Developer's Guide to Syndicating News and Blogs

Developing Feeds with

RSS *and* Atom

O'REILLY®

Ben Hammersley

O'REILLY®

Developing Feeds with RSS and Atom

RSS and Atom are among the fastest-growing technologies on the Web. The RSS 1.0 and 2.0 standards and the up-and-coming Atom are XML-based formats that help web developers to describe and syndicate web site content, application developers to share data, scripters to repurpose information, businessmen to stay on top of their world, and consumers to tap into the Internet in a whole new way.

Developing Feeds with RSS and Atom offers webloggers, web developers, and the programmers who support them an explanation of syndication in general and RSS and Atom in particular. Written for web developers who want to offer feeds of their content, as well as developers who want to use the content other people are syndicating, the book explores and explains metadata interpretation, different forms of syndication, and the role of Web Services and the Semantic Web.

This comprehensive volume introduces content syndication's purpose, limitations, and traditions, and answers the question of "why would you consider 'giving your content away' like this?" The book delves into the architecture of content syndication with an overview of the entire system, from content author to end user. You'll follow the flow of data: content, referral data, and publish-and-subscribe calls, with a detailed look at the protocols and standards possible at each step. You'll understand the specifications and learn how to use them to their limits.

Topics covered in this book include:

- Creating XML syndication feeds with RSS 2.0
- Producing feeds in the enterprise
- Moving toward the Semantic Web with RSS 1.0 and RDF metadata
- Introducing Atom
- Extending your vocabulary: an extensive guide to RSS modules and their design
- Parsing syndication feeds
- Designing architectures for content syndication

If you're interested in producing your own feeds, using other people's, or keeping up-to-date with the latest web technologies, you'll want this step-by-step guide to RSS and Atom and their implementation.

www.oreilly.com

ISBN 0-596-00881-3 US $39.95 CAN $55.95

9 780596 008819 6 36920 00881 1

Developing Feeds with RSS and Atom

Ben Hammersley

O'REILLY®
Beijing · Cambridge · Farnham · Köln · Paris · Sebastopol · Taipei · Tokyo

Developing Feeds with RSS and Atom
by Ben Hammersley

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor:	Simon St.Laurent
Production Editor:	Mary Anne Weeks Mayo
Cover Designer:	Ellie Volckhausen
Interior Designer:	David Futato

Printing History:

April 2005: First Edition, with content from *Content Syndication with RSS*.

This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN: 0-596-00881-3
[M]

Table of Contents

Preface

This book is about RSS and Atom, the two most popular content-syndication technologies. From distributing the latest web site content to your desktop and powering loosely coupled applications on the Internet, to providing the building blocks of the Semantic Web, these two technologies are among the Internet's fastest growing.

There are millions of RSS and Atom feeds available across the Web today; this book shows you how to read them, how to create your own, and how to build applications that use them. It covers:

- RSS 2.0 and its predecessors
- RSS 1.0 and the Semantic Web
- Atom and the latest generation of feed technology
- How to create and parse feeds
- Extending RSS and Atom through modules
- Using RSS and Atom on the desktop, on the Web, and in the enterprise
- Building RSS- and Atom-based applications

Audience

This book was written with two somewhat interrelated groups in mind:

Web developers and web site authors
: This book should be read by all web developers who want to share their site with others by offering feeds of their content. This group includes everyone from webloggers and amateur journalists to those running large-budget, multiuser sites. Whether you're working on projects for multinational news organizations or neighborhood sports groups, with RSS and Atom, you can extend the reach, power, and utility of your product, and make your life easier and your work more productive. This book shows you how.

Developers

This book is also for developers who want to use the content other people are syndicating and build applications that produce feeds as their output. This group includes everyone from fan-site developers wanting the latest gaming news and intranet builders needing up-to-date financial information on the corporate Web, to developers looking to incorporate news feeds into artificially intelligent systems or build data-sharing applications across platforms. For you, this book delves into the interpretation of metadata, different forms of content syndication, and the increasing use of web services technology in this field. We'll also look at how you can extend the different flavors of RSS and Atom to fit your needs.

Depending on your interests, you may find some chapters more necessary than others. Don't be afraid to skip around or look through the index. There are all kinds of ways to use RSS and Atom.

Assumptions This Book Makes

The technology used in this book is not all that hard to understand, and the concepts specific to RSS and Atom are fully explained. The book assumes some familiarity with HTML and, specifically, XML and its processing techniques, although you will be reminded of important technical points and given places to look for further information. (Appendix A provides a brief introduction to XML if you need one.)

Most of the code in this book is written in Perl, but the examples are commented sufficiently to make things clear and easily portable. There are also some examples in PHP and Ruby. However, users of any language will get a lot from this book: the explanations of the standards and the uses of RSS and Atom are language-agnostic.

How This Book Is Organized

Because RSS and now Atom come in a number of flavors, and there are lots of ways to use them, this book has a lot of parts.

Chapter 1 explains where these things came from and why there is so much diversity in what seems on the surface to be a relatively simple field. Chapter 2 and 3 look at what you can do with RSS and Atom without writing code or getting close to the data. Chapter 2 looks at these technologies from the ordinary user's perspective, showing how to read feeds with a number of tools. Chapter 3 digs deeper into the challenge of creating RSS and Atom feeds, but does so using tools that don't require any programming.

The next four chapters look at the most common varieties of syndication feeds and how to create them. Chapter 4 examines RSS 2.0, inheritor of the 0.91 line of RSS. Chapter 5 looks at RSS 1.0, and its rather different philosophy. Chapter 6 explores

the many modules available to extend RSS 1.0. Chapter 7 looks at a third alternative: the recently emerging Atom specification.

Chapters 8 through 11 focus on issues that developers building and consuming feeds will need to address. Chapter 8 looks at the complex world of parsing these many flavors of feeds, and the challenges of parsing feeds that aren't always quite right. Chapter 9 looks at ways to integrate feeds with publishing models, particularly publish-and-subscribe. Chapter 10 demonstrates a number of applications for feeds that aren't the usual blog entries or news information, and Chapter 11 describes how to extend RSS 2.0 or RSS 1.0 with new modules in case the existing feed structures don't do everything you need.

Finally, there are two appendixes. Appendix A provides a quick tutorial to XML that should give you the foundation you need to work with feeds, while Appendix B provides a list of sites and software you can explore while figuring out how best to apply these technologies to your projects.

Conventions Used in This Book

The following font conventions are used in this book:

Italic is used for:

- Unix pathnames, filenames, and program names
- Internet addresses, such as domain names and URLs
- New terms where they are defined

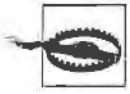
`Constant width` is used for:

- Command lines and options that should be typed verbatim
- Names and keywords in programs, including method names, variable names, and class names
- XML element tags and URIs

`Constant width italic` is used for:

- Replaceable items, such as variables or optional elements, within syntax lines or code

This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Using Code Examples

The examples from this book are freely downloadable from the book's web site at *http://www.oreilly.com/catalog/deveoprssatom*.

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books *does* require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation *does* require permission.

We appreciate, but don't require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Developing Feeds with RSS and Atom*, by Ben Hammersley. Copyright 2005 O'Reilly Media, Inc., 0-596-00881-3."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari Enabled

When you see a Safari® Enabled icon on the cover of your favorite technology book, it means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at *http://safari.oreilly.com*.

Comments and Questions

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (in the United States or Canada)
(707) 829-0515 (international or local)
(707) 829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/deveoprssatom

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our web site at:

http://www.oreilly.com

Acknowledgments

Thanks, as ever, go to my editor Simon St.Laurent and my technical reviewers Roy Owens, Tony Hammond, Timo Hannay, and Ben Lund. Thanks also to Mark Pilgrim, Jonas Galvez, Jorge Velázquez for their lovely code. Bill Kearney, Kevin Hemenway, and Micah Dubinko earned many thanks for their technical-reviewing genius on the first edition. Not to forget Dave Winer, Jeff Barr, James Linden, DJ Adams, Rael Dornfest, Brent Simmons, Chris Croome, Kevin Burton, and Dan Brickley. Cheers to Erhan Erdem, Dan Libby, David Kandasamy, and Castedo Ellerman for their memories of the early days of CDF and RSS and to Yo-Yo Ma for his recording of Bach's Cello Suite No.1, to which much of this book was written.

But most of all, of course, to Anna.

A Short History of RSS and Atom

In the Developer's Bars of the world—those dark, sordid places filled with grizzled coders and their clans—a special corner is always reserved for the developers of content-syndication standards. There, weeping into their beer, you'll find the veterans of a long and difficult process. Most likely, they will have the Thousand Yard Stare of those who have seen more than they should. The standards you will read about in this book were not born fresh and innocent, of a streamlined process overseen by the Wise and Good. Rather, the following chapters have been dragged into the world and tempered through brawls, knife fights, and the occasional riot. What has survived, it is hoped, is hardy enough to prosper for the foreseeable future.

To fully understand these wayward children, and to get the most out of them, it is necessary to understand the motivations behind the different standards and how they evolved into what they are today.

HotSauce: MCF and RDF

The deepest, darkest origins of the current versions of RSS began in 1995 with the work of Ramanathan V. Guha. Known to most simply by his surname, Guha developed a system called the Meta Content Framework (MCF). Rooted in the work of knowledge-representation systems such as CycL, KRL, and KIF, MCF's aim was to describe objects, their attributes, and the relationships between them.

MCF was an experimental research project funded by Apple, so it was pleasing for management that a great application came out of it: ProjectX, later renamed HotSauce. By late 1996, a few hundred sites were creating MCF files that described themselves, and Apple HotSauce allowed users to browse around these MCF representations in 3D. Documentation still exists on the Web for MCF and HotSauce. See *http://www.eclectica-systems.co.uk/complex/hotsauce.php* and Example 1-1 for more.

Example 1-1. An example of MCF

```
begin-headers:
MCFVersion: 0.95
name: "Eclectica"
end-headers:

unit: "tagging.mco"
name: "Tagging and Acrobat Integration"
default_genl_x: -109
default_genl_y: -65
typeOf: #"SubjectCategory"

unit: "http://www.nplum.demon.co.uk/temptin/temptin.htm"
name: "TemptIn Information Management Template"
genls_pos: ["tagging.mco" -85 -137]
```

Example 1-1. An example of MCF (continued)

```
unit: "http://www.nplum.demon.co.uk/temptin/tryout.htm"
name: "Download Try-out Version"
genls_pos: ["tagging.mco" -235 120]
```

It was popular, but experimental, and when Steve Jobs's return to Apple's management in 1997 heralded the end of much of Apple's research activity, Guha left for Netscape.

There, he met with Tim Bray, one of the original XML pioneers, and started moving MCF over to an XML-based format. (XML itself was new at that time.) This project later became the Resource Description Framework (RDF). RDF is, as the World Wide Web Consortium (W3C) RDF Primer says, "a general-purpose language for representing information in the World Wide Web." It is specifically designed for the representation of metadata and the relationships between things. In its fullest form, it is the basis for the concept known as the *Semantic Web*—the W3C's vision wherein computers can understand the meaning of, and the relationships between, documents and other data. You can read *http://en.wikipedia.org/wiki/Semantic_Web* for more details.

Channel Definition Format

In 1997, XML was still in its infancy, and much of the Internet's attention was focused on the increasingly frantic war between Microsoft and Netscape.

Microsoft had been watching the HotSauce experience, and early that year the Internet Explorer development team, along with some others, principally a company called Pointcast, created a system called the Channel Definition Format (CDF).

Released on March 8, 1997, and submitted as a standard to the W3C the very next day, CDF was XML-based and described both the content and a site's particular ratings, scheduling, logos, and metadata. It was introduced in Microsoft's Internet Explorer 4.0 and later into the Windows desktop itself, where it provided the backbone for what was then called Active Desktop. The CDF specification document is still online at *http://www.w3.org/TR/NOTE-CDFsubmit.html*, and Example 1-2 shows a sample.

Example 1-2. An example CDF document

```
<!DOCTYPE Channel SYSTEM "http://www.w3c.org/Channel.dtd" >
<Channel HREF="http://www.foosports.com/foosports.cdf" IsClonable=YES >

<IntroUrl
VALUE="http://www.foosports.com/channel-setup.html" />
<LastMod VALUE="1994.11.05T08:15-0500" />
<Title VALUE="FooSports" />
<Abstract VALUE="The latest in sports and atheletics from FooSports" />
<Author VALUE="FooSports" />
```

Example 1-2. An example CDF document (continued)

```
<Author VALUE="FooSports" />
<LastMod VALUE="1994.11.05T08:15-0500" />

<Schedule>
<StartDate VALUE="1994.11.05T08:15-0500" />
<EndDate VALUE="1994.11.05T08:15-0500" />
<IntervalTime DAY=1 />
<EarliestTime HOUR=12 />
<LatestTime HOUR=18 />
</Schedule>
</Item>

</Channel>
```

Very soon after its release, the potential of a standard, XML-based syndication format became apparent. By April 14, 1997, just over a month since Microsoft gave the standard its first public viewing, Dave Winer's UserLand Software released support for the format into its Frontier product. Written by Wes Felter, and built upon by Dave Winer, it would be the company's first foray into XML-based syndication, but by no means its last. UserLand was to become a major character in our story.

CDF was an exciting technology. It had arrived just as XML was being lauded as the Next Big Thing, and that combination—of a useful technology with a whole new thing to play with—made it rather irresistible for the nascent weblogging community. CDF, however, was really designed for the bigger publishers. A lot of the elements were overkill for the smaller content providers (who, at any rate, didn't consider themselves *content providers* at all), and so a lot of webloggers started to look into creating a simpler specification.

Weblogging?

