Highway Patrol Live Incidents website to mobile users through Short Messaging Service (SMS)
Protocols Used	Not applicable
Data Formats	RSS
Tools Featured	PHP's DOM Extensions
APIs Used	411Sync

And, on this project, we will learn how to use the dark arts. No, I'm not talking about controlling mindless zombies through black magic. I'm talking about screen-scraping.

In this project, we will screen-scrape from the California Highway Patrol website. The CHP maintains a website of traffic incidents. This site auto-refreshes every minute, ensuring the user gets live data about accidents throughout the state of California. This is very valuable if you are in front of a computer. If you are out and about running errands, it would be useless unless you had an Internet-enabled personal digital assistant. Even then, the site uses JavaScript and frames, and may not work on many PDA web browsers. More widely available than Internet PDAs are cell phones with text messaging through Short Messaging Service (SMS) capabilities. Our mashup will make the CHP data available through SMS instead of a web browser.

Our application will take the information from the CHP Incidents website and modify it into an RSS file. We will then sign-up for an account at 411Sync.com to make this feed available to us just by sending an SMS message to a phone number. When a user sends an SMS message with a keyword that we have reserved to 411Sync.com, they will hit our feed and return it to the user in another SMS. Our mashup makes data more accessible, and we don't have to invest in new hardware to send SMS messages.

411Sync.com makes an HTTP request to our script, so this project will require a publicly accessible web server to use for development and deployment. This is the only project in this book that has such a requirement.

Screen Scraping the PHP Way

Screen-scraping has always been a dubious practice and the trigger of many lawsuits. Many website owners, not surprisingly, jealously guard their content and are resentful of anyone who sends spiders to grab data and use it on their own site. They view screen-scrapers as necrotic vampires who steal users, siphon-off advertising revenue, and suck-up server and bandwidth resources without giving anything back. On the other side, screen scrapers have argued that if content providers don't want to share their data, they should not put it in such a public place like the World Wide Web.

Amongst honest web developers, screen-scraping without prior approval is generally frowned upon. From a legal standpoint, depending on where you are, restrictions on screen-scraping usually fall into the realm of civil contract law through website Terms and Conditions clauses. Those who wish to screen-scrape should, first and foremost, get permission to do so. Otherwise, they should not be surprised if they receive a Cease and Desist letter from the scrapee's attorney. Yes, we have permission from the California Highway Patrol to screen scrape for this book.

Even if you get over the legal hurdles, you have technical hurdles to overcome. Earlier, screen scraping for the web often involved writing a script with a language such as Perl that loaded the information in memory. Through complex and cryptic regular expressions, the program would come through the text and look for what it needed. If the program was interested in data that was stored in the third column of a table, it would have to go through all of the `<table>`, `<tr>`, and `<td>` tags, maintaining a count of where it is at any point, if it's interested in the data it encountered, and if it is, store it somewhere else. Screen scrapers would have to take into account malformed HTML. If the page was changed at all, it's time for a rewrite.

Luckily, these days of tedium are long past. There are two key differences that have made it much easier to screen scrape today than it was years ago.

First, it is more common to have a website that is generated through programs than hand-coded sites. Customer relationship management systems, document management systems, bulletin board forums, and plain database web applications are just some of the examples from where we may want to extract data. Not only is there more data out there, but having programs generate the pages theoretically reduces the chance of human coding errors. Less coding errors mean scraping programs have to worry less about inconsistencies.

The second thing that has changed is the widespread adoption of the Document Object Model. In the first chapter, we used a SAX-based parser to parse an XML document. We touched on the difference between SAX-based parsers and DOM-based parsers. The key difference being that a SAX-based parsers parse as it is reading the data, while a DOM-based parser loads the document into memory before taking any action. When we look at the data source later, we will see that the CHP site is a good candidate for a DOM-based parser, which is what we will use. Before we start actually mucking around with a DOM-based parser, we should know what a Document Object Model even is.

A DOM is an official W3C specification that describes an object model representation of an XML document. Whatever is holding the model knows all of the elements in a document, and their relation to each other, like parent, child, and siblings. Simply holding the document in memory is not enough; whatever loads the document also provides hooks for a developer to access data in the document. The DOM is a programmatic way to access parts of an XML document.

A practical, specific example is how a web browser sees an HTML document. The browser knows there is a root element named `HTML`. There are two child elements named `HEAD` and `BODY`. Within the `BODY`, there may be elements such as `P` and `TABLE`. In `TABLE`, there are `TR` and `TD`. The browser knows that some `TD` elements are siblings of each other, and which specific `TR` element is a `TD` element's parent. The browser knows how each piece of a web page relates to every other piece.

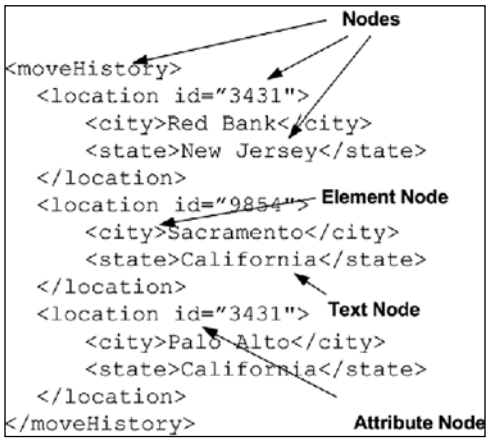
All of the elements, values, and attributes in a document are often referred to as a DOM tree, because when diagrammed out visually, elements and their parents and children often look like branches in a tree.

A DOM-based parser works in a similar way. It loads the entire XML document in memory, and through different commands, you can extract values or navigate to another part of the document. All operations are done against the document in memory. You can also create nodes and insert them into the model.

PHP 5 has a built in DOM-based parser in the DOM XML functions (`http://www.php.net/manual/en/ref.dom.php`). The parser is actually a collection of classes that represent components of a DOM, and the documented DOM functions are methods in those classes. The classes are interfaces defined in the W3C DOM Level 3 specification. The DOM Functions actually follow the official specifications fairly closely. The official specs can be found at `http://www.w3.org/TR/2003/WD-DOM-Level-3-Core-20030226/DOM3-Core.html`. The following table summarizes the essential classes in the W3C DOM and their implementation in PHP DOM.

DOM Class	PHP Equivalent	Purpose
Document	DOMDocument	Represents the entire XML document. You can think of it as the root element. In parsing, this is the object you first start with.
Node	DOMNode	Every "part" of an XML document. In the DOM, everything is a type of node—elements are Element Nodes, attributes are Attribute Nodes, etc. Node methods operate on nodes in a high level, like traversing through the tree and naming.
NodeList	DOMNodeList	In the DOM, methods that return nodes are usually in a NodeList, which is just what it sounds like—a list of node objects. In PHP, these functions usually return an array of nodes.
Element	DOMElement	An element in the XML document other than the root; for example, "div".
Attribute	DOMAttribute	An attribute within an, for example, class="content".
Text	DOMText	In the DOM, a Text object is a text value of the element or attribute.
ProcessingInstruction	DOMProcessingInstruction	Parser processing instructions.

The following diagram shows common node types in an a simple XML snippet:



As the PHP DOM functions implement a specification based on classes, they work slightly differently than other PHP functions. The official PHP documentation takes a little bit of adjustment to find what you're used to, and because the functions are object methods, you must first have the object created in order to use them.

When you look at a DOM function in the documentation, you'll notice that the class name is also part of the function name. The classes in PHP implement required interfaces of the DOM. A class's methods are relevant to the DOM component that they are named after. For example, the `DOMElement` class is equivalent to the Element interface of the DOM, and affect the elements in a document. In PHP.net's function reference, other functions are usually organized alphabetically in their category. DOM XML functions in PHP.net are grouped by class before being alphabetized.

If you explore through the DOM XML documentation, you'll see that objects are very, very important. Most of the functions take in and return other DOM XML objects, and not just the simple data types of other PHP functions. Beginners will like to have the documentation open and handy to see which objects are returned with a function because the return type obviously affects which functions are available. We'll see this in action as we parse through some code.

Parsing with DOM Functions

The first step in parsing with the DOM Functions is to load the XML document into the `DOMDocument` class. There is several functions you can use to do this. Each function is in `DOMDocument`, and are used for a specific purpose.

Function	Purpose
DOMDocument->load()	Loads a XML file
DOMDocument->loadHTMLFile()	Loads an HTML page
DOMDocument->loadXML()	Loads a string of XML text
DOMDocument->loadHTML()	Loads a string of HTML text

The first two functions, `load()` and `loadHTMLFile()`, are used to load an XML file or HTML page. They take the URL of the resource as the input parameter. You can pass either a path to a file on the local server or a fully qualified URL to another server.If you pass a URL, the functions will execute a `GET` request to the server hosting the file. The resource, for example loaded must have the appropriate content type. That is, the content type must be text/xml or text/html, respectively.

The next two functions, `loadXML()` and `loadHTML()`, serve the same purpose, but work in a different way. Instead of reaching across a network to get a file like the former two, these two functions load a string representation of XML or HTML. For example, if your XML or HTML is stored in a variable, use these functions to load them.

Basic Element and Attribute Parsing

Once you have a document loaded, you can start using the functions in the `DOMDocument` class to extract information. Let's walk through an example session. In the examples code for this project, there is a file named `domxmlexample.php`. This small example shows the basic procedure for using DOM XML. This file has a small XML block stored in a variable. Our code then extracts data from this block using PHP's DOM XML functions.

```
<?php
  $xml = '<posts>
  <date>February 26, 2007</date>
  <post author="James Edward Farley, III">
     <title>Damn Intarwebs!</title>
     <date>April 23, 2007</date>
  </post>
  <post author="Elizabeth Rankin">
     <title>Chinese New Year in New York City</title>
     <date>March 3, 2007</date>
  </post>
  <post author="Evan Spiegel">
     <title>I got an AppleTV</title>
     <date>May 2, 2007</date>
  </post>
  </posts>';
```

This example uses a small piece of XML that one might find in a blog feed. The root element is named posts, and each individual post is wrapped in a post element. The author is an attribute in the post element. Title and date information are child elements of the post.

```
$aDom = new DOMDocument();

  try {
     @$aDom->loadXML($xml);
  } catch (Exception $ex) {
     $aDom = false;
  }
```

The `DOMDocument` object is instantiated, but is empty. We use `loadXML()` to load the XML string into the `DOMDocument`.

Now we can use methods on the `DOMDocument` object to extract data from itself. The first example will pull all of the dates from the document.

```
<p>
  All Dates:<br />

<?php
  $allDates = $aDom->getElementsByTagName('date');

  foreach ($allDates as $date) { ?>
           Date: <?= $date->nodeValue ?> <br />
<?php } ?>

</p>
```

The first line calls `getElementsByTagName()` and gets all elements in the document with a tag name of `date`. This includes all `date` elements within each post and the `date` element right after the root element:

```
<date>February 26, 2007</date>
<date>April 23, 2007</date>
<date>March 3, 2007</date>
<date>May 2, 2007</date>
```

In the DOM, `getElementsByTagName()` will return a `NodeList` object, which is a List of nodes. Lists are a special type of data structure found in other languages, like Java, that are a type of array. In PHP, a `NodeList` is implemented similarly to an array of `DOMElement` objects. To use the dates we extracted, we must loop through this array. A node holds its property inside the `nodeValue` property. In this example, we echo out this property to the browser.

The lesson here is that `getElementsByTagName()` will return all elements regardless of where they are and how they are nested. If we want to isolate the dates that are nested within posts, we need to get the posts first.

```
<p>
Post Dates:<br />

<?php
$posts = $aDom->getElementsByTagName('post');

foreach ($posts as $post) {
   $postDate = $post->getElementsByTagName('date'); ?>
       Date: <?= $postDate->item(0)->nodeValue ?>
<?php } ?>
</p>
<p>
```

The first line of PHP code will put an array of node objects into a variable named `$posts`.

```
<post author="James Farley, III">
<post author="Elizabeth Rankin">
<post author="Evan Spiegel">
```

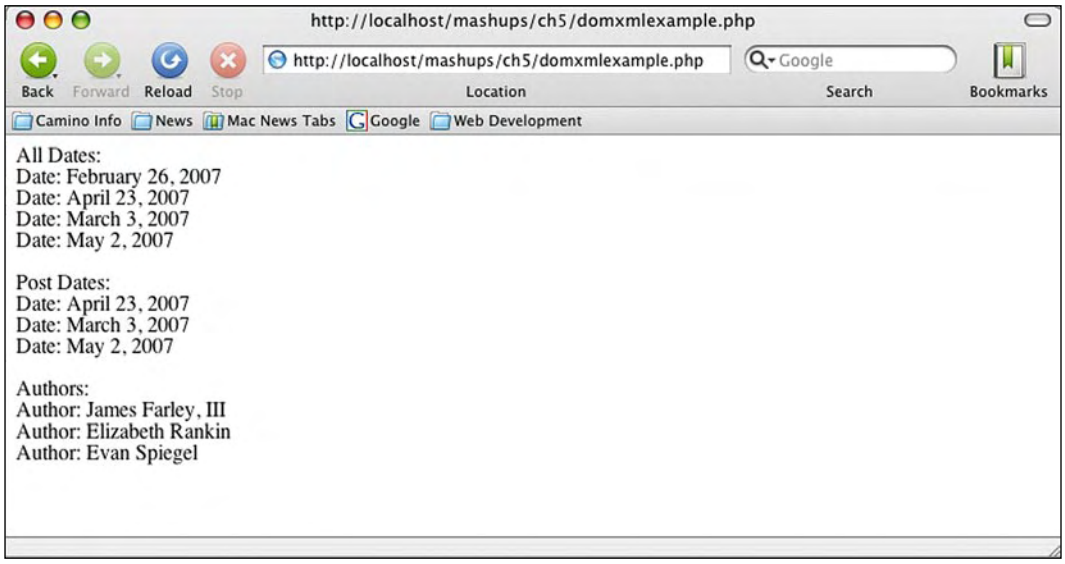
Every child element of `post` is now available to us. We loop through the `$posts` array and extract data from each `post` element. Now, though we are limited to methods that are available in `DOMElement`, and because `DOMElement` implements `DOMNode`, the `DOMNode methods`, too. Looking at the `DOMElement` documentation, `getElementsByTagName()` is also a method in `DomElment`, so we'll use it. Every time we loop through `$posts`, we'll call `getElementsByTagName()` again, but this time, it's isolated to the post element we are in. `$postDate` will hold this new `DOMNodeList`. We could loop through this again, but as we know that there is one and only one date element nested in post, we can access it directly using the `DOMNodeList's item()` method.

Even though a `DOMNodeList` functions similarly to an array, it isn't exactly the same. You cannot directly access values by placing the array index within brackets like for other PHP arrays. Instead, the `item()` method will return the array element. Pass the index value that you wish to extract to `item()`. As the only `date` element is in the first position, we give `item()` an index parameter of zero. This gives us a `DOMElement` in return, and we can use the nodeValue property again to get the value.

```
Authors:<br />
<?php foreach ($posts as $post) { ?>
Author: <?= $post->getAttribute('author') ?><br />
<? } ?>
```

Another common task is to retrieve attributes of an XML element. `DOMElement's getAttribute()` method will accomplish this. This method takes a name of the attribute that you wish to look for in a `DOMElement` and return its value. In our sample XML, the post author's name is an attribute of post. We can loop through `$posts` again. This time for each post element, we call `getAttribute()` and pass the string "author," which is what we are seeking. The documentation states that `getAttribute()` will simply return the attribute value (and not a `DomAttribute` object), so there is no need to access any further properties or methods. If you need the node, there is a companion method named `getAttributeNode()`, that will return the node instead of the attribute value.

Loading this example page in a browser will return the screenshot below:



Testing the Schema

The previous example works fine when the XML document is well structured. However, if the document is not well-formed, a script error may occur. For example, if you use method to retrieve an element, but the element does not exist, subsequent calls to any `DOMElement` methods on it will result in an non-existent object error. To help prevent this, the DOM Functions include some methods that queries and validates the XML schema. All of these methods return true on success or false on failure. You can use these methods to test for an element or object before calling methods on it.

Object	Method	Description
DOMDocument	relaxNGValidate	Validates an RNG file for well-formedness
DOMDocument	relaxNGValidateSource	Validates a string of RNG data for well-formedness
DOMDocument	schemaValidate	Validates an XML file for well-formedness
DOMDocument	schemaValidateSource	Validates a string of XML data for well-formedness
DOMDocument	Validate	Validates a string of XML based on its DTD
DOMElement	has Attribute	Checks whether the element has an attribute
DOMElement	hasAttributeNS	Checks whether the element has an attribute within a given namespace
DOMImplementation	hasFeature	Checks whether the document implementation has a certain W3C feature
DOMNode	has Attributes	Checks whether the node has any attributes
DOMNode	hasChildNodes	Checks whether the node has any child nodes
DOMNode	isDefaultNamespace	Checks whether a given URI is the document's default namespace
DOMNode	isSameNode	Checks whether two given nodes are the same node
DOMNode	isSupported	Checks whether the document has a certain W3C feature given a version
DOMText	isWhitespaceInElementContent	Checks whether the text node has whitespace

More About PHP's Implementation of the DOM

The importance of the DOM functions being completely object-oriented cannot be overstated. If you look at the official DOM specification, you can see there are a lot of special properties in each class. The Node class has many properties that deal with other nodes. For example, `firstChild` holds the first child node and `nextSibling` holds the next sibling node. There are properties that let you move up, down, and laterally from any node in the DOM.

The majority of the DOM classes are interfaces for the `Node` class. This means that if you have an object that extends `Node`, you will have access to these navigation properties. This not only gives you the ability to navigate through the document easily, but it gives you options on how you can extract data.

For example, `DOMElement` extends `DOMNode`. If you have a `DOMElement` object, you can grab all of the child nodes back as a `DOMNodeList` simply by using `DOMNode's childNodes` property. `DOMNode's` traversal properties is summarized in the following table:

Property Name	Return Type	Description
parentNode	DOMNode	The parent node that contains the DOMNode
childNodes	DOMNodeList	A DOMNodeList of all direct child nodes of this DOMNode
firstChild	DOMNode	The first child node of the DOMNode
lastChild	DOMNode	The last child node of the DOMNode
previousSibling	DOMNode	The node immediately preceding the DOMNode
nextSibling	DOMNode	The node immediately following the DOMNode

One of the things we did in our previous example was to grab the dates only associated with a post. We grabbed the posts, looped through it, extracted another `NodeList` using `getElementsByTagName`, now use the `item` method and pass it the assumed index position.

```
<?php
  $posts = $aDom->getElementsByTagName('post');
  foreach ($posts as $post) {
     $postDate = $post->getElementsByTagName('date'); ?>
        Date: <?= $postDate->item(0)->nodeValue ?><br />
<?php } ?>
```

We could have also written it this way:

```
<?php
  $posts = $aDom->getElementsByTagName('post');
  foreach ($posts as $post) {
     $childNodes = $post->childNodes; ?>
  Date: <?= $childNodes->item(1)->nodeValue ?><br />
<?php } ?>
```

In this example, we grab the `NodeList` of posts again and loop through it. This time, however, we don't have to call `getElementsByTagName` a second time. Instead, we grab the child nodes using `DOMNodes`'s `childNodes` property. This also returns a NodeList, so we can use the item method to grab the first element. The main advantage to this method is that we are using a property (`childNodes`) instead of a function call (`getElementsByTagName`) to get the same information. Accessing properties is theoretically faster than calling a function because there is no thinking necessary from the parser—the data is just passed and assigned.

Beware of White Space

We use an index position of 1 for the `item` method because in the XML created, line breaks after the post element. The first item returned, `index(0)` is the line break itself. It is `DOMText` object containing white space. What we really want is the second item, which is the `date` element. This is a common pitfall when working with all XML documents. Even though you may see an element right before or after the element you're interested in, there may be white space character data objects in-between.

Using a full HTML file, let's take a look at another example on using these properties. The example file named `examplehtml.html` is a small, valid HTML file with two lists.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
   <title>Test ID Page</title>
</head>
<body>
   <p>This is a test of lists.</p>
   <ul id="myList">
      <li>First Item</li>
      <li>Second Item</li>
      <li>Third Item</li>
   </ul>
   <ul id="anotherList">
      <li>You</li>
      <li>Won't</li>
      <li>See</li>
      <li>This</li>
      <li>At</li>
      <li>First</li>
   </ul>
   </body>
</html>
```

Note the first unordered list has an `id` attribute of `myList`, and the second unordered list has an `id` attribute of `anotherList`. The file `domxmlexample2.php` is a PHP script that does some basic extraction from this HTML file.

```
<?php
  $aDom = new DOMDocument();
  try {
     $aDom->loadHTMLFile('examplehtml.html');
  } catch (Exception $ex) {
     $aDom = false;
  }
```

The beginning of this script does the same thing as the first example. We create a new DOMDocument object and load some XML into it. This time, we use `loadHTMLFile()` to load an HTML file.

```
$firstUL = $aDom->getElementById('myList');
$secondUL = $firstUL->nextSibling;
?>
```

In this next section of code, we grab the two unordered list nodes. The first list is retrieved using the `getElementById()` method which takes the value of an `id` attribute as the parameter and returns a `DOMElement`. The second unordered list could be extracted using the same method. However, as we know the second list is on the same DOM tree level as the first list, and it also immediately follows the first list, we can use `DOMNode's nextSibling` property to retrieve it.

```
Looping through the first list:<br />
<?php foreach ($firstUL->childNodes as $value) { ?>
   <?= $value->nodeValue ?><br />
<?php } ?>
```

This `foreach` loop uses the `childNodes` property, which is a `DOMNodeList`, to retrieve the items in the first unordered list. Each `DOMNode` in the loop gets assigned to `$value`. The `nodeValue` property in each `DOMNode is` the value of each `li` element. Echoing out the `nodeValue` property will display the `li` value on the screen.

```
Looping through the second list:
<?php foreach ($secondUL->childNodes as $value) { ?>
   <?= $value->nodeValue ?><br />
<?php } ?>
```

The second list's `ul` element was extracted using the `nextSibling` property. This property is a `DOMNode`. This is different from `$firstUL` because `getElementById()` returns a `DOMElement`. Nevertheless, for our purposes, this is good enough because the core navigation properties and values are in the `DOMNode` object. Like the previous `foreach` loop, this block gets the child nodes of the element back, and those `nodeValue` properties are used to display the value of the `li` element.

