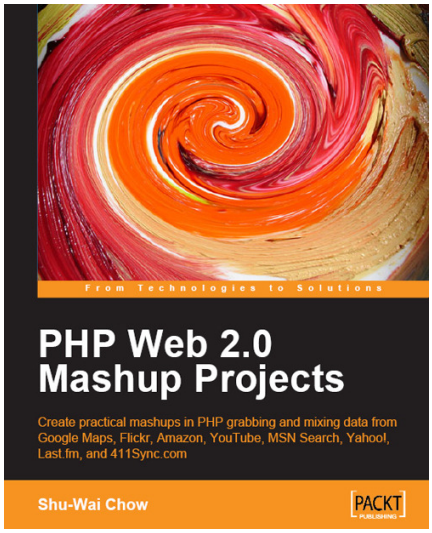
PHP Web 2.0 Mashup Projects

Practical PHP Mashups with Google Maps, Flickr, Amazon, YouTube, MSN Search, Yahoo!

Shu-Wai Chow



Chapter No. 4

"Your Own Video Jukebox"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, **Chapter No. 4 "Your Own Video Jukebox"**

A synopsis of the book's content

Information on where to buy this book

About the Author

Shu-Wai Chow has worked in computer programming and information technology for the past eight years. He started his career in Sacramento, California, spending four years as the webmaster for Educaid, a First Union Company, and another four years at Vision Service Plan as an application developer. Through the years, he has become proficient in Java, JSP, PHP, ColdFusion, ASP, LDAP, XSLT, and XSL-FO. Shu has also been the volunteer webmaster and a feline adoption counselor for several animal welfare organizations in Sacramento.

He is currently a software engineer at Antenna Software in Jersey City, New Jersey, and is finishing his studies in Economics at Rutgers, the State University of New Jersey. Born in the British Crown Colony of Hong Kong, Shu did most of his alleged growing up in Palo Alto, California. He lives on the Jersey Shore with seven very demanding cats, four birds that are too smart for their own good, a tail-less bearded dragon, a betta who needs her tank cleaned, a dermestid beetle colony, a cherished Fender Stratocaster, and a beloved, saint-like fiancé.

PHP Web 2.0 Mashup Projects:

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.

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.

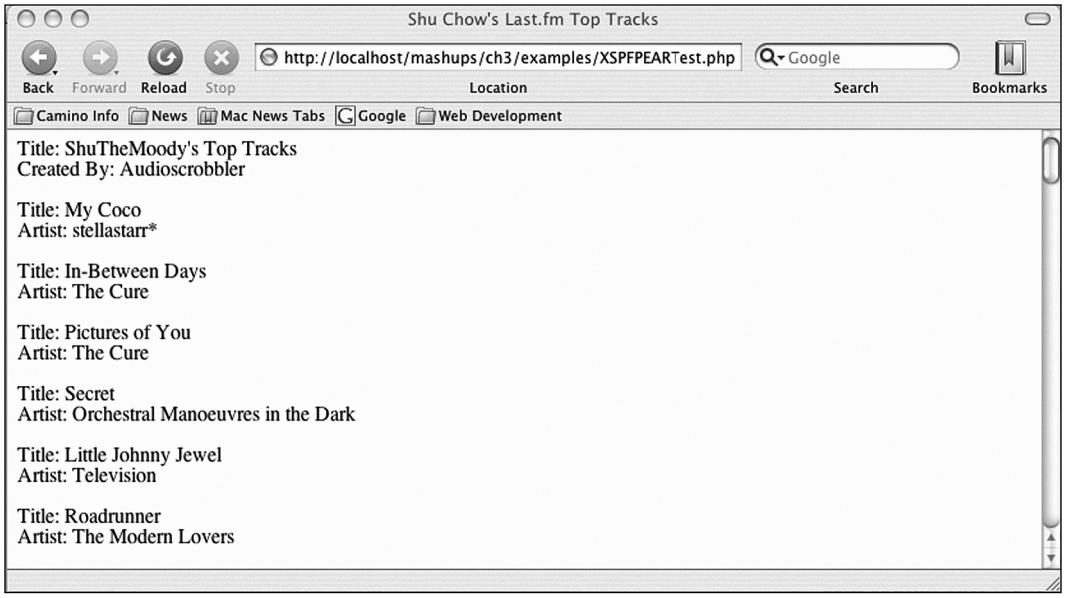
```
<html>
<head>
   <title>Shu Chow's Last.fm Top Tracks</title>
</head>
<body>
    Title: <?= $xspfObj->getTitle() ?><br />
    Created By: <?= $xspfObj->getCreator() ?>
```

