From Technologies to Solutions

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

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

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)

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

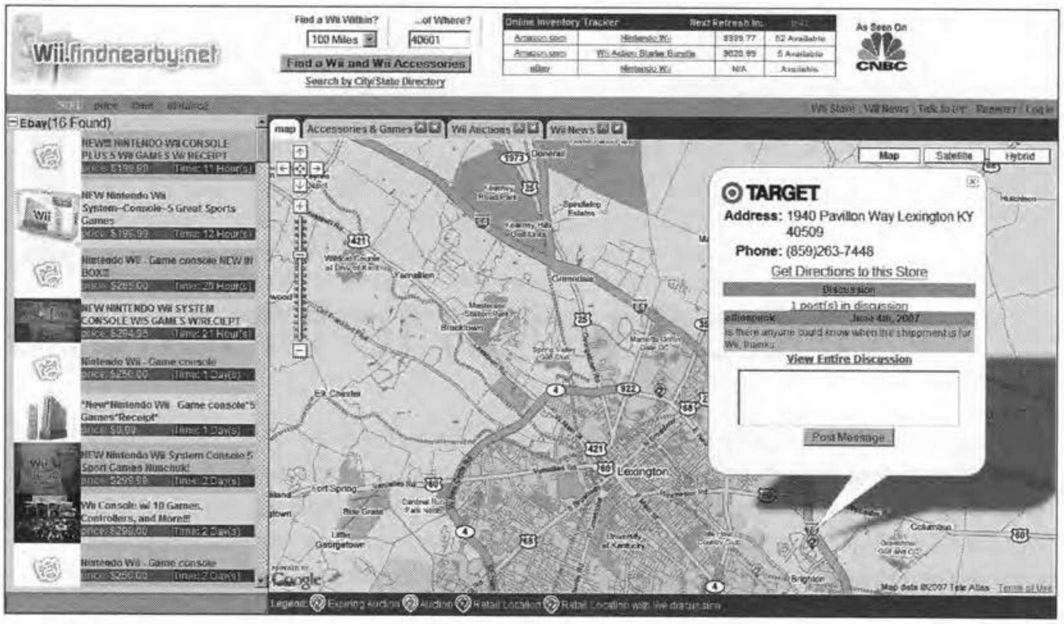
You can contact us at `questions@packtpub.com` if you are having a problem with some aspect of the book, and we will do our best to address it.

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.



Wii.findnearby.net

Find a Wii Within? 100 Miles ...of Where? 40601

Find a Wii and Wii Accessories

Search by City/State Directory

Online Inventory Tracker		Next Refresh In:	
Amazon.com	Nintendo Wii	$339.77	52 Available
Amazon.com	Wii Action Starter Bundle	$620.99	5 Available
eBay	Nintendo Wii	N/A	Available

As Seen On CNBC

Ebay(16 Found)

- NEW!!! NINTENDO Wii CONSOLE PLUS 5 Wii GAMES W/ RECEIPT — Price: $199.99 — Time: 11 Hour(s)
- NEW Nintendo Wii System–Console–5 Great Sports Games — Price: $195.99 — Time: 12 Hour(s)
- Nintendo Wii - Game console NEW IN BOX!!! — Price: $265.00 — Time: 20 Hour(s)
- NEW NINTENDO Wii SYSTEM CONSOLE Wi5 GAMES W/RECIEPT — Price: $294.95 — Time: 21 Hour(s)
- Nintendo Wii - Game console — Price: $250.00 — Time: 1 Day(s)
- *New*Nintendo Wii - Game console*5 Games*Receipt* — Price: $0.00 — Time: 1 Day(s)
- NEW Nintendo Wii System Console 5 Sport Games Nunchuk! — Price: $299.98 — Time: 2 Day(s)
- Wii Console w/ 10 Games, Controllers, and More!!! — Price: $298.00 — Time: 2 Day(s)
- Nintendo Wii - Game console — Price: $250.00 — Time: 2 Day(s)