RSS 1.0, RSS 2.0, and Atom are all deeply entrenched within the weblogging community, and I refer to weblogging, webloggers, and weblogs themselves frequently within this book. If you've never heard of the activity, it is easily explained. A *weblog* is, at heart, a personal web site, consisting of diary-like entries displayed in reverse chronological order. *Weblogging*, or blogging for short, is the activity of writing a weblog, or blog, upon which a weblogger, or blogger, spends his time. Weblogging is extremely popular: at time of writing, in late 2004, there are an estimated four million weblogs being written worldwide. The vast majority of these produce a syndication feed.

For good examples of weblogs, visit *http://www.weblogs.com* for a list of recently updated sites. My own weblog is found at *http://www.benhammersley.com/weblog/index.html*.

O'Reilly has also published a book on weblogging, *Essential Blogging*.

- NewsGator Outlook Edition (*http://www.newsgator.com/outlook.aspx*) is an RSS reader extension for Microsoft Outlook. Many people swear by it, and it features synchronization with an online version for when you're away from your main machine.

Feed-Based Search Engines

For anyone wanting to start a search engine, RSS and Atom are godsent. It's no surprise that a handful of search engines have started up fed solely by syndication feeds. They're usually based around weblogs and the news sites that publish feeds, and can be extraordinarily up to date.

- Feedster (*http://www.feedster.com/*) is the original, and perhaps the best. It claims to index close to one million feeds.
- Medlogs (*http://www.medlogs.com/*) is a good example of feeds powering a specialist search engine. This one aggregates medical news.
- Bulkfeeds (*http://bulkfeeds.net/*) is much like a smaller Feedster, only based in Japan and concentrating on Japanese feeds. It's a fine example of the internationalization of feed technology.

Finding Feeds to Read

Identifying sites that make feeds available can be tricky. There is no standard place to publish a feed, nor is there any particular filename or path to look for one. Of course, there are various methods for sites to identify their feeds, but none are universal. Nevertheless, if you can't see an explicit link to a feed, here's a few things you can try:

- Look for the traditional feed icon, the white writing on an orange background, usually reading "XML" (see Figure 2-7). There are variants on this theme, but they're all recognizable.

Figure 2-7. The garden variety eggplant

- View Source on the site's main page. If you see a line within the head section of the code that reads:

```
<link rel="alternate" type="application/rss+xml" title="RSS" href= "http://www.
example.org/rss.xml"/>
```

  the href part is the URL you want. This is a called an Auto-Discovery link and is discussed in Chapter 9.

The Basic Structure

The top level of an RSS 2.0 document is the `rss version="2.0"` element. This is followed by a single `channel` element. The `channel` element contains the entire feed contents and all associated metadata.

Required Channel Subelements

There are 3 required and 16 optional subelements of `channel` within RSS 2.0. Here are the required subelements:

`title`
: The name of the feed. In most cases, this is the same name as the associated web site or service.

```
<title>RSS and Atom</title>
```

`link`
: A URL pointing to the associated resource, usually a web site. The link must be an IANA-registered URI scheme, such as http://, https://, news://, or ftp://, though it isn't necessary for a application developer to support all these by default. The most common by a large margin is http://. For example:

```
<link>http://www.benhammersley.com</link>
```

`description`
: Some words to describe your channel.

```
<description>This is a nice RSS 2.0 feed of an even nicer weblog</description>
```

Although it isn't explicitly stated in the specification, it is highly recommended that you *do not* put anything other than plain text in the `channel/title` or `channel/description` elements. There are some existing feeds with HTML within those elements, but these cause a considerable amount of wailing, and at least a small amount of gnashing of teeth. *Do not do it.* Use plain text only in these elements. The following sidebar, "Including HTML Within title or description," gives a fuller account of this, but in my opinion it's a bad idea.

Optional Channel Subelements

There are 16 optional channel subelements of RSS 2.0. Technically speaking, you can leave these out altogether. However, I encourage you to add as many as you can. Much of this stuff is static; the content of the element never changes. Placing it into your RSS template or adding another line to a script is little work for the additional value of your feed's metadata. This is especially true for the first three subelements listed here:

Including HTML Within title or description

Since the early days of RSS 0.91, there's been an ongoing debate about whether the `item/title` or `item/description` elements may, or should, contain HTML. In my opinion, they should not, for both practical and philosophical reasons. Practically speaking, including HTML markup requires the client software to be able to parse or filter it. While this is fine with many desktop agents, it restricts developers looking for other uses of the data. This brings us to the philosophical aspect. RSS's second use, after providing headlines and content to desktop readers and sites, is to provide indexable metadata. By combining presentation and content (i.e., by including HTML markup within the `description` element), you could disable this feature.

However, my opinion lost out on this one. RSS 2.0 now allows for entity-encoded HTML within the `item/description` tag. It doesn't mention anything, in either direction, regarding `item/title`, and people are basically making it up as they go along. With that in mind, I still state that `item/title` at least should be considered plain text.

If you want to put HTML within the `item/description` element, you can do it in two ways:

Entity encoding

With entity encoding, the angle brackets of HTML tags are converted to their respective HTML entities, `<` and `>`. If you need to show angle brackets as literal characters, the ampersand character itself should be encoded as well:

```
This is a &lt;em&gt;lovely left angle bracket:&lt;/em&gt; &amp;lt;
```

Within a CDATA block

The alternative is to enclose the HTML within a CDATA block. This removes one level of entity encoding, as in:

```
<![CDATA[This is a <em>lovely left angle bracket:</em> &lt;]]>
```

Either approach is acceptable according to the specification, and there is no way for a program to tell the difference between the two, or to tell if the description is actually just plain text that *resembles* encoded HTML. This is a major problem with the RSS 2.0 specification, as you'll see when we talk about parsing feeds. Atom and RSS 1.0 both have their own ways around this issue.

`language`

The language the feed is written in. This allows aggregators to index feeds by language and should contain the standard Internet language codes as per RFC 1766.

```
<language>en-US</language>
```

`copyright`

A copyright notice for the content in the feed:

```
<copyright>Copyright 2004 Ben Hammersley</copyright>
```

managingEditor
: The email address of the person to contact for editorial enquiries. It should be in the format: *name @example.com (FirstName LastName)*.

```
<managingEditor>ben@benhammersley.com (Ben Hammersley)</managingEditor>
```

webMaster
: The email address of the person responsible for technical issues with the feed:

```
<webMaster>techsupport@benhammersley.com (Geek McNerdy)</webMaster>
```

pubDate
: The publication date of the content within the feed. For example, a daily morning newspaper publishes at a certain time early every morning. Technically, any information in the feed should not be displayed until after the publication date, so you can set pubDate to a time in the future and expect that the feed won't be displayed until after that time. Few existing RSS readers take any notice of this element in this way, however. Nevertheless, it should be in the format outlined in RFC 822:

```
<pubDate>Sun, 12 Sep 2004 19:00:40 GMT</pubDate>
```

lastBuildDate
: The date and time, RFC 822–style, when the feed last changed. Note the difference between this and channel/pubDate. lastBuildDate must be in the past. It is this element that feed applications should take as the "last time updated" value and not channel/pubDate.

```
<pubDate>Sun, 12 Sep 2004 19:01:55 GMT</pubDate>
```

category
: Identical in syntax to the item/category element you'll see later. This takes one optional attribute, domain. The value of category should be a forward-slash-separated string that identifies a hierarchical location in a taxonomy represented by the domain attribute. Sadly, there is no consensus either within the specification or in the real world as to any standard format for the domain attribute. It would seem most sensible to restrict it to a URL; however, it needn't necessarily be so.

```
<category domain="Syndic8">1765</category>
```

generator
: This should contain a string indicating which program created the RSS file:

```
<generator>Movable Type v3.1b3</generator>
```

docs
: A URL that points to an explanation of the standard for future reference. This should point to *http://blogs.law.harvard.edu/tech/rss*:

```
<docs>http://blogs.law.harvard.edu/tech/rss</docs>
```

`cloud`

The `<cloud/>` element enables a rarely used feature known as "Publish and Subscribe," which we shall investigate fully in Chapter 9. It takes no value itself, but it has five mandatory attributes, themselves also explained in Chapter 9: `domain`, `path`, `port`, `registerProcedure`, and `protocol`.

```
<cloud domain="rpc.sys.com" port="80" path="/RPC2" registerProcedure= "pingMe"
protocol="soap"/>
```

`ttl`

ttl, short for Time-to-Live, should contain a number, which is the minimum number of minutes the reader should wait before refreshing the feed from its source. Feed authors should adjust this figure to reflect the time between updates and the number of times they wish their feed to be requested, versus how up to date they need their consumers to be.

```
<ttl>60</ttl>
```

`image`

This describes a feed's accompanying image. It's optional, but many aggregators look prettier if you include one. It has three required and two optional subelements of its own:

`url`

The URL of a GIF, JPG, or PNG image that corresponds to the feed. It is, quite obviously, required.

`title`

A description of the image, normally used within the ALT attribute of HTML's `<img>` tag. It is required.

`link`

The URL to which the image should be linked. This is usually the same as the `channel/link`.

`width` *and* `height`

The width and height of the icon, in pixels. The icons should be a maximum of 144 pixels wide by 400 pixels high. The emergent standard is 88 pixels wide by 31 pixels high. Both elements are optional.

```
<image> <title>RSS2.0 Example</title> <url>http://www.exampleurl.com/example/
images/logo.gif</url> <link>http://www.exampleurl.com/example/index.html</link>
<width>88</width> <height>31</height> <description>The World's Leading Technical
Publisher</description> </image>
```

`rating`

The PICS rating for the feed; it helps parents and teachers control what children access on the Internet. More information on PICS can be found at *http://www.w3.org/PICS/*. This labeling scheme is little used at present, but an example of a PICS rating would be:

```
<rating>(PICS-1.1 "http://www.gcf.org/v2.5" labels on "1994.11.05T08:15-0500"
until "1995.12.31T23:59-0000" for "http://w3.org/PICS/Overview.html" ratings
(suds 0.5 density 0 color/hue 1))</rating>
```

`textInput`

An element that lets RSS feeds display a small text box and Submit button, and associates them with a CGI application. Many RSS parsers support this feature, and many sites use it to offer archive searching or email newsletter sign-ups, for example. `textInput` has four required subelements:

`title`

The label for the Submit button. It can have a maximum of 100 characters.

`description`

Text to explain what the `textInput` actually does. It can have a maximum of 500 characters.

`name`

The name of the text object that is passed to the CGI script. It can have a maximum of 20 characters.

`link`

The URL of the CGI script.

```
<textInput> <title>Search</title> <description>Search the Archives</
description> <name>query</name> <link>http://www.exampleurl.com/example/
search.cgi</link> </textInput>
```

`skipDays` *and* `skipHours`

A set of elements that can control when a feed user reads the feed. `skipDays` can contain up to seven `day` subelements: Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, or Sunday. `skipHours` contains up to 24 `hour` subelements, the numbers 1–24, representing the time in Greenwich Mean Time (GMT). The client should not retrieve the feed during any day or hour listed within these two elements. The elements are ORed not ANDed: in the example here, the application is instructed not to request the feed during 8 p.m. on any day, and never on a Monday:

```
<skipDays><day>Monday</day></skipDays> <skipHours><hour>20</hour></skipHours>
```

item Elements

RSS 2.0 can have any number of `item` elements. The `item` element is at the heart of RSS; it contains the primary content of the feed. Technically, `item` elements are optional, but a syndication feed with no `item`s is just a glorified link. Not having any `item`s doesn't mean the feed is invalid, just extremely boring.

All `item` subelements are optional, with the proviso that at least one of `item/title` or `item/description` is present. You can use this feature to build lists (more on that later).

With `item`, there are the 10 standard `item` subelements available:

`title`

Usually, this is the title of the story linked to by the `item`, but it can also be seen as a one-line list item. There is controversy over whether HTML is allowed within this element; for more information, see the sidebar "Including HTML Within title or description."

`link`

The URL of the story the item is describing.

`description`

A synopsis of the story. The `description` can contain entity-encoded HTML. Again, as with `item/title`, see the pertinent sidebar "Including HTML Within title or description."

`author`

This should contain the email address of the resource's author referred to within the `item`. The specification's example is in the format `user@example.com (firstname lastname)` but isn't explained further:

```
<author>ben@benhammersley.com (Ben Hammersley)</author>
```

`category`

Exactly the same as `channel/category`, but it pertains to the individual `item` only:

```
<category domain="the twisted passages of my mind">up/to_the_left/there</category>
```

`comments`

This should contain the URL of any comments page for the item; it's primarily used with weblogs:

```
http://www.example.com/comments.cgi?post=12345
```

`enclosure`

This describes a file associated with an `item`. It has no content, but it takes three attributes: `url` is the URL of the enclosure, `length` is its size in bytes, and `type` is the standard MIME type for the enclosure. Some feed applications can download these files automatically. The original idea was for configuring a feed aggregator to automatically download large media files overnight, thereby deferring the extra bandwidth required. This is an underused feature of RSS 2.0 because most aggregators don't support it, but in 2004, it became the focus of a lot of development around the idea of podcasting. See the sidebar "Including HTML Within title or description" for details.

```
<enclosure url="http://www.example.com/hotxxxpron.mpg" length= "34657834"
type="video/mpeg"/>
```

`guid`

Standing for Globally Unique Identifier, this element should contain a string that uniquely identifies the item. It must never change, and it must be unique to the object it is describing. If that content changes in any way, it must gain a new guid. This element also has the optional attribute `isPermalink`, which, if `true`, denotes that the value of the element can be taken as a URL to the object

referred to by the item. Therefore, if no item/link element is present, but the isPermalink attribute is set to true, the application can take the value of guid in its place. The specification doesn't say what to do if both are present and aren't the same, but it seems sensible to give preference within any application to the item/link element.

```
<guid isPermalink="true">http://www.example.com/example.html</guid>
```

pubDate

The publication date of the item. Again, as with channel/pubDate, any information in the item shouldn't be displayed until after the publication date, but few existing RSS readers take any notice of this element in this way. The date is in RFC 822 format.

```
<pubDate>Mon, 13 Sep 2004 00:23:05 GMT</pubDate>
```

source

This should contain the name of the feed of the site from which the item was derived, and the attribute url should be the URL of that other site's feed:

```
<source url="http://www.metafilter.com/rss.xml">Metafilter</source>
```

Example 4-1 shows these parts assembled into an RSS 2.0 XML document.