```
What's the first item? <b><?= $firstUL->firstChild->nodeValue ?></b>
```

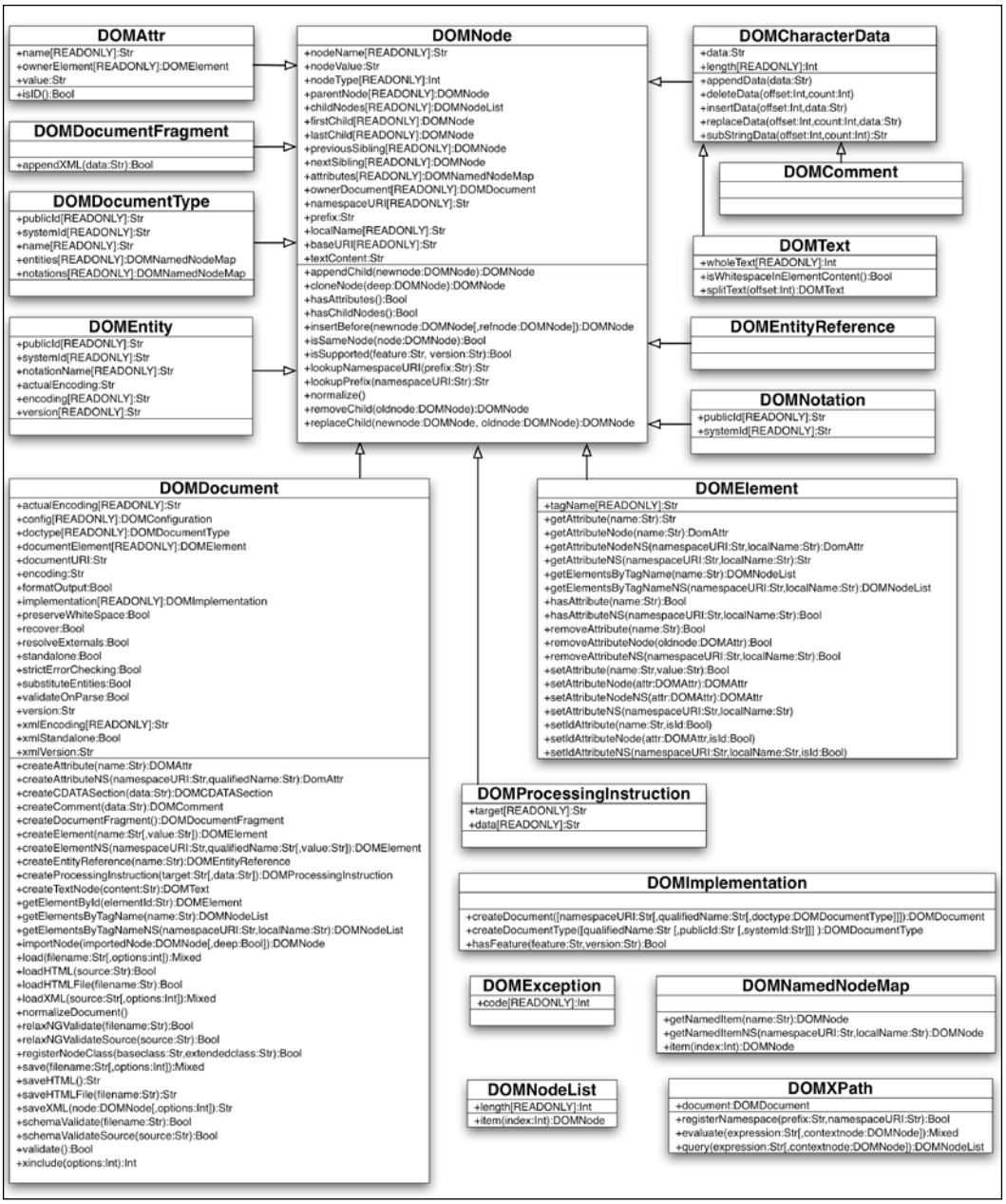
Finally, this last line gets the first item of the first unordered list and displays it. `$firstUL` is the first unordered list element. `firstChild` property holds the `li` element. Finally, we display its `nodeValue` property to get the information about the list. The purpose of the last line is to show how to access elements far away from your current element. Properties, because they are objects, can be chained. It is perfectly reasonable to traverse the DOM tree like such:

```
$testNode->parentNode->previousSibling->firstChild
```

Although wordy and long, this would go up one level from the node you are in, across one level to the previous node on the same level, and then down one level. If you think of this in familial relationships, if you are the `testNode`, then your parent's sibling's child would be your cousin.

Running this example script in the browser will show how this parsing results:

The DOM section in the official PHP documentation lists the implemented DOM classes, their methods, and their properties. Sometimes, though, it's easier to see things graphically. The following diagram shows the PHP implemented DOM classes as a UML class diagram.



This diagram shows the inheritance chain of each object, and every project's properties and methods. At a glance, we can see which class can use the `DOMNode` properties and methods thanks to inheritance as well as the data types we are dealing with. Class diagrams are not that hard to interpret. Each box is a class. A class is divided up into three sections. The top section is the name of the class. The middle section lists the properties. The bottom section lists the methods. If the middle or bottom section is empty, the class does not have any properties or classes, respectively.

Properties are listed in the following format:

```
VisibilityIndicator PropertyName:DataType
```

Visibility indicators are a plus sign (+) for a public property, hash (#) for protected, and a minus sign (-) for private. In the case of PHP DOM, all properties are public. If a property has `[READONLY]` after the property name, it means that the property cannot be changed by the developer. In `DOMNode`, there is a property listed as `+ownerDocument[READONLY]:DOMDocument`. This means that there is a public property named `ownerDocument`. It is read only and accessing it gives you a `DOMDocument` data type.

Methods are listed in this format:

```
VisibilityIndicator MethodName(parameters):returnType
```

The visibility indicators are the same as the properties, and all are public in PHP DOM. The method name is self-explanatory. The parameters are in a `parameterName:datatype` pair. You shouldn't worry about the parameter name because it is used by the function. It is just listed to give you a hint on what to pass. If a parameter is not in brackets ([]) then it is required. Parameters in brackets are optional. Finally, the returned value's data type is given. Let's look at a method in DOMDocument: `+loadXML(source:Str[,options:Int]):Mixed`. This is a public `method` named `loadXML`. It has one required string parameter named `source`, and an optional one named `options` which is an integer. It returns a mixed data type. You'll have to consult the documentation to find out exactly what is the source and options, and what type of "mixed" data you get back.

An arrow from one class to another shows inheritance, with the parent being the class that is being pointed. `DOMText's` parent is `DOMCharacterData`. `DOMCharacterData's` parent is `DOMNode`.

Now that we are familiar with the tool we are going to use, it's time to look at the API and our source data.

411Sync.com API

411Sync.com is a site that provides an interface between your cell phone and information that may normally not be in a format that is mobile-friendly. Some cell phone-managed services it offers include:

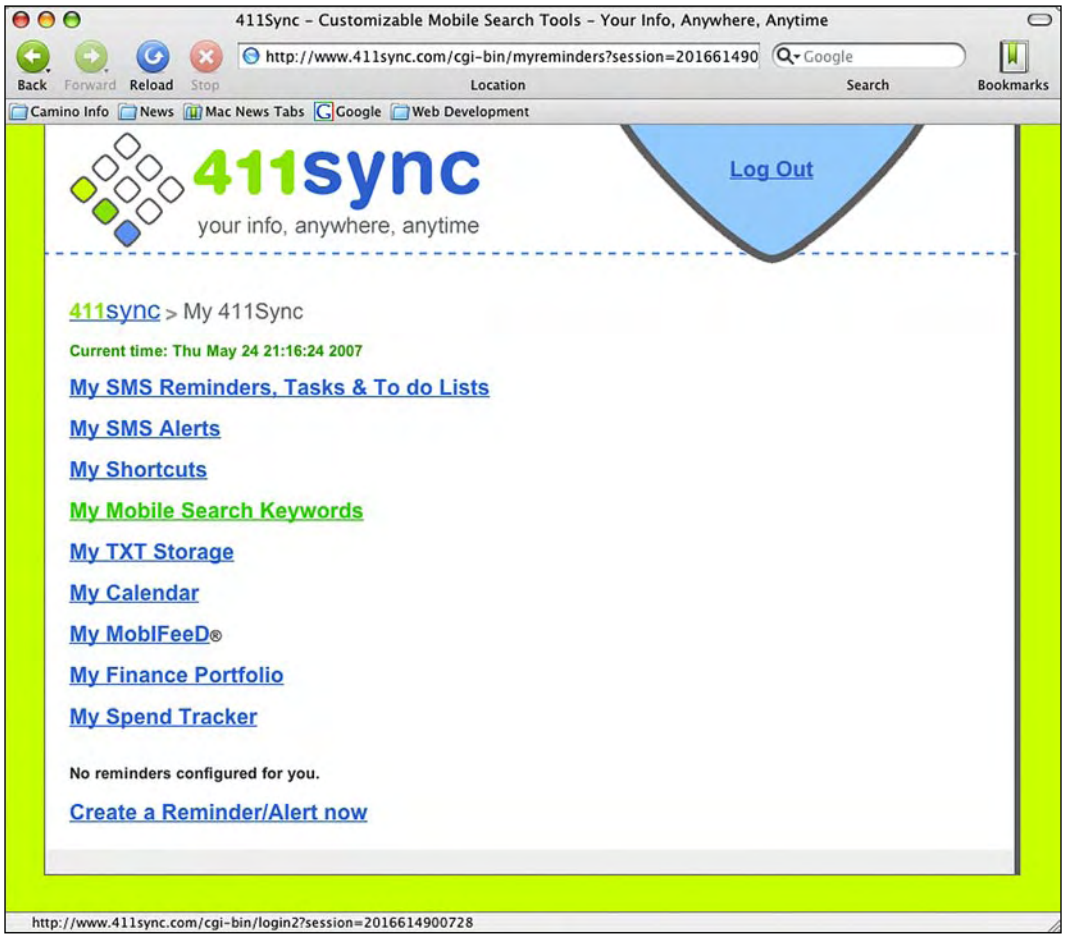
- Create and schedule personal reminders
- A calendar
- Stock portfolio alerts
- Spending trackers
- Text message storage
- Mobile search keywords

The mobile search keywords feature is what we will be using for our mashup to deliver content. It is a nifty, free way for any web developer to create content to be delivered through SMS. Users send a request to 411Sync's phone number. 411Sync then makes a request to your web server, passing any parameters you need. 411Sync takes whatever results come up, formats it, and sends it back to the phone.

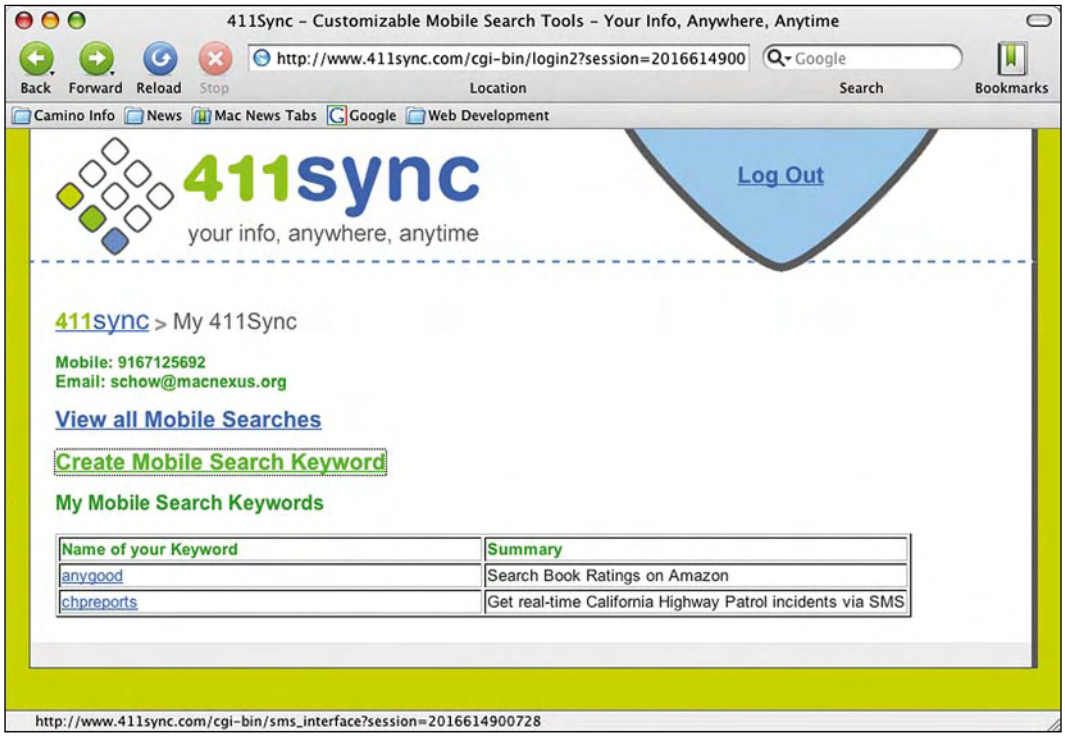
The mobile search keywords feature has developer documentation, located at `http://www.411sync.com/cgi-bin/Developer`, which details the fields the form requires, what they mean, and more information about the necessary RSS response. By parsing and delivering RSS files from developers, 411Sync makes mobile development easy. While it would normally take a lot of time and money to invest in an SMS infrastructure, small developers can have access to this channel by simply making a common RSS file. Debugging is also simpler because all we have to do is make the RSS file.

Creating Your Mobile Search Keyword

Before this happens, you, as the web developer, need to register for a free account with 411Sync.com. You can do this at `http://www.411sync.com/cgi-bin/Register`. You will need your cell phone number and to know your cell carrier. After you have registered, click on **Log-in**. You will be taken to your services homepage.



Click on **My Mobile Search Keywords** to view the search keywords you have created and to create new ones.



Once you create some mobile search keywords, they will appear here for you to edit. For our purposes, you will have to create a new one by clicking on the **Create Mobile Search Keyword** link. This will take you to a form that will create your keyword.

411Sync will publicize your keyword for you on their site directory. Therefore, most of the questions on the keyword creation page asks you what your site is about. Not all of these questions affect the way your search works. However, there are three that must be filled out and require some thought in order for your keyword to work.

Name Your Keyword

This question identifies your service. It is the body of the SMS message from the user to 411Sync. Therefore, it should be descriptive as to what the service is, and as short as possible while meeting the minimum character length requirement, which is currently at six. As this keyword identifies your feed, 411Sync will check to make sure this value is unique in their system.

The keyword I selected for my service is "chpreports".

Format the Users will Use when They Use Your Search

While this question does not directly affect keyword behavior, it will make you think about what the user has to put in. This question essentially asks what is the entire SMS message the user must send to 411Sync. The first word should be the name of the keyword. If there are any variables, they should be named and enclosed in curly brackets.

Later, we will see we need one variable to identify the geographic region we are interested in. Therefore when you create a service, make sure this question is filled in with the keyword and a variable name in curly brackets. In the example, I created a mobile search with the `chpreports {regionCode}`.

HTTP Location of the XML Data

This question asks where your RSS file will be. This should point to the script we are going to create. Note that if the user needs to enter in a variable in order to use your search, you need to manually set up the URL here to handle a variable. For example, in the previous question we identified one variable, `regionCode`. (That we need. "regionCode" is purely informational!) In the script, we do not access it as `$_GET['regionCode']`. 411Sync will take whatever extra parameters it sees and just append it to this URL.

In creating my URL, I set this question to:

```
http://www.shuchow.com/mashups/ch4/feed.php?rid=
```

As parameters are just passed to the URL, we catch it here. We set up one URL and at the end, we start the query string parameter. In this case, it is named rid. When 411Sync is activated and it sends a parameter and `GET` request to the service, that parameter will get stuck at the end of this URL. Then, our script will be able to access it using `$_GET['rid']`.

In more detail, the complete life cycle of a person's request is:

1. User sends a SMS message to 411Sync. The message contains the keyword that identifies your service. The message may also be followed by some parameter. 411Sync sends this parameter to your service.
2. 411Sync's servers send an HTTP `GET` request to the URL you specified.
3. Your script executes, using any `GET` parameters 411Sync passed to it.
4. Your script sends its response back to 411Sync as an RSS file.
5. 411Sync parses this RSS file, extracting the `title` elements.
6. 411Sync passes the content from the title elements and sends an SMS message back to the user.

Therefore, the key thing we need to create is an RSS file. There are a few restrictions, though. Some of these restrictions are outlined in the documentation, some are undocumented, while some are just part of the nature of mobile development.

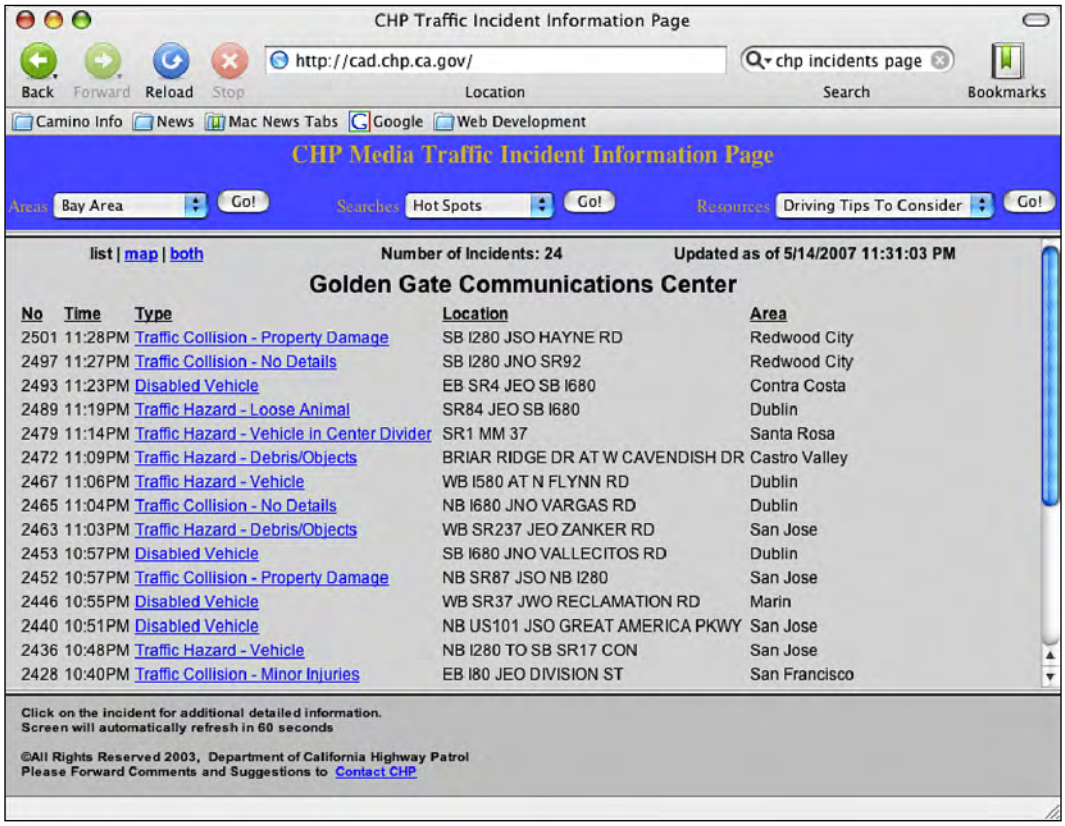
First, the 411Sync parser takes everything in the title elements of the RSS file and delivers it to the user. Our RSS file will have to stuff everything in these elements. Second, while the documentation says that the service can deliver 255 characters to the user, these are `UNICODE` characters. Delivery in ASCII will be significantly less. Therefore, saving space in our response is critical. Finally, remember that typing on a 9-digit keyboard is still somewhat difficult. Our application needs to require just the minimum amount from the user.

We will need to define which parameters we need to create this file. The content for our RSS file will be traffic information coming from the California Highway Patrol.

California Highway Patrol Incident Page

This section will examine the California Highway Patrol Incident page. A user can choose the area they are interested in, and the page will retrieve incidents for that area. We want to let users request traffic reports for the same areas, so we need to examine how this page creates content.

We will do this by dissecting the general HTML structure of the **Incident Page**. The **Incident Page** is located at `http://cad.chp.ca.gov`.



This page is divided into three frames. The bottom frame is just an informational and legal footer. There is a top frame that holds some navigation. The pull-down menu in the upper left corner labelled **Areas** lists the urban centres of California. The traffic incident content is in the middle frame. This data appears to be displayed in a table. There are columns for the time of the incident, type of incident, location, and the general neighborhood. When the areas pull-down is activated and a new area is selected, the content page refreshes with the area you selected. From this behavior, we can guess that the pull down, through a JavaScript event, sends some sort of information somewhere, probably in the form of an HTTP `GET` or `POST` request, and the content page changes.

In your web browser, view the frame source code for the top header. In Mozilla and Opera, you can do this by right clicking on the top frame and selecting **Frame | View Source**. For Safari or Internet Explorer, you can do this by right clicking on the top frame and selecting **View Frame Source**, or **View Source** in the case of Internet Explorer.

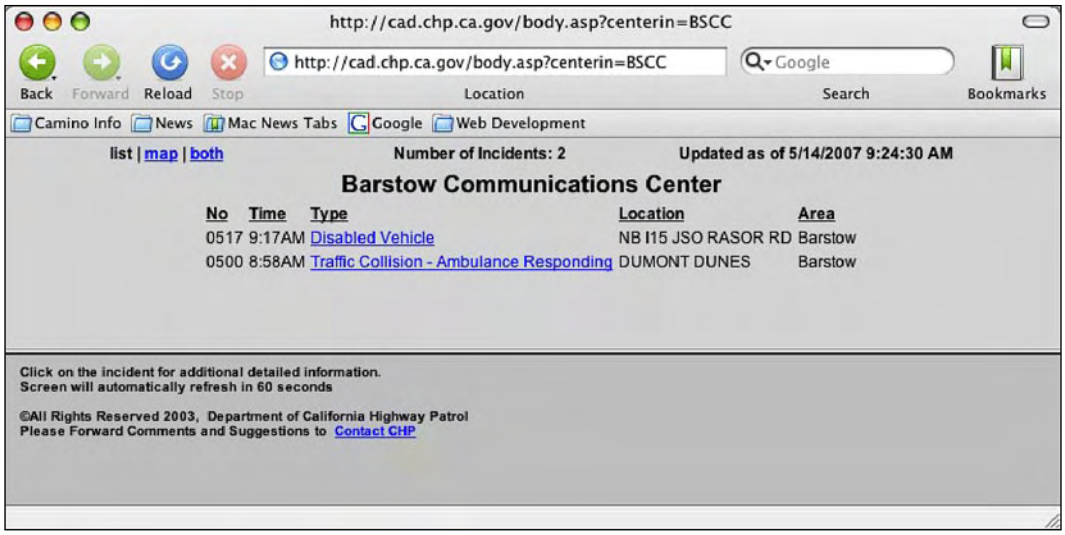
Let's zero in on the section that creates the **Areas** pull-down menu.

```
<form name="areas" action="body.asp" target="bodyx">
   <TR>
      <TD ALIGN="left" >
      <FONT COLOR="#CFB53F" SIZE="-1">Areas</font>
      <SELECT name = "centerin" onchange="document.areas.submit()">
         <Option Value="BFCC">Bakersfield</Option>
         <Option Value="BSCC">Barstow</Option>
         <Option Value="GGCC">Bay Area</Option>
         …
         <Option Value="VTCC">Ventura</Option>
         <Option Value="YKCC">Yreka</Option>
         <Option Value="ca">State Map</Option>
      </Select>
      <input type="submit" value="Go!">
      </TD>
   <TR>
</form>
```

There are several things to take away from this:

- The menu is encased in one form named `areas`. The action goes to `body.asp`, which is in the same directory of this page. The HTTP method is not specified, so it will default to a `GET` request.
- The form's only items are a `SELECT` tag and a submit button.
- The select element is named `centerin`. On an `onchange` event, the form is submitted. So, we can assume the submit button is only used for browsers without Javascript turned on.
- The menu options have a four-character value. The first two characters appear to be tied to the geographical area. The last two characters are `CC`.

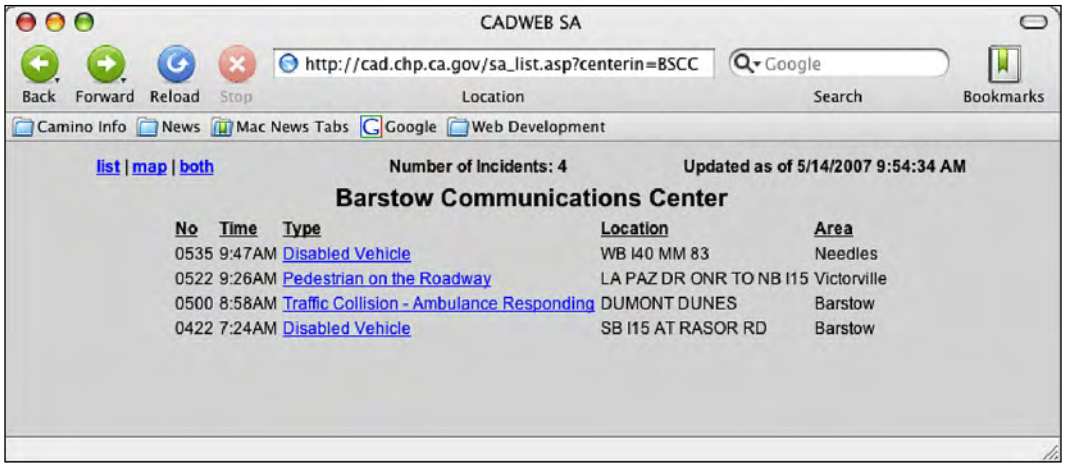
Based on this, we can try to try to make a request to `body.asp` with a `GET` parameter of `centerin` and a value from one of the menu items. Let's try a URL of `http://cad.chp.ca.gov/body.asp?centerin=BSCC`. This uses the value for Barstow, California.



This page looks similar to the home page except there is no navigation menu. We're interested to see how the top frame, now with the content, is constructed. We can check out the source for the whole frameset.