TARGET

Address: 1940 Pavilion Way Lexington KY 40509

Phone: (859)263-7448

Get Directions to this Store

Discussion

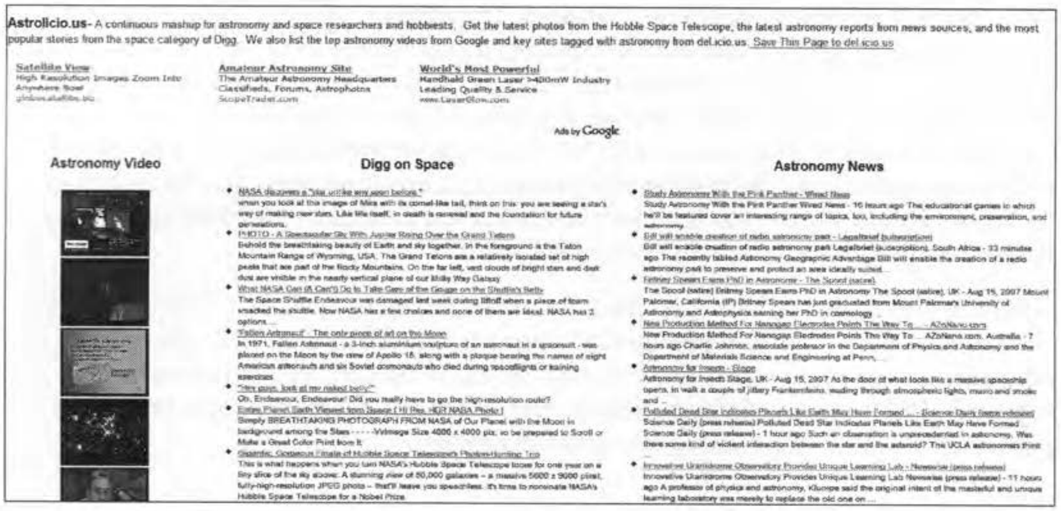
1 post(s) in discussion

June 4th, 2007

Is there anyone could know when the shippment is for Wii. thanks

View Entire Discussion

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.



Astrolicio.us- A continuous mashup for astronomy and space researchers and hobbiests. Get the latest photos from the Hobble Space Telescope, the latest astronomy reports from news souces, and the most popular stories from the space category of Digg. We also list the top astronomy videos from Google and key sites tagged with astronomy from del.icio.us. Save This Page to del.icio.us

Satellite View
High Resolution Images Zoom Into Anywhere Now!

Amateur Astronomy Site
The Amateur Astronomy Headquarters Classifieds, Forums, Astrophotos
ScopeTrader.com

World's Most Powerful
Handheld Green Laser >400mW Industry Leading Quality & Service
www.LaserGlow.com

Ads by Google

Astronomy Video

Digg on Space

- NASA discovers a "star unlike any seen before."
  when you look at this image of Mira with its comet-like tail, think on this: you are seeing a star's way of making new stars. Like life itself, in death is renewal and the foundation for future generations.
- PHOTO - A Spectacular Sky With Jupiter Rising Over the Grand Tetons
  Behold the breathtaking beauty of Earth and sky together. In the foreground is the Teton Mountain Range of Wyoming, USA. The Grand Tetons are a relatively isolated set of high peaks that are part of the Rocky Mountains. On the far left, vast clouds of bright stars and dark dust are visible in the nearly vertical plane of our Milky Way Galaxy
- What NASA Can (& Can't) Do to Take Care of the Gouge on the Shuttle's Belly
  The Space Shuttle Endeavour was damaged last week during liftoff when a piece of foam smacked the shuttle. Now NASA has a few choices and none of them are ideal. NASA has 3 options ...
- "Fallen Astronaut" - The only piece of art on the Moon
  In 1971, Fallen Astronaut - a 3-inch aluminium sculpture of an astronaut in a spacesuit - was placed on the Moon by the crew of Apollo 15, along with a plaque bearing the names of eight American astronauts and six Soviet cosmonauts who died during spaceflights or training exercises
- "Hey guys, look at my naked belly!"
  Oh, Endeavour, Endeavour! Did you really have to go the high-resolution route?
- Entire Planet Earth Viewed from Space (Hi Res. HDR NASA Photo)
  Simply BREATHTAKING PHOTOGRAPH FROM NASA of Our Planet with the Moon in background among the Stars - - - - -Wimage Size 4000 x 4000 pix. so be prepared to Scroll or Make a Great Color Print from It
- Gigantic, Gorgeous Finale of Hubble Space Telescope's Photo-Hunting Trip
  This is what happens when you turn NASA's Hubble Space Telescope loose for one year on a tiny slice of the sky above: A stunning view of 50,000 galaxies – a massive 5600 x 9000 pixel, fully-high-resolution JPEG photo – that'll leave you speechless. It's time to nominate NASA's Hubble Space Telescope for a Nobel Prize

Astronomy News

- Study Astronomy With the Pink Panther - Wired News
  Study Astronomy With the Pink Panther Wired News - 16 hours ago The educational games in which he'll be featured cover an interesting range of topics, too, including the environment, preservation, and astronomy.
- Bill will enable creation of radio astronomy park - Legalbrief (subscription)
  Bill will enable creation of radio astronomy park Legalbrief (subscription), South Africa - 33 minutes ago The recently tabled Astronomy Geographic Advantage Bill will enable the creation of a radio astronomy park to preserve and protect an area ideally suited ...
- Britney Spears Earns PhD in Astronomy - The Spoof (satire)
  The Spoof (satire) Britney Spears Earns PhD in Astronomy The Spoof (satire), UK - Aug 15, 2007 Mount Palomar, California (IP) Britney Spears has just graduated from Mount Palomar's University of Astronomy and Astrophysics earning her PhD in cosmology ...
- New Production Method For Nanogap Electrodes Points The Way To ... - AZoNano.com
  New Production Method For Nanogap Electrodes Points The Way To ... AZoNano.com, Australia - 7 hours ago Charlie Johnson, associate professor in the Department of Physics and Astronomy and the Department of Materials Science and Engineering at Penn, ...
- Astronomy for Insects - Stage
  Astronomy for Insects Stage, UK - Aug 15, 2007 As the door of what looks like a massive spaceship opens, in walk a couple of jittery Frankensteins, wading through atmospheric lights, music and smoke and ...
- Polluted Dead Star Indicates Planets Like Earth May Have Formed ... - Science Daily (press release)
  Science Daily (press release) Polluted Dead Star Indicates Planets Like Earth May Have Formed ... Science Daily (press release) - 1 hour ago Such an observation is unprecedented in astronomy. Was there some kind of violent interaction between the star and the asteroid? The UCLA astronomers think ...
- Innovative Uranidrome Observatory Provides Unique Learning Lab - Newswise (press release)
  Innovative Uranidrome Observatory Provides Unique Learning Lab Newswise (press release) - 11 hours ago A professor of physics and astronomy, Klumpe said the original intent of the masterful and unique learning laboratory was merely to replace the old one on ...