Example 4-1. An example RSS 2.0 feed

```
<?xml version="1.0"?>
<rss version="2.0">
<channel>
  <title>RSS2.0Example</title>
  <link>http://www.exampleurl.com/example/index.html</link>
  <description>This is an example RSS 2.0 feed</description>
  <language>en-gb</language>
  <copyright>Copyright 2002, Oreilly and Associates.</copyright>
  <managingEditor>example@exampleurl.com</managingEditor>
  <webMaster>webmaster@exampleurl.com</webMaster>
  <rating> </rating>
  <pubDate>03 Apr 02 1500 GMT</pubDate>
  <lastBuildDate>03 Apr 02 1500 GMT</lastBuildDate>
  <docs>http://blogs.law.harvard.edu/tech/rss</docs>
  <skipDays><day>Monday</day></skipDays>
  <skipHours><hour>20</hour></skipHours>
  <category  domain="http://www.dmoz.org">Business/Industries/Publishing/Publishers/
Nonfiction/Business/O'Reilly_and_Associates/</category>
  <generator>NewsAggregator'o'Matic</generator>
  <ttl>30<ttl>
  <cloud domain="http://www.exampleurl.com" port="80" path="/RPC2"
registerProcedure="pleaseNotify" protocol="XML-RPC" />

  <image>
    <title>RSS2.0 Example</title>
    <url>http://www.exampleurl.com/example/images/logo.gif</url>
    <link>http://www.exampleurl.com/example/index.html</link>
    <width>88</width>
```

Example 4-1. An example RSS 2.0 feed (continued)

```
    <height>31</height>
    <description>The World's Leading Technical Publisher</description>
  </image>

  <textInput>
    <title>Search</title>
    <description>Search the Archives</description>
    <name>query</name>
    <link>http://www.exampleurl.com/example/search.cgi</link>
  </textInput>

  <item>
    <title>The First Item</title>
    <link>http://www.exampleurl.com/example/001.html</link>
    <description>This is the first item.</description>
    <source url="http://www.anothersite.com/index.xml">Another Site</source>
    <enclosure url="http://www.exampleurl.com/example/001.mp3" length="543210"
                      type"audio/mpeg"/>
    <category domain="http://www.dmoz.org">Business/Industries/Publishing/Publishers/
Nonfiction/Business/O'Reilly_and_Associates/</category>
    <comments>http://www.exampleurl.com/comments/001.html</comments>
    <author>Ben Hammersley</author>
    <pubDate>Sat, 01 Jan 2002 0:00:01 GMT</pubDate>
    <guid isPermaLink="true">http://www.exampleurl.com/example/001.html</guid>
  </item>

  <item>
    <title>The Second Item</title>
    <link>http://www.exampleurl.com/example/002.html</link>
    <description>This is the second item.</description>
    <source url="http://www.anothersite.com/index.xml">Another Site</source>
    <enclosure url="http://www.exampleurl.com/example/002.mp3" length="543210"
                            type"audio/mpeg"/>
    <category domain="http://www.dmoz.org">Business/Industries/Publishing/Publishers/
Nonfiction/Business/O'Reilly_and_Associates/</category>
    <comments>http://www.exampleurl.com/comments/002.html</comments>
    <author>Ben Hammersley</author>
    <pubDate>Sun, 02 Jan 2002 0:00:01 GMT</pubDate>
    <guid isPermaLink="true">http://www.exampleurl.com/example/002.html</guid>
  </item>

  <item>
    <title>The Third Item</title>
    <link>http://www.exampleurl.com/example/003.html</link>
    <description>This is the third item.</description>
    <source url="http://www.anothersite.com/index.xml">Another Site</source>
    <enclosure url="http://www.exampleurl.com/example/003.mp3" length="543210"
                        type"audio/mpeg"/>
    <category domain="http://www.dmoz.org">Business/Industries/Publishing/Publishers/
Nonfiction/Business/O'Reilly_and_Associates/</category>
    <comments>http://www.exampleurl.com/comments/003.html</comments>
    <author>Ben Hammersley</author>
```

One key difference between the development of RSS and the development of Atom is that Atom's whole design process is held out in the open, on the Atom-Syntax mailing list just mentioned and on the Atom wiki. The wiki (*http://www.intertwingly.net/wiki/pie/FrontPage*) is a great place to find the latest developments, issues, ideas, and pointers to the latest specification documents. It is well worth exploring, once you've finished reading.

The Structure of an Atom Feed

Because Atom is really two standards, one for syndication and one for the remote retrieval, creation, and editing of online resources (or, to put it more simply, a weblog API), an Atom document is deeply structured. The syndication format, our bailiwick here, defines two document formats: the Atom feed and the Atom entry.

An Atom feed is made up of none or more Atom entries (although it's most probably going to have at least one entry), plus some additional metadata. An Atom entry is just that: one single indivisible piece of "content." This single indivisible piece of content is what gives Atom its name and is the key to understanding the whole Atom project.

The Atom entry

Before we get too excited, let's look at an Atom entry document (Example 7-1).

Example 7-1. An Atom entry document

```
<?xml version="1.0" encoding="utf-8"?>
<entry version="draft-ietf-atompub-format-05: do not deploy"
 xmlns="http://purl.org/atom/ns#draft-ietf-atompub-format-05">

<title>Example Entry Document</title>

<link
rel="alternate"
type="text/html"
href="http://example.org/example_entry"
hreflang="en"
title="Example Entry Document"
/>

<edit href="http://example.org/edit?title=example_entry">

<author>
        <name>Ben Hammersley</name>
        <uri>http://www.benhammersley.com</uri>
        <email>ben@benhammersley.com</email>
</author>

<contributor>
        <name>Albert Einstein</name>
```

Note that this configuration defines the directory's index page as *index.shtml* because it isn't a good idea to make your browser seek out SSI directives in every page it serves. Rather, you should tell it to look for SSI directives solely in pages that end with a certain file extension by adding the following lines to your *httpd.conf* file

```
AddType text/html .shtml
AddHandler server-parsed .shtml
```

This makes Apache search any file ending in *.shtml* (the traditional extension for such things) for SSI directives and replace them with their associated files before serving them to the end user.

This approach has a disadvantage: if you want to add SSI directives to an existing page, you have to change the name of that page. All links to that page will therefore be broken in order to get the SSI directives to work. So, if you're retrofitting a site with RSS, the other method is to use the `XBitHack` directive within your *httpd.conf* file:

```
XBitHack on
```

`XBitHack` tells Apache to parse files for SSI directives if the files have the execute bit set. So, to add SSI directives to an existing page, rather than having to change the filename, you just need to make the file executable using `chmod`.

How Often to Read the Feed

Feeds do change, it is true. People update their sites at all hours, and it would be lovely to have the very latest headlines. Currently, however, it isn't a good idea to keep requesting a new feed every few minutes. Etiquette and convention limit requests for a new file to once every 60 minutes, unless the feed's publisher specifically allows you to grab it more often, or unless said publisher is using Publish and Subscribe (see Chapter 9).

In many cases, even requesting the feed every hour is too much. Feeds that change only once a day require downloading only once a day. It's a simple courtesy to pay attention to these conventions.

Server-Side Includes with Microsoft IIS

Microsoft's Internet Information Services (IIS) server package comes with server-side includes enabled: by default, it processes any file ending in *.stm*, *.shtm*, or *.shtml*. However, files are processed only if they're inside directories with Scripts or Execute access permissions.

CHAPTER 10

Unconventional Feeds

Equation (1.2-9) is a second order, nonlinear, vector, differential equation which has defied solution in its present form. It is here therefore we depart from the realities of nature to make some simplifying assumptions...

—Roger R. Bate, Jerry E. White, and Donald Mueller
Fundamentals of Astrodynamics, 1971

By now you should be confident in your understanding of both RSS and Atom, how the standards work, how to parse them, and how to fit them into the rest of the Internet. Before we finish this book with an explanation of how the standards can be extended, we have a good opportunity to show our knowledge in action. This chapter, therefore, is a collection of recipes for creating and using feeds that do things slightly more interesting and useful than simply transporting headlines and articles.

Apache Logfiles

Like many people, I have a weblog and quite a considerable amount of other online writings sitting on my own web server. However once I've written something and it's been indexed by the search engines, people tend to find it, and I feel a vague social responsibility to keep it up there. But things being as they are, and me tinkering as I do, documents go missing. Keeping track of the "404 Page Not Found" errors spit out by your server is, therefore, a good thing to do.

This script, therefore, goes through a standard Apache logfile and produces an RSS 2.0 feed of pages other people have found to be missing.

Walking Through the Code

Let's start with the usual Perl good form of `strict;` and `warnings;` and then load up the marvellous `Date::Manip` module. This is perhaps a little overkill for its use here, but it does allow for some extremely simple and readable code. This is a CGI

CHAPTER 10

Unconventional Feeds

Equation (1.2-9) is a second order, nonlinear, vector, differential equation which has defied solution in its present form. It is here therefore we depart from the realities of nature to make some simplifying assumptions...

—Roger R. Bate, Jerry E. White, and Donald Mueller
Fundamentals of Astrodynamics, 1971

By now you should be confident in your understanding of both RSS and Atom, how the standards work, how to parse them, and how to fit them into the rest of the Internet. Before we finish this book with an explanation of how the standards can be extended, we have a good opportunity to show our knowledge in action. This chapter, therefore, is a collection of recipes for creating and using feeds that do things slightly more interesting and useful than simply transporting headlines and articles.

Apache Logfiles

Like many people, I have a weblog and quite a considerable amount of other online writings sitting on my own web server. However once I've written something and it's been indexed by the search engines, people tend to find it, and I feel a vague social responsibility to keep it up there. But things being as they are, and me tinkering as I do, documents go missing. Keeping track of the "404 Page Not Found" errors spit out by your server is, therefore, a good thing to do.

This script, therefore, goes through a standard Apache logfile and produces an RSS 2.0 feed of pages other people have found to be missing.

Walking Through the Code

Let's start with the usual Perl good form of `strict;` and `warnings;` and then load up the marvellous `Date::Manip` module. This is perhaps a little overkill for its use here, but it does allow for some extremely simple and readable code. This is a CGI

application, so we need that module, and we're producing RSS, so XML::RSS is naturally required too.

```
use strict;
use warnings;
use Date::Manip;
use XML::RSS;
use CGI qw(:standard);
```

First off, let's set up the feed. Because this is the simplest possible form of RSS— just a list, really—it is a perfect fit for RSS 2.0. Then we give it a nice title, link and description, as per the specification:

```
my $rss = new XML::RSS( version => '2.0' );
$rss->channel(
    title       => "Missing Files",
    link        => "http://www.example.org",
    description => "Files found to be missing from my server"
);
```

On my host at least, logfiles are split daily and named after the date. Since I'm fixing the missing files as they appear, I only want to parse yesterday's file. So let's use Date::Manip's functions to return yesterday's date, then create the logfile path, and open a filehandle to it. You will need to change this line to reflect your own setup:

```
my $yesterdays_date = &UnixDate( "yesterday", "%Y%m%d" );
my $logfile_file = "/web/logs/ben/benhammersley.com/$yesterdays_date.log";
open( LOGFILE, "< $logfile_file" );
```

Now, go into a loop, taking the logfile line by line and using the mother of regular expressions to split it up into its requisite parts. This is a very useful line to take note of: you can change this section to convert this script to monitor your Apache logfiles for just about anything.

```
while (<LOGFILE>) {
    my (
        $host,      $ident_user, $auth_user, $date,     $time,
        $time_zone, $method,     $url,       $protocol, $status,
        $bytes,     $referer,    $agent
      )
      = /^(\S+) (\S+) (\S+) \[([^:]+):(\d+:\d+:\d+) ([^\]]+)\] "(\S+) (.+?) (\S+)" (\
S+) (\S+) "([^"]+)"
"([^"]+)"$/;
```

But today, we're interested in 404 errors. So, the script uses the XML::RSS module to add items for every error found:

```
my $cleaned_status = $status || "111";

if ( $cleaned_status == "404" ) {
    $rss->add_item(
        title       => "$url",
        link        => "$url",
        description => "$referer"
```

```
        );
    }
}
```

Then all that's left to do is to close the filehandle, print out the correct MIME type for RSS, and print out the feed we've built:

```
close(LOGFILE);
print header('application/xml+rss');
print $rss->as_string;
```

The Entire Listing

```
#!/usr/bin/perl
use strict;
use warnings;
use Date::Manip;
use XML::RSS;
use CGI qw(:standard);

my $rss = new XML::RSS( version => '2.0' );
$rss->channel(
    title       => "Missing Files",
    link        => "http://www.example.org",
    description => "Files found to be missing from my server"
);

my $yesterdays_date = &UnixDate( "yesterday", "%Y%m%d" );
my $logfile_file = "/web/logs/ben/benhammersley.com/$yesterdays_date.log";
open( LOGFILE, "< $logfile_file" );

while (<LOGFILE>) {
    my (
        $host,      $ident_user, $auth_user, $date,     $time,
        $time_zone, $method,     $url,       $protocol, $status,
        $bytes,     $referer,    $agent
      )
      = /^(\S+) (\S+) (\S+) \[([^:]+):(\d+:\d+:\d+) ([^\]]+)\] "(\S+) (.+?) (\S+)" (\
S+) (\S+) "([^"]+)"
"([^"]+)"$/;

    my $cleaned_status = $status || "111";

    if ( $cleaned_status == "404" ) {
        $rss->add_item(
            title       => "$url",
            link        => "$url",
            description => "$referer"
        );
    }
}

close(LOGFILE);
print header('application/xml+rss');
print $rss->as_string;
```

Code TODOs to RSS

I live by my to-do list. Without it, I'd be a drippy mess of procrastination, and you wouldn't be reading this book right now. It's good to be organized, and especially so with code. Working on the scripts and chapters in this book, I've started to leave messages to myself within them. I mark these out as a new line starting with `TODO:`.