```
<HTML>
 <HEAD>
 </HEAD>
  <FRAMESET ROWS="55%,*" >
  <FRAME NAME="sa" SRC="sa_list.asp?centerin=BSCC&style=l"
  SCROLLING="auto" MARGINWIDTH="2" MARGINHEIGHT="2" BORDER="1">
  <FRAMESET ROWS="*" BORDER=YES FRAMESPACING=20>
    <FRAME NAME="ii" SRC="./footer_default.asp" SCROLLING="auto"
    MARGINWIDTH="2" MARGINHEIGHT="2">
  </FRAMESET>
 </FRAMESET>
 <BODY>
 </BODY>
</HTML>
```

Here, we see the source of the top frame is a page named `sa_list.asp`. It is passed URL parameters of `centerin`, the original name of the pull-down menu, with a value of `BSCC`, which we used to test. Another value of `style=1` is passed. Let's try to hit that page with those query parameters.



Sure enough, this is the page we are looking for. If you change the value of `centerin` in the address bar to some other option from the pull-down menu, you will get a page for another area. Let's take a look at the HTML of `sa_list.asp`:

```
<html>
 <head>
  <meta http-equiv="Refresh" content="30">
  <meta name content="text/html charset=iso-8859-1">
  <title>CADWEB SA</title>
   <link rel="STYLESHEET" type="text/css" href="Inc/cadweb.css">
 </head>
 <BODY class="sa" ONLOAD="if((navigator.appName=='Netscape') &&
 (navigator.appVersion.charAt(0)=='3') && (navigator.appVersion.
 indexOf('Win')+navigator.appVersion.indexOf('Mac')!=-2))document.
 bgColor=document.bgColor;">
 <table border="0" align="center">
   <tr>
      <td class="Head" width="33%" align="Left">
         <a href="set_nav.asp?centerin=BSCC&style=l"
         target="bodyx">list</a> |
         <a href="set_nav.asp?centerin=BSCC&style=m"
         target="bodyx">map</a> |
         <a href="set_nav.asp?centerin=BSCC&style=b"
         target="bodyx">both</a>
```

```
         </td>
         <td class="Head"  width="33%" align="Left">
         Number of Incidents: 4</td>
         <td class="Head"  width="33%" align="right">
         Updated as of 5/27/2007 10:05:00 AM
         </td>
      </tr>
   </table>
   <table border="0" align="center">
      <tr>
         </td>
         <td class="HeadT" colspan="5" align="center">Barstow
         Communications Center</td>
      </tr>
      <tr>
         <td class="HeadUl">No</td>
         <td class="HeadUl">Time</td>
         <td class="HeadUl">Type</td>
         <td class="HeadUl">Location</td>
         <td class="HeadUl">Area</td>
      </tr>
      <tr>
         <td class="T">0535</td>
         <td class="T" nowrap> 9:47AM</td>
         <td class="T">
           <A HREF=./iiqr.asp?Center=INCC&LogNumber=0535D0527&t=Disabled
           %20Vehicle&l=WB%20I40%20MM%2083&b= TARGET="ii">Disabled Vehicle
         </td>
         <td class="T">WB I40 MM 83</td>
         <td class="T">Needles</td>
      </tr>
      <tr>
         <td class="T">0522</td>
         <td class="T" nowrap> 9:26AM</td>
         <td class="T">
            <A HREF=./iiqr.asp?Center=INCC&LogNumber=0522D0527&t=
            Pedestrian%20on%20the%20Roadway&l=LA%20PAZ%20DR%20ONR%20TO%20
            NB%20I15&b=4296%207C TARGET="ii">Pedestrian on the Roadway
         </td>
         <td class="T">LA PAZ DR ONR TO NB I15</td>
         <td class="T">Victorville</td>
       </tr>
      <tr>
         <td class="T">0500</td>
         <td class="T" nowrap> 8:58AM</td>
         <td class="T">
            <A HREF=./iiqr.asp?Center=INCC&LogNumber=0500D0527&t=
            Traffic%20Collision%20-%20Ambulance%20Responding&l=
            DUMONT%20DUNES&b= TARGET="ii">Traffic Collision -
            Ambulance Responding
```

```
        </td>
        <td class="T">DUMONT DUNES</td>
        <td class="T">Barstow</td>
      </tr>
     <tr>
        <td class="T">0422</td>
        <td class="T" nowrap> 7:24AM</td>
        <td class="T">
           <A HREF=./iiqr.asp?Center=INCC&LogNumber=0422D0527&t=
           Disabled%20Vehicle&l=SB%20I15%20AT%20RASOR%20RD&b=
           TARGET="ii">Disabled Vehicle
        </td>
        <td class="T">SB I15 AT RASOR RD</td>
        <td class="T">Barstow</td>
     </tr>
  </table>
  </body>
  </html>
```

This page is divided into two tables. The top table is the header of the page. We can ignore it for our purposes. The second table is the content. It is a table of active incidents. This is what we are going to have to extract.

The first row in the table is the header row. It tells us what each column holds. The columns are: an identifier number, time of the incident, the type of incident, exact location, and the general area. Each subsequent row holds an incident. Each cell in an incident row has a class attribute whose value is `T`. This appears to be the best identifier for our data. We now have a good idea of how this page is generated and how it is structured, so we can create our feed.

Your Site is an API

This investigation of the California Highway Patrol website hints at a notion that many developers are adapting these days—the website you create is an API. Not only should your site be friendly for others to mash up, but also for desktop applications like screen readers and user style sheets. By thinking of your site as a content API, you increase adaptation of your site. To facilitate this, make sure your HTML code is well-formed and structurally there is good separation between content and presentation.

Mashing Up

We now have some key bits of information about this page. Let's recap what we have discovered and learned:

1. 411Sync.com parses an RSS file on your server and passes it to the user. The service can take a variable from the user and pass to your page.
2. The page of incidents is at `http://cad.chp.ca.gov/body.asp?centerin=XXCC`, where "`XXCC`" is a four character code that defines the area to pull.
3. The incidents are in the second table of that page.
4. The columns of the incidents are always in the order ID Number, type, location, and general area.
5. Each incident cell has a class attribute whose value is `T`.

Based on these factors, the format needs of 411Sync.com, and user-friendliness for mobile users, we are going to break down our application into the following components:

- `clsIncident.php`—An incident object, this is our "Model" component. All this class will do is hold information about an incident—time, location, etc.
- `clsDomParser.php`—We are going to take a page from our event-based SAX Parser from Chapter 1. While parsing with PHP DOM is not really event-based parsing, there are basic things that all parsers will have to do. For example, the parser will need to load the XML data. In here, we can also dictate whether the parser should load an XML data versus HTML data, and whether the data itself is a physical file or a string of data. As these tasks are universal for all PHP DOM parsers, we will create an abstract class, and all parsing logic specific to the implementation will be in another class that extends this one.
- `clsCHPDomParser.php`—This class will act as a "Controller" object. Its job is to interact with the outward-facing, "View" page and use the other objects to create the content. The other classes will have no or very little knowledge of the CHP site and structure. This class, though, will directly work with the `sa_list.asp` page.
- `feed.php`—This is the only page that 411Sync.com will directly request. In other words, this is our "View" page. This page's job is to take the parameter from 411Sync.com, pass it to the other helper classes, wait for the results from these helper classes, and return the result as an RSS file to the user.

The Incident Class

This class is located inside the `classes` directory of the example code. This is the only model class we need.

```
<?php
   class Incident {
   private $incidentType;
   private $address;
   private $area;
   public function getIncidentType() { return $this->incidentType; }
   public function getAddress() { return $this->address; }
   public function getArea() { return $this->area; }
   public function setIncidentType($t) {$this->incidentType = $t; }
   public function setAddress($a) { $this->address = $a; }
   public function setArea($a) { $this->area = $a; }
   }
?>
```

Besides the parser class we will create later, this is the only class that truly knows anything about the CHP site. Even then, its knowledge is limited to the structure of incident data. All this class will do is hold information about an incident, and provide getter and setter functions for the information. We're going to keep the incident type, address, and general area. In the interest of saving space, our first application will not retrieve the time of the incident.

The DOM Parser Class

The event-based SAX Parsers from Chapter 1 defined parsing functions for child objects to implement, set up the parser, and had a generic function that executed the parsing of the documents. Our new parent DOM parser class, named DomParser with a file name of `clsDomParser.php` in the examples code, will function in a similar fashion. It will hold generic functions that all parsers must execute.

In the PHP DOM case, the generic tasks for this class will be to create the DOM and decide whether to load the data as HTML or XML. The decision on what to use is not made here. That will be tied to the implementation class. All this class will do is call the appropriate PHP functions, depending on what the executing class wants.

```
<?php
  abstract class DomParser {
     protected $dom;
```

We start by defining the class as an abstract class. Next, we define a class variable to hold the DOM. There is no need for other classes to work with the DOMDocument directly, so we make this a protected variable ;

```
public function parse($url, $mode) {
   $returnMe = true;
   switch($mode) {
      case "HTML":
         try {
             @$this->dom->loadHTMLFile($url);
             }
         catch (Exception $ex) {
                    $this->dom = null;
                    $returnMe = false;
                              }
          break;
      case "XML":
      try {
           $this->dom->loadXML($url);
          }
      catch (Exception $ex)
           {
            $this->dom = null;
            $returnMe = false;
           }
         break;
       default:
            $this->dom = null;
            $returnMe = false;
   }
   return $returnMe;
}
```

This parse function simply calls the appropriate function to load the data into the DOMDocument. It takes two parameters—the URL where the XML is stored and a string value of `XML` if we want to load XML, and a string value of `HTML` if we want to load it as HTML. A `switch` statement decides which appropriate load function to call. If the load was successful, the function returns `true`. If it failed, the `try-catch` blocks will set the return value to `false`.

Note that for the HTML load, we use the `@` symbol before the load function call on DomDocument. The `@` symbol before a function call will suppress any warnings that may be outputted to the browser. More often than not, when you develop in PHP, you want to see all errors and warnings that happen. This is one case where you do not want to see this. If the HTML data is not valid according to its DOCTYPE element, `DomDocument` will report it and output it to the screen. Any validation error will trigger this output and it is probably correct to say that the majority of web pages out on the internet are not valid. In fact, earlier, when we examined code from the CHP site, there was a missing closing `TR` element in the navigation header and on `sa_list.asp`, there are missing closing `A` tags. If you do not use the `@` symbol, your results will probably be riddled with PHP warnings to your user. In our case, those errors will cause a fatal error in our output because our feed would no longer be valid RSS.

Luckily, the side affect for using @ in this case is minimal. `DomDocument's` ability to load malformed HTML is very forgiving and shouldn't affect execution of code.

```
    public function __construct() {
        $this->dom = new DomDocument();
    }
}
?>
```

Lastly, the constructor is defined. When the object is instantiated, it will automatically create the `DomDocument` and assign it to the class property.

This class is now ready to be used by anything that wants to parse using PHP DOM.

The CHP DOM Parser Class

The CHPDomParser class is named `clsCHPDomParser.php` in the examples code. This class holds the code to parse the actual CHP web page. It must also interact with other objects and the exposed web page. It has the tightest coupling with the PHP site and holds the actual business logic in our application.

```
<?php
  class CHPDomParser extends DomParser {
     public function getRSSItems($rc) {
        $rc = strtoupper(trim($rc));
        $items = "";
```

The first function is the only public function in this class besides the constructor and will be called by the view page after object instantiation. This function calls other functions in this class to create the content. The content is held in the variable named `$items`, which is returned to the caller. It takes one parameter, `$rc`, which is the region code.

```
if ($rc == 'HELP' || $rc == '') {
    $items = $this->getHelpItems();
```

To make our application user friendly, we should define some help text. This help text is defined in the function named `getHelpItems()`, which we call here. The help text will be returned if the user sends a code of "HELP," or if the user does not send any parameters. The help text return is triggered by the first `if` block.

```
} else {
        $chpURL = $this->getCHPURL($rc);
```

If the request from the user is not for help, then we will prepare to make a request to the CHP site. Our first task will be to create the URL to `sa_list.asp`. The URL creation is handled by the function named `getCHPURL()`. Part of the URL is the region code, so we pass the region code submitted to this function.

```
if ($this->parse($chpURL, 'HTML')) {
      $incidentObjects = $this->getIncidentObjects();
      $items = $this->incidentObjectsToItems($incidentObjects);
                                    }
else {
      $items = "An error has occurred  Please try again.";
     }
```

Next, we try parsing `sa.list.asp`. The bulk of the work is done in the function `getIncidentObjects()`. This will parse the incident rows and return an array of Incident objects. We then pass this array to `incidentObjectsToItems()`, which takes the array and formats it into a text string to be returned to the user. This text string is populated into `$items`. If the load fails, `$items` is populated with a generic error message.

```
}
 return $items;
}
```

`$items` is then returned to the caller.

```
private function getIncidentObjects()
{
         $incidentsArr = array();
         $counter = 0;
         $rows = $this->dom->getElementsByTagName('tr');
         foreach ($rows as $row)
           {
         $tds = $row->getElementsByTagName('td');
```

`getIncidentObjects()` is the main parsing function. It starts off with some initialization declarations. `$inicidentsArr` holds the array of incidents and will be returned to the caller. Another variable called `$counter` will be used to keep track of which column in the table we are in.

The basic premise is that the `td` elements we are interested in are within `tr` elements. Therefore, we grab the `tr` elements in the whole document using `getElementsByTagName()` on the `DOMDocument`. We then loop through that NodeList and execute `getElementsByTagName()` on the `tr` node searching for the `td` elements.

```
foreach ($tds as $td)
{
   if ($td->getAttribute('class') == 'T')
   {
     //we are in the data section
```

Now, we must loop through the columns on each row. You may be wondering why we didn't just use and get all `td` elements directly in the beginning. The reason why is that we must keep track of our position in this table in order to identify which `td` element holds an incident type, what holds an address, and which holds an area. We know that the third `td` after a row is an incident type. We know that the fourth `td` after a row is an address, and so forth. If we grabbed all `td` elements at the beginning, the numbering would be off due to other table on the whole page or any other `td` elements used to control layout.

We are only interested in the `td` elements of the incident table. Therefore, we check to see if there is a `class` attribute and whether it has a value of `T`. If that is true, let's go ahead and count the columns using the `$counter` variable and a `switch` statement.

```
switch($counter)
{
   case 0:
       //1st - start of a new incident
       $incident = new Incident();
       break;
```

If the counter is zero, it means we are in the first column, starting a new row and that requires a new `incident` object. Here, we create one:

```
case 1:
    //2nd - the time.  Skip.
    break;
```

If the counter is one, we are in the time column. We can skip this because we made a decision not to extract it to save space.

```
case 2:
    //3rd - The Incident Type
    $incident->setIncidentType($td->nodeValue);
    break;
```

A counter value of two means we are in the third column, which is the incidents type column. We grab the node value and store it into the `incident` object that was created.

```
case 3:
   //4th - Address
    $incident->setAddress($td->nodeValue);
    break;
case 4:
    //5th -> Area
    $incident->setArea($td->nodeValue);
    array_push($incidentsArr, $incident);
    $incident = null;
    break;
```

We repeat the same procedure for the fourth and fifth columns. The fifth column is the end of the row, so we need to also do some clean up. We take the `incident` object and place it into the `$incidentsArr`, which is going to be returned to the caller. We then destroy the `incident` object, ready to be created the next time a new row is encountered.

```
}
if ($counter == 4)
 {
   $counter = 0;
 }
else
 {
  $counter++;
 }
```

This block increments or resets the counter when it encounters each "T" table cell.

```
            } //If
          } //Inner foreach
        }  // Outer foreach
        return $incidentsArr;
    } //Function
```

Finally, the incidents array is returned and the function is closed.

```
    private function incidentObjectsToItems($incidents) {
        $itemsString = "";
        foreach ($incidents as $incident) {
            $itemsString .= $this->abbreviateIncidentType($inciden
                   t->getIncidentType()) . ": " . $incident->getAddress()
                   . ", " . $incident->getArea() . "--";
         }
         return $itemsString;
}
```

This function simply loops through an incidents array and formats it into a concise and compact string. The only thing to look out for is that we pass the incident type to `abbreviateIncidentType()`. This function abbreviates the incidents to give us more space. More on how this function works when we get to it.

```
    private function getHelpItems() {
     $help = 'Fresno-FR, Los Angeles-LA, Monterey-MT, Sacramento-ST, ';
     $help .= 'Redding-RD, San Diego-BO, San Francisco Bay Area-GG';
     return $help;
    }
```

This is the function that generates the help text. This is simply a list of acceptable region codes. Even though the pull-down menu has many more regions, we're going to limit the ones shown to the user just to keep the help text under the maximum payload size.

```
    private function abbreviateIncidentType($type) {
        $returnType = $type;
        switch($type) {
            case strstr($type, 'Traffic Hazard'):
                $returnType = 'TH';
                break;
            case strstr($type, 'Collision'):
```

```
            $returnType = 'COL';
            break;
         case strstr($type, 'Hit and Run'):
            $returnType = 'H/R';
            break;
         case strstr($type, 'Hit & Run'):
            $returnType = 'H/R';
            break;
         case strstr($type, 'Disabled Vehicle'):
            $returnType = 'DV';
            break;
         default:
            $returnType = str_replace('a', '', $type);
            $returnType = str_replace('e', '', $returnType);
            $returnType = str_replace('i', '', $returnType);
            $returnType = str_replace('o', '', $returnType);
            $returnType = str_replace('u', '', $returnType);
      }
       return $returnType;
   }
```

This function abbreviates the incident types. Some of the incident type labels are quite long. We do this simply to shorten the data returned. In case something slips by, the default `case` in this `switch` statement removes the vowels from the string.

```
   private function getCHPURL($rc) {
      $prefix = 'http://cad.chp.ca.gov/sa_list.asp?style=l';
      $returnMe = $prefix . '&centerin=' . $rc . 'CC';
      return $returnMe;
   }
}
?>
```

At the end of the class is the `getCHPURL()` function. This function simply injects the region code submitted by the user into the URL format that `sa_list.asp` requires.

Now we have our helper classes defined, we can tie them all together using the exposed view page.

Creating the Feed Page

The feed page's job is just to include all necessary class files, call `getRSSItems()` from `CHPDomParser`, and format the returned values into a valid RSS file.

```
<?php
  $regionCode = $_GET['rid'];
  header('Content-type: text/xml; charset=UTF-8');
  require_once('classes/models/clsIncident.php');
  require_once('classes/clsDomParser.php');
  require_once('classes/clsCHPDomParser.php');
```

The first few lines prepare our script for the execution of code. We grab the `GET` variable named `rid` and set it to a local variable named `$regionCode`. We then set the content type of this file to XML using a header change. Finally, we include all of the classes we just created.

```
  $parserObj = new CHPDomParser();
  $text = $parserObj->getRSSItems($regionCode);
?>
```

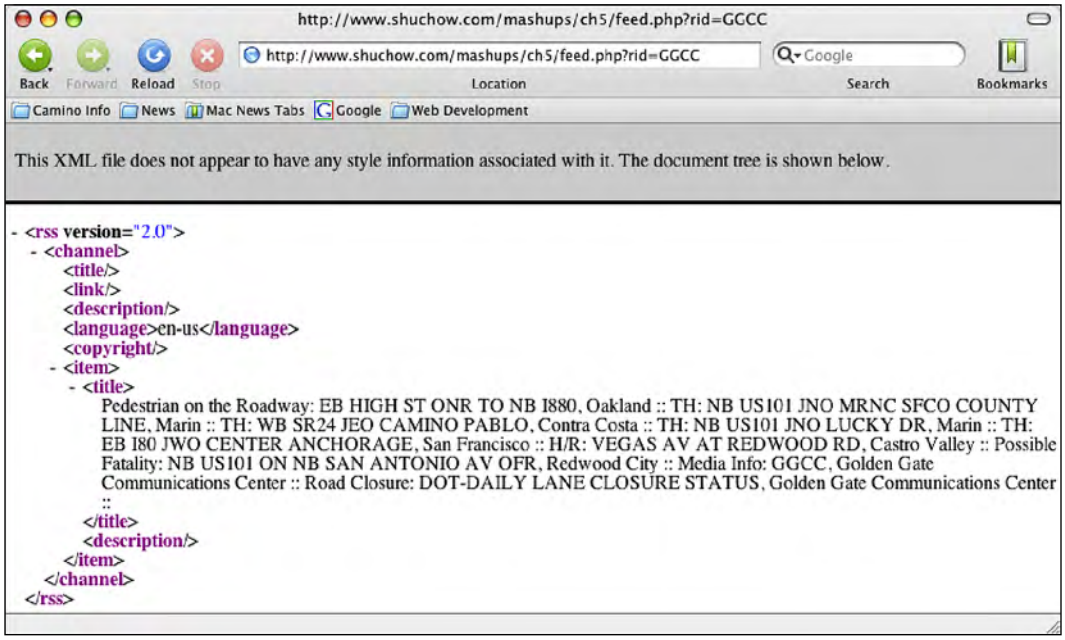
Next, all we have to do to initialize the load, and parsing is to create the `CHPDomParser` object and then call `getRSSItems()`, passing the `$regionCode`. The items are returned and set in the variable named `$text`. That concludes the major PHP code in this file.

```
<?= '<?xml version="1.0" encoding="utf-8" ?>' ?>
<rss version="2.0">
  <channel>
    <title></title>
    <link></link>
    <description></description>
    <language>en-us</language>
    <copyright></copyright>
    <item>
    <title><?= $text ?></title>
    <description></description>
    </item>
  </channel>
</rss>
```

The RSS file is quite simple. Like the 411Sync documentation says, it passes the values in the `title` element back to the user. All we have to do, then, is output `$text` into the `title` element.

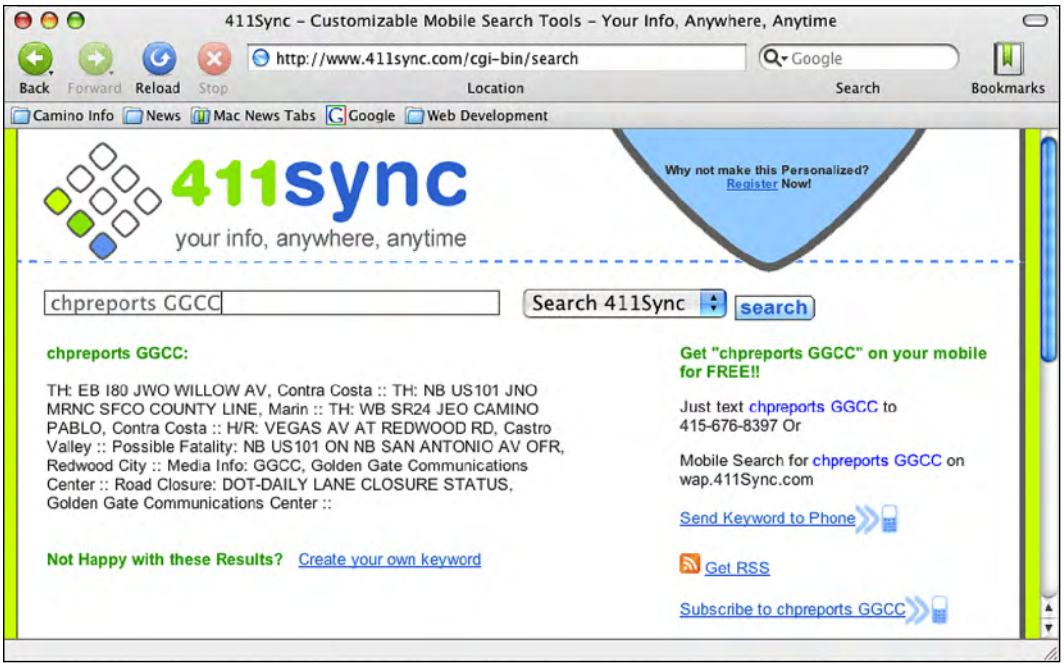
Testing and Deploying

Our application is now complete. We can now test and deploy our application. After installing it into a publicly accessible web server, you can simulate the URL that 411Sync.com will use to hit your script. Don't forget to add the region code variable after `rid=` or "`help`".



If this test is successful, we can be confident that our code executes properly and is successfully retrieving the data from the CHP site.

Next, we can see if 411Sync.com can successfully get to your page. 411Sync.com makes all of its searches available through a web interface. It is located at `http://www.411sync.com/cgi-bin/search`. In the search box, enter in your keyword and the region code then press **Search**.



If this test is successful, try to send the same SMS message from your phone to 415-676-8397. You should soon receive an SMS message back with traffic information.

Summary

In this chapter, we used the new PHP DOM extension found in PHP 5 to screen scrape information. This is a far more sophisticated and powerful way to extract information from websites than old methods that often would hard code a lot of structural information. After getting permission from the California Highway Patrol, we deconstructed the Live Incidents site to see how it works. With this knowledge, we used the PHP DOM extension to extract traffic incident data from the Live Incidents website, reformatted it, and prepared it for delivery through SMS. 411Sync.com makes creating mobile content very easy. All we had to do was create an RSS feed. Through keywords, 411Sync.com directs requests to our RSS feed and sends it back to the mobile device.

6
London Tube Photos

Project Overview

What	Plot London Tube station locations on Google Maps. When a station's icon is clicked, search Flickr for photos of the station and display them on the map.
Protocols Used	REST
Data Formats	XML, RDF, JSON
Tools Featured	SPARQL, RDF API for PHP, `XMLHttpRequest` Object (AJAX)
APIs Used	Google Maps, Flickr Services

We have used a lot of techniques and APIs in our projects. For the most part, things have mashed up together fairly easily with minimal issues. One of the reasons for this is that we have relied on PHP to create the presentation for our mashups. This simplifies the architecture of our mashup and gives us a lot of control. Many APIs, though, are JavaScript-based, and hence, any mashup will rely heavily on JavaScript for the presentation. This introduces a lot of other issues that we will have to deal with. In this mashup, we will encounter some of those issues, and look at ways to work around them. PHP will remain an important part of our mashup, but take a smaller role than it has played so far.

In this mashup, we will present a geographically-centric way to present pictures from the photo-sharing site, Flickr. When a user loads our application, they will be presented with a Google map of London. A pull-down menu of all the London Tube lines will be available. The user will select a line, and the application will load all of the Tube stations onto the map and display them with markers. If the user clicks on a marker, the name of the station will appear as a popup on the map. In the background, a search query against Flickr will be initiated, and any pictures of the station will appear in the popup as a thumbnail. Clicking on the photo will take the user to the photo's page on Flickr.

JavaScript is not the only new tool that we will integrate into our toolbox. Before we can work on the user interface, we will need to populate data into our application. We need to find out which Tube stations belong to which line, and where those stations are located. Many websites have one of those things or the other, but not both. If we used them, not only are we dealing with two data sources, but we'd have to resort to screen scraping again. Fortunately, there is one place that has both pieces of information. This source is in Resource Description Format, an XML format that we glanced at, earlier in Chapter 3. In this mashup, we will take a much closer look at RDF, and how to extract data from it using a young query language called SPARQL (SPARQL Protocol and RDF Query Language).

Preliminary Planning

Note that it would not have been wise to pre-plan mashups, but this application will be much more complex, and will definitely require some forethought. Previously, our APIs have worked in the background delivering data. We use PHP to retrieve data from an API, receive it in whatever format it gives us, format the response into either HTML output to the user, or another format to retrieve data from another API. PHP gives us a lot of flexibility in the way our application is designed.

This time, one API, Google Maps, is a JavaScript API. Another, Flickr Services, is still server based. The two cannot talk directly to each other, and we are going to have to play within the rules set by each one. More than ever, we are going to have to take a close look at everything before we attempt to write a single line of code.

At this point, this is what we know:

1. We need to find a data source for the Tube stations. We need to find the names of the stations in each line, and some piece of information we can use to geographically identify it on a map. The latter will be dictated more by the capability of the tool on the receiving end. In other words, as we are going to use Google Maps, we are going to have to see how Google Maps places markers on its map, and we will have to massage the source data to Google Map's liking.
2. We will use the Google Maps API solely for presentation. JavaScript cannot call PHP functions or server side code directly, nor can PHP call JavaScript functions. However, we can use PHP to write JavaScript code on the fly, and we do have the JavaScript `XMLHttpRequest` object available. The `XMLHttpRequest` object can call server resources by sending a `GET` or `POST` request without the page reloading. We can then dynamically update the page in front of the user. This process is popularly known as AJAX, or Asynchronous JavaScript and XML.

Looking at the Flickr Service's documentation page at `http://www.flickr.com/services/api/`, we find we have an incredible variety of formats and protocols to choose from. All of our major request protocols, REST, XML-RPC, and SOAP are there. In addition to these, we can have our choice of JSON or serialized PHP for the response format. There is also a huge list of language kits already built. You can use these kits to call Flickr directly from PHP, ColdFusion, Java, etc. Unfortunately, JavaScript is not on that list.

Finding Tube Information

Our biggest problem is finding the initial Tube data. Without this first step, we cannot create our mashup. The first logical step is to look at the official Tube site at `http://www.tfl.gov.uk/tube/`. Poking around, we see a lot of colorful maps of the lines, but nothing machine readable—no feeds and not even a pull-down menu with stations. It looks like the official site will be a poor choice as a source of data.

We should look at the Google Maps API to see what it can even accept. The documentation homepage is at `http://www.google.com/apis/maps/documentation/`. This site has many examples as well as class, methods, and properties references. Looking around, we see that a Google Map marker is represented by a class called `GMarker`. There are many examples on how to create a marker like so:

```
marker = new GMarker(point);
map.addOverlay(marker);
```

That's wonderful, but what is a point that is passed to the `GMarker` class? Looking at the documentation reference, we find that it is a `GLatLng` object, which is an object that has two simple properties—the longitude of the marker and the latitude of the marker. It looks like the most direct way to create a marker is through latitude and longitude coordinates.

Ruling out the official Tube site, we still need to find longitude and latitude information for sites. With some searching, I stumbled upon Jo Walsh's site, `frot.org`. Ms. Walsh has done a lot of work with open geographical data, and is currently an officer in the Open Source Geospatial Foundation (`http://www.osgeo.org/`). On her site, she talks about mudlondon, an IRC bot she created. As part of this bot, she compiled an RDF file of all London Tube stations. The file is located at `http://space.frot.org/rdf/tube_model2.rdf`. The first half of this file is information about each station, including latitude and longitude positions. The second half of this file maps out each line and their station. These two pieces of information are exactly what we need. After contacting her, she was gracious enough to allow us to use this file for our mashup.

Being an XML-based file, we can create our own parser like we did before. However, some more searching reveals an RDF parser for PHP. This should save us some effort.

There is one problem with this approach. The RDF file itself is over 500 kilobytes in size. It would be perfectly reasonable to treat this RDF file like an RSS 1.1 feed and load and parse it at run time. However, this file is not a blog's stream. Tube stations do not change very often. To save bandwidth for Ms. Walsh, and dramatically speed up our application, we should eliminate this load and parse. One solution is to save this file directly onto our file system. This will give us a great speed boost. Another speed boost can be gained if we retrieved the data from a database instead of parsing the file every time. XML parsers are a fairly new addition into the PHP feature set. They are not as mature as the database extensions. The nature of XML parsing also has an overhead to it compared to just retrieving data from a database. It would appear that we should use RDF parsing to populate a database at first, and then in our application, load the data dynamically from a database.

Integrating Google Maps and Flickr Services

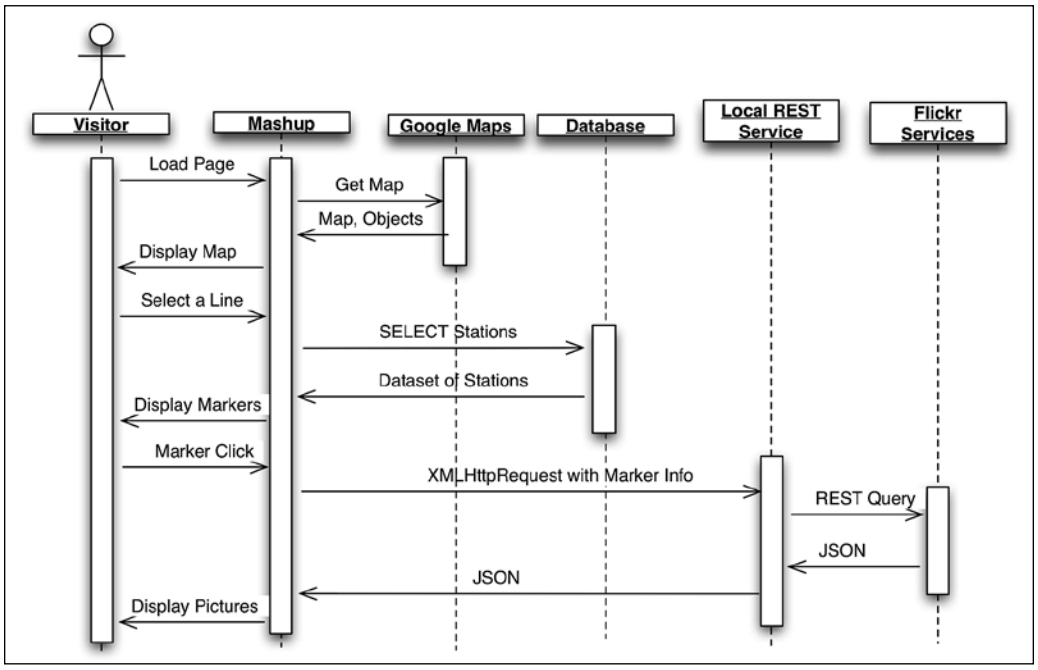
Now that we have the data and know generally how to create markers with that data, we need to look at how to bridge a JavaScript call in Google Maps to a server call in Flickr Services. Flickr Services has a REST-based endpoint available. This means that all we would need to do is send a `GET` or `POST` request to the endpoint, supplying our parameters, and we would get data back. Moreover, one return option is JavaScript Object Notation, JSON. Theoretically, we can use the `XMLHttpRequest` object in JavaScript to send a `GET` request, and get JavaScript directly back from the server. We can then use this JavaScript to dynamically change our page. This would really make things easy.

The main obstacle to this is that we cannot make the `XMLHttpRequest GET/POST` request directly against Flickr Services. This is because cross-scripting attacks are a security problem. To counter this, all web browsers prevent a site from sending an `XMLHttpRequest` against another site. An `XMLHttpRequest` can only go back to the server from where the page was served.

To get around this, we can set up our own REST service that sits on our server. When the user clicks on a marker, the `XMLHttpRequest` goes back against our REST service. Our REST service then calls Flickr Service, and we merely pass the Flickr response back to the client.

Application Sequence

We now have a plan of attack and a preliminary architecture for our application. We can create a Unified Modeling Language sequencing diagram to illustrate what will happen when a visitor uses our mashup.



If you do not know UML, do not worry. This diagram keeps the UML notation simplified and is easy to understand. This is basically a fancy way of summarizing the steps that a user goes through to load a set of pictures from Flickr. While there are just three things a user must do, this diagram shows sequentially what happens behind the scenes.

This diagram gives us a good idea of what we are dealing with in terms of technology. Let's take a look at some of the new formats we will encounter.

Resource Description Framework (RDF)

Recall from Chapter 3, we described RSS 1.1 as being RDF-based. What exactly is RDF? Many call RDF "metadata about data" and then go on to describe how it has evolved beyond that. While RDF and its usage has certainly evolved, it is important to not to forget the "metadata about data" aspect because it captures the essence of what RDF is.

The purpose of RDF is to describe a web resource, that is, to describe something on the Internet. For example, if a shopping website lists the price of something, what exactly is a price? Is it in American Dollars? Mexican Pesos? Russian Rubles? For a website, what exactly is a timestamp? Should a machine parser treat a timestamp in 12-hour notation different from a timestamp in 24-hour notation? XML, at a very high level, was supposed to allow groups to standardize on a transaction format. Implementation details were left to the parties of interests because XML is just a language. RDF is the next evolution of that original goal. It gives us a framework for that implementation. By defining what a timestamp is, any machine or human that encounters that RDF document will know, without any ambiguity, what that timestamp is, what it means, and what format it should be in.

The basic concepts and syntax of RDF is fairly simple and straightforward. RDF groups things in what it calls **triples**. A triple basically says, "A something has a property whose value is something". Triples use the grammar concepts of subject, predicate, and object. In the sentence, "The page has a size of 21 kilobytes", the page is the subject. The predicate is the property, in this case, size. The object is the value of that property, 21 kilobytes. Typically in RDF, the subject is represented by an `about` attribute of a parent element. The property and value are represented by element and value pairs under that parent element. The page size sentence could be represented as follows in XML notation:

```
<rdf:RDF
   Xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   Xmlns="http://www.example.org/pageProperties"
>
<rdfDescription rdf:about="http://www.shuchow.com/thecats.html">
   <creator>Shu Chow</creator>
   <title>My Cats</title>
   <lastMod>01/24/22007</lastMod>
   <size>21 kilobytes</size>
</rdf:Description>
</rdf:RDF>
```

In RDF, every element must be namespaced. The `rdf` namespace is required, and must point to `http://www.w3.org/1999/02/22-rdf-syntax-ns`. This gives us access to the core RDF elements that structure this document as an RDF document. In this short document, we access the RDF elements three times—once as the root element of the document, once more to identify a resource using the `Description` element, and once more to identify the specific resource with the `about` attribute. In human language form, the title can be stated as, "A web resource at `http://www.shuchow.com/thecats.html`, has a title property, whose value is 'My Cats'". Even more casually, we can say, "The page's title is 'My Cats'".

Breaking it down into subject, predicate, and object:

- The subject is `http://www.shuchow.com/thecats.html`.
- The predicate is `title`. This may also be expressed as a URI.
- The object is "My Cats".

In RDF, subject and predicates must be URIs. However, like in the preceding example, predicates can be namespaced. Values can be either URIs, or, more commonly, literals. Literals are string values within the predicate elements.

There is another RDF element that we will encounter in our mashup. In the previous example, it was obvious from context that the web resource was an HTML page. The RDF Schema specification has a `type` element `resource` attribute that classifies subjects as programming objects (as opposed to triples objects), like PHP or Java objects.

```
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns="http://www.example.org/pageProperties"
>
<rdfDescription rdf:about="http://www.shuchow.com/thecats.html">
   <creator>Shu Chow</creator>
   <title>My Cats</title>
   <lastMod>01/24/22007</lastMod>
   <size>21 kilobytes</size>
   <rdf:type rdf:resource=
                       "http://www.example.org/objects#An_HTML_Page" />
</rdf:Description>
</rdf:RDF>
```

The `resource` attribute is always a URI. Combined with the `type` element, they tell us that in order to find out what exactly this resource is, we should visit the value of the `resource` attribute. In this example, the resource is described at `http://www.example.org/objects#An_HTML_Page`, which presumably describes an HTML page.

Knowing just the simple nature of triples can get us started with RDF. Within the core RDF specification, there are a few more elements that pertain to grouping of collections. However, as the specification is designed to be scaled and expanded, there are not many more elements beyond that. Namespacing of extensions is the source of RDF's power. For our mashup, we will encounter a few more extensions, and we will examine them closer when we encounter them. For now, we have the basic skills to read and use our latitude/longitude data source.

Common extensions to RDF and their applications can turn RDF into a very deep subject. To learn more about RDF, the W3C has created an excellent primer located at `http://www.w3.org/TR/rdf-primer/`. Be warned that one can get easily wrapped up in the philosophical underpinnings of RDF—the official specification is actually six separate documents.

SPARQL

RDF is designed to be a data store. It follows that as soon as RDF came out, people wanted a way to query, like a traditional database. SPARQL is a new RDF query language that has recently become a W3C recommendation. You can think of SPARQL as writing a query, loosely akin to SQL for databases, to parse an XML file, specifically an RDF file. The results returned to you are row and column tables just like in SQL.

Most people learned SQL with the aid of a command line client that queried a database. This allowed us to experiment and play with query structures. Fortunately for SPARQL, there is something similar; SPARQLer, located at `http://www.sparql.org/sparql.html`, is an interactive web tool that allows you to specify an RDF document on the web as an input and write SPARQL queries against it. It will display the query results to us much like the results from a database client. As we go through our initial discussion of SPARQL, we will use this query tool and an example document RDF document at `http://www.shuchow.com/mashups/ch6/pets.rdf`. This RDF document is a list of all the animals that my pay check feeds.

Analyzing the Query Subject

In the database world, before you start writing queries, you need to understand the schema a little, either by entity-relationship diagrams (if you had good documentation) or by simply using `SHOW TABLES` and `EXAMINE` SQL commands. You'll need to do the same thing with SPARQL. Sometimes the host will have documentation, but often, you will just need to read the RDF file to get a general feel for the document. Let's start this exercise by opening the RDF file we will be working with at `http://www.shuchow.com/mashups/ch6/pets.rdf`. Your browser will either download this file to your hard drive, or it will open it in-window. If it opens up in-window, it will probably apply a stylesheet to it to pretty up the presentation. In this case, you will need to view the source of the document to see all the tags and namespace prefixes.

This RDF file is very straightforward and simple. We start off with the root element, followed by the namespaces:

```
<rdf:RDF
    xmlns:mypets="http://www.shuchow.com/"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
>
```

The namespaces `rdf` and `rdfs` are tied to w3.org resources, which tells us that they are industry standards. `mypets`, however, is tied to shuchow.com, the file's domain. This means that it's probably a proprietary vocabulary created by the shuchow.com organization to support the information. To find out more, we could visit the site. Doing so should lead us to some documentation on some of the syntax we will encounter.

The rest of the file is basically a list of pets wrapped around `Description` elements with some details as child elements. The `about` attribute in the `Description` element points to the exact subject of this item.

```
<rdf:Description rdf:about="http://www.shuchow.com/thecats.html#avi">
   <mypets:name>Avi</mypets:name>
   <mypets:age>6</mypets:age>
   <mypets:gender>F</mypets:gender>
   <rdfs:type rdf:resource="http://www.shuchow.com/#parrot"/>
</rdf:Description>
```

The name, age, and gender of each pet are the value of their respective elements. Each of these elements is namespaced to `mypets`. The `type` of the item is a URI pointing to a location that describes what this "thing" is. For this file, it is an imaginary URI used only as a way to separate the types of animals in my house. In the real world, this may also not point to a real file, or it may have a complex RDF taxonomy definition behind it. These `Description` blocks are repeated for each pet.

Anatomy of a SPARQL Query

If you know SQL, it should be easy to understand the first few lines of a SPARQL query. Let us take a look at a simple SPARQL query to understand its parts. Suppose we want to extract one specific piece of information about a specific pet. Let's say we wish to extract Saffy's age. We know in the document that the age is the value of the `mypets:age` element. We also know that the name of the pet, Saffy, is in the `mypets:name` element. We need a query that will extract the value of `mypets:age` restricted by the value of `mypets:name`.

This SPARQL query will give us this information:

```
PREFIX mypets: <http://www.shuchow.com/>
SELECT ?age
FROM <http://www.shuchow.com/mashups/ch6/pets.rdf>
WHERE {
   ?Description mypets:name "Saffy" .
   ?Description mypets:age ?age
}
```

There are a couple of syntactical things we need to state before we look at this query. First, in SPARQL, URIs are surrounded by less than and greater than brackets. Second, SPARQL queries rely on variables to name values. Variable names are denoted with a question mark at the beginning.

The first line of this query is a `PREFIX` statement. `PREFIX` statements are required for every namespace that the query will encounter in the RDF document. In `pets.rdf`, there are actually three namespace declarations. However, to extract the age, we touch `mypets:name` and `mypets:age`, and they share a common namespace. Therefore, in our query, we only need to prefix the `mypets` namespace. The format is the `PREFIX` keyword, followed by the namespace name as given in the RDF document, a colon, and finally the namespace value also as given in the RDF document.

The next line is the `SELECT` statement. In the `SELECT` statement, list the names of the SPARQL query variables you wish to extract. In SQL, `SELECT` statements are followed by the names of the table columns or aliases. In SPARQL, variables are defined, and their values set, in the `WHERE` clause. `SELECT` statements specify those variables you wish to pluck. We will look at how to define SPARQL variables very shortly. To keep things simple, this example uses the name of the element we are interested in, `age`, as the variable name, `?age`. However, `SELECT ?mangoes` would have also given us the same results as long as the second line in the `WHERE` clause was changed to `?Description mypets:age ?mangos`. If you wish to extract multiple variables, list each variable out in the `SELECT` statement, separated by spaces.

The next statement is the `FROM` statement. In SPARQL, this statement is optional. It is used to point to the source of the RDF data. In many parsers, the location of the RDF document is made outside of the SPARQL query. For example, some parsers take the URL of the RDF document as a constructor argument. The `FROM` statement, although not necessary, is like a comment for the query. It tells us that this query is written for this specific RDF document. Like programmer comments, although not necessary, it is good form to include this statement. In SPARQLer, we have the option of either putting the source URL in the query or in a separate field.

Writing SPARQL WHERE Clauses

Finally, we get to the `WHERE` clause. In SQL, a `WHERE` clause narrows down and refines the data we are looking for. In SPARQL, it does the same thing. It also gives a sense of structure for the query and parser. In a SQL database, a table has a defined, consistent schema. A RDF document is a flat file. From a parser's standpoint, there really is no guarantee of any sort of structure. A SPARQL `WHERE` clause gives the parser an idea of how objects and properties are organized and how they relate to each other.

Basic Principles

Recall the three parts of a RDF triple, and what they represent:

- A Subject tells us the "thing" that this triple is about.
- A Predicate specifies a property of the subject.
- An Object is the value of the predicate.

A triple is simply each part, written out, in one line and separated by a string.

A SPARQL `WHERE` clause is just a series of triples strung together. Further, each part of a triple can be substituted with a variable.

For example, let's say there is a cat named Gilbert. He has green eyes.

In a simple RDF, he can be represented like such:

```
<rdf:Description rdf:about='http://www.example.com/
cats#GilbertTheCat'>
   <name>Gilbert</name>
   <eyeColor>Green</eyeColor>
</rdf:Description>
```

In triple form, this can be presented like such:

```
rdf:Description  name   "Gilbert"
```

This isolates the cat who's `name` value is "Gilbert." The item we are focusing on is the subject. This is represented by the `rdf:Description` element. `Name` is the property of the subject, which makes it the predicate. The value of the `name`, the object in this triple, is "Gilbert". To specify the literal value of a triple's object, we wrap the value around with quotes.

In queries, we can replace the subject with a variable.

```
?catObject  name   "Gilbert"
```

Now, `?catObject` holds a reference to the cat who's name is Gilbert. We can use this variable to access other properties of Gilbert the cat. To access Gilbert's eye color, we could use two triples strung together:

```
?catObject  name       "Gilbert" .
?catObject  eyeColor ?eyeColor
```

To string together triples in a SPARQL query, use a period. This acts as a concatenation operator, much like a period is used in PHP.

In this grouping, the first triple will place the subject, Gilbert The Cat, in the `?catObject` variable. The second triple's subject is the variable `?catObject`. That means the predicate and object of the second triple will use this subject. This second triple will place Gilbert's eye color in the `?eyeColor` variable. To return the `eyeColor` variable in the SPARQL resultset, we need to specify it in the `SELECT` statement.

In SPARQL `WHERE` clauses, the key concept to remember is that all variables reference the same thing**.** The order of the `WHERE` statements matters very little. It is what each variable's value is at the end of execution that matters.

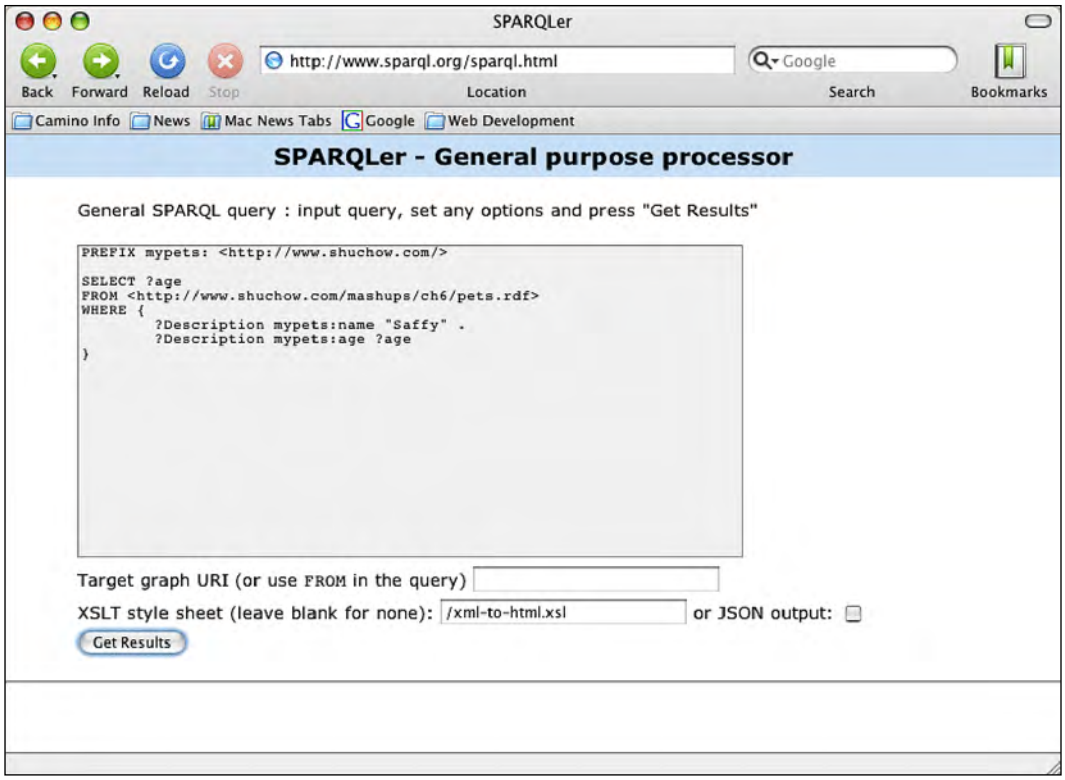
A Simple Query

This is the same principle that is applied to our earlier query that extracts Saffy's age in our pets RDF document.

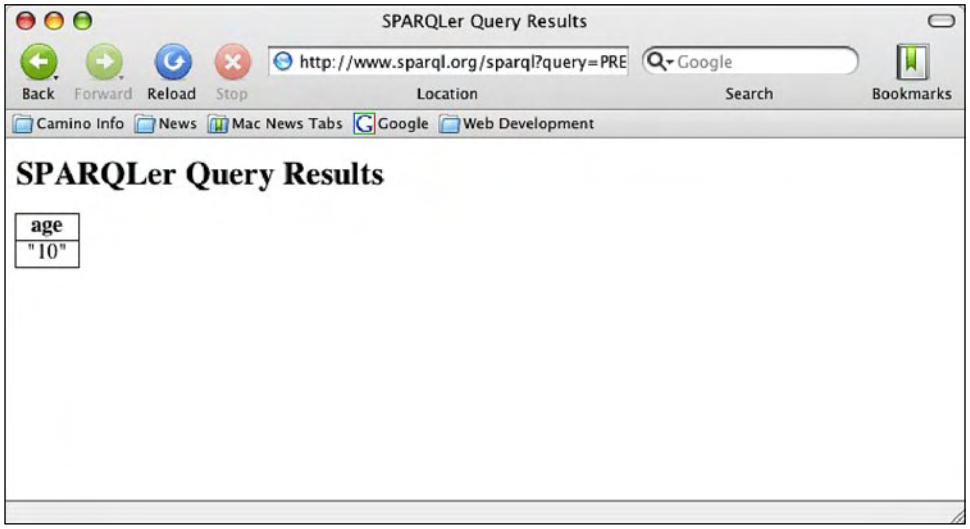
To see this in action, let's load up the online XML parser. Bring up SPARQLer (`http://www.sparql.org/sparql.html`) in a web browser. You will be presented with a simple form. The text area is where the SPARQL query you want to run is entered. As long as you have a FROM clause in the query, you can leave the **Target graph URI** field blank. The other options on the form can also be left blank. Enter the age query into the query text area in the form:

```
PREFIX mypets: <http://www.shuchow.com/>

SELECT ?age
FROM <http://www.shuchow.com/mashups/ch6/pets.rdf>
WHERE {
   ?Description mypets:name "Saffy" .
   ?Description mypets:age ?age
}
```



Click on the **Get Results** button. SPARQLer will go out to retrieve `pets.rdf`, load it, and then proceed to parse it.



The result will show that Saffy's age is 10.

The first triple finds the item that has a name (designated by the `mypets:name` element) with a literal value of `Saffy`. The subject of this item is placed in the `?Description` variable. Note that in the predicate of both triples in the `WHERE` clause, the namespace is included with the element name. This is another important thing to remember when writing SPARQL queries—if the element name in the RDF document has a namespace prefix, you must also include that prefix in the SPARQL query, along with declaring the namespace in a `PREFIX` statement.

Not only does this first clause zero-in on `Saffy`, but it sets the context of our search and places it into the `?Description` variable. This is extremely important in SPARQL because every clause requires a subject. Thanks to this clause, we can use `?Description` as the subject for other `WHERE` clauses.

The second statement says the following:

> "*The subject of this triple is referenced by* `?Description` *(which we already set in the first triple). The predicate of this subject that I'm interested in is* `mypets:age`*. Place the object of this triple into a variable named* `?age`."

It is wordy to think of the query like this, but necessary. When learning and using SPARQL, it's very important that we keep in mind the notion of triples. It's very easy to fall back into a SQL mindset and think, "This clause gets me the station name based on the element". However, what's really going on is more complicated than that. The element name is useless unless the subject is defined throughout your query.

During the parsing process, the parser finds that `?age` is represented by "10" in the document. The `?age` variable is returned because it is specified in the `SELECT` statement.

This example returned just one pet by using the pet's name. We can place no restrictions on the value and return all the results. This would be like a SQL SELECT query without a `WHERE` clause (`SELECT ColumnName FROM TableName`).