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.

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

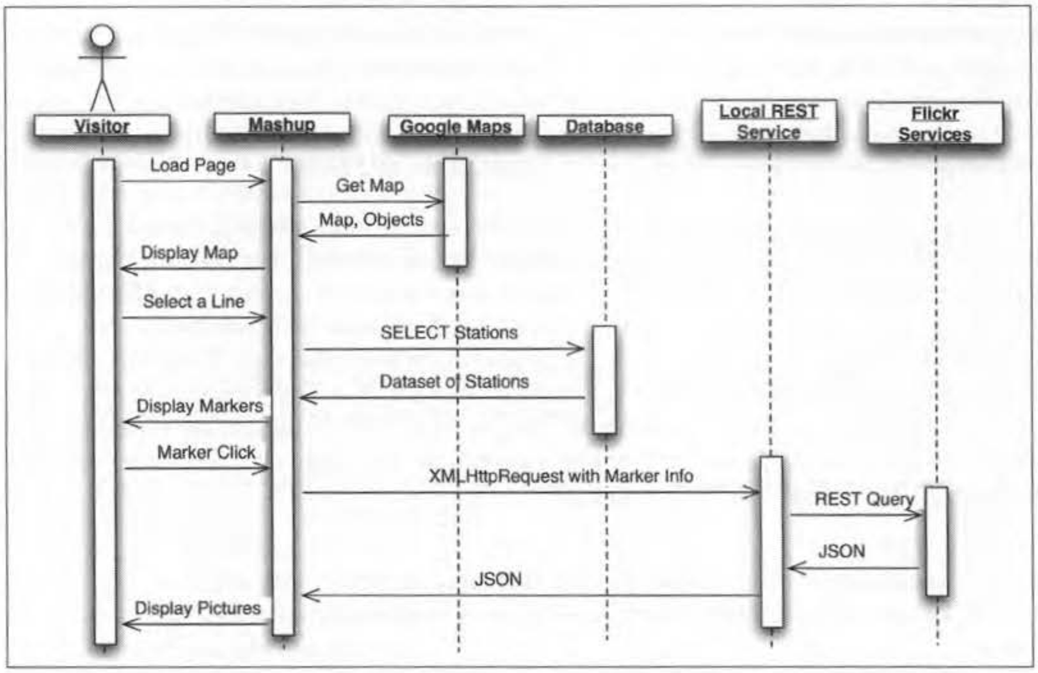
Now that we have the data and know generally how to create markers with that data, we need to look at how to bridge a JavaScript call in Google Maps to a server call in Flickr Services. Flickr Services has a REST-based endpoint available. This means that all we would need to do is send a `GET` or `POST` 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.

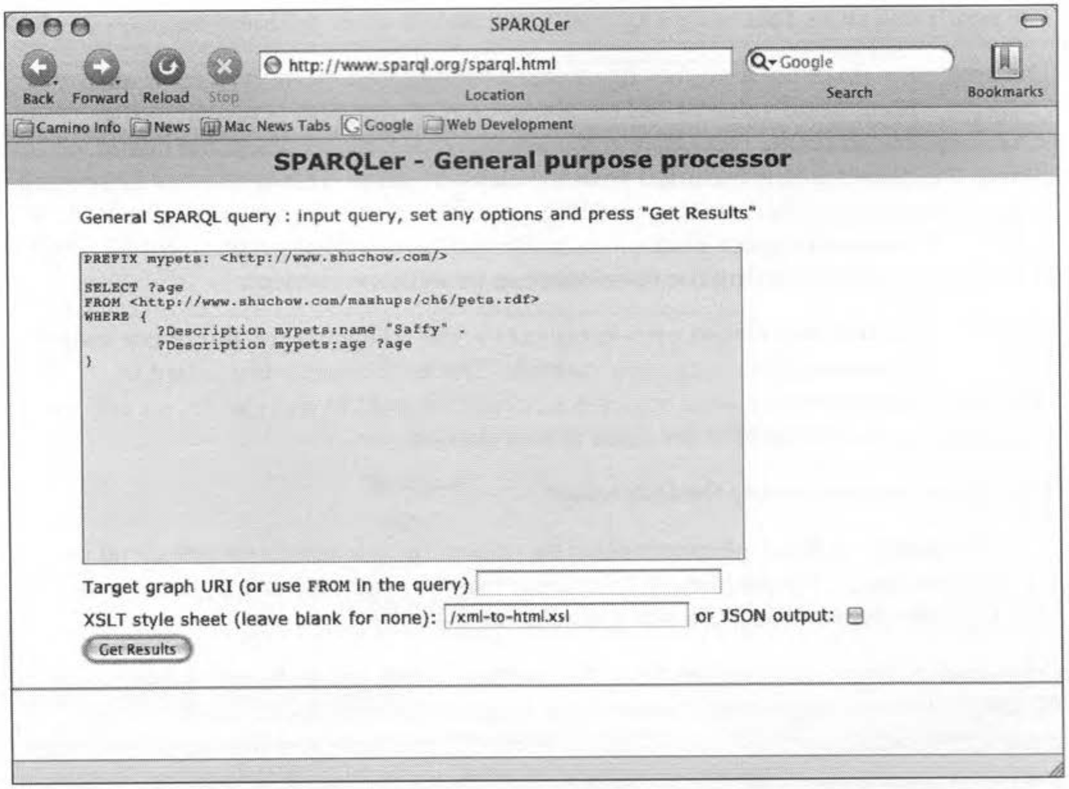
In SPARQL `WHERE` clauses, the key concept to remember is that all variables reference the same thing. The order of the `WHERE` statements matters very little. It is what each variable's value is at the end of execution that matters.