I then have this script run over all of my working directories, parsing each file and looking for those lines. The result is a nice to-do list in my reader application and a morning's work all set out.

Walking Through the Code

We'll do the usual start, with `strict;`, `warnings;`, `CGI` and `XML::RSS`. Let's use the `File::Find` module to do the directory-traversing:

```
use strict;
use warnings;
use XML::RSS;
use CGI qw(:standard);
use File::Find;
```

Up near the top, we set the root directory, below which everything will be parsed. You need to change this to match your own setup, but remember that the script checks every file. Setting it to something too high up your filesystem tree might slow the script down magnificently.

```
my $start_directory = "/Users/ben/Code/";
```

Now, set up the RSS feed, as per usual. Here, we're using a cunning trick, technically against the RSS 2.0 specification, of making the `link` element a `file://` URL. This usually works very well, bringing up your OS's file manager, or the application associated with that particular file, but remember that technically, it's not supposed to work at all. Still, for a locally produced, locally consumed feed, a little slackness on the specification is probably allowed.

```
my $rss = new XML::RSS( version => '2.0' );
$rss->channel(
    title       => "TODOs from $start_directory",
    link        => "file:/$start_directory",
    description => "Files with TODO messages, from $start_directory"
);
```

Now, use the `File::Find` module to traverse the directories, opening a filehandle for each one, slurping it in and searching each line with a regular expression for `TODO`. If it's found, add an `item` to the RSS feed.

```
find( \&search_and_rss, $start_directory );

sub search_and_rss {
```

```
    open( CODEFILE, "< $File::Find::name" );

    while (<CODEFILE>) {
        if ( $_ =~ m/TODO/ ) {
            $rss->add_item(
                title       => "$File::Find::name",
                link        => "file:\/\/$File::Find::name",
                description => "$_ at Line $. of $File::Find::name"
            );
```

As a test for the script, I've left a TODO line within itself. It's just a suggestion that a section of code to add the file's last modification date to the RSS feed might be a nice idea; I leave this as an exercise for the reader. Now, close the filehandle:

```
            #TODO: Put in date code here

        }
    }
    close(CODEFILE);
}
```

And finally, we serve up the feed:

```
print header('application/xml+rss');
print $rss->as_string;
```

The Entire Listing

```
#!/usr/bin/perl

use strict;
use warnings;
use XML::RSS;
use CGI qw(:standard);
use File::Find;

my $start_directory = "/Users/ben/Code/";

my $rss = new XML::RSS( version => '2.0' );
$rss->channel(
    title       => "TODOs from $start_directory",
    link        => "file:/$start_directory",
    description => "Files with TODO messages, from $start_directory"
);

find( \&search_and_rss, $start_directory );

sub search_and_rss {

    open( CODEFILE, "< $File::Find::name" );

    while (<CODEFILE>) {
        if ( $_ =~ m/TODO/ ) {
            $rss->add_item(
```

```
                title       => "$File::Find::name",
                link        => "file:\/\/$File::Find::name",
                description => "$_ at Line $. of $File::Find::name"
            );

            #TODO: Put in date code here
        }
    }
    close(CODEFILE);
}

print header('application/xml+rss');
print $rss->as_string;
```

Daily Doonesbury

The morning isn't quite the same without the daily dose of *Doonesbury*, the Pulitzer Prize winning cartoon strip by Garry Trudeau. But, frankly, the nearest newspaper stand is far too far to go before I've had morning coffee and typing in *http://www.doonesbury.com* on an empty stomach is a bit much. Why not have it delivered via RSS? You just need a web server, or a proper operating system capable of running CGI scripts, and the following code. This is tremendously simple stuff—but distinctly pleasing to have set up for your morning coffee.

Walking Through the Code

We'll go with the traditional start: `strict;`, `warnings;`, `CGI` with `DATE::Manip`, and the joy that is `XML::RSS` all being loaded for your coding pleasure. So far, so simple.

```
use strict;
use warnings;
use CGI qw(:standard);
use Date::Manip;
use XML::RSS;
```

The *Doonesbury* site, thankfully, names its image files after the date, in the format YYMMDD. So, first, you work out the date, then add a single entry containing escaped HTML that displays the image file you want.

```
my $todays_date = &UnixDate( "today", "%y%m%d" );
my $this_year = &UnixDate("today","%Y");

my $rss = new XML::RSS( version => '2.0' );

$rss->add_item(
    title => "Doonesbury for $todays_date",
    link  => "http://images.ucomics.com/comics/db/$this_year/db$todays_date.gif",
    description => '<img src="http://images.ucomics.com/comics/db/2004/'
      . "db$todays_date.gif"
      . '"/>'
);
```

Then it's just a matter of adding in the channel information and serving the thing out with the correct MIME type:

```
$rss->channel(
    title       => "Today's Doonesbury",
    link        => "http://www.doonesbury.com",
    description => "Doonesbury, by Garry Trudeau"
);

print header('application/xml+rss');
print $rss->as_string;
```

You can obviously use this sort of code to produce RSS feeds of any online comic strip. Do the author a favor, however: don't make the feed public, visit his site once in a while, and buy some schwag. You're getting great stuff for free—with the added convenience you've added, for sure—and it's nice to give something back.

The Entire Listing

```
#!/usr/bin/perl

use strict;
use warnings;
use CGI qw(:standard);
use Date::Manip;
use XML::RSS;

my $todays_date = &UnixDate( "today", "%y%m%d" );

my $rss = new XML::RSS( version => '2.0' );

$rss->add_item(
    title => "Doonesbury for $todays_date",
    link  => "http://images.ucomics.com/comics/db/2004/db$todays_date.gif",
    description => '&lt;img src="http://images.ucomics.com/comics/db/2004/'
      . "db$todays_date.gif"
      . '"/&gt;'
);

$rss->channel(
    title       => "Today's Doonesbury",
    link        => "http://www.doonesbury.com",
    description => "Doonesbury, by Garry Trudeau"
);

print header('application/xml+rss');
print $rss->as_string;
```

Amazon.com Wishlist to RSS

Being perfection herself, my wife loves books, and as a loving and dutiful husband, her Amazon wishlist is required reading for Christmas, birthdays, and all other occasions. But keeping track of the wishlist is a pain if I have to trudge over to Amazon every time. Far better to have my feed reader do it for me, with the help of a little script. Figure 10-1 shows my wishlist to give you an idea of what one looks like.

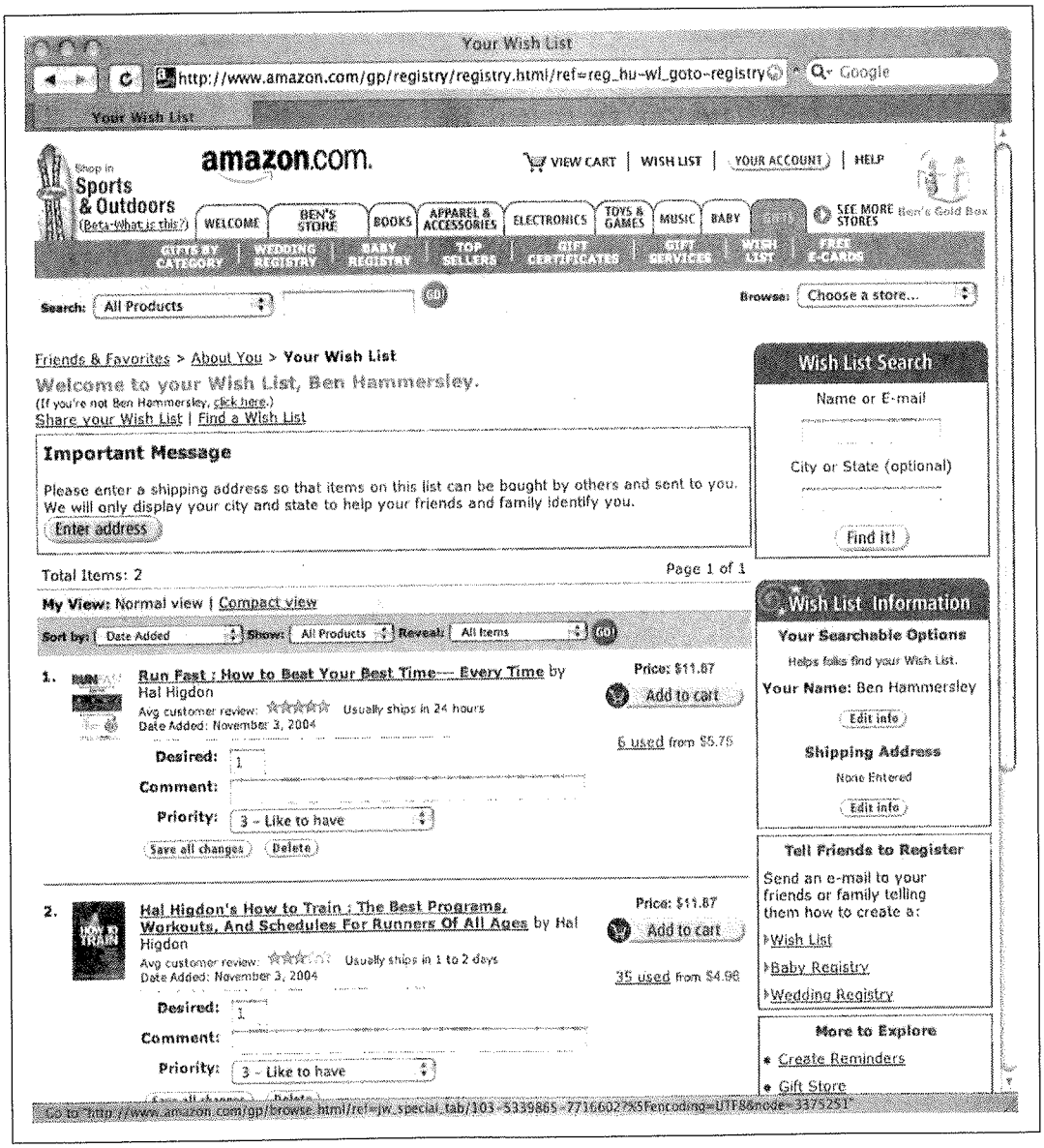


Figure 10-1. My Amazon.com wishlist page

This feed uses the Amazon Web Services API to do its evil work. This can be either REST- or SOAP-based, so you can choose your own preferred poison. For fun, I'll do

this using the REST interface, and then using `XML::Simple` to parse the XML. My idea of fun might not be the same as yours, of course.

Walking Through the Code

As always, we fire up the script with the loading of the modules and the setting of some global variables: the obligatory `use strict;` and `use warnings;`, and the required Amazon API subscription key. You'll need to get your own from *http://www.amazon.com/gp/aws/landing.html*.

```
use strict;
use warnings;
use XML::RSS;
use XML::Simple;
use LWP::Simple qw(!head);
use CGI qw(:standard);
use Getopt::Long;
use Date::Manip;

my $amazon_subscription_id = "xxxxxxxxxxxxxxxxxx";
my $rss = new XML::RSS( version => '2.0' );
```

As this script is running as a CGI application, it requires the Amazon Wishlist ID as a parameter. This allows you to use the same script many times over for each of the lists you want to monitor. Let's make the parameter compulsory, naturally.

```
my $cgi = CGI::new( );
my $list_id = $cgi->param('list');
```

Now, run the query via the Amazon Web Services REST interface. This takes a specifically formed URI and returns an XML document. We'll retrieve this using the `LWP::Simple` module:

```
my $query_url =
"http://webservices.amazon.com/onca/
xml?Service=AWSProductData&SubscriptionId=$amazon_subscription_
id&Operation=ListLookup&ProductPage=15&ListType=WishList&ListId=$list_
id&ResponseGroup=Request,ListItems";

my $wishlist_in_xml = get("$query_url");
```

and place it into the Parser:

```
my $parser = XMLin("$wishlist_in_xml") or die ("Could not parse file");
```

This produces an array of the items within the wishlist, albeit still stored within the XML format returned by the REST query. That format looks like this:

```
&lt;ListItem&gt;
        &lt;ListItemId&gt;I2MNVARC9PUUA7&lt;/ListItemId&gt;
        &lt;DateAdded&gt;2004-11-03&lt;/DateAdded&gt;
        &lt;QuantityDesired&gt;1&lt;/QuantityDesired&gt;
        &lt;QuantityReceived&gt;0&lt;/QuantityReceived&gt;
        &lt;Item&gt;
```

```
                &lt;ASIN&gt;0875963528&lt;/ASIN&gt;
                &lt;ItemAttributes&gt;
                        &lt;Title&gt;Hal Higdon's How to Train&lt;/Title&gt;
                &lt;/ItemAttributes&gt;
        &lt;/Item&gt;
&lt;/ListItem&gt;
```

So now you need to take the relevant bits of that information and throw it into the feed. For the taking is the title, the date, and the ASIN number—Amazon's internal system for identifying its stock. Use this to provide a link. For a change of pace, let's use a different XML parsing method, too. `XML::Simple` is tremendously useful for this sort of thing.