```
PREFIX mypets: <http://www.shuchow.com/>
SELECT ?name
FROM <http://www.shuchow.com/mashups/ch6/pets.rdf>
WHERE {
   ?Description mypets:name ?name
}
```

Go back to SPARQLer and enter this query. This `WHERE` clause will execute and place all of the `mypets:name` values into a variable named `?name`. Our `SELECT` statement returns this variable back to us.

Your SPARQLer result set should look like this:

name
"Pim Pim"
"Saffy"
"Manfred"
"Lizzie Borden"
"Tera-San"
"Moose"
"Hoser"
"Mathilda"
"Opal"
"Wozniak"
"Dolly"
"Avi"
"Snowball"

Querying for Types

In the first query, we used a literal value of the name `Saffy` to find what we were looking for. Simply searching on a literal value is often not a reliable approach. Earlier, we noted that the RDF Schema vocabulary allows us to classify subjects as programming objects using the `type` element. This next example will show how to restrict on this element.

Let's say we wish to grab the names of all parrots. Our `WHERE` clause needs to do the following:

- Find the parrots in the RDF document.
- Extract their names.

The `type` element is still the predicate. However, this element does not have a value we can use as the triple object. Instead, the `resource` attribute value is the object in this triple. `resource` is a URI that points to a description of what a parrot is. Remember that triple objects can be either a literal value or a URI. Again, this particular example URI is only an example to identify, not a formal vocabulary definition, which it sometimes can be. This combination says "This subject is a parrot". From there, we can extract the `name` element as we did before.

The restriction requirement is similar to what we have been doing. The triple associated with it will use a URI instead of quoted literals like the previous examples. We can specify this simply by specifying the URI in the query using greater than/less than signs.

This triple is simply this:

```
?Description rdfs:type <http://www.shuchow.com/#parrot>
```

Sometimes, you may find this resource attribute starts with a local anchor, the pound sign (#) followed by the value like so:

```
<rdfs:type rdf:resource="#value"/>
```

This pound sign is a reference to the document itself, much like it is used in HTML anchor tags to reference locations within the same document.

Simply the object of "`#value`" does not qualify as a full URI in SPARQL triples. As the pound sign is a redundant reference, we must also include the absolute path to the file we are querying in the triple. Assuming the page at `http://www.example.com/this.rdf`, to search on these values, you would need to include the full URI back to the document, along with the value after the pound sign:

```
?Subject ns:predicate <http://www.example.com/this.rdf#value>
```

The complete SPARQL query looks like this:

```
PREFIX mypets: <http://www.shuchow.com/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?name
FROM <http://www.shuchow.com/mashups/ch6/pets.rdf>
WHERE {
   ?Description rdfs:type <http://www.shuchow.com/#parrot> .
   ?Description mypets:name ?name
}
ORDER BY ?name
```

Running the query returns these results:

Name
"Avi"
"Dolly"
"Hoser"
"Moose"

Ordering, Limiting, and Offsetting

Note that in this query, we added an `ORDER BY` clause. SPARQL supports a set of clauses that follow a `WHERE` clause, which organizes the returned dataset. In addition to `ORDER BY`, we can use `LIMIT` and `OFFSET` clauses.

An `ORDER BY` clause works very similarly to SQL's `ORDER BY` clause. This clause sorts the returned dataset by the variable that follows the clause. The results returned are ordered alphabetically if they are strings or ordinal if they are numeric. Ascending and descending options can be specified by using the `ASC` and `DESC` functions, respectively.

```
ORDER BY ASC(?name)
ORDER BY DESC(?name)
```

The ascending and descending clauses are optional. If they are left out, the default is ascending order.

SPARQL also supports the `LIMIT` and `OFFSET` keywords much like PostgreSQL, MySQL, and other relational database management systems. Both `LIMIT` and `OFFSET` are followed by integers. `LIMIT` will limit the number of results returned to the integer passed to it. `OFFSET` will shift the start of the returned results to the position of the integer, with the first returned result being position zero.

For example, `pets.rdf` has 13 animals in the list. If we want to get the 7th and 8th pets, in by alphabetical order, we can use `LIMIT` and `OFFSET` in conjunction with `ORDER BY`.

```
PREFIX mypets: <http://www.shuchow.com/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?name
FROM <http://www.shuchow.com/mashups/ch6/pets.rdf>
WHERE {
   ?Description mypets:name ?name
}
ORDER BY ?name
LIMIT 2
OFFSET 6
```

Note that order matters when you use `ORDER BY`, `LIMIT`, or `OFFSET`. These three clauses must be in that order after the `WHERE` clause. For example, this will not work:

```
OFFSET 6
ORDER BY ?name
LIMIT 4
```

UNION and DISTINCT

The `UNION` keyword joins multiple `WHERE` groupings together, much like `UNION` in SQL. The returned results will be a combination of the `WHERE` groupings. To use a `UNION` clause, wrap the individual groupings within curly brackets. Join them with the `UNION` keyword. Place all of this within the regular `WHERE` curly brackets.

For example, this query will retrieve the names of all parrots and male pets:

```
PREFIX mypets: <http://www.shuchow.com/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?name
FROM <http://www.shuchow.com/mashups/ch6/pets.rdf>
WHERE {
    {
      ?Description rdfs:type <http://www.shuchow.com/#parrot>
      ?Description mypets:name ?name
    }
    UNION
    {
        ?Description mypets:gender "M" .
        ?Description mypets:name ?name
    }
}
ORDER BY ?name
```

Name
"Avi"
"Dolly"
"Hoser"
"Hoser"
"Manfred"
"Moose"
"Moose"
"Snowball"
"Wozniak"

This union does not give us exactly the query we want. Hoser and Moose, male parrots, are in both the first clause and the second. SPARQL supports another SQL keyword, `DISTINCT`, that will exclude a row based on a column if it has already been included in a previous clause.

Simply add the `DISTINCT` keyword you wish to insure uniqueness on, and the results will reflect the change.

```
SELECT DISTINCT ?name
```

name
"Avi"
"Dolly"
"Hoser"
"Manfred"
"Moose"
"Snowball"
"Wozniak"

More SPARQL Features

The queries we will write later will require more complexity, but the features we have discussed are more than we will need for our mashup. SPARQL, however, has many more advanced features including:

- Querying more than one RDF document (if the parser supports it).
- The ability to filter returned results using special operators and a subset of XPATH functions.

The Working Draft document that fully outlines all of SPARQL's features can be found at `http://www.w3.org/TR/rdf-sparql-query/`. Although it is still in W3C draft stage, many parsers give great support to the language. In future mashups, if you encounter complex RDFs, it would not hurt to be familiar with SPARQL's advanced features to see if it is a viable solution to extract data.

RDF API for PHP (RAP)

Now we know a bit about RDF and SPARQL, we need a way to actually execute SPARQL queries in an application. There are not any core PHP functions for RDF, but there is a very powerful third party library called RDF API for PHP (RAP). RAP is an open source project, and can do just about anything you require with RDF. RAP is basically a collection of RDF models. Each model suits a specific purpose.

A model named MemModel is a RDF file stored in memory. Another model named DbModel, is a used to persist RDF models in a relational database. Each model has specific methods that fit its purpose. DbModel has methods to automatically insert and retrieve the model into and out of a relational database.

All models inherit methods from a generic abstract class called Model. These are generic utility methods that apply to all models. For example, all models need to load a RDF file to do anything with it. The `load()` method accomplishes this. All models can be represented graphically using the `visualize()` method, which creates a graphical representation of the RDF file. Version 0.94 includes a method named `sparqlQuery()` that accepts a SPARQL query and executes it against the model. We will be using this method to create a SPARQL client.

The project home page is located at `http://sites.wiwiss.fu-berlin.de/suhl/bizer/rdfapi/`. You can download the latest version from there. Documentation is also available, and is very extensive. Download the code, and unzip it. It will create a directory named `rdfapi-php`. Then, place `rdfapi-php` in a directory in your application structure. This directory must be accessible by Apache, and terms of location and permissions.

We will use a few of the previous example SPARQL queries as examples for RAP. In the examples code, the file named `rapExample.php` executes two SPARQL queries. Let's take a look at this file to see the steps required to use RAP for SPARQL queries.

The file has some preliminary setup PHP code at the top.

```
define("RDFAPI_INCLUDE_DIR", "Absolute/Path/To/rdfapi-php/api/");
require_once(RDFAPI_INCLUDE_DIR . "RdfAPI.php");

//Create SPARQL Client
$sparqlClient = ModelFactory::getDefaultModel();
$sparqlClient->load('http://www.shuchow.com/mashups/ch6/pets.rdf');
```

The very first thing we need to do is create a global variable named `RDFAPI_INCLUDE_DIR`. The value of this is the absolute path to the `rdfapi-php/api` directory you just installed. We then use this global variable to include the `RdfAPI.php` file. These two lines are required for every use of the RAP library.

Next, we create a default model object. The default model is a generic model that all other models inherit from. It is created in the statement that calls `getDefaultModel()`. The default model object includes the basic methods we will need.

The last line in this block loads the RDF file using the default model's `load()` method. Here, we load a remote file, but you can also keep a RDF file locally.

Remember, the FROM clause is not used in a SPARQL query. The file you pass here is actually the real RDF source. Being able to load remote files obviously means we can use this library on all RDF-based mashups, and can get RDF data at run time.

After this, we can create a query and execute it.

```
$query = '
PREFIX mypets: <http://www.shuchow.com/>
SELECT ?age
FROM <http://www.shuchow.com/mashups/ch6/pets.rdf>
WHERE {
       ?Description mypets:name "Saffy" .
       ?Description mypets:age ?age
}';

$result = $sparqlClient->sparqlQuery($query);
if ($result != false) {
     foreach ($result as $cat) {
        if ($cat != "") {
          echo "Age: " . $cat['?age']->getLabel();
        }
     }
}
```

In this block, we put our SPARQL query into a variable named $query. We pass that to the sparqlQuery method. This method is in the default model. It accepts a SPARQL query and executes it against the RDF file in memory. Its return value is an array of objects. The key in each array is a variable that we added to the SELECT clause of the query, including the question mark. These are Resources objects in the RAP library. The getLabel() method in the Resources object returns the value of the variable.

To grab multiple variables, we just use the other keys in our foreach loop.

```
$query = '
PREFIX mypets: <http://www.shuchow.com/>
SELECT ?name ?age
FROM <http://www.shuchow.com/mashups/ch6/pets.rdf>
WHERE {
       ?Description mypets:name ?name
       ?Description mypets:age ?age
}
LIMIT 5
```

```
';

$result = $sparqlClient->sparqlQuery($query);

if ($result != false) {
    foreach ($result as $cat) {
        if ($cat != "") {
            echo "Name: " . $cat['?name']->getLabel() . ", Age: " .
               $cat['?age']->getLabel() . "<br />";
        }
     }
}
```

Running this code produces this output on screen:

```
Name: Snowball, Age: 14
Name: Lizzie Borden, Age: 14
Name: Saffy, Age: 10
Name: Pim Pim, Age: 12
Name: Tera-San, Age: 6
```

RAP is quite a powerful tool. We only used a small portion of its features. If RDF is a big part of your applications, it is certainly worthwhile exploring this extensive library.

XMLHttpRequest Object

The next technologies we will look at depart from the server-oriented tools we have used. You have probably heard of AJAX, Asynchronous JavaScript and XML transfer. At the least, you have probably seen it on sites like Google Mail and Yahoo! Mail. AJAX allows web browsers to interact with a server without refreshing the page. Combined with dynamic HTML, it has created a new level of interactivity between users and websites. With the near instantaneous data changes in front of a user, web applications have never been more like desktop applications.

Another benefit to AJAX is that it can severely decrease the traffic between web browser and web server. When we take a look at the amount of data being passed to Google Maps, we will see why constant refreshes would slow down the application too much.

As we discuss AJAX and `XMLHttpRequest`, we'll build a very simple web application. This application will take input from the user, pass it to a server, the server will send back an XML document to the browser, and using JavaScript, we will change the page dynamically. The client component of this application is in the examples code as `ajaxTest.html` The corresponding server component is named `ajaxResponse.php`.

The HTML page, without the JavaScript code, is very basic.

```
<html>
<head>
<script type="text/javascript" language="JavaScript">
…
</script>
</head>
<body>
<form name=»theForm» action=»#»>
   <input type=»text» name=»inputField» size=»10» />
   <input type=»button» value=»Click Me» />
</form>
<h1>Server Response Area</h1>
<span id=»ServerResponse»>Nothing yet</span>
</body>
</html>
```

This page is simply a form with a paragraph underneath it which will be updated using JavaScript.

`ajaxResponse.php` is just as simple. This script will take a query parameter named field, and pass it back to the requester as a very simple XML document.

```
<?php
   header("Content-type: text/xml; charset=UTF-8");
?>
<?= '<?xml version="1.0" encoding="utf-8" ?>' ?>
<response>
   <textField>You've entered: <?= htmlentities($_GET['field'])?>
   </textField>
</response>
```

The key here is that the page will use a query parameter named `field`.

XMLHttpRequest Object Overview

The `XMLHttpRequest` object is the heart of AJAX. This is an object built into all modern web browsers (version 5.0 and above) to control HTTP requests. This object is similar to other objects built into web browsers, say the form object to control all form elements, or the `window` object to control the web browser window. All AJAX really is the technique of using `XMLHttpRequest` to make an HTTP request to the server, triggered by some JavaScript event, after the page has loaded. The server returns some data, and the `XMLHttpRequest` object passes the server response to some JavaScript function on the page. Again, using JavaScript, page stylesheet information and the web browser Document Object Model (DOM) is changed dynamically. Let's walk through the life cycle of a simple `XMLHttpRequest`.

Using the Object

The lifecycle is started by a JavaScript event. This can be anything the application needs it to be—a mouseover, a page load, a button click, etc. Once triggered, the steps that take place are:

1. Create the `XMLHttpRequest` object.
2. Define the destination server information (URL, path, port, etc.) of the HTTP request that we are going to make.
3. Much like the web services we used earlier, we need to define the content that we are going to send in our HTTP request. This may be just a blank string.
4. Specify the callback function, and build it.
5. Use the object's `send()` method to send the request.
6. In the callback, catch the server response and use it to change the page.

Creating the Object

There are two ways to create the `XMLHttpRequest` object, depending on which browser the visitor is using. If the user has a Mozilla browser (Firefox, Camino), Safari, or Opera, we just create a `new XMLHttpRequest()` to create an object. If they are using Internet Explorer 6, we need to use ActiveX to create a Microsoft.`XMLHTTP` object, which is a clone of `XMLHttpRequest`. Use these two methods to place the returned objects into a global JavaScript variable. We can use JavaScript to detect the presence of the `XMLHttpRequest` object or Active X to determine which method we should use.

```
g_xmlHttp = null;
function createXMLHttpRequest() {
   if (window.XMLHttpRequest){
      g_xmlHttp = new XMLHttpRequest()
   }
   else if (window.ActiveXObject) {
      g_xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
   }
}
```

This function should be called at the start of the request.

Making the HTTP Request

The start of our HTTP Request should be after the user does something. We will trigger the request when the user triggers a key release on the text field. That is, when the user presses on a key, the web application will call our JavaScript function that communicates to the server.

```
<input type="text" name="inputField" size="10"
onkeyup="sendRequest()" />
```

The function is named `sendRequest()` here. We now need to write this function. This function will create the `XMLHttpRequest` object, define the server parameters, define callback function that will be executed when a server response is captured, and then actually send the request.

```
function sendRequest() {
    createXMLHttpRequest();
    var url = "/mashups/ch6/examples/ajaxResponse.php?field=" +
              document.theForm.inputField.value;
    g_xmlHttp.onreadystatechange = parseResponse;
    g_xmlHttp.open("GET", url, true);
    g_xmlHttp.send(null);
}
```

The first statement in this function calls `createXMLHttpRequest()`, which creates the `XMLHttpRequest` object and places it in the global variable `g_xmlHttp`. The second line places the URL to the service in a variable. This is a virtual URL to the service. You can also make an absolute URL to the service, but we'll discuss later why an absolute URL is unnecessary. The last part of this statement places the value of the input text box we had into a query parameter named `field`, which is what our service is waiting for.

The next three statements use `XMLHttpRequest` methods and properties. `onreadystatechange` is a property that holds the JavaScript callback function for this object. Set this to the name of the function, without opening and closing parentheses, that will be executed when the server responds. You can only select one callback function. To execute more, you will need to create a facade wrapper function that executes the others, and set the facade function as the callback.

`open` gets the object ready to send the request. The first two parameters are required. The first parameter is the HTTP method to use. The second is the URL. The third parameter is whether the object should be in asynchronous mode. It is optional, but it is a good idea to set this to `true` because the default value is `false`, and we do want to be in asynchronous mode. Otherwise, we would be in synchronous mode, which means that the rest of the JavaScript does not execute until `XMLHttpRequest` receives a response from the server.

`send` actually sends the request. `send` takes one required parameter, the body of the request. In this example, we are sending a null because we are just doing a `GET` request. The request does not have a body. If we were doing a `POST`, we would construct the parameters in a separate string and pass it as `send`'s parameter. After `send` is called, the HTTP request is made and the callback function executes.

Creating and Using the Callback

There are two main jobs of the callback function. The first is to capture the server response. The second is to do something with that response.

We start off our function with a couple of checks to make sure the data from the server has indeed arrived. If we didn't do this, the rest of our code will execute prematurely and without all the necessary parts from the server response.

The first `if` statement checks the `readyState` property of the `XMLHttpRequest` object. As the request executes and processes, this value gets changed. There are five possible values of this property:

readyState value	Meaning
0	Uninitialized
1	Loading
2	Loaded
3	Interactive
4	Completed

Only when the value is 4 is the data completely ready to be parsed and used by the web application.

The second `if` statement checks to see `XMLHttpRequests`' status property. This is the same code that reports 404 for missing file, 500 for internal server error, etc. A 200 is a successful transaction. We need to make sure the request is executed successfully, or the data might be useless.

```
function parseResponse() {
   if (g_xmlHttp.readyState == 4) {
      if (g_xmlHttp.status == 200) {
          var response = g_xmlHttp.responseXML;
          var outputArea = document.getElementById("ServerResponse").
                           firstChild;
          var responseElements =
                           response.getElementsByTagName("textField");
          outputArea.nodeValue =
                           responseElements[0].firstChild.nodeValue;
       }
   }
}
```

The first line after the nested `if` statement captures the value in the `responseXML` property of the `XMLHttpRequest` object and places it in a variable. This property is where the browser keeps the response from the server. If you were to inspect it, you would see the direct XML from the server.

The second statement captures the node of the HTML page of where we are going to output the response. We use JavaScript's `getElementById()` function and traverse down the DOM.

We can use the same DOM functions in JavaScript to extract the information from the server response. This is what we do in the third statement. We know what we are interested in is located in the `textField` element of the response. We zero in on that and get that node.

Each DOM element keeps the text it displays in a property called `nodeValue`. In the fourth statement, we set the output area's `nodeValue` to the `nodeValue` of the response. This changes the webpage every time it is executed.

If you type in the text field of `ajaxTest.php`, you can see this code in action.

In our code, we checked for an HTTP status of 200. While this is good practice, it requires the HTTP network protocol to be present in order work. This means you must load the page in a web browser through HTTP. If you load the page through the file system (i.e. through `file:///ajaxTest.php`,instead of `http://localhost/.../ajaxTest.php`), status check will fail, and the code will not execute properly.

This is the standard way of triggering an AJAX application, and it works very nicely. The DOM parsing, however, can get messy. There are two DOMs you must parse—the local web page and the server response. Fortunately, you may have some alternatives to parsing the server response.

First off, `responseXML` has a sister property, `responseText`, that works exactly the same way. `responseText` holds the server response if it is any text string instead of XML. You can immediately use the response text instead of traversing through a DOM to get what you want. If you are merely a front-end developer for a much larger web development, and the company manifesto is to transfer everything via XML, this might not be an available option for you. Or, if your web service is used by third parties, it may be best to keep it as XML. However, if you are writing a very simple service to support just your application, know that you do not have to structure everything in XML. You can just pass a simple text string back and use `responseText` on the client end instead.

If your web service response is too complicated for a simple text string, you may want to consider formating your text response in JavaScript Object Notation (JSON) to send this result back to the page. It will still be a text response, so you can use `responseText` and skip the parsing. JSON gives you the structure of XML with the simplicity of a text string. This next section will introduce us to JSON.

Debugging AJAX

Debugging the request and response from the server can be tricky. We can't use a regular IDE. We need something to watch the HTTP streams. Luckily, if you are using Firefox, there is a Greasemonkey script that will do just that. Greasemonkey is a Firefox extension that allows users to write their own JavaScript and code against a site when they visit it. It can be found at `https://addons.mozilla.org/firefox/748/`. Once you have that install, download the `XMLHttpRequest` debugging tip at `http://blog.monstuff.com/archives/000250.html`. This tool will watch everything that comes out from the browser, and everything going in. Other helpful extensions for Firefox include LiveHTTPHeaders, which show the request and response HTTP headers, and Firebug, a general JavaScript and CSS debugger. For Internet Explorer, a commercial tool called HTTPWatch is available to watch HTTP requests.

JavaScript Object Notation (JSON)

JavaScript Object Notation is simply a transfer format, much like SOAP or XML-RPC. Unlike those two formats, JSON is not XML based. It is JavaScript code that is loosely based on a C-style definitions and formats. Although called JavaScript Object Notation, many server side languages have built parsers to interpret JSON format. Given this and its lightweight nature, it has become a popular alternative to XML when communicating between a web browser and a client. JSON's home page is at `http://www.json.org`.

JavaScript Objects Review

Let's quickly review JavaScript objects first. To define a class in JavaScript, you simply treat it as if it was a function. To give the class properties, use the keyword `this`, followed by a dot, followed by the name of the property. To give the class methods, also use `this`, followed by a dot, the name of the function, an equal sign, the keyword `function` and then the function definition. For example, this could be a cat object in JavaScript:

```
function Cat (name) {
   this.name = name;
   this.gender;
   this.age;
   this.eat = function() {
                          alert("Yum");
                         }
   this.sleep = function() {
```

```
                                alert("zzzz…");
                                }
}
```

This class definition requires a name as a constructor because it is the only required parameter in the class definition. Cats can be instantiated like so:

```
aCat = new Cat("Quincy");
anotherCat = new Cat("Buddy");
```

JavaScript objects are pretty basic. There are no accessor keywords. Everything is public. You can access or set properties simply by using dot notation on the object.

```
aCat.gender = "F"; //Quincy is now a female
anotherCat.name = "Gilbert";  //Buddy just got a name change.
```

Note the dot notation we use to access the object properties. We use the same dot notation when we access JSON properties.

JSON Structure

To delimit object definitions, the object is named followed by an equals sign. The properties of the object are then enclosed in curly brackets. JSON properties are name/value pairs separated by a colon.

JSON properties support the following data types:

Type	Format	Examples
Number	Integer, float, or real. The actual number.	1, 2.8217
String	Double quoted value.	"A Value", "Another Value"
Boolean	True/false, no quotation marks.	true, false
Array	Square bracket delimited list.	[34, 498, 12]
Object	Curly Brackets.	{ property one: value one }
Null	Null.	Null

The JavaScript cat structure above can be represented and expanded in JSON like so:

```
var cat = {
   name: "Quincy",
   gender: "F",
   age: 4,
   spayed: true,
```

```
   collar: {
            charm: "bell",
            color:  "green"
           }
}
```

If this cat was represented using XML, it would be a bit more cumbersome and definitely eat more bytes:

```
<cat>
   <name>Quincy</name>
   <gender>F</gender>
   <age>4</age>
   <spayed>true</spayed
   <collar>
       <charm>bell</charm>
       <color>green</green>
   </collar>
</cat>
```

Accessing JSON Properties

In the above example, the properties of the cat can be easily accessed through dot notation with `cat` as the parent object. Her name is found by using the variable `cat.name`, her age is at `cat.age`, etc. The example file `jsonExample.html` shows how dot notation is used to access a property of a JSON object that is in the response. You simply drill down further with the name of the object as a dot notation level. The code displays Quincy's collar color using the variable `cat.collar.color`.

```
function getColor() {
   alert("Quincy's Collar Color: " + cat.collar.color);
                    }
```

JavaScript is a typeless language (meaning you do not have to specify which data type a variable is), so we can use properties directly through dot notation. The only thing that may need a conversion or alteration step are JSON arrays. For example, let's insert an array of fur colors into the above example.

```
age: 4,
furcolor: ["white", "orange"],
spayed: true,
```

The `furcolor` is still accessible through dot notation, but there will be some twists. If you access the array directly, you will get a string of the array elements separated by commas. `cat.furcolor` will be "white, orange". To access individual elements, attach the element number in brackets after the array name, like you would a normal JavaScript array. `cat.furcolor[0]` will have a value of "white". `cat.furcolor[1]` will have a value of "orange." You can also check the length of the array by accessing `.length` after the array name in dot notation. `cat.furcolor.length` will have a value of 2.

Serializing the JSON Response

As given in the example, the cat is already a serialized JavaScript object. The curly brackets that immediately enclose the properties give this away. This means that we can work directly with the data through dot notation.

Very frequently, though, you will receive a string representation of a JSON object. One such situation is if the JSON object is stored in a `XMLHttpRequest` object's `responseText` property. Sure, structurally the object is in JSON. However, the data is cast as a string.

To turn the JSON string into a JavaScript object, pass it through the JavaScript `eval()` method.

```
var cat = '{"name": "Quincy", "gender": "F", "age": 4, "spayed": true,
"color": ["white", "orange"], "collar": { "charm": "bell", "color":
"green" }}';
var quincyObj = eval('(' + cat + ')');

function getColor() {
   alert(«Quincy's Collar Color: « + quincyObj.collar.color);
}
```

The `eval()` method executes whatever is passed to it. As we are passing in something that is formatted as an object, it will return an object. This unserialized to serialized example is in a file named `jsonTest.html`.

Note that in the call to `eval()`, we have to wrap the string within literal string parentheses. This is because while the code looks like a JavaScript object, `eval()` treats the opening curly bracket in the string as a generic block opening, and not as the start of an object. Placing it within parentheses will put the parser into expression parsing mode, which correctly will parse it as a JavaScript object.

Be careful with `eval()`

Be careful when you use `eval()`. It blindly executes any code passed to it, so make sure you fully trust the source of the input.

Finally, we get to our APIs. We only have two we need to look at—the Google Maps API and Flickr Web Services.

Google Maps API

The Google Maps API allows third party developers to use the features of Google Maps on their own sites. Anything you can do as a user of Google Maps can be done using the Google Maps API. The Google Maps documentation home page is located at `http://www.google.com/apis/maps/documentation/`. The documentation is quite extensive. We will take a look at how the API basically works, and concentrate on the features we will use in our mashup. Just knowing how the API is organized is the key step in searching for information and using the Google Maps API in future projects.

The Google Maps API requires an API key. You can register for it for free at `http://www.google.com/apis/maps/signup.html`. This key is used when including the Google Maps API in your page. Before you do anything with Google Maps, you will need to get this API key and put this source tag and in top of your page's `head` tag.