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
}
```


SPARQLer

http://www.sparql.org/sparql.html

SPARQLer - General purpose processor

General SPARQL query : input query, set any options and press "Get Results"

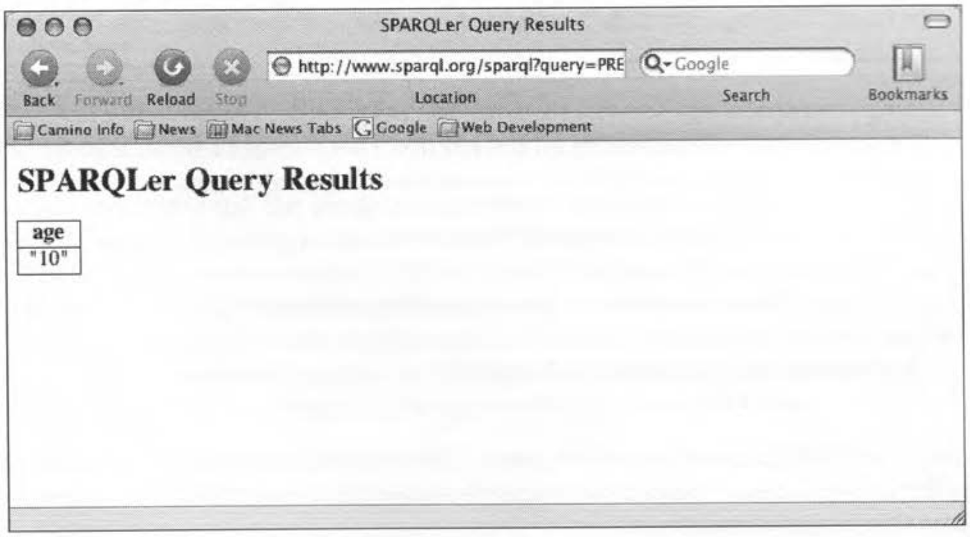
```
PREFIX mypets: <http://www.shuchow.com/>

SELECT ?age
FROM <http://www.shuchow.com/mashups/ch6/pets.rdf>
WHERE {
        ?Description mypets:name "Saffy" .
        ?Description mypets:age ?age
}
```

Target graph URI (or use FROM in the query)

XSLT style sheet (leave blank for none): /xml-to-html.xsl or JSON output:

Get Results

Click on the **Get Results** button. SPARQLer will go out to retrieve `pets.rdf`, load it, and then proceed to parse it.


SPARQLer Query Results

http://www.sparql.org/sparql?query=PRE

SPARQLer Query Results

age
"10"

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;
      }
   }
}
```

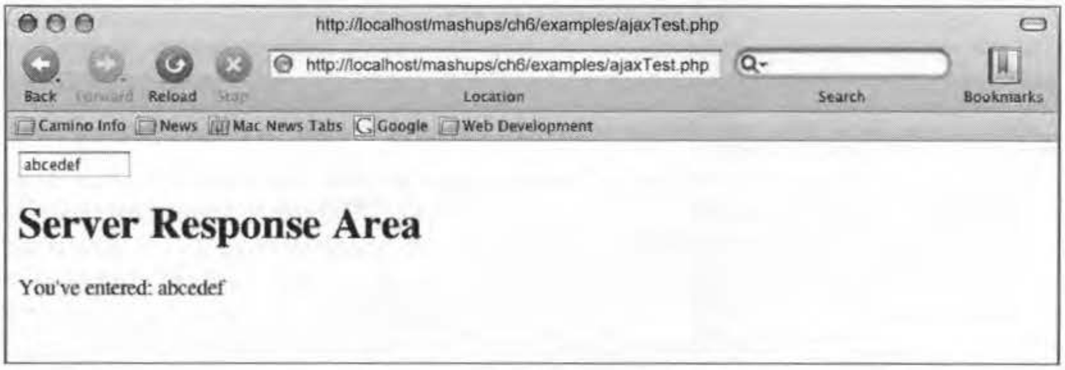
The first line after the nested if statement captures the value in the responseXML property of the XMLHttpRequest object and places it in a variable. This property is where the browser keeps the response from the server. If you were to inspect it, you would see the direct XML from the server.