Now, we'll add in some code to create the correct date, based on the date the item was added to the wishlist. Amazon returns the date in the format YYYY-MM-DD, but RSS 2.0 insists on the format `Sun, 19 May 2002 15:21:36 GMT`. For the sake of sanity, assume all the dates Amazon gives are at exactly midnight.

We need to do some date manipulation, and what better than `Date::Manip` to do this? Under its `UnixDate` function, the code `%g` is exactly the right format for RSS 2.0. Nifty, no?

```
foreach my $item (@{$parser->{'Lists'}->{'List'}->{'ListItem'}}) {

    #my $date = &UnixDate("midnight $item->{'Item'}->{'DateAdded'}","%c, %e %b %Y");

   $rss->add_item(
   title => "$item->{'Item'}->{'ItemAttributes'}->{'Title'}",
   link  => "http://www.amazon.com/exec/obidos/tg/detail/-/$item->{'Item'}->
{'ASIN'}",
    description => "$item->{'Item'}->{'ItemAttributes'}->{'Title'}",
    pubDate => &UnixDate("midnight $item->{'DateAdded'}","%g")

       );

}
```

Then, as ever, it's just a matter of adding in the channel information and serving the thing out with the correct MIME type:

```
$rss->channel(
    title       => "Amazon Wishlist, $list_id",
    link        => "http://www.amazon.com",
    description => "An RSS feed of the Amazon Wishlist, id number $list_id"
);

print header('application/xml+rss');
print $rss->as_string;
```

The Entire Listing

```
#!/usr/bin/perl

use strict;
use warnings;
use XML::RSS;
use XML::Simple;
use LWP::Simple qw(!head);
use CGI qw(:standard);
use Getopt::Long;
use Date::Manip;

my $amazon_subscription_id = "08R1SHPFGCA8VYT9P802";
my $rss = new XML::RSS( version => '2.0' );

my $cgi = CGI::new();
my $list_id = $cgi->param('list');

my $query_url =
"http://webservices.amazon.com/onca/
xml?Service=AWSProductData&SubscriptionId=$amazon_subscription_
id&Operation=ListLookup&ProductPage=15&ListType=WishList&ListId=$list_
id&ResponseGroup=Request,ListItems";

my $wishlist_in_xml = get("$query_url");

my $parser = XMLin("$wishlist_in_xml") or die ("Could not parse file");

foreach my $item (@{$parser->{'Lists'}->{'List'}->{'ListItem'}}) {

    $rss->add_item(
    title => "$item->{'Item'}->{'ItemAttributes'}->{'Title'}",
    link  => "http://www.amazon.com/exec/obidos/tg/detail/-/$item->{'Item'}->
{'ASIN'}",
    description => "$item->{'Item'}->{'ItemAttributes'}->{'Title'}",
    pubDate => &UnixDate("midnight $item->{'DateAdded'}","%g")

        );

}

$rss->channel(
    title       => "Amazon Wishlist, $list_id",
    link        => "http://www.amazon.com",
    description => "An RSS feed of the Amazon Wishlist, id number $list_id"
);

print header('application/xml+rss');
print $rss->as_string;
```

FedEx Parcel Tracker

Christmas is coming, and Santa has outsourced his deliveries to Federal Express. Lucky us, as that means we can use FedEx's online shipment tracker to watch our parcels wend their merry way here. Except the tiresome chore of refreshing the FedEx site is just too much to handle. Let's let a nice script elf take care of it and create a feed for every parcel.

Although FedEx and its rivals do provide APIs, we won't be using them here. FedEx's page is easy enough to scrape by brute force, and it's fun to do so. Of course, when it next changes its page layout, this script will need rejigging. It's easy to see how to do that when it happens.

Walking Through the Code

So, starting with the usual Perl standards of `warnings;`, `strict;`, `XML::RSS`, and `CGI`, let's use `LWP::Simple` to retrieve the page and the marvellous `HTML::TokeParser` to do the dirty work. More on that anon.

```
use warnings;
use strict;
use XML::RSS;
use CGI qw(:standard);
use LWP::Simple 'get';
use HTML::TokeParser;
```

Now let's set up some variables to use later, then fire up the CGI module and grab the tracking number from the query string. To use this script, therefore, you need to request:

```
http://www.example.org/fedextracker.cgi?track=123456789
```

where `123456789` is the tracking number of the parcel:

```
my ( $tag, $headline, $url, $date_line );
my $last_good_date;
my $table_end_check;

my $cgi             = CGI::new();
my $tracking_number = $cgi->param('track');
```

Now we're ready to jingle. Using `LWP::Simple`'s `get` method, pull down the page from the FedEx site. FedEx, bless them, employ openly understandable URLs, so this is easy to set up. Once that's downloaded, throw it into a new instance of the `HTML::TokeParser` module, and we're ready for scraping:

```
my $tracking_page =
  get(
"http://fedex.com/Tracking?action=track&tracknumber_list=$tracking_number&cntry_
code=us"
  );

my $stream = HTML::TokeParser->new( \$tracking_page );
```

Now is as good a time as any to start off `XML::RSS` and fill in some channel details:

```
my $rss = XML::RSS->new( );

$rss->channel(
    title => "FedEx Tracking: $tracking_number",
    link  =>
"http://fedex.com/Tracking?action=track&tracknumber_list=$tracking_number&cntry_
code=us"
);
```

From now on, we're using the `HTML::TokeParser` module, skipping from tag to tag until we get to the section of the HTML to scrape. The inline comments say what we're up to.

```
# Go to the right part of the page, skipping 13 tables (!!!)
$stream->get_tag("table");
$stream->get_tag("table");
$stream->get_tag("table");
$stream->get_tag("table");
$stream->get_tag("table");
$stream->get_tag("table");
$stream->get_tag("table");
$stream->get_tag("table");
$stream->get_tag("table");
$stream->get_tag("table");
$stream->get_tag("table");
$stream->get_tag("table");
$stream->get_tag("table");

# Now go inside the tracking details table
$stream->get_tag("table");
$stream->get_tag("tr");
$stream->get_tag("/tr");
$stream->get_tag("tr");
$stream->get_tag("/tr");
```

By this point, you're at the table to parse, so loop through it, getting the dates and locations. You need to stop at the bottom of the table, so test for a closing `/table` tag. You can do so with a named loop and a `last...if...` command.

You'll notice that in this section, we use those mysterious variables from earlier. Because the table is displayed with the date mentioned only once per day, no matter how many stops the parcel makes on that day, you need to keep track of it.

```
PARSE: while ( $tag = $stream->get_tag('tr') ) {

    $stream->get_tag("td");
    $stream->get_tag("/td");

    # Test here for the closing /tr. If it exists, we're done.

    # Now get date text
    $stream->get_tag("td");
    $stream->get_tag("b");
```

```
my $date_text = $stream->get_trimmed_text("/b");

# The page only mentions the date once, so we need to fill in any blanks
# that might occur.

if ( $date_text eq "\xa0" ) {
    $date_text = $last_good_date;
}
else {
    $last_good_date = $date_text;
}

# Now get the time text
$stream->get_tag("/b");
$stream->get_tag("/td");
$stream->get_tag("td");
my $time_text = $stream->get_trimmed_text("/td");
$time_text =~ s/\xa0//g;

# Now get the status
$stream->get_tag("/td");
$stream->get_tag("/td");
$stream->get_tag("/td");
$stream->get_tag("/td");
$stream->get_tag("td");
my $status = $stream->get_trimmed_text("/td");
$status =~ s/\xa0//g;

# Now get the location
$stream->get_tag("/td");
$stream->get_tag("/td");
$stream->get_tag("/td");
$stream->get_tag("/td");
$stream->get_tag("td");
my $location = $stream->get_trimmed_text("/td");
$location =~ s/\xa0//g;

# Now get the comment
$stream->get_tag("/td");
$stream->get_tag("/td");
$stream->get_tag("/td");
$stream->get_tag("/td");
$stream->get_tag("td");
my $comment = $stream->get_trimmed_text("/td");
$comment =~ s/\xa0//g;

# Now go to the end of the block
$stream->get_tag("/td");
$stream->get_tag("/td");
$stream->get_tag("/tr");

# OK, now we have the details, we need to put them into a feed
# Do what you want with the info:
```

Still inside the loop, create an item for the RSS feed:

```
        if ($status) {
          $rss->add_item(
              title => "$status $location $date_text $time_text",
              link  =>
"http://fedex.com/us/tracking/?action=track&tracknumber_list=$tracking_number",
              description =>
"Package number $tracking_number was last seen in $location at $time_text on $date_
text, with the status,
$status. $comment Godspeed, little parcel! Onward, tiny package!"
          );
      }

      # Stop parsing after the pickup line.
      last PARSE if ( $status eq "Picked up " );
}
```

All that done, you can serve it up nice and festive:

```
print header('application/rss+xml');
print $rss->as_string;
```

The Entire Listing

```
#!/usr/bin/perl

use warnings;
use strict;
use XML::RSS;
use CGI qw(:standard);
use LWP::Simple 'get';
use HTML::TokeParser;

my ( $tag, $headline, $url, $date_line );
my $last_good_date;
my $table_end_check;

my $cgi             = CGI::new( );
my $tracking_number = $cgi->param('track');

my $tracking_page =
  get(
"http://fedex.com/Tracking?action=track&tracknumber_list=$tracking_number&cntry_
code=us"
  );

my $stream = HTML::TokeParser->new( \$tracking_page );

my $rss = XML::RSS->new( );

$rss->channel(
    title => "FedEx Tracking: $tracking_number",
    link  =>
```

```
"http://fedex.com/Tracking?action=track&tracknumber_list=$tracking_number&cntry_
code=us"
);

# Go to the right part of the page, skipping 13 tables (!!!)
$stream->get_tag("table");
$stream->get_tag("table");
$stream->get_tag("table");
$stream->get_tag("table");
$stream->get_tag("table");
$stream->get_tag("table");
$stream->get_tag("table");
$stream->get_tag("table");
$stream->get_tag("table");
$stream->get_tag("table");
$stream->get_tag("table");
$stream->get_tag("table");
$stream->get_tag("table");

# Now go inside the tracking details table
$stream->get_tag("table");
$stream->get_tag("tr");
$stream->get_tag("/tr");
$stream->get_tag("tr");
$stream->get_tag("/tr");

PARSE: while ( $tag = $stream->get_tag('tr') ) {

    $stream->get_tag("td");
    $stream->get_tag("/td");

    # Test here for the closing /tr. If it exists, we're done.

    # Now get date text
    $stream->get_tag("td");
    $stream->get_tag("b");

    my $date_text = $stream->get_trimmed_text("/b");

    # The page only mentions the date once, so we need to fill in any blanks
    # that might occur.

    if ( $date_text eq "\xa0" ) {
        $date_text = $last_good_date;
    }
    else {
        $last_good_date = $date_text;
    }

    # Now get the time text
    $stream->get_tag("/b");
    $stream->get_tag("/td");
    $stream->get_tag("td");
```

```
    my $time_text = $stream->get_trimmed_text("/td");
    $time_text =~ s/\xa0//g;

    # Now get the status
    $stream->get_tag("/td");
    $stream->get_tag("/td");
    $stream->get_tag("/td");
    $stream->get_tag("/td");
    $stream->get_tag("td");
    my $status = $stream->get_trimmed_text("/td");
    $status =~ s/\xa0//g;

    # Now get the location
    $stream->get_tag("/td");
    $stream->get_tag("/td");
    $stream->get_tag("/td");
    $stream->get_tag("/td");
    $stream->get_tag("td");
    my $location = $stream->get_trimmed_text("/td");
    $location =~ s/\xa0//g;

    # Now get the comment
    $stream->get_tag("/td");
    $stream->get_tag("/td");
    $stream->get_tag("/td");
    $stream->get_tag("/td");
    $stream->get_tag("td");
    my $comment = $stream->get_trimmed_text("/td");
    $comment =~ s/\xa0//g;

    # Now go to the end of the block
    $stream->get_tag("/td");
    $stream->get_tag("/td");
    $stream->get_tag("/tr");

    # OK, now we have the details, we need to put them into a feed
    # Do what you want with the info:

    if ($status) {
        $rss->add_item(
            title => "$status $location $date_text $time_text",
            link  =>
"http://fedex.com/us/tracking/?action=track&tracknumber_list=$tracking_number",
            description =>
"Package number $tracking_number was last seen in $location at $time_text on $date_
text, with the status,
$status. $comment Godspeed, little parcel! Onward, tiny package!"
        );
    }

    # Stop parsing after the pickup line.
    last PARSE if ( $status eq "Picked up " );
```

```
}

print header('application/rss+xml');
print $rss->as_string;
```

Google to RSS with SOAP

Google, the search engine of fashion these days, is as close to an authority as you can get on the Internet. If it's not mentioned on Google, the feeling is, it's not really online. And if it is, conversely, then people will find it. Keeping track of things in Google's database, therefore, is an important job. Whether you are trying to find things out or monitoring for other people's discoveries, an RSS feed is the perfect way to do it. Happily, Google provides a nice interface to work with its systems. This makes things very easy.