```
<script src="http://maps.google.com/maps?file=api&amp;v=2&amp;key=Your
Google API Key" type="text/javascript"></script>
```

The API is a JavaScript API based heavily on objects. The central object is the Google Map that you see. Everything that you see on Google Maps including map controls, icons, lines, and the white information window box, are just JavaScript objects added to the map. As we go through the examples in this section, we will build the same page that is in the examples named `googleMapTest.php`.

Creating a Map

The Map is created by instantiating the `GMap2` class. The only required parameter in the `GMap2`'s constructor is an HTML container to place the map. Typically, this is an empty `div` tag. The Google Map will be displayed in the space occupied by this tag. This places a lot of importance on this container. You can use CSS to position the map on the page, and the size of the container determines the size of the map.

Let's take a look at a simple example:

```
<html>
<head>
   <title>Google Maps Scratch</title>
   <script src="http://maps.google.com/maps?file=api&amp;v=2&amp;
   key=YOUR_GOOGLE_API_KEY" type="text/javascript"></script>

    <script type="text/javascript">
    var g_map;

    function load() {
         if (GBrowserIsCompatible()) {
             g_map = new GMap2(document.getElementById("map"));
          }
    }
     </script>
</head>
<body onload="load()">
     <div id="map" style="width: 800px; height: 600px"></div>
</body>
</html>
```

This simple page would create a Google Map. We declare a global variable named `g_map` to hold the Google Map. The load function is run when the onload event is triggered. In the load function, a Google JavaScript function is called, `GBrowserIsCompatible`, to check for browser compatibility. If it passes, we create the map by instantiating `GMap2`. We pass the container using the JavaScript DOM function `getElementById` to the `GMap2` constructor. As the size of the div element is 800 by 600 pixels, this map will also be 800 by 600 pixels.

If you actually ran this code, you would find that it's pretty useless. You would just get a blank, grey map. The problem is that the map doesn't know where to initially center itself. You must specify this by using the map's `setCenter()` method. `setCenter()` can actually be called at any time, and can be triggered by any event. It accepts a `GLatLng` object as its parameter.

Geocoding

As you work with Google Maps, you will find that it relies heavily on latitude and longitude coordinates to do anything on the map. The problem is that in every day communication, we use addresses more often than latitude/longitude coordinates. The process of translating from an address to a latitude/longitude coordinate is known as geocoding. To make using Google Maps a lot easier, the API provides an object named `GClientGeocoder` to geocode for us.

To create a geocoder, first instantiate the `GClientGeocoder` object. This object has a method named `getLatLng()`, which takes two parameters. The first parameter is a string of the address you wish to look up. The second is a callback function that is called after the server returns the results.

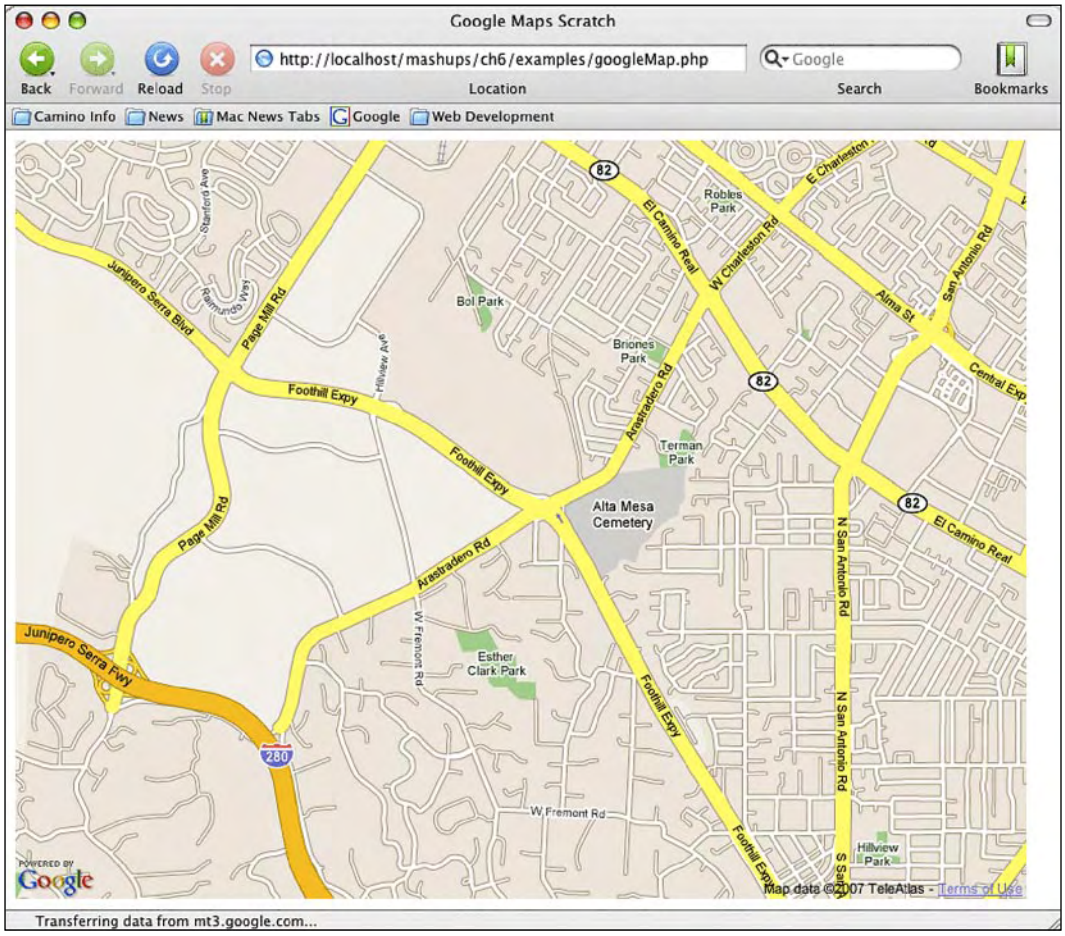
Google's servers pass a `GLatLng` object to the callback function. A `GLatLng` object simply holds latitude and longitude coordinates as properties. If you need to create a `GLatLng` object, there are two parameters you must pass—the latitude and longitude. These properties can be accessed again by using this object's `lat()` and `long()` methods.

A small inconvenience in using `getLatLng()` is that this method doesn't actually return a `GLatLng` object to the caller. However, because one is passed to the callback function, you have to create a callback function in order to use the geocoding results. Going back to our code, we can make a small modification to the JavaScript to make it center on an address.

```
<script type="text/javascript">
   var g_map;
   function load() {
     if (GBrowserIsCompatible()) {
          var  geocoder = new GClientGeocoder();
          g_map = new GMap2(document.getElementById("map"));
          geocoder.getLatLng(
             "780 Arastradero Road, Palo Alto, CA 94306 USA",
              centerMapCallback);
     }
   }
   function centerMapCallback(returnedPoint){
          g_map.setCenter(returnedPoint, 14);
   }
</script>
```

In this modified script, we create a `GClientGeocoder` in the load function. We create the map like before. After that, we call `getLatLng()`, passing an address, and the callback function, `centerMapCallback`.

In `centerMapCallback()`, we catch the `GLatLng` object in the parameter and pass it to the map's `setCenter()` method to do the actual centering. The second parameter, whose value is 14, is the zoom level. When the API calls for a zoom level, you can supply an integer from zero to seventeen. The higher the number, the closer the zoom will be.



We will not be doing any geocoding in this mashup, but you should still familiarize yourself with `GClientGeocoder`. We will be using `GLatLng` quite a bit. Both objects are very important to the Google Maps API. You will find that a mashup often needs both of these objects.

Markers

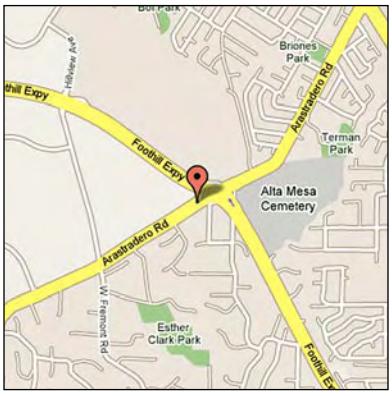
One frequent use of `GLatLng` is that they are parameters for markers. Markers are the pointers Google Maps use to identify a specific place on the map. Each marker is an instance of the `GMarker` class.

To create a basic marker on the map, you only need to do two things: 1) Create the `GMarker` object, and 2) Add it to the map.

In our example, we can add a marker to the address simply by adding two extra lines to do those tasks in our callback function.

```
function centerMapCallback(returnedPoint){
                        var marker = new GMarker(returnedPoint);
                        g_map.setCenter(returnedPoint, 14);
                        g_map.addOverlay(marker);
}
```

The first line instantiates the `GMarker` and places it in a local variable named `marker`. The second line zooms to the map center as before. The third line adds `marker` to the Google Map.



`GMarker` can take a second parameter, a `GMarkerOptions` object. This is an object whose sole purpose is to tweak the marker. Using it, you can do things like add your own customer icons or make the marker draggable. All you have to do is set the properties of the `GMarkerOptions` object.

Consult the `GMarkerOptions` documentation at `http://www.google.com/apis/maps/documentation/reference.html#GMarkerOptions` for everything you can do to markers.

Events

In the Google Maps API Class References documentation, notice that some objects have events associated with them. These objects are things the user sees and can interact with, like the map itself, lines, and markers. This allows you to fire off JavaScript functions whenever the user does something.

Events are managed by the `GEvent` namespace. To register an event, you must add it to the `GEvent` object using the `addListener()` method. `addListener()` takes three parameters. First, it takes the object that you want the event to be active. Second, it takes the kind of event (click, drag, etc.) that is available on the object. Finally, it takes a handler function that fires when the event is triggered.

Let's add an event to our marker. Adding a few more lines to our callback function, we can add an alert box that pops up when our marker is clicked.

```
function centerMapCallback(returnedPoint){
   var marker = new GMarker(returnedPoint);
   g_map.setCenter(returnedPoint, 14);
   g_map.addOverlay(marker);
   GEvent.addListener(marker, "click", function() {
                                          alert("Marker clicked!");
   } );
}
```

`GEvent` is not an object that we create, so we do not need to instantiate it. It is automatically instantiated when we load the Google Maps API. When the click event is triggered on `marker`, the handler function is executed.

InfoWindow Box

An alert box is pretty bland. What's more useful is the white popup box that often appears when using Google Maps. These popup boxes look like comic book speech balloons. They point to a specific location on the map, and contain helpful information about that location. In the Google Maps API, these boxes are known as `InfoWindows`.

`InfoWindows` are represented in the API by the `GInfoWindow` class. The most important thing to know about `InfoWindows` is that for each Google map, there is one and only one `InfoWindow`. This has two implications to us. First, when the `InfoWindow` comes and goes from the user's view, all that is happening is that visibility of `InfoWindow` is being toggled. This is done either through built-in events of the API like, like clicking on the `InfoWindow`'s close window button, or programmatically by the developer, like calling the `InfoWindow`'s `show()` or `hide()` functions.

Second, events just share and update the same `InfoWindow`. When you see an `InfoWindow` take on new content, like what happens when you switch from one marker to another in Google Maps, the `InfoWindow`'s content is being changed through JavaScript DOM methods. We will have to do the same when we use `InfoWindow` boxes in our mashup.

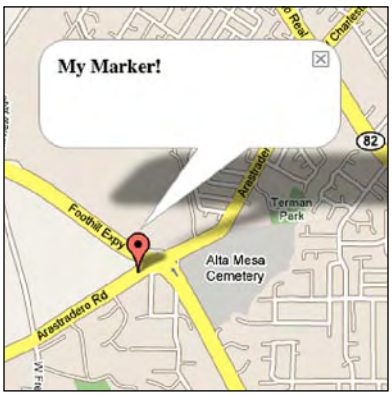
Let's modify our example script further. Instead of getting a JavaScript alert box, let's display an `InfoWindow` box when the user clicks on the marker.

Remember, every map already has an `InfoWindow` box associated with it when you instantiate the map. Therefore, there is no need to create a `GInfoWindow` object. All we have to do is order it to appear in the exact place that we want.

You can set an `InfoWindow` box over a specific point by passing a `GLatLng` object over the point to the `GInfoWindow`'s `reset()` method, then make it appear using the object's `show()` method. However, there is a quicker way to do this. Making the `InfoWindow` box appear over a marker is one of the most common things to do in Google Maps. It's so common, the Google Maps API Team created methods on the `GMarker` object that does just this. The beauty is that the method is on the marker, so it will appear over the marker automatically. You do not have to track down the latitude/longitude of the marker.

We can simply modify the event handler to show the `InfoWindow` instead of an alert.

```
GEvent.addListener(marker, "click", function() {
   marker.openInfoWindowHtml("<div><b>My Marker!</b></div>");
} );
```



`InfoWindow`'s size is the width and height of the largest HTML container inside. Therefore, you can control the size by adding a height and/or width CSS properties to the enclosing container. For example, you can make a roughly 200 pixels by 300 pixels `InfoWindow` by putting a `div` tag that is 200 pixels by 300 pixels like so:

```
.openInfoWindowHtml("<div style=\"width:220px; height:250px;\">
                    Some HTML</div>");
```

Version 2.5 and above of the API also has added support for tabs in the `InfoWindow`. To turn an `InfoWindow` into tabs, create a `GInfoWindowTab` for each tab. This class's constructor takes two parameters. The first is the label of the tab, the second is the content. Place all of these `GInfoWindowTab` objects in a JavaScript array. The `GMarker` class also has support for a method named `openInfoWindowTabs()`. This method takes an array of `GInfoWindowTab` objects. Calling it will open an `InfoWindow`, but the window will be in a tab interface, with the objects as the content.

Our callback function can be tweaked a bit to use tabs in the `InfoWindow`:

```
function centerMapCallback(returnedPoint){
 var tabsArray = new Array();
 tabsArray[0] = new GInfoWindowTab("One", "<p>Content for Tab 1</p>");
 tabsArray[1] = new GInfoWindowTab("Two", "<p>Content for Tab 2</p>");

 var marker = new GMarker(returnedPoint);
 g_map.setCenter(returnedPoint, 14);
 g_map.addOverlay(marker);
 GEvent.addListener(marker, "click", function() {
     marker.openInfoWindowTabs(tabsArray);
 } );
}
```

This concludes the basic features of Google Maps. There are plenty of other features available. Some of the powerful features include:

- The ability to draw lines on the map, similar to when Google Maps gives directions.
- A REST interface for the service returning XML, allowing you to use the Google Maps database on server-side applications.
- A Marker Manager to handle large amounts of markers at different zoom levels.
- Override the map tiles from Google Maps using the `GMapTiles` object.

If you use Google Maps API heavily in mashups, you should also be aware of the many options objects available to you. They give you the flexibility to go beyond many other mashups that use the API. For example, with the `GMarkerOptions` object, you can create custom markers on your map.

Even without these advanced features, you will be able to do a lot with Google Maps. We certainly have more than enough to create our mashup.

Flickr Services API

Flickr, focusing on photo sharing, is one of the oldest community-driven sites out there. They were also an early adopter of web APIs for third party developers. These things have given them a large user base and a very rich API. Flickr Services is probably the most flexible web API we have seen. The API home page is located at `http://www.flickr.com/services/api/`. You will need a free developer key to use this API. As Flickr! is a subsidiary of Yahoo!, you will also need a free Yahoo! account. You will be prompted for both at `http://www.flickr.com/services/api/keys/`. From there, you can also sign up for both.

Like the other APIs from social-sharing sites we have seen, Flickr Services' API focuses not only on their subject matter, but also has many methods that deal with community features. There are an abundant group of methods that allow you to query information about Flickr's community. Assuming someone has allowed it on their privacy settings, you can get a person's blog entries and favourite photos, among other things. There is also an API dealing with Flickr Group's information. They allow you to find photos and information from people with a similar interest.

Certainly, the two largest groups of methods have to do with photos and photosets. A user can arrange their photos into photosets for organizational purposes. Flickr, like Last.fm and YouTube, relies heavily on user tags. Their photo search is influenced by what is tagged by people.

Probably the most impressive thing about Flickr Services is the choices you have in request and response formats. For request, you can use any of the three most popular formats—REST, SOAP, and XML-RPC. For responses, you can choose Flickr's own XML schema, SOAP, XML-RPC, JSON, or even serialized PHP objects. Regardless of the format you choose for request and responses, Flickr Services has a consistent method of doing things. All requests take the same parameters and return the same data. You just need to format and parse differently for each one.

Executing a Search

Because Flickr Services is so consistent, the best way to get an overview of it is to walk through an example. In our mashup, we will need to concentrate on the group of photo methods. In particular, we need one to search photos based on user tags. Let's try and execute a search like we will be doing for our mashup.

For our request and response, we'll look to keep things simple. We will send the service request using REST. Our web application is PHP driven, so a serialized PHP response would be intriguing. However, as JavaScript will be doing a lot of the work, we will use JSON. The straight XML response from a REST call would also be acceptable, but it would be nice to avoid the DOM parsing that would be required with it.

The method names are fairly self-explanatory and give us a lot of clues on what the method does. Looking at the documentation for the method `flickr.photos.search` at `http://www.flickr.com/services/api/flickr.photos.search.html`, we see it is exactly what we need to search photos.

The URL for all Flickr REST requests is `http://api.flickr.com/services/rest/`. Following this URL are the parameters of the method in a `GET` request format. There are two required parameters for all REST requests—`method` and `api_key`. The value of `method` is the name of the method that you wish to call. The value of `api_key` is your Flickr API Key. To call `flickr.photos.search`, our complete URL would be:

```
http://api.flickr.com/services/rest/?method=flickr.photos.
search&api_key=YOUR_FLICKR_API_KEY
```

A methods documentation page lists all the parameters the method can take. `flickr.photos.search`s' available parameters are quite extensive. This gives us a lot of ability to tweak our search. According to the documentation, the only required parameter is `api_key`. However, this is sort of misleading because we also need to supply a search term. We can search tags using the `tags` parameter, or a free text search using the `text` parameter. Even though both are optional parameters, we need to include one or the other. Otherwise, Flickr will return a message saying that empty searches are not supported.

To use tags, supply a comma delimited list of terms you wish to search. A text search is just a free text string. Either way, when using REST, remember to URL encode your terms.

```
http://api.flickr.com/services/rest/?method=flickr.photos.
search&api_key=YOUR FLICKR_API_KEY&text=fender%20stratocaster
```

If you use XML-RPC or SOAP, use the exact same parameters as listed in the documentation and format the parameters and values as required by the respective format. For SOAP, the endpoint is at `http://api.flickr.com/services/soap/`. For XML-RPC, the service endpoint is at `http://api.flickr.com/services/xmlrpc/`.

Interpreting Service Results

If you hit the above URL in a web browser, after adding your API key, the search will execute and you will receive a live response from the server.

```
<?xml version="1.0" encoding="utf-8" ?>
<rsp stat="ok">
<photos page=»1» pages=»20» perpage=»100» total=»1904»>
   <photo id="412962278" owner="43203076@N00" secret="63e7e2e1f0"
    server="183" farm="1" title="Doin' Studio Time" ispublic="1"
    isfriend="0" isfamily="0" />
   <photo id="412463850" owner="63895350@N00" secret="26b97edbb5"
    server="172" farm="1" title="Norby with his Fender Stratocaster"
    ispublic="1" isfriend="0" isfamily="0" />
   <photo id="411598583" owner="75859527@N00" secret="657eb806c8"
    server="172" farm="1" title="Hocus Pocus" ispublic="1"
    isfriend="0" isfamily="0" />
    …
</photos>
</rsp>
```

The returned format is in a standard format returned by Flickr whenever it returns photos. By default, a call returns 100 results per "page". The `photos` element groups individual `photo` elements in a "page". Each `photo` element represents a photo returned in the search results. You can change the page you are on by passing a `page` parameter to the call. Alternatively, you can also change the number of photos returned in a page with the `per_page` parameter in the call.

Each `photo` element is basically a collection of attributes about the photo. These attributes are very important. We need to know them in order to load the photo.

Attribute	Description
Id	Unique ID of the photo.
Owner	Owner ID of the person that owns this picture.
Secret	A secondary identifier used to help identify the photo.
Server	The server on which this photo is stored.
Farm	The server farm on which this photo is stored.
Title	The title of the picture.
isPublic	Boolean indicating whether the owner is publicly sharing the photo.
isFriend	Boolean indicating whether the owner is on your list of friends.
isFamily	Boolean indicating whether the owner is on your list of family members.

The last three booleans take either a 1 or 0 value. They also require the service caller to be authenticated in using the authentication methods in the API.

This is what we want, but it is in the wrong format. We want the results back in JSON. To get results in JSON, we need to pass a format parameter to the service call. In this case, the value of that parameter is `json`.

```
http://api.flickr.com/services/rest/?method=flickr.photos.search&api_
key=YOUR FLICKR_API_KEY&text=fender%20stratocaster&format=json
```

Adding the parameter will give us this response from the server.

```
jsonFlickrApi({
   "photos": {
       "page":1,
       "pages":20,
       "perpage":100,
       "total":"1904",
       "photo":[
       {«id»:»412962278», «owner»:»43203076@N00»,
       «secret»:»63e7e2e1f0», «server»:»183», «farm»:1,
       «title»:»Doin\u2019 Studio Time», «ispublic»:1, «isfriend»:0,
       «isfamily»:0},
       {«id»:»412463850», «owner»:»63895350@N00»,
       «secret»:»26b97edbb5», «server»:»172», «farm»:1, «title»:
       »Norby with his Fender Stratocaster», «ispublic»:1,
       «isfriend»:0, «isfamily»:0},
```

```
        {«id»:»411598583», «owner»:»75859527@N00»,
        «secret»:»657eb806c8», «server»:»172», «farm»:1, «title»:»Hocus
        Pocus», «ispublic»:1, «isfriend»:0, «isfamily»:0},
        …
        ]
    }
})
```

Each method's documentation page documents the returned XML format of the call. From there, it is easy to take an educated guess at the JSON equivalent. Generally, element attributes in the XML document are object properties in the JSON document. Nested elements are translated into nested objects. The subject of search results, whether they are things like blog entries, users in a group, or like in this case, photos, are returned as JSON arrays. If you have trouble estimating the exact translation of a method, you can always manually make the request in your browser like we did here.

Note that the JSON results are encapsulated in a call to `jsonFlickrApi`. By default, the API assumes that you want to pass the JSON results to a JavaScript callback function. If you have a function named `jsonFlickrApi` in your application, the JavaScript engine will pass the JSON object to that function when it receives the response. The engine will then automatically execute the function. This can be a controller in your JavaScript for the service's return value. However, you do need to create a function named `jsonFlickrApi`, and it must be set-up to act on the returned JSON code. If you choose not to use this, you can turn this automatic callback off by sending a true (`1`) value to the `nojsoncallback` parameter in your call. This will give the exact same text string without the `jsonFlickrApi()`.

```
http://api.flickr.com/services/rest/?method=flickr.photos.search&api_
key=YOUR_FLICKR_API_KEY&text=fender%20stratocaster&format=json&nojson
callback=1
```

Retrieving a Photo or a Photo's Page

Now that we have the results, we can use the data to retrieve photos from Flickr. Image URLs in Flickr have the following format:

```
http://farm{FARM-ID}.static.flickr.com/{SERVER-ID}/{ID}_
{SECRET}{SIZE}.jpg
```

With the exception of the size, all the other variables can be extracted directly from `flickr.photos.search`'s web service call response.

The `FARM-ID` is the farm attribute. `SERVER-ID` is the server attribute. ID is the id attribute. `SECRET` is the secret attribute in the `XML`. `SIZE` is the size of the photo you want. It is an underscore followed by one character. The character can take on any of the following letters:

Suffix	Meaning	Max Pixels on Side
_o	Original size	*
_b	Large	1024
None	Medium	500
_m	Small	240
_t	Thumbnail	100
_s	Small Square	75 px x 75 px

One of the first photo's XML is returned as:

```
<photo id="411598583" owner="75859527@N00" secret="657eb806c8"
server="172" farm="1" title="Hocus Pocus" ispublic="1" isfriend="0"
isfamily="0" />
```

We can use this information to construct a URL to a small version of the photo:

```
http://farm1.static.flickr.com/172/411598583_657eb806c8_m.jpg
```

Original size works a little differently. They have their own secret code in an attribute named `originalsecret` and you must include the file type extension, which you can get from another attribute named `original_format`. To get these attributes, you need to request them in your original request in the extras parameter. This parameter takes a comma-delimited list of attributes that may not be included in the default response.

```
http://api.flickr.com/services/rest/?method=flickr.photos.
search&api_key=YOUR_FLICKR_API_KEY&text=fender%20stratocaster&form
at=json&nojsoncallback=1&extras=originalsecret,original_format
```

Consult a method's documentation to see if any extra parameters are available.

A URL to the photo's web page works in a similar way. The URL takes the following format:

```
http://www.flickr.com/photos/{USER-ID}/{PHOTO-ID}
```

The documentation outlines several different things that you can link to, for example, you can construct URLs to a photoset or a user's profile.

Mashing Up

We have toured a lot of technologies for this mashup. Some of these are pretty cutting-edge, but necessary to incorporate a relatively new specification. Not surprisingly, your data sources are not always going to be from web APIs. Staying flexible and searching for new technologies to use in your applications is important. At last, we have the knowledge to start building the application.

The database is a good place to begin. Recall from our sequence diagram that a visitor directly and indirectly interacts with several different components of our application at any one time. Many of the components rely on the Google Map to be built first, but the map relies on the database as a source for marker locations.

Building and Populating the Database

Our mashup needs three things: Tube stations, lines of the Tube system, and which stations belong to which line. We also need to keep in mind that a station can belong to more than one line. As our source of data is from the Tube Station RDF document, let's take a close look at the document to see what's available to us.

Examining the File

The first half of the page consists of stations. A typical station looks like this:

```
<rdf:Description rdf:about="http://london.openguides.org/index.
cgi?id=Acton_Town_Station;format=rdf#obj">
<os:y>179613</os:y>
<dc:subject>Tube</dc:subject>
<name>Acton Town Station</name>
<dc:title>Acton Town Station</dc:title>
<rdfs:type rdf:resource="http://www.w3.org/2003/01/geo/wgs84_
pos#SpatialThing"/>
<geo:long>-0.280009</geo:long>
<space:connects rdf:resource="http://london.openguides.org/index.
cgi?id=Turnham_Green_Station;format=rdf#obj"/>
<os:x>519478</os:x>
<rdfs:seeAlso rdf:resource=»http://london.openguides.org/index.
cgi?id=Acton_Town_Station;format=rdf#obj»/>
<geo:lat>51.502833</geo:lat>
</rdf:Description>
```

We need at least a name and a latitude/longitude pair for Google Maps. The `name`, `geo:long`, and `geo:lat` elements appear to give this to us. We will definitely need to extract these. Putting this "thing" in a subject/predicate/object context, the `rdf:about` attribute would give us the subject. Should the need arise, we can use that as a unique identifier. We also see there is a `type/resource` element that may identify this item as a tube station; this may also be useful.

Nowhere in this document do we find an actual list of lines. However, the last half of this document is interesting. They are a collection of blocks, but smaller than a station block.