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.


http://localhost/mashups/ch6/examples/ajaxTest.php
Back
Forward
Reload
Stop
http://localhost/mashups/ch6/examples/ajaxTest.php
Location
Search
Bookmarks
Camino Info
News
Mac News Tabs
Google
Web Development
abcedef

Server Response Area

You've entered: abcedef

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.

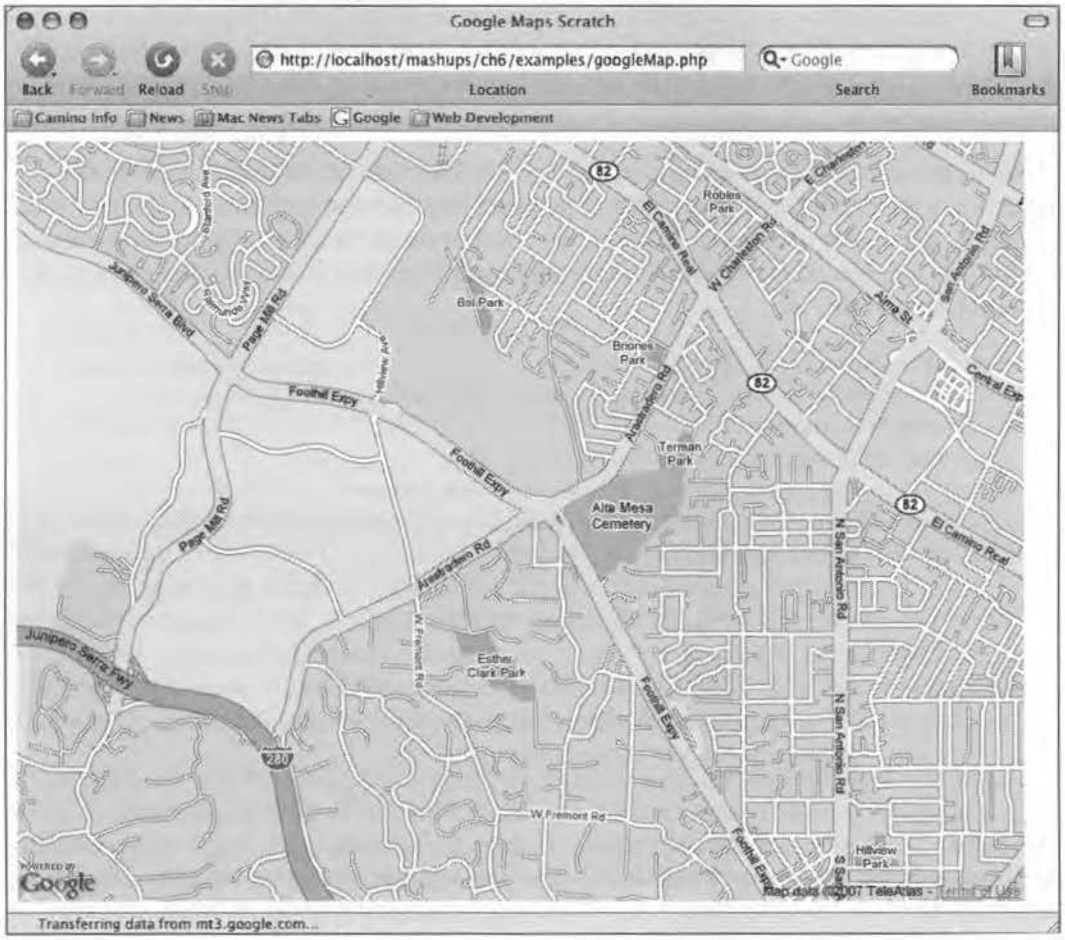
Google's servers pass a `GLatLng` object to the callback function. A `GLatLng` object simply holds latitude and longitude coordinates as properties. If you need to create a `GLatLng` object, there are two parameters you must pass—the latitude and longitude. These properties can be accessed again by using 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.

Foothill Expy
Arastradero Rd
Terman Park
Alta Mesa Cemetery
W Fremont Rd
Esther Clark Park

`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.

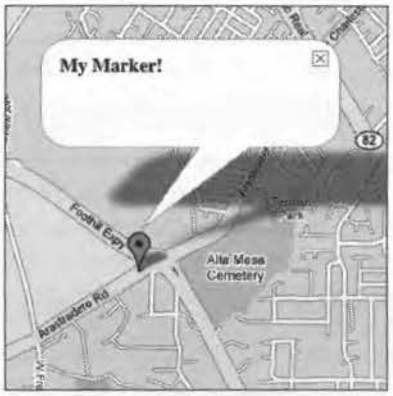
Let's modify our example script further. Instead of getting a JavaScript alert box, let's display an `InfoWindow` box when the user clicks on the marker.