Walking Through the Code

We start, as ever, with the usual pragmas, `CGI` and `XML::RSS`. This time, you also need `SOAP::Lite` and `HTML::Entitie`.

```
use warnings;
use strict;
use XML::RSS;
use CGI qw(:standard);
use HTML::Entities ();
use SOAP::Lite;
```

We'll set up the query term, and the Google API key, from the CGI input. Then, we fire up the `SOAP::Lite` by pointing it at Google's WSDL file. WSDL files contain the finer details of a SOAP interface, telling the script exactly where to send commands and so on.

```
my $query = param("q");
my $key   = param("k");

my $service = SOAP::Lite -> service('http://api.google.com/GoogleSearch.wsdl');
```

Now, run the search using the SOAP interface we just set up; then set up the feed:

```
my $result = $service -> doGoogleSearch ($key, $query, 0, 10, "false", "",
"false","", "latin1", "latin1");

my $rss = new XML::RSS (version => '2.00');

$rss->channel(  title  => "Google Search for $query",
                link => "http://www.google.com/search?q=$query",
                description => "Google search for $query",
                        language => "en",
                      );
```

Now, it's just a matter of going through the results and using them as values for the feed's items. To make it valid XML, use the HTML::Entities module to encode any stray characters that might need it:

```
foreach my $element (@{$result->{'resultElements'}}) {
        $rss->add_item(
                title   => HTML::Entities::encode($element->{'title'}),
                link    => HTML::Entities::encode($element->{'URL'})
                );
        }
```

And finally, serve the feed:

```
print header('application/xml+rss'), $rss->as_string;
```

The Entire Listing

```
#!/usr/bin/perl

use warnings;
use strict;
use XML::RSS;
use CGI qw(:standard);
use HTML::Entities ();
use SOAP::Lite;

my $query = param("q");
my $key   = param("k");

my $service = SOAP::Lite -> service('http://api.google.com/GoogleSearch.wsdl');

my $result = $service -> doGoogleSearch ($key, $query, 0, 10, "false", "", "false",
"", "latin1", "latin1");

my $rss = new XML::RSS (version => '2.00');

$rss->channel(  title  => "Google Search for $query",
                link => "http://www.google.com/search?q=$query",
                description => "Google search for $query",
                        language => "en",
                      );

foreach my $element (@{$result->{'resultElements'}}) {
        $rss->add_item(
                title   => HTML::Entities::encode($element->{'title'}),
                link    => HTML::Entities::encode($element->{'URL'})
                );
        }

print header('application/xml+rss'), $rss->as_string;
```

Last-Modified Files

There are lots of reasons to have an ever-updating list of recently modified files: security, for one. However, it's also a useful system for helping you group working files in your mind. I'm forever loosing track of files I'm working on—especially overnight—and this feed helps a great deal. For collaborative working, it's a godsend because you can see what other activity is going on automatically. I also have one of these pointed at a shared directory where friends drop music and silly MPEGs. Altogether, for something so simple, it's remarkably useful.

It's a CGI script that takes a single parameter `path`, which should be equal to the absolute path on your filesystem that you wish to look under—for example, `http://www.example.org/lastmodified.cgi?path=/users/ben`.

Walking Through the Code

Let's trek once again into the `warnings;`, `strict;`, `XML::RSS`, and `CGI`, plus `Date::Manip` for the dateline and `File::Find` for its directory traversing capabilities. Lovely. All set? Good.

```
use warnings;
use strict;
use XML::RSS;
use CGI qw(:standard);
use File::Find;
use Date::Manip;
```

By now you should be getting the hang of this. We're firing up the CGI module, snaffling the parameter we are passing to it, then setting up the feed. No great mystery here, in other words. Really, this is somewhat the point: feeds are very simple things. It's the ideas of how to use them that are valuable.

```
my $cgi             = CGI::new( );
my $start_directory = $cgi->param('path');

my $rss = new XML::RSS( version => '2.0' );
$rss->channel(
    title       => "Last modified from $start_directory",
    link        => "file:/$start_directory",
    description =>
"A list of all of the files in $start_directory, with their modification dates added
in"
);
```

Here's the real meat of the script. We're using the `find` function from the `File::Find` module, to traverse the directory we've given it and throw each individual file it finds into a small subroutine:

```
find( \&search_and_rss, $start_directory );

sub search_and_rss {
```

Here's an interesting thing. You can use File::Find and the standard stat function to grab the last-modified date of each file and then use Date::Manip to convert it from the standard epoch seconds to the required format. Note that, unlike most other date strings, Date::Manip requires the word "epoch" in front of the raw time value to help it along. This caused me to do a lot of shouting before I remembered it.

```
my $last_modified_date = ( stat($File::Find::name) )[9];
my $parsed_date        = &ParseDate("epoch $last_modified_date");
my $pubDate            = &UnixDate( $parsed_date, "%g" );
```

It's then just a matter of creating each item, and, once the whole directory has been traversed, serving up the feed:

```
    $rss->add_item(
        title       => "$File::Find::name",
        link        => "file:\/\/$File::Find::name",
        description => "$File::Find::name",
        pubDate     => "$pubDate",
    );

}

print header('application/xml');
print $rss->as_string;
```

The Entire Listing

```
#!/usr/bin/perl

use warnings;
use strict;
use XML::RSS;
use CGI qw(:standard);
use File::Find;
use Date::Manip;

my $cgi             = CGI::new( );
my $start_directory = $cgi->param('path');

my $rss = new XML::RSS( version => '2.0' );
$rss->channel(
    title       => "Last modified from $start_directory",
    link        => "file:/$start_directory",
    description =>
"A list of all of the files in $start_directory, with their modification dates added
in"
);

find( \&search_and_rss, $start_directory );

sub search_and_rss {
```

```
    my $last_modified_date = ( stat($File::Find::name) )[9];
    my $parsed_date        = &ParseDate("epoch $last_modified_date");
    my $pubDate            = &UnixDate( $parsed_date, "%g" );

    $rss->add_item(
        title       => "$File::Find::name",
        link        => "file:\/\/$File::Find::name",
        description => "$File::Find::name",
        pubDate     => "$pubDate",
    );
}

print header('application/xml');
print $rss->as_string;
```

Installed Perl Modules

Another interesting way to use a Feed is to keep track of changing system configurations. The following is a simple example. I have two machines running my scripts; one is my laptop and the other a server in a rack somewhere in North America. I write mostly in Perl and use a lot of modules. It's useful to me to keep track of what's installed, and what's not, on each; this script produces a feed of the modules I have installed.

Walking Through the Code

We'll use the usual modules plus one more, `ExtUtils::Installed`. Written by Alan Burlison, this module really does all the work here. Again, this is somewhat the point of this chapter. Feeds aren't difficult things, and they are incredibly useful interfaces to list-like data.

```
use warnings;
use strict;
use CGI qw(:standard);
use XML::RSS;
use ExtUtils::Installed;
```

`ExtUtils::Installed` creates an array of all the installed modules; run the shell command `uname -n` to find the name of the machine we're running the script on:

```
my $installed    = ExtUtils::Installed->new( );
my $machine_name = `uname -n`;
```

Now, set up the feed as usual, using the machine name to label it:

```
my $rss = new XML::RSS( version => '2.0' );
$rss->channel(
    title       => "Perl Modules on $machine_name",
    link        => "http://www.benhammersley.com/tools/",
    description => "A list of installed Perl modules on $machine_name"
);
```

Finally, go through the array, pulling out the name and version number of each module and creating the items. Then serve it up. Simple.

```
foreach my $module ( $installed->modules( ) ) {
    my $version = $installed->version($module) || "???";

    $rss->add_item(
        title       => "$module $version",
        description => "$module $version",
        link        => "http://search.cpan.org/search%3fmodule=$module",
    );
}

print header('application/rss+xml');
print $rss->as_string;
```

The Entire Listing

```
#!/usr/bin/perl

use warnings;
use strict;
use CGI qw(:standard);
use XML::RSS;
use ExtUtils::Installed;

my $installed    = ExtUtils::Installed->new( );
my $machine_name = `uname -n`;

my $rss = new XML::RSS( version => '2.0' );
$rss->channel(
    title       => "Perl Modules on $machine_name",
    link        => "http://www.benhammersley.com/tools/",
    description => "A list of installed Perl modules on $machine_name"
);

foreach my $module ( $installed->modules( ) ) {
    my $version = $installed->version($module) || "???";

    $rss->add_item(
        title       => "$module $version",
        description => "$module $version",
        link        => "http://search.cpan.org/search%3fmodule=$module",
    );
}

print header('application/rss+xml');
print $rss->as_string;
```

The W3C Validator to RSS

Of all the tasks of Hercules, the one where he had to keep his web site's XHTML validated was the hardest. Without wanting to approach the whole Valid XHTML Controversy, we can still safely say that keeping a site validated is a pain. You have to validate your code, most commonly using the W3C validator service at *http://validator.w3.org*, and you have to keep going back there to make sure nothing has broken.

You have to do that unless, of course, you're subscribed to a feed of validation results. This script does just that, providing an RSS interface to the W3C validator.

You pass the URL you want to test as a query in the feed URL, like so: `http://www.example.org/validator.cgi?url=http://www.example.org/index.html`.

Walking Through the Code

We're using the traditional Perl start plus `LWP::Simple` and `XML::Simple`, which will parse the results coming back from the validator. Note that, in the classic gotcha, `LWP::Simple` and `CGI` clash, so we have to add those additional flags to prevent a type mismatch.

```
use warnings;
use strict;
use XML::RSS;
use CGI qw(:standard);
use LWP::Simple 'get';
use XML::Simple;
```

Now, grab the URL from the query string, and use `LWP::Simple` to retrieve the results. The W3C provides an XML output mode for the validator, and this is what we're using here. It is, however, classed as beta and flakey, and might not always work.

```
my $cgi = CGI::new();
my $url = $cgi->param('url');

my $validator_results_in_xml =
  get("http://validator.w3.org/check?uri=$url;output=xml");
```

Curiously enough, the top of the XML that is returned causes `XML::Simple` to throw an error. Use a split function to trim off this broken section:

```
my ( $broken_xml_to_ignore, $trimmed_validator_results_in_xml ) =
  split ( />/, $validator_results_in_xml );
```

Now, place the valid XML into an `XML::Simple` object, and parse it:

```
my $parsed_validator_results = XMLin($trimmed_validator_results_in_xml);
```

Now is a good a time as any to set up the top of the feed:

```
my $rss = new XML::RSS( version => '2.0' );

$rss->channel( title => "XHTML Validation results for $url",
link  => "http://validator.w3.org/check?uri=$url",
description => "w3c validation results for $url" );
```

Then it's a simple matter of running through each error message the validator gives and turning it into a feed item:

```
foreach my $error ( @{ $parsed_validator_results->{'messages'}->{'msg'} } ) {
    $rss->add_item(
        title       => "Line $error->{'line'} $error->{'content'}",
        link        => "http://validator.w3.org/check?uri=$url",
        description => "Line $error->{'line'} $error->{'content'}",
    );
}
```

Finally, serve it up:

```
print header('application/xml+rss');
print $rss->as_string;
```

The Entire Listing

```
#!/usr/bin/perl

use warnings;
use strict;
use XML::RSS;
use CGI qw(:standard);
use LWP::Simple 'get';
use XML::Simple;

my $cgi = CGI::new( );
my $url = $cgi->param('url');

my $validator_results_in_xml =
  get("http://validator.w3.org/check?uri=$url;output=xml");

my ( $broken_xml_to_ignore, $trimmed_validator_results_in_xml ) =
  split ( />/, $validator_results_in_xml );

my $parsed_validator_results = XMLin($trimmed_validator_results_in_xml);

my $rss = new XML::RSS( version => '2.0' );

$rss->channel( title => "XHTML Validation results for $url",
link  => "http://validator.w3.org/check?uri=$url",
description => "w3c validation results for $url" );

foreach my $error ( @{ $parsed_validator_results->{'messages'}->{'msg'} } ) {
    $rss->add_item(
        title       => "Line $error->{'line'} $error->{'content'}",
        link        => "http://validator.w3.org/check?uri=$url",
```

```
            description => "Line $error->{'line'} $error->{'content'}",
        );
    }

    print header('application/xml+rss');
    print $rss->as_string;
```

Game Statistics to Excel

Halo 2, a game for the XBox console, can be played as a multiplayer online game over the XBox-Live network. If you do this, your players statistics become available online as an RSS 2.0 feed. Example 10-1 shows an example of such a feed, courtesy of player "nedrichards."