```
<rdf:Description rdf:about="http://space.frot.org/a_space/id5276761">
<rdfs:type rdf:resource="http://space.frot.org/rdf/space.owl#Tube_
Line"/>
<rdf:predicate rdf:resource="http://frot.org/space/0.1/connects"/>
<rdf:subject rdf:resource="http://london.openguides.org/index.
cgi?id=North_Ealing_Station;format=rdf#obj"/>
<dc:title>Piccadilly Line</dc:title>
<rdf:object rdf:resource="http://london.openguides.org/index.
cgi?id=Ealing_Common_Station;format=rdf#obj"/>
</rdf:Description>
```

They appear to be a list of spatial relationships described in a triple format. The rdfs/resource pair tells us it is a tube line. However, there are many of these in each line. What gives this away are the `rdf:predicate`, `rdf:subject`, and `rdf:object` tags. These items tells us that in this line, the subject, which directly correlates to the `rdf:about` attributes of the stations, connects (according to `rdf:predicate`) to the object, which also directly correlates to the `rdf:about` attributes. Basically, these items tell us that the subject station connects to the object station in a certain line. They are drawing the line map for us using a triple.

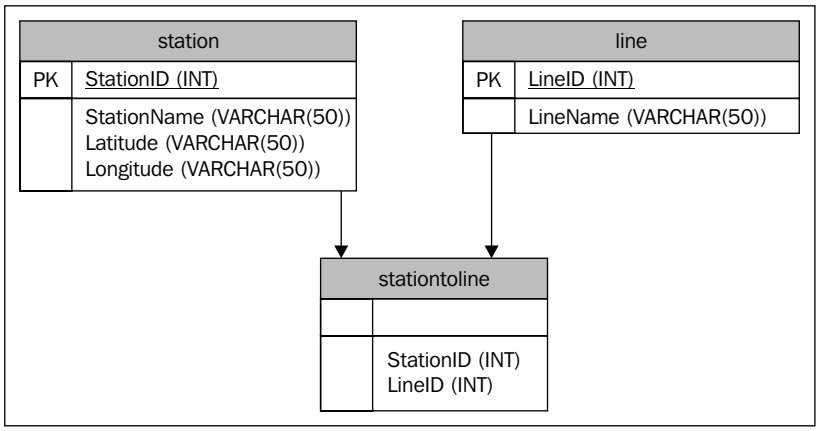
Therefore, we can simply pick these out to get the line stations. As `rdf:subject` elements are the start of the connection chain, we can just pick out the `rdf:subject` and filter by `dc:title` to get all of the stations in a line.

This is the only hint of the presence of Tube lines. However, all we really need to do is extract the name of the line and the stations to which they belong.

Creating Our Database Schema

A line has many stations and a station can belong to more than one line. This sounds like a job for a join table. We'll keep things simple and just extract the name, latitude, and longitude for the stations, and just the line name for the line.

Our database schema will look like this:



We have included an SQL file in the examples code named `londontube.sql`. This file will create a database with foreign key constraints. You can run this file directly in an SQL import tool, like the MySQL command line, or phpMyAdmin, to create this database. For all other RDMS setups, create a database named londontube and mimic the schema.

Don't forget to give at least `SELECT` and `INSERT` permissions for a user on this database, and a password!

Building SPARQL Queries

To populate these tables from RDF, we will need a SPARQL query for each one. First, we will need to populate all stations. Second, we will need a SPARQL query to populate all the lines. After we insert lines and stations, we need to use the SQL IDs that were generated and insert them into the stationtoline junction table.

As we create these, we can double check our work back at SPARQLer. Be sure to change the Data URL field to the London Tube RDF at `http://space.frot.org/rdf/tube_model2.rdf`.

Stations Query

Our stations query must extract the name, latitude, and longitude from the RDF document. We can do this with the following query:

```
PREFIX : <http://xmlns.com/foaf/0.1/>
PREFIX geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT DISTINCT ?stationName ?lat ?long
FROM <tube_model2.rdf>
WHERE {
       ?type rdfs:type
       <http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing> .
       ?type :name ?stationName .
       ?type geo:lat ?lat .
       ?type geo:long ?long
}
ORDER BY ?stationName
```

In the top, we define the prefixes we will need. Note the very first prefix. The name of the station is in the `name` element, which does not have a prefix. It falls into the default namespace. You have to declare default namespace prefixes if they are used in SPARQL. To do this, create the `PREFIX` statement as you normally would, but the namespace portion is just an empty colon.

The `SELECT` statement tells the parser to grab three variables, `?stationName`, `?lat`, and `?long`. The `WHERE` clause refines the search and sets those variables.

The first triple narrows the search to stations. Remember when we looked at stations in the RDF document, it had a `type/resource` pair that identifies it as a station? The type was in the `rdfs` namespace, but its `resource` attribute was in the `rdf` namespace. Even though we do not explicitly use the `rdf` namespace in this first `WHERE` clause, the value is in that namespace, so therefore we also need to give it a `PREFIX` declaration. This statement sets the subject for our other clauses in the variable named `?type`.

The three other triples set the variables we asked for in the `SELECT` statement. They essentially work the same way. They use the subject in `?type` to find the predicate, which are the elements we want. The object of these triples is placed into the `?stationName`, `?lat`, and `?long` variables.

Lines Query

This one is easier than it may first appear. Our lines query must get all of the lines in the system. However, the RDF file does not have a section of just lines. It does have the section where it describes all of the connections in a line, though. We can simply grab all of these connection items and use the `DISTINCT` keyword on the line name to make sure we only get one of each.

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
FROM <tube_model2.rdf>
SELECT DISTINCT ?lineName
WHERE {
   ?type rdfs:type <http://space.frot.org/rdf/space.owl#Tube_Line> .
   ?type dc:title ?lineName
}
ORDER BY ?lineName';
```

The `WHERE` clause by itself, gets all of the line names from the `dc:title` element based on a type/resource combination like the previous query. However, the `DISTINCT` keyword filters out all the repeat instances.

Lines to Stations Query

Remember previously that RDF items do not have relationships like SQL does *per se*. We can work around this by using queries to find the subject of the child object. We will have to do this to map the relationship between lines and stations.

```
PREFIX : <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT DISTINCT ?lineName ?stationName
FROM <tube_model2.rdf>
WHERE {
   ?line rdfs:type <http://space.frot.org/rdf/space.owl#Tube_Line> .
   ?line dc:title ?lineName .
   ?line rdf:subject ?infourl .
   ?infourl dc:title ?stationName
}
ORDER BY ?lineName ?stationName';
```

This query is asking for a line name and a station name pairing. All stations that belong to a line should be included with that line. If a line has twelve stations, there should be twelve entries with that line in the result set, with each station having on entry in that pair.

The first line in the `WHERE` clause is simple enough. It sets the subject of all Tube lines in the `?line` variable. The second line sets the `?lineName` variable, which we want to extract, by making it the object of the `dc:title` predicate. It gets interesting in the third line.

In the RDF document, these connection items have a subject element.

```
<rdf:subject rdf:resource="http://london.openguides.org/index.
cgi?id=Wapping_Station;format=rdf#obj"/>
```

These subject elements tell us that subject of this connection item is the resource attribute value. The third line in the `WHERE` clause, then, sets the `rdf:resource` value in a variable named `?infourl`. Remember earlier when we looked at the stations we noted the `rdf:about` attribute in the `Description` elements of the stations could be used as a unique identifier for those stations? This is where it comes in handy. These identifiers are used in `rdf:resource` in these connection items.

In the fourth line, we use this station identifier URL as the subject to grab the station name. This fourth line looks for all subjects with the unique station URL and operates on those items. In other words, it looks back at the station items in the first half of the page.

Finally, back in the `SELECT` statement, we add a `DISTINCT` keyword to `?lineName`. This is because a connection between two stations is actually represented twice in our document. You'll find a statement that says, "Station A is connected to Station B", and later on in the document, you'll find "Station B connects to Station A". This is no accident, but will cause each connection to be listed twice. `DISTINCT` will eliminate that.

We have successfully worked around the issue of relationships. Though your classic foreign key constraints in SQL are not available, we do have identifiers in this file that we can play with. Fortunately, we have a well designed document, but this may not always be the case. You may have to query more than one document, or you may have to get extra complicated with your SPARQL `WHERE` clauses.

Database Population Script

Now that we have our SPARQL queries, it's time to actually use them to populate our database. We will write a procedural script that uses RDF API for PHP to do just that.

As RAP is objected oriented, we'll use a model-centric approach for this script. In the example chapter code, this section will go over the code in the script named `populateDB.php`. In the `classes/models` directory, there are two files, `clsLine.php` and `clsStation.php`. They represent the line table in the database and the station table. They are just containers. Each column in the database is represented by properties in the class, and each property has a public getter and setter method to access it.

The `clsLine.php` file looks like this:

```
<?php
  class Line {
     private $lineId;
     private $lineName;
     public function getLineId() { return $this->lineId; }
     public function getLineName() { return $this->lineName; }
     public function setLineId($i) { $this->lineId = $i; }
     public function setLineName($n) { $this->lineName = $n; }
  }
?>
```

`clsStation.php` looks like this:

```
<?php
  class Station {
     private $stationId;
     private $stationName;
     private $lat;
     private $long;
     public function getStationId() { return $this->stationId; }
     public function getStationName() { return $this->stationName; }
     public function getLat() { return $this->lat; }
     public function getLong() { return $this->long; }
     public function setStationId($i) { $this->stationId = $i; }
     public function setStationName($n) { $this->stationName = $n; }
     public function setLat($l) { $this->lat = $l; }
     public function setLong($l) { $this->long = $l; }
  }
?>
```

These "plain old PHP objects" are generic enough to reuse later in our application.

Based on our database schema, our `populateDB.php` needs to take the following steps:

1. Get all lines from the RDF file.
2. Insert all the lines into the table.
3. Remember the table primary key that was generated by the insert.
4. Get all stations from the RDF file.
5. Insert all stations into the table.
6. Remember the table primary key that was generated by the insert.
7. Get all stations in a line from the RDF file.
8. Use the primary keys there were generated from the inserts and insert them correctly into the stations-to-line junction table based on the query from the RDF file.

Our script starts off with the standard initialization and preparation code that RAP requires. In addition, we include the two model object definitions.

```
define("RDFAPI_INCLUDE_DIR", "Absolute/Path/To/rdfapi-php/api/");
require_once(RDFAPI_INCLUDE_DIR . "RdfAPI.php");
require_once('classes/models/clsLine.php');
require_once('classes/models/clsStation.php');
```

Next, the SPARQL client is created. We pass the URL to the tube document into the `load()` method.

```
//Create SPARQL Client
$sparqlClient = ModelFactory::getDefaultModel();
$sparqlClient->load('http://space.frot.org/rdf/tube_model2.rdf');
```

We need to create a database connection. Modify this section as necessary if you are not using MySQL, and customize it to the user.

```
//Create MySQL Client
$mySQLConn = @mysql_connect("127.0.0.1", "DB USER NAME", "DB USER
PASSWORD")  or die("Couldn't connect to the MySQL server.");
$db = mysql_select_db("londontube", $mySQLConn) or die("Couldn't
connect to the londontube database.");
```

Now it's time to create some functions that will query the RDF document.

The first is the `getAllStations()` function. This function will query the RDF document and return an array of station objects.

```
function getAllStations(&$sparqlClient) {
   $returnArray = array();
   $query = '
   PREFIX : <http://xmlns.com/foaf/0.1/>
   PREFIX geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>
   PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
   PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
   SELECT DISTINCT ?stationName ?lat ?long
   FROM <tube_model2.rdf>
   WHERE {
          ?type rdfs:type
          <http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing> .
          ?type :name ?stationName .
          ?type geo:lat ?lat .
          ?type geo:long ?long
   }
   ORDER BY ?stationName';
   $result = $sparqlClient->sparqlQuery($query);
   if ($result != "false") {
       foreach ($result as $station) {
          if ($station != "") {
             $stationObj = new Station();
             $stationObj->setStationName($station['?stationName']-
             >getLabel());
             $stationObj->setLat($station['?lat']->getLabel());
             $stationObj->setLong($station['?long']->getLabel());
             $returnArray[$station['?stationName']->getLabel()] =
             $stationObj;
           }
       }
   }
   return $returnArray;
}
```

This function starts off with the SPARQL query that we built earlier and uses the SPARQL client passed to it to execute it against the loaded RDF document. Remember that the query gets the name, latitude, and longitude. The results set comprises a row for each station. The `foreach` loops through this results set. It

places each results object into a RAP resource object named `$station`. For each station in the results set, a new station is instantiated. Using the setter methods, the results in `$station` populate each station object's name, latitude, and longitude. It then places this object into the array to be returned, with the name of the station as the key. Without any integer identifiers in RDF, we are going to have to use the next best thing. The names of the stations and lines are going to have to be the keys.

The same principle applies to `getAllLines()`, which grabs all of the station lines in the RDF document.

```
function getAllLines(&$sparqlClient) {
   $returnArray = array();
   $query = '
   PREFIX dc: <http://purl.org/dc/elements/1.1/>
   PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
   FROM <tube_model2.rdf>
   SELECT DISTINCT ?lineName
   WHERE {
          ?type rdfs:type
                   <http://space.frot.org/rdf/space.owl#Tube_Line> .
           ?type dc:title ?lineName
   }
   ORDER BY ?lineName';
   $result = $sparqlClient->sparqlQuery($query);
   if ($result != "false") {
      foreach ($result as $line) {
          if ($line != "") {
             $lineObj = new Line();
             $lineObj->setLineName($line['?lineName']->getLabel());
             $returnArray[$line['?lineName']->getLabel()] = $lineObj;
          }
      }
   }
   return $returnArray;
}
```

The same principle applies to `getAllLines()`, which grabs all of the station lines in the RDF document. Again, the line name is the key in this array.

Lastly, we create a function that finds the station-to-line relationships.

```
function getLinesAndStations(&$sparqlClient) {
   $returnArray = array();
   $i = 0;
   $query = '
   PREFIX : <http://xmlns.com/foaf/0.1/>
   PREFIX dc: <http://purl.org/dc/elements/1.1/>
   PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
   PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
   SELECT DISTINCT ?lineName ?stationName
   FROM <tube_model2.rdf>
   WHERE {
      ?line rdfs:type <http://space.frot.org/rdf/space.owl#Tube_Line>
      ?line dc:title ?lineName .
      ?line rdf:subject ?infourl .
      ?infourl dc:title ?stationName
   }
   ORDER BY ?lineName ?stationName';
   $result = $sparqlClient->sparqlQuery($query);
   if ($result != "false") {
      foreach ($result as $relationship) {
          if ($relationship != "") {
              $returnArray[$i]['line'] = $relationship['?lineName']-
              >getLabel();
              $returnArray[$i]['station'] = $relationship[
             '?stationName']->getLabel();
             $i++;
           }
       }
   }
   return $returnArray;
}
```

This array starts off, like the other two, by using SPARQL to query the loaded RDF document. However, the returned array is different from the other two. We did not create any model objects to hold relationships, so nothing like that is used. Instead, we return a multi-dimensional array. An integer is the index, and each value is an associative array inside it. The associative array has the line and station name grouping together.

Now we have three functions that return three arrays. Let's call them and start working on the arrays.

```
$linesArr = getAllLines($sparqlClient);
$stationsArr = getAllStations($sparqlClient);
$joinArr = getLinesAndStations($sparqlClient);
```

This block will store the arrays in `$linesArr`, `$stationsArr`, and `$joinArr`. First, we will operate on the `$linesArr` array.

```
foreach ($linesArr as $line) {
   $sql = 'INSERT INTO line (LineName) VALUES (\'' . addslashes(
          $line->getLineName()) . '\')';
   $e = mysql_query($sql, $mySQLConn);
   $line->setLineId(mysql_insert_id($mySQLConn));
                              }
```

This `foreach` loop will insert each Line object in the `$linesArr` array into the database. The last statement in the code will get the new ID number from the insert and store it in the object property. Another `foreach` loop does the same thing with the stations.

```
foreach ($stationsArr as $station) {
   $sql = 'INSERT INTO station (StationName, Latitude, Longitude)
   VALUES (\'' . addslashes($station->getStationName()) . '\', \''
   . addslashes($station->getLat()) . '\', \'' . addslashes($station-
   >getLong()) . '\')';
   $e = mysql_query($sql, $mySQLConn);
   $station->setStationId(mysql_insert_id($mySQLConn));
                                    }
```

After this is done, we still have our arrays of lines and stations. Now, however, each object's ID property is set with the primary key number assigned from the database. We need to use this property when we populate the join table.

```
foreach ($joinArr as $key => $value) {
   $sql = 'INSERT INTO stationtoline (LineID, StationID) VALUES (' .
   $lines[$value['line']]->getLineId() . ', ' . $stations[$value[
   'station']]->getStationId() . ')';
   $e = mysql_query($sql, $mySQLConn);
}
```

Remember that `$join` is a multivariable array, and `'line'` is the key in the associative array that has the line name, and `'station'` is the key with the station name. We use these keys to grab the object in `$linesArr` and `$stationsArr`. Once

we have these objects, it's just a matter of using the ID getter method to grab the database primary key ID for that station or line. These are used in the SQL statement for the insert.

Run this file once in your web browser and you will have a fully populated database full of London Tube station information. It's time to create the web front end to our mashup.

The TubeSource Database Interface Class

This mashup will always have a pull-down menu of all stations. Once the user selects a line, the page will refresh itself and the line's stations will be marked with markers. This implies two things:

1. We need a function to pull the names of the Tube lines from the database.
2. We need a function to pull the station names from the database based on lines.

We'll create a database interface class for this. It will be the source of all Tube information from the database. In the examples, this file is in the `classes` directory and named `clsTubeSource.php`. Anything that interfaces with the database will occur in this class.

```
class TubeSource {
   private $dbConn;
   public function getAllLines() {
       $returnArray = array();
       $sql = 'SELECT LineID, LineName FROM line';
       $e = mysql_query($sql, $this->dbConn);
       while ($row = mysql_fetch_array($e)) {
          $lineObj = new Line();
          $lineObj->setLineId($row['LineID']);
          $lineObj->setLineName($row['LineName']);
          array_push($returnArray, $lineObj);
       }
       return $returnArray;
   }
   public function getStationsByLine($lineid) {
       $returnArray = array();
       $sql = 'SELECT S.StationName, S.Latitude, S.Longitude FROM
               stationtoline AS SL
```

```
                INNER JOIN station AS S
                ON SL.StationID = S.StationID
                WHERE SL.LineID = ' . $lineid;
            $e = mysql_query($sql, $this->dbConn);
            while ($row = mysql_fetch_array($e)) {
               $stationObj = new Station();
               $stationObj->setStationName($row['StationName']);
               $stationObj->setLat($row['Latitude']);
               $stationObj->setLong($row['Longitude']);
               array_push($returnArray, $stationObj);
            }
            return $returnArray;
        }
        public function __construct(&$dbConn) {
            $this->dbConn = $dbConn;
        }
    }
    ?>
```

This class takes a database connection object in its constructor. Its two methods, `getAllLines()` and `getAllStationsByLine()`, return arrays of Line objects and Station objects, respectively. They work with and populate the model classes in a similar fashion as the SPARQL queries did. `getAllStationsByLine()` takes the primary key ID of the line as a parameter, and uses it in the `WHERE` clause.

The Main User Interface

At this point, we can create the main user interface page to see how our mashup is progressing. Let's create the functionality to draw a Google Map and draw the markers when a user selects a line. This page needs to do the following:

1. Create and display Google Map.
2. Contain the JavaScript to display the station markers.
3. Call the `TubeSource` database class.
4. Present the user with a pull-down menu of stations populated with data from `TubeSource`.

This basic form of the home page is named `index-Basic.php.` We'll walk through the portions of the page that handle all of the listed functionality. Later, we will modify the page to add the Flickr calls to get the photos.

```
<?php
$googleKey = 'YOUR GOOGLE API KEY';
require_once('classes/models/clsLine.php');
require_once('classes/models/clsStation.php');
require_once('classes/clsTubeSource.php');
```

This page starts with some preliminary initialization. The Google API key is set in a variable. All of our model classes are included as well as the `TubeSource` class.

```
//Create MySQL Client
  $mySQLConn = @mysql_connect("127.0.0.1", "tubeapp", "tubular")  or
  die("Couldn't connect to the MySQL server.");
  $db = mysql_select_db("londontube", $mySQLConn) or die("Couldn't
  connect to the londontube database.");
//Create a DB abstrction object
  $tubeSourceObj = new TubeSource($mySQLConn);
```

We need to create the database code. Here, the database client is created and `TubeSource` is instantiated with the client.

```
$linesArr = $tubeSourceObj->getAllLines();
  if ($_GET['line']) {
     $stationsArr = $tubeSourceObj->getStationsByLine($_GET['line']);
  }
?>
```

The next few lines end the preliminary PHP code. The first makes a call to `TubeSource`'s `getAllLines()` to get all the lines. The returned array of Line objects, in `$linesArr`, will be used to created the pull-down menu.

If a `GET` parameter was passed to this page, we'll make a call to `TubeSource`'s other method, `getStationsByLine()`. This will get us the Station objects of a line stored in an array.

Next, we start our HTML and JavaScript.

```
<html>
  <head>
    <title>London Tube Stations</title>
```

```
<script src="http://maps.google.com/maps?file=api&amp;v=2&amp;
key=<?= $googleKey ?>"
   type="text/javascript"></script>
 <script type="text/javascript">
var g_map;
```

The JavaScript starts off with a declaration of a few global variables to hold information throughout the application.

```
function load() {
   if (GBrowserIsCompatible()) {
       var point = null;
       g_map = new GMap2(document.getElementById("map"));
```

The load function will be executed by the body onload event. The purpose of this function is to create the Google Map and draw any markers if needed. This loads the map into the `g_map` global variable.

```
       g_map.addControl(new GSmallMapControl());
       g_map.addControl(new GMapTypeControl());
       g_map.setCenter(new GLatLng(51.5099983215,
       -0.134690001607), 11);
```

These three lines operate on our map. The first two add some controls. There are a whole series of controls you can add to a Google Map. The first line adds a small version of the pan and zoom commands you see on Google Maps. The second line adds Map Type Control buttons to the upper right corner of the map. These buttons control whether the map is a typical street map, a satellite map, or a hybrid.

The third line centres the map to a location. Through research, trial, and error, I found the latitude and longitude of downtown London. We pass the coordinates to a `GLatLng` object, set a nice zoom level of 11 to most of London, and pass that to the `setCenter()` method.

```
<?php if ($_GET['line'] && count($stationsArr) > 0) {
     foreach ($stationsArr as $station) { ?>
          point = new GLatLng(<?= $station->getLat() ?>,
          <?= $station->getLong() ?>);
          g_map.addOverlay(createMarker(point,
          '<?= addslashes($station->getStationName()) ?>'));
<?php } } ?>

      }
}
```

This section creates the markers. We use PHP to help us. If a line `GET` parameter was passed to the page and the array of stations is not empty, then we need to create a marker for each station. Still in PHP, we loop through using a `foreach` loop. A `GLatLng` object, represented by `point`, is created with the PHP object's latitude and longitude properties. If we just use this point and pass it to the map's `addOverlay` method, we would create a marker on the map. However, we want to do a little extra with it, like create an event.

We use this `point` and pass it to another function, `createMarker()`. This function creates a marker, adds an event listener to it, then returns the same marker.

```
// Creates a marker at the given point with the given number label
   function createMarker(point, stationName) {
    var marker = new GMarker(point);
    GEvent.addListener(marker, "click", function() {
       marker.openInfoWindowHtml("<div style=\"width:220px;
       height:250px;\"><b>" + stationName + "</b></div>");
                                                    });
    return marker;
}
```

A marker is created in the first line of the function. Remember that the `GEvent` object is created when you call the Google Map. Its job is to watch for events on all Google Map objects. We tell it to listen for a click on this marker through the `addListener()` method.

In the callback function parameter, we define what's going to happen when the marker is clicked. Here, we tell the map to open the `InfoWindow` using `openInfoWindowHtml()`. We provide HTML as the parameter using the station name. When opened, the `InfoWindow` will appear over the marker. The name of the station will be the only content in the window.

```
    </script>
</head>
<body onload="load()" onunload="GUnload()">
```

In our body tag, we initiate map creation by calling `load()`. We also add a call to `GUnload()` when the page is exited. `GUnload()` is part of the Google Maps API. Its job is to close up any memory leaks. It is always a good idea to call this at an `onunload` page event whenever you are using Google Maps.