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>");
} );
```



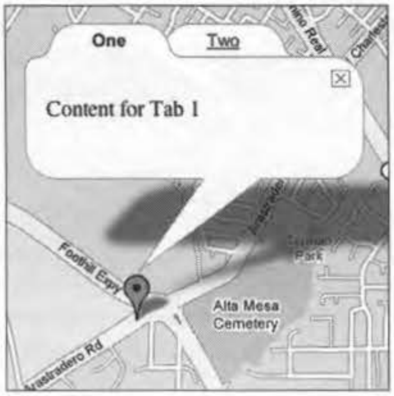
InfoWindow's size is the width and height of the largest HTML container inside. Therefore, you can control the size by adding a height and/or width CSS properties to the enclosing container. For example, you can make a roughly 200 pixels by 300 pixels InfoWindow by putting a div tag that is 200 pixels 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.searchs`' 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.

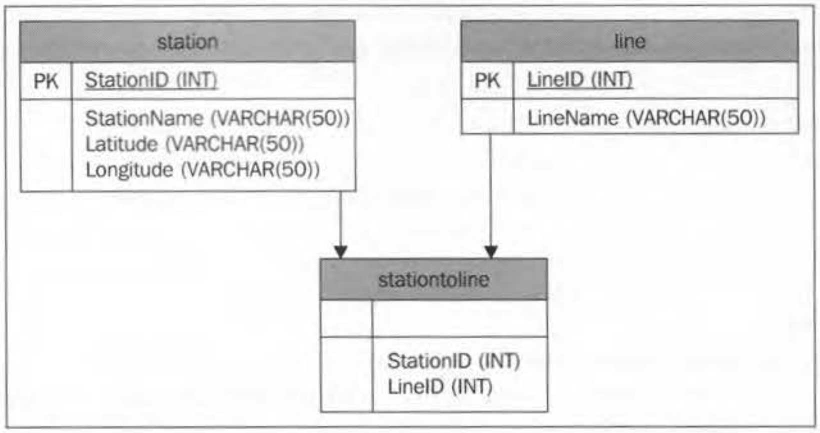
Therefore, we can simply pick these out to get the line stations. As `rdf:subject` elements are the start of the connection chain, we can just pick out the `rdf:subject` and filter by `dc:title` to get all of the stations in a line.

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.

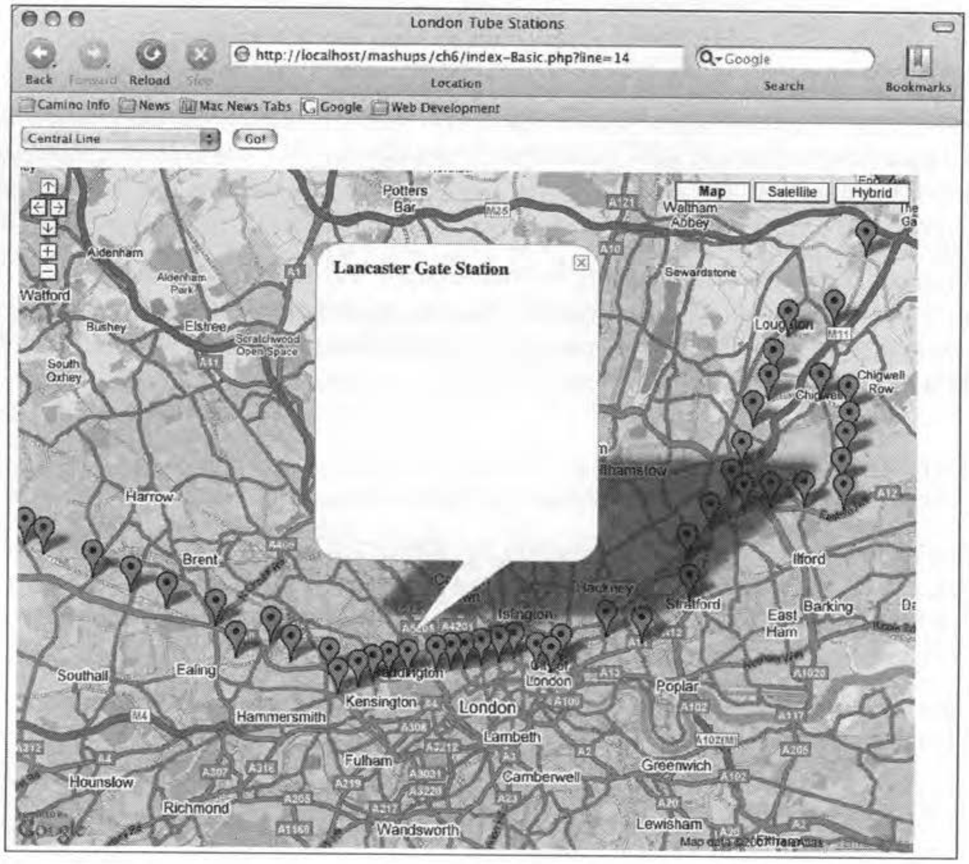
```
<form name="selectionForm" action="index-Basic.php" method="get">
<select name="line">
   <option value="">Select a Line</option>
```

```
<?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**.