Example 10-1. An RSS 2.0 feed from Halo 2

```
<?xml version="1.0" encoding="utf-8"?>
<rss version="2.0">
  <channel>
    <title>nedrichards's recent games</title>
    <link>http://www.bungie.net/stats</link>
    <description>Halo2 games played recently (courtesy of Bungie.Net)</description>
    <language>en-us</language>
    <pubDate>Tue, 21 Dec 2004 15:16:54 GMT</pubDate>
    <docs>http://blogs.law.harvard.edu/tech/rss</docs>
    <generator>BungieTest RSS Generator</generator>
    <webMaster>webmaster@bungie.net</webMaster>
    <item>
      <title>Arranged Game: Rockets on Headlong</title>
      <link>http://bungie.net/stats/gamestats.aspx?gameid=30084649&player=nedrichards</
link>

       <pubDate>Thu, 16 Dec 2004 22:49:33 GMT</pubDate>
       <guid>http://bungie.net/stats/gamestats.aspx?gameid=30084649&player=nedrichards</
guid>

       <description>Game played at Thu, 16 Dec 2004 22:49:33 GMT<br/><br/>Playlist:
Arranged
Game<br/>Rockets on Headlong<br/><br/><b>Gamertag (Team): Score, Kills, Deaths,
Assists</b><br/>Evil Ted Hed (4): 25, 25, 14, 1<br/>Cranium Oxide (0): 21, 21, 12,
2<br/>nedrichards (2): 15, 15, 25, 1<br/>smokingdrum (3): 12, 12, 18, 2<br/>Frizby
(1): 11, 11, 24, 1<br/></description>
        </item>
        <item>
           <title>Team Skirmish: 1 Flag CTF Fast on Burial Mounds</title>
           <link>http://bungie.net/stats/gamestats.
aspx?gameid=30068566&player=nedrichards</link>

           <pubDate>Thu, 16 Dec 2004 22:28:23 GMT</pubDate>
           <guid>http://bungie.net/stats/gamestats.
aspx?gameid=30068566&player=nedrichards</guid>
```

Example 10-1. An RSS 2.0 feed from Halo 2 (continued)

```
            <description>Team game played at Thu, 16 Dec 2004 22:28:23 GMT<br/><br/>
Playlist:
Team Skirmish<br/>1 Flag CTF Fast on Burial Mounds<br/><br/><b>Gamertag (Team):
Score, Kills, Deaths, Assists</b><br/>Frizby (1): 0, 3, 8, 1<br/>BoP Andy (0): 0, 4,
2, 1<br/>nedrichards (1): 0, 3, 7, 1<br/>smokingdrum (1): 0, 2, 8, 2<br/>Ninja Master
35 (0): 0, 10, 6, 1<br/>Snake9000 (1): 0, 11, 8, 2<br/>BEACONHILLGAMER (0): 0, 9, 5, 1<br/
>Spartan
736 (0): 0, 7, 6, 2<br/></description>
         </item>
         <item>
           <title>Team Slayer: Team Slayer on Foundation</title>
            <link>http://bungie.net/stats/gamestats.
aspx?gameid=30060296&player=nedrichards</link>

           <pubDate>Thu, 16 Dec 2004 22:17:44 GMT</pubDate>
            <guid>http://bungie.net/stats/gamestats.
aspx?gameid=30060296&player=nedrichards</guid>

            <description>Team game played at Thu, 16 Dec 2004 22:17:44 GMT<br/><br/>
Playlist:Team Slayer<br/>Team Slayer on Foundation<br/><br/><b>Gamertag (Team): Score,
Kills, Deaths, Assists</b><br/>AgelessPainter (0): 22, 22, 4, 3<br/>smokingdrum (1):
12, 12, 14, 0<br/>sfpipeman1 (0): 11, 11, 8, 2<br/>DARKNIGHT377 (0): 11, 11, 6, 1<br/>
nedrichards(1): 9, 9, 12, 5<br/>BlindJokerCard (0): 6, 6, 6, 6<br/>
Frizby (1): 4, 3, 15, 4<br/>Wang2 (1): 0, 0, 9, 4<br/></description>
            </item>
      </channel>
</rss>
```

Each `item` contains the statistics for a single game. I've removed most of the games from the example to save space.

Such statistics are perfect fodder for analysis, and so Samuel Radakovitz—a program manager for Excel within Microsoft—has built a workbook for Excel 2003 that can import Halo2 feeds and produce beautiful reports from them. It is getting ever more powerful, and the latest version can be downloaded from *http://www.isamrad.com/Halo2RSS/*.

Feeds by SMS

In most of the examples so far, we have written the parsed feed to the screen of a computer. This is just the start. You can use the same basic structure to output to just about anything that handles text. Example 10-2 is a script that sends the top headline of a feed to a mobile phone via the Short Message Service (SMS). It uses the `WWW::SMS` module, outputting to the first web-based free SMS service it can find that works.

Example 10-2. rsssms.pl sends the first headline title to a mobile phone via SMS

```
#!/usr/local/bin/perl
use strict;
use warnings;
use LWP::Simple;
use XML::Simple;
use WWW::SMS;

# Take the command line arguments, URL first, then complete number of mobile
my $url=$ARGV[0];
my $number=$ARGV[1];

# Retrieve the feed, or die disgracefully
my $feed_to_parse = get ($url) or die "I can't get the feed you want";

# Parse the XML
my $parser = XML::Simple->new( );
my $rss = $parser->XMLin("$feed_to_parse");

# Get the data we want
my $message = "NEWSFLASH:: $rss->{'channel'}->{'item'}->[0]->{'title'}";

# Send the message
my @gateway = WWW::SMS->gateways( );
my $sms = WWW::SMS->new($number, $message);
foreach my $gateway(@gateway) {if ($sms->send($gateway)) {
          print 'Message sent!';
            last;
     } else {
          print "Error: $WWW::SMS::Error\n";
     }}
```

You can use the script in Example 10-2 from the command line or crontab like so:

```
perl rsssms.pl http://full.urlof/feed.xml 123456789
```

You can see how to set this up on crontab to send the latest news at the desired interval. But how about using the system status module, mod_systemstatus, to automatically detect and inform you of system failures? Perhaps you can use something like Example 10-3.

Example 10-3. mod_systemstatusSMS.pl

```
#!/usr/local/bin/perl

use strict;
use warnings;
use LWP::Simple;
use XML::Simple;
use WWW::SMS;
```

Example 10-3. mod_systemstatusSMS.pl (continued)

```
# Take the command line arguments, URL first, then complete number
my $url=$ARGV[0];
my $number=$ARGV[1];

# Retrieve the feed, or die gracefully
my $feed_to_parse = get ($url) or die "I can't get the feed you want";

# Parse the XML
my $parser = XML::Simple->new(  );
my $rss = $parser->XMLin("$feed_to_parse");

# initialise the $message
my $message;

# Look for downed servers
foreach my $item (@{$rss->{'item'}}) {
    next unless ($item->{'ss:responding'}) eq 'false';
    $message .= "Emergency! $item->{'title'} is down.";
           }

# Send the message
if ($message) {
my @gateway = WWW::SMS->gateways(  );
my $sms = WWW::SMS->new($number, $message);
foreach my $gateway(@gateway) {if ($sms->send($gateway)) {
          print 'Message sent!';
     } else {
          print "Error: $WWW::SMS::Error\n";
     }}
     };
```

Again, run from cron, this little beasty will let you monitor hundreds of machines—as long as they are generating the correct RSS—and inform you of a server outage via your mobile phone.

This combination of selective parsing, interesting output methods, and cron allows you to do many things with RSS feeds that a more comprehensive system may well inhibit. Monitoring a list of feeds for mentions of keywords is simple, as is using RSS feeds of stock prices to alert you of falls in the market. Combining these techniques with Publish and Subscribe systems (discussed in Chapter 9) gives you an even greater ability to monitor the world. Want an IRC channel to be notified of any new weblog postings? No problem. Want an SMS whenever the phrase "Free Beer" appears in your local feeds? Again, no problem.

Podcasting Weather Forecasts

The podcasting technique described in Chapter 4 is a hotbed of development at the moment. One idea put forward was to use it to deliver weather forecasts via a web service and a text-to-speech application.

Jorge Velázquez put together a script to do just that. Released at *http://www.jorgev.com/archives/000115.html*, it requires an account with weather.com and an installation of `Lame` (from *http://lame.sourceforge.net/*) and `text2wave` on the server.

How to Use It

The URL accepts two parameters, `locid` and `unit`. The `locid` is the weather.com location identifier for the city you require. For U.S. cities, this can be a zip code, and for non-U.S. cities, it is a special weather.com code. (e.g., 92126 for San Diego, CA, or ITXX0067 for Rome, Italy). The unit parameter is optional and can be either `m` for metric or `s` for imperial measurements. It defaults to imperial.

The location code for Florence, Italy, is ITXX0028, so the URL for the feed would be *http://www.example.com/cgi-bin/weather.cgi?locid=ITXX0028*. Simple.

The Code Itself

Jorge's code is, in his own words, very simple. This is how he describes it:

> A brief explanation of how the script works, it's actually quite simple: I first call into weather.com's XML Data Feed. Then I use XPath to extract the data in which I am interested. I format this information into a text file, which I then pass onto the text2wave utility which performs the text-to-speech conversion. Finally, since wav files are so huge, I convert it to MP3 using the LAME encoder. Piece of cake, eh?

```
#!/usr/bin/perl -w
# 2004-12-13 Jorge Velázquez

use strict;
use CGI qw(:standard);
use XML::RSS;
use XML::XPath;
use LWP::Simple;
use File::Temp;
use File::Basename;

# partner and key information
my $par = 'partneridhere';
my $key = 'keyhere';

# get parameters
my $cgi   = CGI::new();
my $locid = $cgi->param('locid');
my $unit  = $cgi->param('unit');
```

```
if ( not $unit ) {
    $unit = 's';
}
my $mp3dir = 'enter/mp3/path/here';

# query weather info, current conditions with 2 day forecast
my $xml =
  get(
"http://xoap.weather.com/weather/local/
$locid?cc=*&dayf=2&prod=xoap&par=$par&key=$key&unit=$unit"

  );

#load it into XPath object
my $xp = XML::XPath->new($xml);

# extract unit information from feed
my $ut = $xp->findvalue('/weather/head/ut');
my $ud = $xp->findvalue('/weather/head/ud');
my $us = $xp->findvalue('/weather/head/us');

# create rss 2.0 feed
my $rss = new XML::RSS( version => '2.0' );

# channel information
$rss->channel(
    title       => 'Weather Podcast',
    link        => 'http://www.weather.com/',
    description => 'Local weather podcasting feed',
    language    => 'en'
);

# get current weather conditions
my $dnam   = $xp->findvalue('/weather/loc/dnam');
my $ccobst = $xp->findvalue('/weather/cc/obst');
my $cclsup = $xp->findvalue('/weather/cc/lsup');
my $cctemp = $xp->findvalue('/weather/cc/tmp');
my $ccwind = $xp->findvalue('/weather/cc/wind/s');
my $cct    = $xp->findvalue('/weather/cc/t');
my $ccbarr = $xp->findvalue('/weather/cc/bar/r');
my $ccbard = $xp->findvalue('/weather/cc/bar/d');

# convert the units to words
my $utword = $ut eq 'F' ? 'fahrenheit' : 'celsius';

# build the text file for converting to speech
my $text =
  "Current conditions for $dnam. $cct. The temperature is $cctemp $utword.";

# write the file and get the stats
my $filename = &writefile($text);
my $basename = basename $filename;
my $filesize = ( stat $filename )[7];
```

```
# add rss item for current weather
$rss->add_item(
    title       => "Current weather conditions for $ccobst",
    link        => "http://www.weather.com/weather/local/$locid",
    description => $text,
    pubDate     => $cclsup,
    enclosure   => {
        url    => "$mp3dir$basename",
        length => $filesize,
        type   => 'audio/mpeg'
    }
);

# get forecast timestamp
my $lsup = $xp->findvalue('/weather/dayf/lsup');

# iterate through forecast days
my $nodeset = $xp->find('/weather/dayf/day');
foreach my $node ( $nodeset->get_nodelist ) {

    # get the items of interest to our script
    my $t   = $node->findvalue('@t');
    my $dt  = $node->findvalue('@dt');
    my $hi  = $node->findvalue('hi');
    my $low = $node->findvalue('low');
    my $d   = $node->findvalue('part[@p="d"]/t');
    my $n   = $node->findvalue('part[@p="n"]/t');

    # build the text file for converting to speech
    $text = "Weather forecast for $t, $dt in $dnam. ";
    if ( $d eq 'N/A' ) {
        $text .= "Daytime conditions not available. ";
    }
    else {
        $text .= "Daytime $d. ";
    }
    if ( $n eq 'N/A' ) {
        $text .= "Nighttime conditions not available. ";
    }
    else {
        $text .= "Nighttime $n. ";
    }
    if ( $hi eq 'N/A' ) {
        $text .= "The high is unavailable. ";
    }
    else {
        $text .= "The high is $hi $utword. ";
    }
    if ( $low eq 'N/A' ) {
        $text .= "The low is unavailable. ";
    }
    else {
        $text .= "The low is $low $utword. ";
    }
```

```
    # write the file and get the stats
    $filename = &writefile($text);
    $basename = basename $filename;
    $filesize = ( stat $filename )[7];

    # add this forecast
    $rss->add_item(
        title       => "Weather forecast for $t, $dt at $dnam",
        link        => "http://www.weather.com/weather/local/$locid",
        description => $text,
        pubDate     => $lsup,
        enclosure   => {
            url    => "$mp3dir$basename",
            length => $filesize,
            type   => 'audio/mpeg'
        }
    );
}

print header('application/rss+xml');
print $rss->as_string;

sub writefile {

    # write the text file
    my $fhtxt = new File::Temp( suffix => '.txt' );
    print $fhtxt $_[0];
    close $fhtxt;

    # write the wav file
    my $fhwav = new File::Temp( suffix => '.wav' );
    system "text2wave", "-o", $fhwav->filename, $fhtxt->filename;

    # convert it to mp3
    my $fhmp3 =
      new File::Temp( unlink => 0, dir => '../mp3', suffix => '.mp3' );
    system "lame", "-h", $fhwav->filename, $fhmp3->filename;

    # return the file name
    return $fhmp3->filename;
}
```

Having Amazon Produce Its Own RSS Feeds

Andrew Odewahn, an editor at O'Reilly, spotted how you can have Amazon.com produce RSS feeds from any search within its own site. So, he says:

> You could subscribe to a search for books on "software engineering," "Java," and "history of europe" to easily keep track with what's going on.

To do this, you only need play with a URL. The URL for an RSS 0.91 feed of a keyword search for "software engineering" is:

> *http://xml.amazon.com/onca/xml3?t=webservices-20&dev-t=amznRss&KeywordSearch=software%20engineering&mode=books&bcm=&type=lite&page=1&ct=text/xml&sort=+salesrank&f=http://xml.amazon.com/xsl/xml-rss091.xsl*

As you can see, this is actually calling the Amazon web services system and then formatting it by passing it through an XSLT stylesheet.

Andrew goes on to say,

> By default, the search results are sorted by Amazon rank. After hacking around a bit, I found that you can change this default to any of a variety of other orders. For example, you can modify the feed so that books are sorted by publication date to get an automatically updated list of new books published on a particular topic.