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>
```

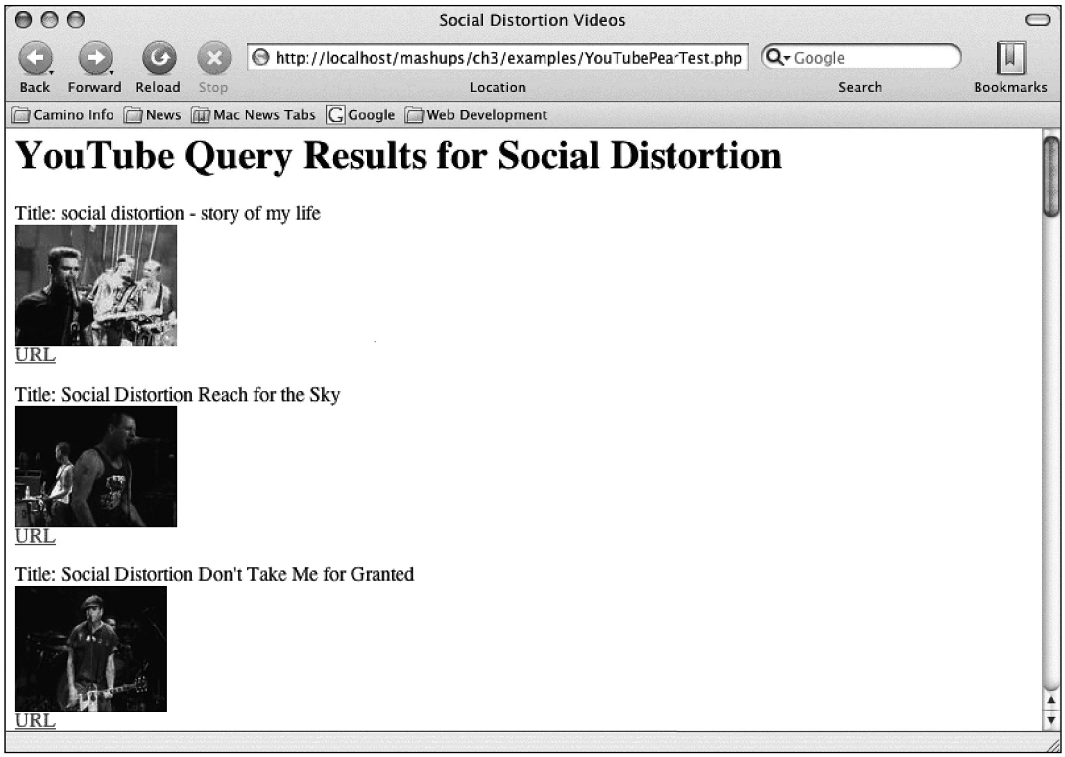
Next, we will loop through the videos. In order to get an array of video objects, we first need to parse the XML response. We do that using Services_YouTube's xpath method, which will use the powerful XPATH query language to go through the XML and convert it into PHP objects. We pass the XPATH query 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.

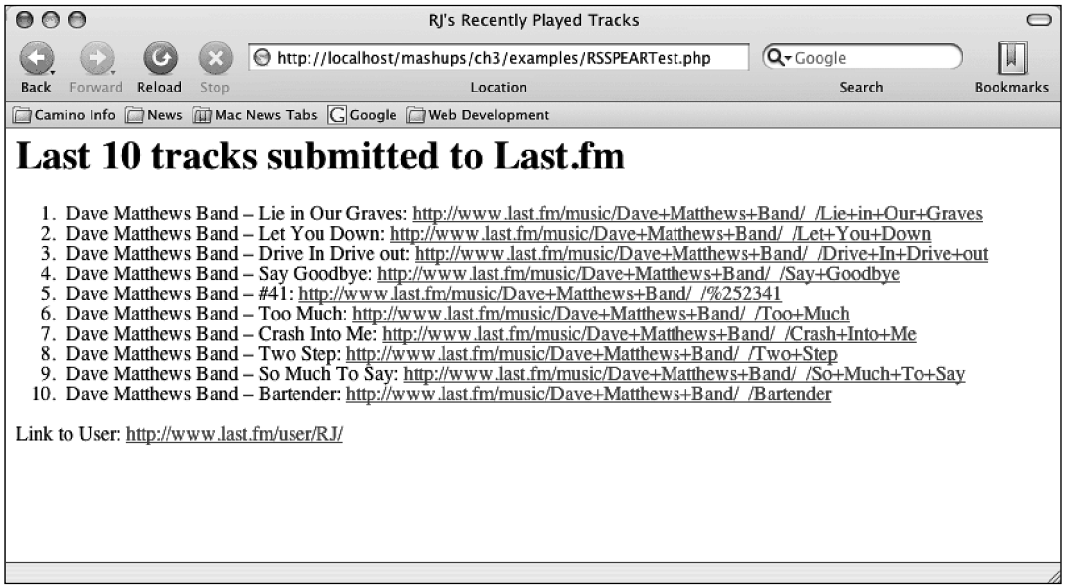
```
      Link to User:
      <a href="<?= $channelInfo['link'] ?>"><?=
                                          $channelInfo['link'] ?></a>
   </body>
</html>
```

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.

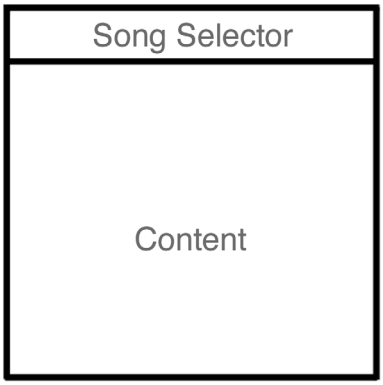
Mashup Architecture

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.

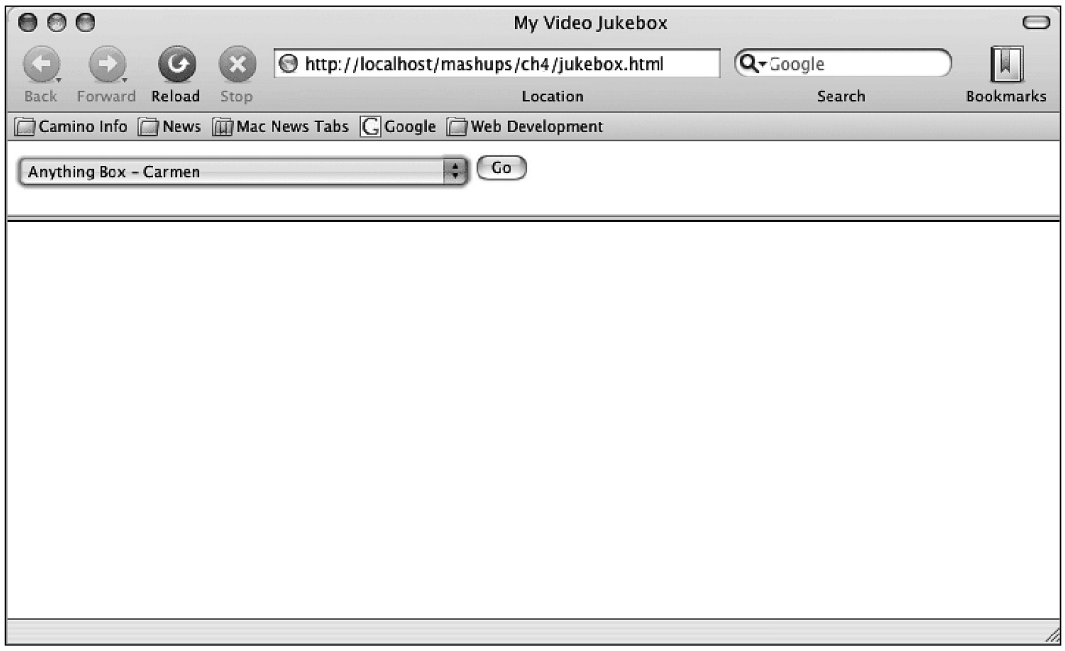
```
<?php } else { ?>
   No Results Found
<?php } ?>
</body>
</html>
```

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.

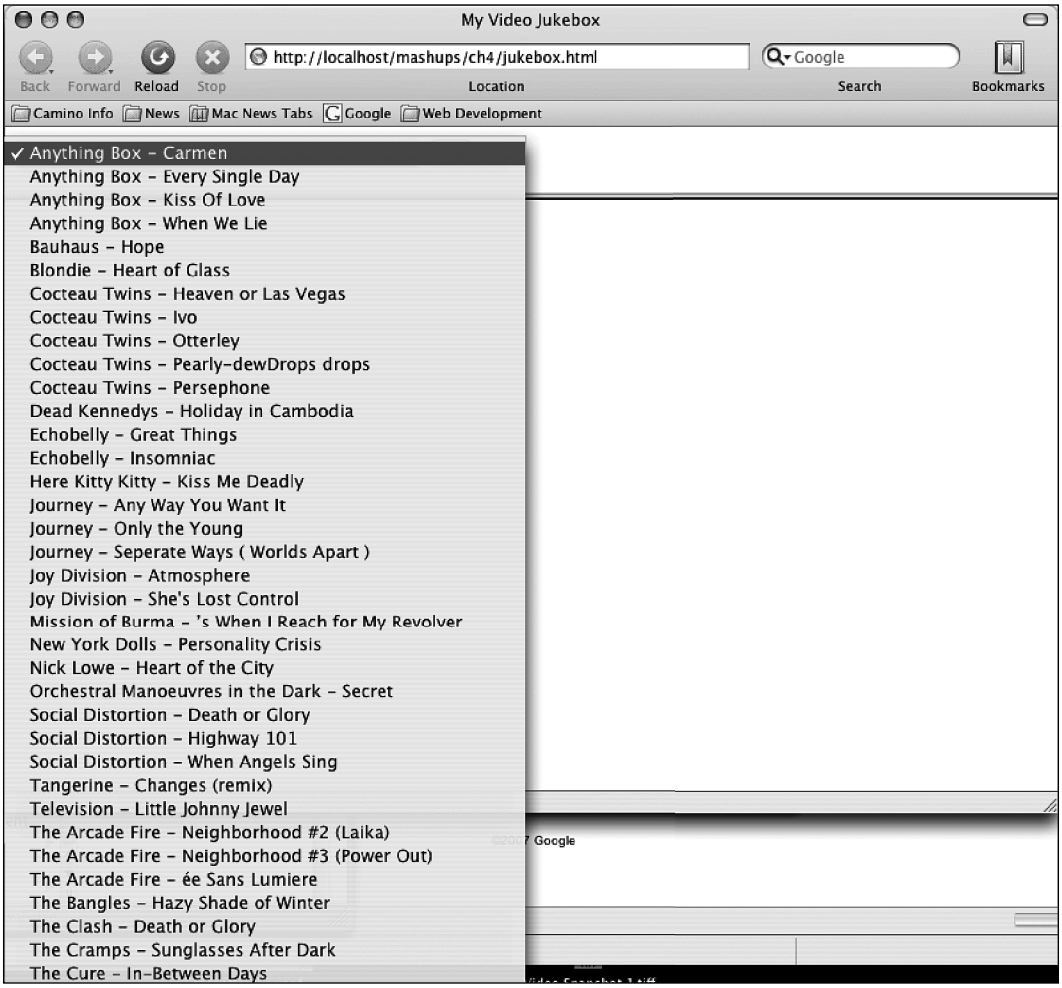