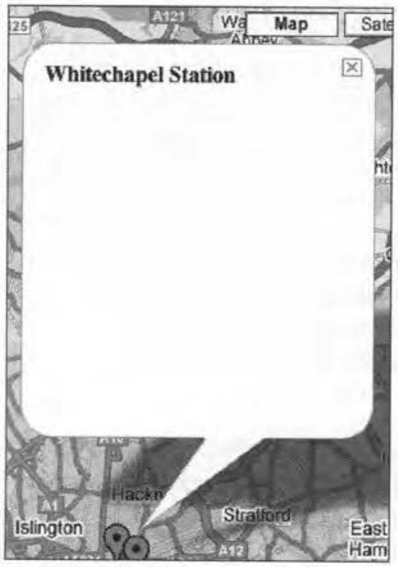
Race conditions are a notion that originated in the electronics design, but has been adopted by software designers. In simple terms, it is when execution of a code happens before a prerequisite is met. This is a constant pitfall in multi-threaded languages like C, C++, and Java. PHP, being a single threaded 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.

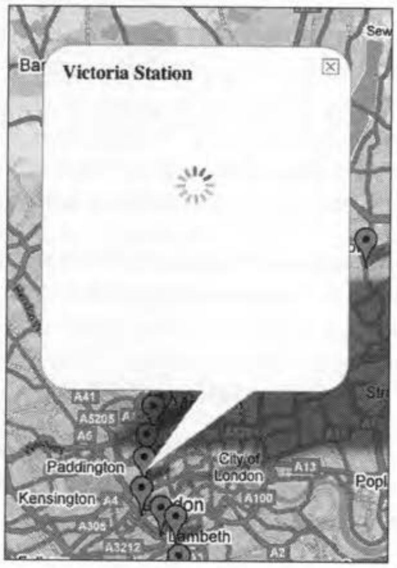
If you copy a file to a location with PHP's `copy()` function, the operating system needs to finish copying before you can work on the copy. Otherwise, operations on the copy will fail. This might not be an issue with 4 kilobyte text files, but imagine moving a 700 megabyte CD image. Even with a fast 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 = "&nbsp;" +
        "<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;
   }
}
```

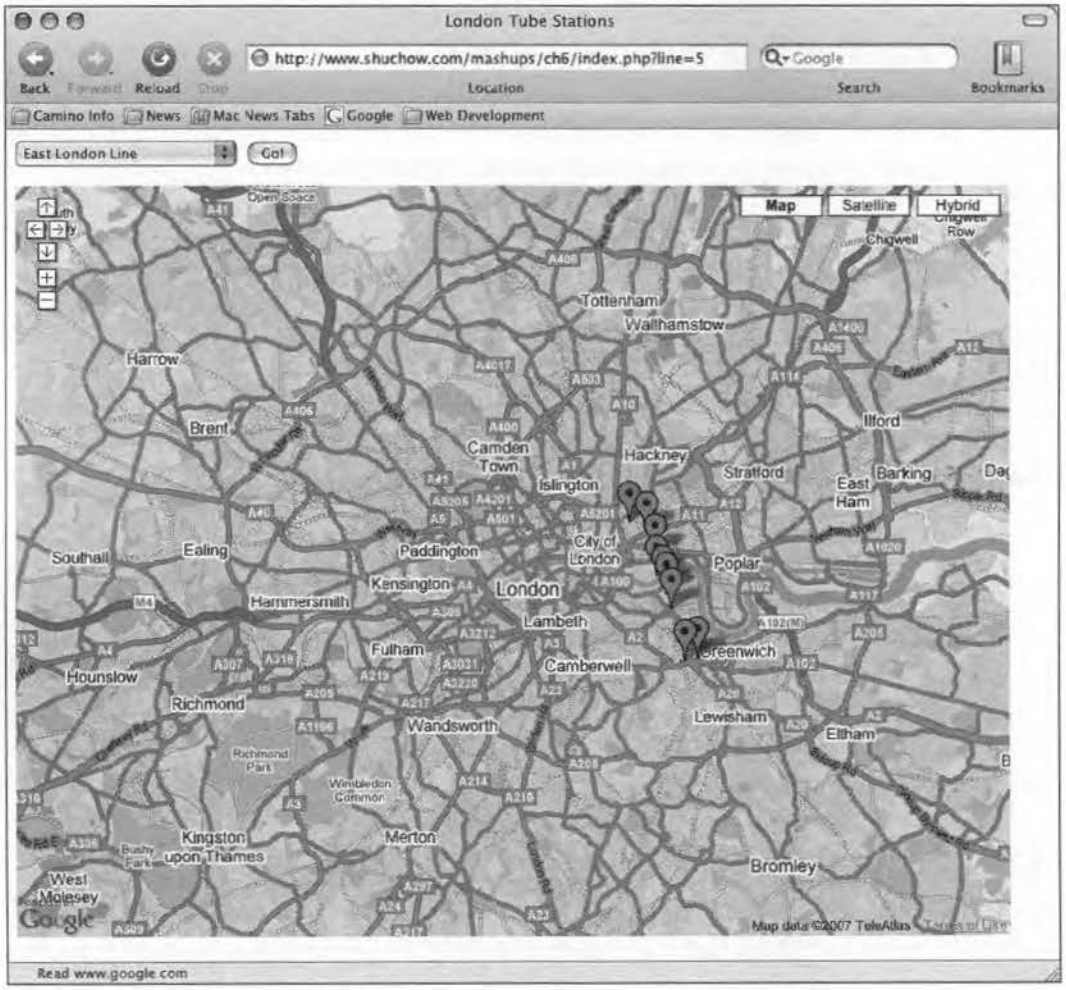
This function is the last function called by the event listener.