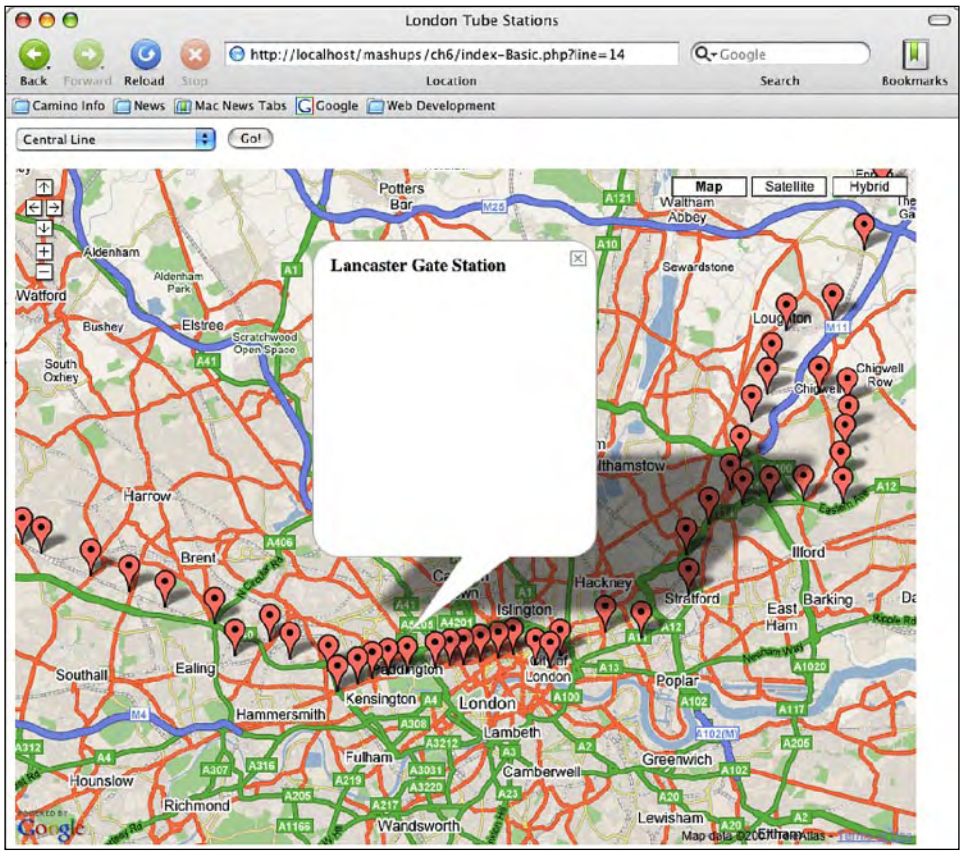
```
<form name="selectionForm" action="index-Basic.php" method="get">
<select name="line">
   <option value="">Select a Line</option>
```

```
<?php foreach($linesArr as $lines) { ?>
   <option value="<?= $lines->getLineId() ?>" <?= $_GET['line'] ==
   $lines->getLineId() ? "selected=\"selected\"" : "" ?>><?= $lines-
   >getLineName() ?></option>
<?php } ?>
</select>
<input type="submit" value="Go!" />
</form>
```

This code block draws the form object that we use for line selection. The PHP `foreach` loop loops through the array of Tube lines to grab each line object.

```
    <div id="map" style="width: 800px; height: 600px"></div>
</body>
</html>
```

At this point, the map is functional. You have a mashup that can draw stations in the London Tube system. You can navigate around, select lines, and click on markers to see what stations you clicked on.



Using Flickr Services with AJAX

With some slight modification, we can add a call to Flickr Services. Generally, the strategy we want is to make an HTTP request with the `XMLHttpRequest` object when the user clicks on a marker A good place to do this is in the callback function for the marker's event listener. We already know the name of the station, so we can use it as the basis of a search to Flickr.

This is a very acceptable strategy, but there's a huge problem. In general, browsers cannot make an HTTP request with `XMLHttpRequest` to another server. This is done to prevent cross-site scripting attacks, in which a malicious website runs code to steal information about sensitive information between a user and another website. In practice, this means that `XMLHttpRequest` calls can only go back to the server the page originated from. With this limitation, how are we going to use `XMLHttpRequest` to make a call from our website to Flickr Services?

Creating an XMLHttpRequest Proxy

The solution is to create a web service proxy on our server. The web server will execute the Flickr Service call, not the browser. Our `XMLHttpRequest` action will execute a `GET` request on the proxy and pass Flickr Services parameters to the proxy. The proxy will then make a request to Flickr, and pass the response back, unaltered, to the web browser. The browser doesn't know or care that the true data source is from Flickr. In the examples code, in the services directory, the proxy is named `searchFlickr.php`. This is a small file whose sole job is to do just that.

```
<?php
require_once('../classes/RESTParser.php');
$restParser = new RESTParser();
```

We will use the REST interface from Flickr. In this code, we will use the same REST parser that we created from Chapter 1 to handle the REST call.

```
$paramArray = array();
foreach ($_GET as $key => $value) {
   if ($key == 'format' || $key == 'nojsoncallback' || $key = 'text')
   {
       $paramArray[$key] = $value;
   } else {
       die("Unallowed Parameter Passed.");
   }
}
```

We initialize an array of Flickr Services parameters. We will pass this array to the REST parser when we actually make the service call. This array is populated by looping through the `GET` array and adding the array key and values to the `$paramArray`. As this page is open to the entire world, it is a good idea to put some security around it. Here, we are allowing only three Flickr parameters to be passed from the calling page. Otherwise, the script will die.

```
//Add the API Key to the Request
$paramArray['api_key'] = 'YOUR FLICKR SERVICES KEY';
$paramArray['method'] = 'flickr.photos.search';
$paramArray['per_page'] = '4';
$paramArray['page'] = '1';
$paramArray['text'] .= ' London tube';
```

One additional benefit of this approach is that we get to hide our API key on the server. If it was JavaScript making this call, we would have to expose our key in front end code, and anyone can steal it. This isn't as much of a concern with the Google API Key because that key is restricted by domain. However, there is no such restriction for the Flickr Services key. Here, we add it to the parameter array on the server. The key is passed in server-to-server communication, and the user will never see it.

As an additional security measure, we also specify the Flickr method here. This insures that only the `flickr.photos.search` method is called from this script.

We are passing two additional parameters to make the results a little more manageable. `per_page` will limit the results returned from Flickr to just 4 photos per page. We then tell Flickr to return only one page using the `page` parameter. The result is that a maximum of four photos will be returned by Flickr.

The final line in this block adds "London tube" to our search query terms passed to Flickr. This is solely for the purposes of narrowing the search..

```
echo $restParser->callService($paramArray, 'api.flickr.com', '/
services/rest/', 'GET');
?>
```

Finally we pass the array of parameters to the RESTParser's `callService()` method along with the Flickr Services server and endpoint information. The method returns the response from the server, and we just echo it out to the requester.

Modifying the Main JavaScript

Now we can modify our mashup's index page. In the examples code for this chapter, there is a file named `index.php`. This is the full, completed home page for the mashup. It is basically `index-Basic.php` from earlier, but with the Flickr Services calls. We will talk about what is different with this version from the basic version.

The first thing we need to do is add a handful more global variables to track.

```
var g_xmlHttp;
var g_stationName;
var g_flickrString;
var g_map;
```

The first, `g_xmlHttp`, is a container for the `XMLHttpRequest` object. The next two, `g_stationName` and `g_flickrString`, are used to hold information from the Flickr Service response. We will talk about the need for them as we encounter them. Finally, `g_map` is the same Google Map container as before.

Making the XMLHttpRequest

Before we can make `XMLHttpRequest` requests, we need a function to create the `XMLHttpRequest` object when a request is about to be made. This is done through the `createXMLHttpRequest()` function.

```
function createXMLHttpRequest() {
    if (window.XMLHttpRequest){
       g_xmlHttp = new XMLHttpRequest()
    } else if (window.ActiveXObject) {
       g_xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
    }
}
```

This code uses the standard browser check to see if the browser can create a native `XMLHttpRequest` object or, if it is Internet Explorer, create a `Microsoft.XMLHTTP` request object through `ActiveXObject.` The object is then placed in `g_xmlHttp`.

We call `createXMLHttpRequest()` in the first line of a modified Event Listener callback function.

```
GEvent.addListener(marker, "click", function() {
   createXMLHttpRequest();
   g_stationName = stationName;
    retrieveFlickrPhotos(stationName);
   marker.openInfoWindowHtml("<div style=\"width:220px; height:250px;\
   "><b>" + stationName + "</b><p style=\"text-align:center;\">
   <img src=\"images/wait.gif\" style=\"padding-top:50px;\" /></p></
   div>");
});
```

When the AJAX response is returned, the browser will have to update the `InfoWindow`. At that point, the browser will not know the name of the station that was clicked, so we store it in a global variable. This second statement is more for user friendliness than application functionality.

Whenever someone clicks on the marker, the application should make the AJAX request. Therefore, it needs to be included here in the click event. We will have to define the request execution in a new function, `retrieveFlickrPhotos()`. This function will actually create the Flickr search parameters, so we need to pass the name of the station to use as the search term.

```
function retrieveFlickrPhotos(stationName) {
    var url = "services/searchFlickr.php? format=
             json&nojsoncallback=1&text=" + escape(stationName);
    g_xmlHttp.onreadystatechange = parseFlickrSearch;
    g_xmlHttp.open("GET", url, true);
    g_xmlHttp.send(null);
}
```

This function first prepares the URL back to our `searchFlickr.php` service on our server. We add the Flickr Services parameters to this URL. The parameters we pass are summarized as follows:

Parameter	Value	Notes
format	json	We want the response to be in JSON.
nojsoncallback	1	We do not want to automatically execute a callback when the JSON response is received. This functionality will be handled by the XMLHttpRequest callback.
text	The station name and extra search parameters	We need to pass the name through the JavaScript `escape()` function to make it URL-friendly. We also pass the terms "London" and "tube" to narrow our search. The latter is purely to refine our results.

After that, the call to the service is executed. We define a callback named `parseFlickrSearch()` to handle the response.

Race Conditions

After this, we should create `parseFlickrSearch()` and define how we are going to update the `InfoWindow` with Flickr photos. Before we do this, though, we need to talk a little bit about **race conditions**.

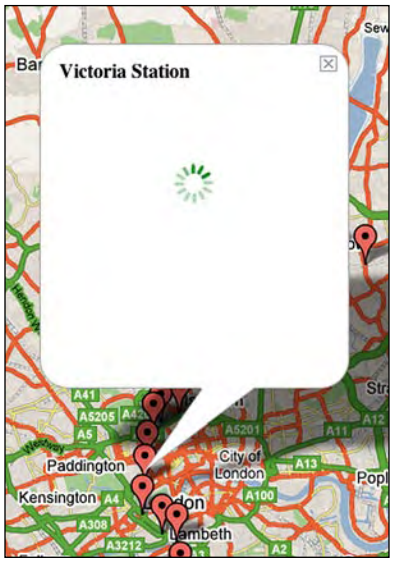
Race conditions are a notion that originated in the electronics design, but has been adopted by software designers. In simple terms, it is when execution of a code happens before a prerequisite is met. This is a constant pitfall in multi-threaded languages like C, C++, and Java. PHP, being a single threaded language, does not usually encounter a lot of race condition issues. One exception to this in PHP and an example of a race condition is in file manipulation.

If you copy a file to a location with PHP's `copy()` function, the operating system needs to finish copying before you can work on the copy. Otherwise, operations on the copy will fail. This might not be an issue with 4 kilobyte text files, but imagine moving a 700 megabyte CD image. Even with a fast RAID, this might take a minute or so to copy, during which time, your script must wait.

In developing with web services, where we have to call other networks, we will need to be cognizant of race conditions due to network latency. AJAX only adds in more complexity. An AJAX application, where code execution takes place on the browser, has no idea what is going on with the web server. If multiple asynchronous requests are made before they are fulfilled by the server, AJAX applications may see strange results. Data retrieved by a request may not match up perfectly to the request that initiated it. In other words, what you see on screen may have been caused by an action several clicks ago. Our code must successfully handle these cases.

We will encounter race conditions at several points when we parse code. There are many strategies we can employ to counter race conditions, and they are usually much customized to a problem. However, solutions often fall into broader categories. One way to combat a race condition is to pre-cache the data during a time when the user is not interacting with the system so things like network latency and system timeouts are not significant. Another solution is to reserve and hold onto a resource so that it will be available when you need it. When we look at our race conditions, we will simply make sure every resource has arrived before we execute code.

The first time we encounter a race condition is when we click on a marker. At this point, the `InfoWindow` opens. The AJAX request has been initiated, yet it must go through our proxy, wait for Flickr's response, and then come back through our proxy. We face some network latency. Meanwhile, our user sees a blank window.



Is anything happening on the left? Did the service find any photos? Did the server time out? The user does not know. This condition is not disastrous, but shows that we have to do something about the timing. A perfectly reasonable way to handle this is to tell the user that something is definitely happening, and be patient. On the right, we add a "loading" graphic in the user interface to tell the user to wait.

If you we clicked on a marker, would you rather see the blank space on the left, or some feedback of status like the one on the right?

To add this, we can simply add an image tag to the HTML string that is passed when we open the `InfoWindow`.

```
marker.openInfoWindowHtml("<div style=\"width:220px; height:250px;\
"><b>" + stationName + "</b><p style=\"text-align:center;\"><img src=\
"images/wait.gif\" style=\"padding-top:50px;\" /></p></div>");
```

Make Your Own Load Graphics

There are many repositories out there with load images for you to download and use for free. If none of them suit your tastes, you can make your own. Fortunately there is a site that can help. Ajaxload.info (`http://www.ajaxload.info/`) offers many basic load designs and lets you customize with any color.

Parsing the AJAX Response

Let's continue with our response parsing code. This section will deal with how we get data out of the call to Flickr Services and how we update the web page. The first step is to create `parseFlickrSearch()`, the callback function that we specified when we made the outgoing HTTP request with `XMLHttpRequest`.

```
function parseFlickrSearch() {
   if (g_xmlHttp.readyState == 4) {
       var results = eval('(' + g_xmlHttp.responseText + ')');
       var photo = results.photos.photo;
       var totalPhotos = results.photos.total;
       var l_flickrString = "";
```

We start off by checking the status of the request. If the request is complete, we continue with the execution of our code. Little did we know previously that by waiting, we were dealing with a race condition.

The first statement after the `if` statement places the Flickr response, stored in the `XMLHttpRequest` property `responseText`, in the `results` variable. This is after the code has been executed through `eval()`.

The next line goes straight to the list of photos returned. Remember the first few lines of a Flickr Service Response:

```
<?xml version="1.0" encoding="utf-8" ?>
<rsp stat="ok">
<photos page=»1» pages=»20» perpage=»100» total=»1904»>
   <photo id="412962278" owner="43203076@N00" secret="63e7e2e1f0"
   server="183" farm="1" title="Doin' Studio Time" ispublic="1"
   isfriend="0" isfamily="0" />
```

The service returns one `photo` element for each photo found. In JSON, this is treated as an array. Therefore, think of `photo` as an array of photo objects.

We set a variable, `totalPhotos`, to manage the total number of photos returned. We set one last local variable, `l_flickrString`, to store the local results from Flickr. This is a local variable that will be appended to the global `g_flickrString`.

```
        g_flickrString = "<div><b>" + g_stationName + "</b><br />"
```

The HTML that will be in the `InfoWindow` is stored in the variable `g_flickrString`. Here, we start the string by repeating the name of the station, which was stored in a global variable earlier when the marker was first clicked.

```
if (totalPhotos > 0) {
for (x = 0; x < totalPhotos; x++) {
       l_flickrString = " " +
       "<a href='http://www.flickr.com/photos/PHOTO_OWNER/
       PHOTO_ID' />" +
       "<img src='http://farmPHOTO_FARM.static.flickr.com" +
       "/PHOTO_SERVER/PHOTO_ID_PHOTO_SECRET_t.jpg' border='0' /></a>";
       l_flickrString = l_flickrString.replace(/PHOTO_OWNER/g,
       photo[x].owner);
       l_flickrString = l_flickrString.replace(/PHOTO_ID/g,
       photo[x].id);
       l_flickrString = l_flickrString.replace(/PHOTO_FARM/g,
       photo[x].farm);
       l_flickrString = l_flickrString.replace(/PHOTO_SERVER/g,
       photo[x].server);
       l_flickrString = l_flickrString.replace(/PHOTO_SECRET/g,
       photo[x].secret);
       g_flickrString = g_flickrString + l_flickrString;
}
```

Here is where the population of Flickr data actually takes place. The `if` clause makes sure some results were returned. If there are results returned, we loop through the photo objects using a `for` loop and limited to the frequency of loops to `totalPhotos`. Each loop through creates a string containing the URL to the picture returned and the anchor tag to the photo's page. This string is stored in the `l_flickrString` variable. For readability, we use a few placeholders for the Flickr values in the string, then we use the JavaScript `replace()` method to exchange these placeholders with the actual values from the `photo` array. At the end,. `l_flickrString` is attached to the global `g_flickrString`.

```
          } else {
                 g_flickrString = g_flickrString + "<p>No photos found
                 for this station.</p>";
          }
       }
}
```

After this, we close the `if-else` block. The `else` statement says if no results were found, update `g_flickrString` with a message telling the user that the search came up empty. This function's sole job was to create the string of HTML that will be in `InfoWindow`. Let's take a look at updating `InfoWindow` with this string.

The main population happens in `updateInfoBox()`.

```
function updateInfoBox() {
   if (g_flickrString == undefined) {
      var timeout = window.setTimeout("updateInfoBox()", 3000);
   } else {
   g_map.getInfoWindow().getContentContainers()[0].innerHTML = "<div>"
   + g_flickrString + "</div>";
            //Cleanup
            g_flickrString = null;
            g_stationName = null;
   }
}
```

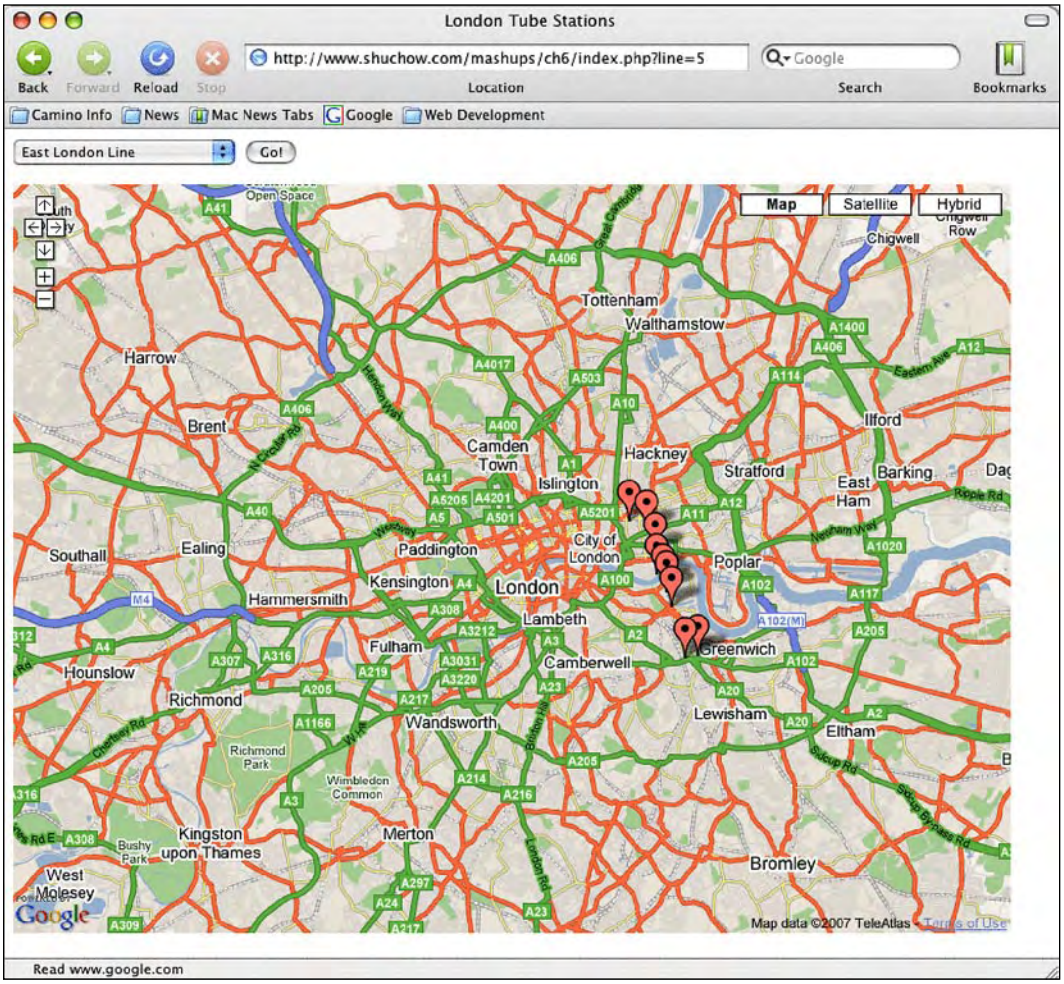
This function is the last function called by the event listener.

```
GEvent.addListener(marker, "click", function() {
   createXMLHttpRequest();
   g_stationName = stationName;
   retrieveFlickrPhotos(stationName);
   marker.openInfoWindowHtml("<div style=\"width:220px; height:250px;\
   "><b>" + stationName + "</b><p style=\"text-align:center;\"><
   img src=\"images/wait.gif\" style=\"padding-top:50px;\" /></p></
   div>");
   updateInfoBox();
});
```

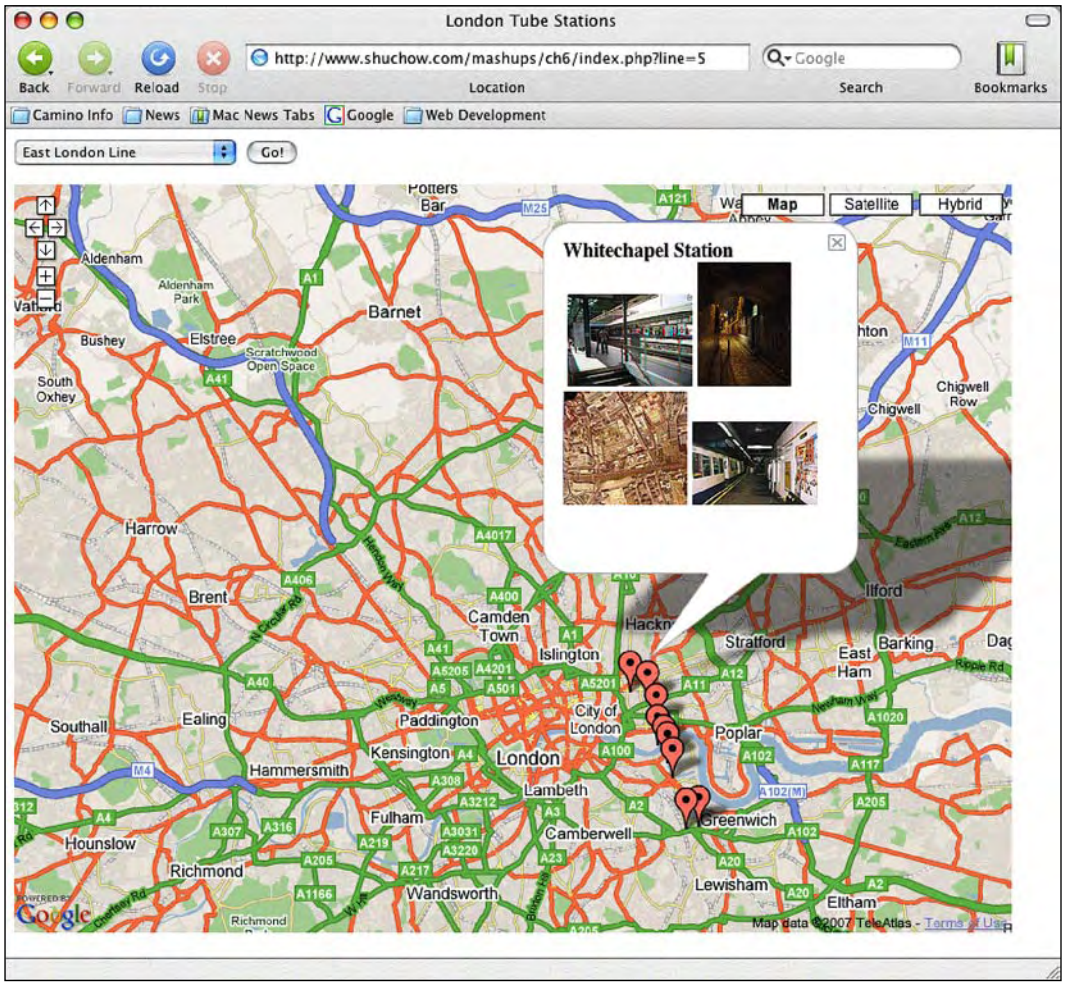
However, remember the service call happens elsewhere. While the information is being retrieved, the window is already there. This is another race condition. If we call `g_flickrString` before it is set, you will find it is undefined. If `g_flickrString` is empty, use the `setTimeout()` JavaScript function to call itself after three seconds. This delay in execution is a frequent tactic used in AJAX implementations.

If results were found, we get the DOM node of the `InfoWindow` box. This was done using the DOM Inspector in Firefox. After this, we can append `g_flickrString` into the node. Finally, we clean up the global variables by setting them to null.

At long last, our mashup is complete. We can take it out for a test drive. Load the web page in your browser and select a line with the pull-down menu. The markers for the line will appear.



Click on one of the markers.



The `InfoWindow` will pop up. Through AJAX, our application is already searching for our station at Flickr. When it finds it, the first four photos are added to the `InfoWindow`.

Summary

We have covered a lot of technologies in this chapter. We learned how to read RDF documents and how to extract data from them using SPARQL and RAP for RDF. These standards are fairly new. However, given the desire to put as much as possible into RSS, these technologies are certainly bound to take off.

When we created the front end application, there were more new technologies including AJAX to communicate from the server to the device. The biggest pitfall in this AJAX application was race conditions. We examined how to overcome those with various techniques.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

P

R

S

T

U

V

W

X

Y