Place these three files into the same, web-server accessible directory on a web server. You can start using the mashup by launching `jukebox.php` in your web browser.

Using the Mashup

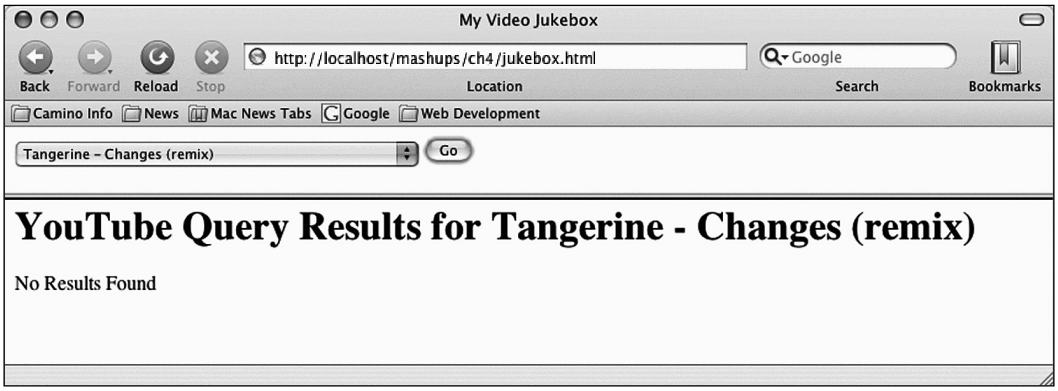
Using the mashup is quite simple. Load `jukebox.html` into your web browser.



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.

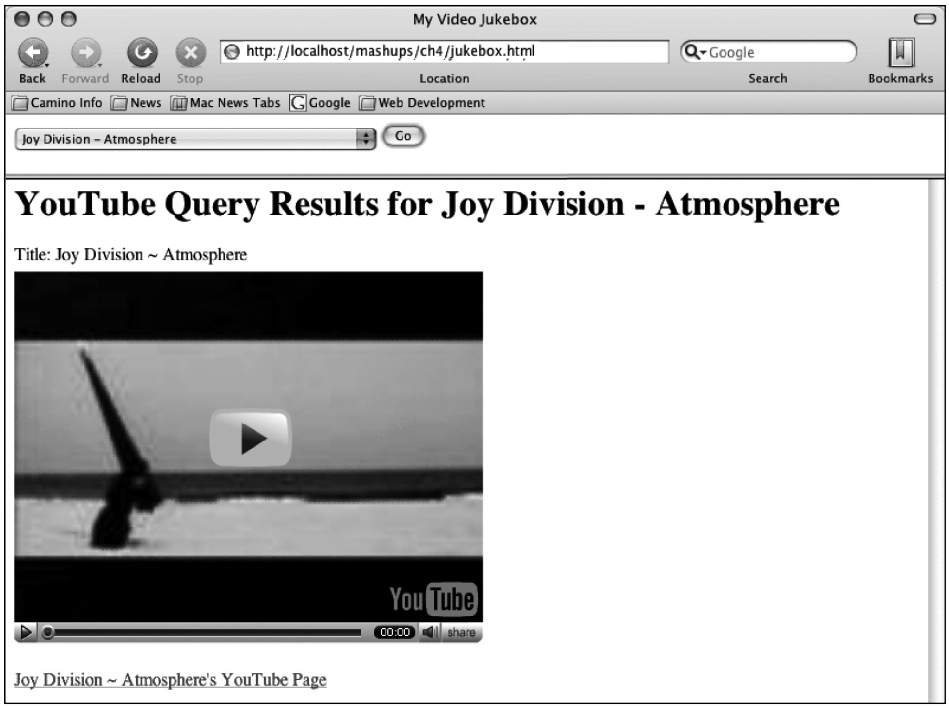


Click on the menu. You will see a list of the songs from the two feeds. Select one and press the **Go** button.



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.

Where to buy this book

You can buy **PHP Web 2.0 Mashup Projects: Practical PHP Mashups with Google Maps, Flickr, Amazon, YouTube, MSN Search, Yahoo!** from the Packt Publishing website: http://www.packtpub.com/php-web-20-mashups/book

Free shipping to the US, UK, Europe, Australia, New Zealand and India.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com

For More Information: http://www.packtpub.com/php-web-20-mashups/book