```
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();
});
```

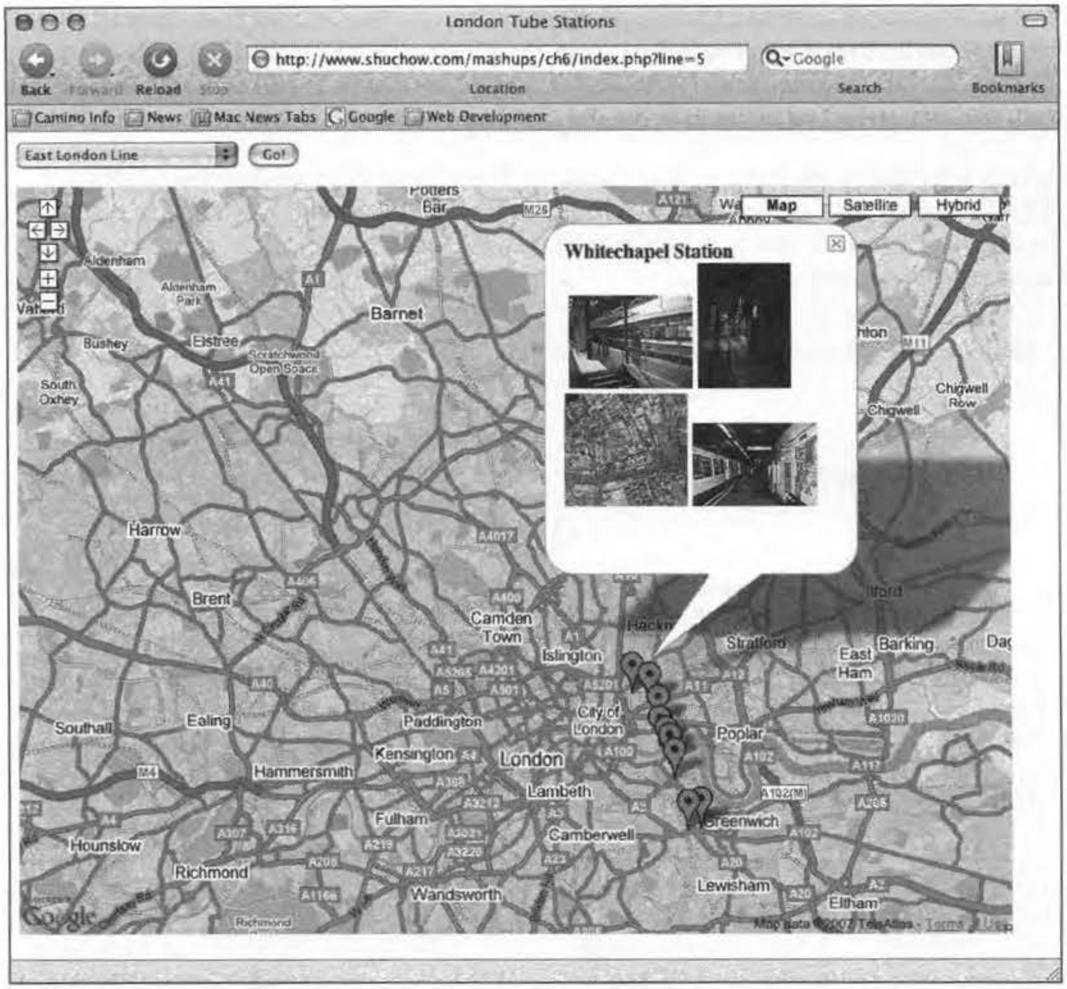
However, remember the service call happens elsewhere. While the information is being retrieved, the window is already there. This is another race condition. If we call g_flickrString before it is set, you will find it is undefined. If g_flickrString is empty, use the setTimeout() JavaScript function 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.

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.

This book is a practical tutorial with five detailed and carefully explained real-world PHP projects. Each project begins with an overview of the technologies and protocols needed for the project, and then dives straight into the tools used and details of creating new and effective mashup applications.

Who this book is written for

If you feel confident with your PHP programming, familiar with the basics of HTML and CSS, unafraid of XML, and interested in mashing things up, this is the book for you!

There are a lot of formats and protocols, web services and web APIs encountered in this book—you do not need to know anything about them or about AJAX; you will find all you need in the book.

What you will learn from this book

- Expand your website and applications using mashups
- Gain a thorough understanding of mashup fundamentals
- Clear, detailed walkthrough of key PHP mashup building technologies
- Five fully-implemented example mashups with full code
- Product lookup on Amazon.Com from their code in the Internet UPC database
- A fully customized search engine with MSN Search and Yahoo!
- A personal video jukebox with YouTube and Last.FM
- Real-time traffic incident data via SMS and the California Highway Patrol!
- Display pictures sourced from Flickr in Google maps

$ 39.99 US
£ 24.99 UK

Prices do not include local sales tax or VAT where applicable

ISBN 978-1-847190-88-8
53999
9 781847 190888

PACKT PUBLISHING

"Community Experience Distilled"

Packt Publishing, Birmingham - Mumbai
www.PacktPub.com