To do this, scroll over the URL until you see the "&sort=+salesrank" part. To change the sort order, replace "salesrank" with the option you'd prefer. Here's a list of possible options found on Amazon's Web Services page, *http://www.amazon.com/gp/aws/landing.html*:

- Publication Date → daterank
- Featured Items → pmrank
- Sales Rank → salesrank
- Customer reviews → reviewrank
- Price (Lo-Hi) → pricerank
- Price (Hi-Lo) → inverse-pricerank

So, to get a feed of all the new books published in Software Engineering, just replace "salesrank" with "daterank" and click update. From now on, you easily see all the new stuff coming out in one place. For example:

> *http://xml.amazon.com/onca/xml3?t=webservices-20&dev-t=amznRss&KeywordSearch=Ben%20Hammersley&mode=books&bcm=&type=lite&page=1&ct=text/xml&sort=+pricerank&f=http://xml.amazon.com/xsl/xml-rss091.xsl*

produces a feed of the books I have written, in order of price. This technique highlights a very good point: it is well worth looking at complex URLs quite carefully: they're invariably hackable.

Cross-Poster for Movable Type

By now, the traditional use of feeds as a form of content syndication is beginning to look somewhat old-fashioned. But fear not: here's a use that is as traditional as can be.

I needed a script to check all of the weblogs I write for and to cross-post everything I write onto my own weblog. Bear in mind that I'm not the only author on these other sites.

To do this, check their RSS feeds on a set schedule, grab the content within, build a big entry, and then post it. This code is for a Movable Type installation, but it isn't hard to modify it to fit another weblogging platform.

Walking Through the Code

You open the proceedings by defining all of the libraries and modules. This is the exact same code as I have running on my own server, so you need to modify the following paths to point to your own Movable Type libraries, blog IDs, and so on:

```
use lib "/web/script/ben/mediacooperative.com/lib";
use lib "/web/script/ben/mediacooperative.com/extlib";
use lib "/web/script/ben/lib/perl";
use MT;
use MT::Entry;
use Date::Manip;
use LWP::Simple 'get';
use XML::RSS;

my $MTauthor = "1";
my $MTblogID = "3";
my $MTconfig = "/home/ben/web/mediacooperative.com/mt.cfg";
my $guts     = "";
```

Now, let's set up a list of sites to check. For each site, you need to define only the feed URL and the <dc:creator> or <author> name under which I am posting. Everything else you can get from the feed itself. For example:

http://del.icio.us/rss/bhammersley "bhammersley" http://www.oreillynet.com/feeds/author/?x-au=909 "Ben Hammersley" http://monkeyfilter.com/rss.php "DangerIsMyMiddleName" http://www.benhammersley.com/expeditions/northpole2006/index.rdf "Ben Hammersley"

You can do this with an array of arrays:

```
my @sites_to_check = (
    [ "http://del.icio.us/rss/bhammersley",              "bhammersley" ],
    [ "http://www.oreillynet.com/feeds/author/?x-au=909", "Ben Hammersley" ],
    [ "http://monkeyfilter.com/rss.php", "DangerIsMyMiddleName" ],
    [
        "http://www.benhammersley.com/expeditions/northpole2006/index.rdf",
        "Ben Hammersley"
    ],
);
```

Now, the loop. You go through each feed, downloading, parsing, and so on. Let's start by taking the site_feed_url and the site_author_nym (the name under which I

go on that site) out of the array. This step could be omitted, but for the sake of clarity, we'll leave it here.

```
for my $site_being_checked (@sites_to_check) {

    my $site_feed_url   = @$site_being_checked[0];
    my $site_author_nym = @$site_being_checked[1];
```

Now, retrieve the feed, or go to the next one if it fails:

```
    my $feed_xml = get("$site_feed_url") or next;
```

And now, to parse it. You do so by spawning a new instance of the `XML::RSS` parser and jamming the feed into it:

```
my $rss_parser = XML::RSS->new( );
$rss_parser->parse($feed_xml);
```

To set up for the strange occasion where there might be new content to post, let's query the newly created RSS parser object for its name:

```
my $feed_name = $rss_parser->{channel}->{title};
my $feed_link = $rss_parser->{channel}->{link};
```

Now, go through each of the items within the field, and grab all needed data out of them: the link, title, description, author, and date. Note that you have to include the fallbacks of the `guid`, content, and the various `dc` values to deal with different versions of RSS.

```
foreach my $item ( @{ $rss_parser->{items} } ) {

    my $item_link        = $$item{link} || $$item{guid};
    my $item_title       = $$item{title};
    my $item_description = $$item{description};
    my $item_author      = $$item{author} || $$item{dc}->{creator};
    my $item_date        = $$item{pubDate} || $$item{dc}->{date};
```

Now, check to see if any were written today by me. First, work out what time and date it is now. Then, compare the post's date with the date now, and, if it's less than 24 hours behind, *and* it was written by me, then all is good.

Note: to get this code to work with del.icio.us and any other sites that use date strings with z instead of +00:00 (which `Date::Manip` can't deal with), you have to use a nasty substitution. Sorry about that.

```
my $todays_date = &UnixDate( "now", "%Y-%m-%dT%H:%M:%S+00:00" );
$item_date =~ s/Z/+00:00/;
my $date_delta = DateCalc( "$item_date", "$todays_date", \$err, 1 );
my $parsed_delta = Delta_Format( "$date_delta", exact, '%dh' );

if ( ( $parsed_delta < 1 ) and ( $item_author eq $site_author_nym ) ) {
```

If all the tests turn out to be true, add a bunch of HTML to the `$guts` variable, within which you're building the new entry:

```
            $guts .=
qq|<div id="CrossPoster"><blockquote><a href="$item_link">$item_title</a><br/>posted
to <a href="$feed_link">$feed_name</a><br/></p><p>$item_description</p></blockquote>
</div>|;

        }
    }
}
```

Now, having worked our way through the feeds, if the $guts has anything in it, you need to post it and take care of that end:

```
if ( $guts ne "" ) {

    my $mt    = MT->new( Config => $MTconfig ) or die MT->errstr;
    my $entry = MT::Entry->new;

    $entry->blog_id($MTblogID);
    $entry->status( MT::Entry::RELEASE( ) );
    $entry->author_id($MTauthor);
    $entry->title("Posted elsewhere today");
    $entry->text($guts);
    $entry->convert_breaks(0);
    $entry->save
      or die $entry->errstr;

    # rebuild the site

    $mt->rebuild( BlogID => $MTblogID )
      or die "Rebuild error: " . $mt->errstr;

    # ping aggregators

    $mt->ping($MTblogID);

}
```

The Entire Listing

```
#!/usr/bin/perl

use lib "/web/script/ben/mediacooperative.com/lib";
use lib "/web/script/ben/mediacooperative.com/extlib";
use lib "/web/script/ben/lib/perl";
use MT;
use MT::Entry;
use Date::Manip;
use LWP::Simple 'get';
use XML::RSS;

my $MTauthor = "1";
my $MTblogID = "3";
my $MTconfig = "/home/ben/web/mediacooperative.com/mt.cfg";
my $guts     = "";
```

```
my @sites_to_check = (
    [ "http://del.icio.us/rss/bhammersley",                "bhammersley" ],
    [ "http://www.oreillynet.com/feeds/author/?x-au=909", "Ben Hammersley" ],
    [ "http://monkeyfilter.com/rss.php", "DangerIsMyMiddleName" ],
    [
        "http://www.benhammersley.com/expeditions/northpole2006/index.rdf",
        "Ben Hammersley"
    ],
);

for my $site_being_checked (@sites_to_check) {

    my $site_feed_url   = @$site_being_checked[0];
    my $site_author_nym = @$site_being_checked[1];

    my $feed_xml = get("$site_feed_url") or next;

    my $rss_parser = XML::RSS->new( );
    $rss_parser->parse($feed_xml);

    my $feed_name = $rss_parser->{channel}->{title};
    my $feed_link = $rss_parser->{channel}->{link};

    foreach my $item ( @{ $rss_parser->{items} } ) {

        my $item_link        = $$item{link} || $$item{guid};
        my $item_title       = $$item{title};
        my $item_description = $$item{description};
        my $item_author      = $$item{author} || $$item{dc}->{creator};
        my $item_date        = $$item{pubDate} || $$item{dc}->{date};

        my $todays_date = &UnixDate( "now", "%Y-%m-%dT%H:%M:%S+00:00" );
        $item_date =~ s/Z/+00:00/;
        my $date_delta = DateCalc( "$item_date", "$todays_date", \$err, 1 );
        my $parsed_delta = Delta_Format( "$date_delta", exact, '%dh' );

        if ( ( $parsed_delta < 1 ) and ( $item_author eq $site_author_nym ) ) {

            $guts .=
qq|<div id="CrossPoster"><blockquote><a href="$item_link">$item_title</a><br/>posted
to <a href="$feed_link">$feed_name</a><br/></p><p>$item_description</p></blockquote>
</div>|;

        }
    }
}

if ( $guts ne "" ) {

    my $mt    = MT->new( Config => $MTconfig ) or die MT->errstr;
    my $entry = MT::Entry->new;

    $entry->blog_id($MTblogID);
    $entry->status( MT::Entry::RELEASE( ) );
```

```
    $entry->author_id($MTauthor);
    $entry->title("Posted elsewhere today");
    $entry->text($guts);
    $entry->convert_breaks(0);
    $entry->save
      or die $entry->errstr;

    # rebuild the site

    $mt->rebuild( BlogID => $MTblogID )
      or die "Rebuild error: " . $mt->errstr;

    # ping aggregators

    $mt->ping($MTblogID);
}
```

APPENDIX A

The XML You Need for RSS

The purpose of this appendix is to introduce you to XML. A knowledge of XML is essential if you want to write RSS documents directly, rather than having them generated by some utility. If you're already acquainted with XML, you don't need to read this appendix. If not, read on.

The general overview of XML given in this appendix should be more than sufficient to enable you to work with the RSS documents. For further information about XML, the O'Reilly books *Learning XML* and *XML in a Nutshell* are invaluable guides, as is the weekly online magazine *XML.com*.

Note that this appendix makes frequent reference to the formal XML 1.0 specification, which can be used for further investigation of topics that fall outside the scope of RSS. Readers are also directed to the "Annotated XML Specification," written by Tim Bray and published online at *http://XML.com*, which provides an illuminating explanation of the XML 1.0 specification, and "What is XML?" by Norm Walsh, also published on *XML.com*.

What Is XML?

XML (Extensible Markup Language) is an Internet-friendly format for data and documents, invented by the World Wide Web Consortium (W3C). *Markup* denotes a way of expressing the structure of a document within the document itself. XML has its roots in a markup language called SGML (Standard Generalized Markup Language), which is used in publishing and shares this heritage with HTML. XML was created to do for machine-readable documents on the Web what HTML did for human-readable documents—that is, provide a commonly agreed-upon syntax, so that processing the underlying format becomes a commodity and documents are made accessible to all users.

Unlike HTML, though, XML comes with very little predefined. HTML developers are accustomed to both the notion of using angle brackets (<>) for denoting elements (that

is, syntax) and also the set of element names themselves (such as head, body, etc.). XML shares only the former feature (i.e., the notion of using angle brackets for denoting elements). Unlike HTML, XML has no predefined elements but is merely a set of rules that lets you write other languages like HTML. (To clarify XML's relationship with SGML: XML is an SGML subset. In contrast, HTML is an SGML application. RSS uses XML to express its operations and thus is an XML application.)

Because XML defines so little, it is easy for everyone to agree to use the XML syntax and then build applications on top of it. It's like agreeing to use a particular alphabet and set of punctuation symbols, but not saying which language to use. However, if you come to XML from an HTML background (and have an interest in extending RSS), you may need to prepare yourself for the shock of having to choose what to call your tags!

Knowing that XML's roots lie with SGML should help you understand some of XML's features and design decisions. Note that although SGML is essentially a document-centric technology, XML's functionality also extends to data-centric applications, including RSS. Commonly, data-centric applications don't need all the flexibility and expressiveness that XML provides and limit themselves to employing only a subset of XML's functionality.

Anatomy of an XML Document

The best way to explain how an XML document is composed is to present one. The following example shows an XML document you might use to describe two authors:

```
<?xml version="1.0" encoding="us-ascii"?>
<authors>
    <person id="lear">
        <name>Edward Lear</name>
        <nationality>British</nationality>
    </person>
    <person id="asimov">
        <name>Isaac Asimov</name>
        <nationality>American</nationality>
    </person>
    <person id="mysteryperson"/>
</authors>
```

The first line of the document is known as the *XML declaration*. This tells a processing application which version of XML you are using (the version indicator is mandatory) and which character encoding you have used for the document. In this example, the document is encoded in ASCII. (The significance of character encoding is covered later in this appendix.)

If the XML declaration is omitted, a processor makes certain assumptions about your document. In particular, it expects it to be encoded in UTF-8, an encoding of the Unicode character set. However, it is best to use the XML declaration wherever

possible, both to avoid confusion over the character encoding and to indicate to processors which version of XML you're using.

Elements and Attributes

The second line of the example begins an element, which has been named authors. The contents of that element include everything between the right angle bracket (>) in <authors> and the left angle bracket (<) in </authors>. The actual syntactic constructs <authors> and </authors> are often referred to as the element *start tag* and *end tag*, respectively. Don't confuse tags with elements! Note that elements may include other elements, as well as text. An XML document must contain exactly one root element, which contains all other content within the document. The name of the root element defines the type of the XML document.

Elements that contain both text and other elements simultaneously are classified as *mixed content*. RSS doesn't generally use mixed content.

The sample authors document uses elements named person to describe the authors themselves. Each person element has an attribute named id. Unlike elements, attributes can contain only textual content. Their values must be surrounded by quotes. Either single quotes (') or double quotes (") may be used, as long as you use the same kind of closing quote as the opening one.

Within XML documents, attributes are frequently used for metadata (i.e., "data about data"), describing properties of the element's contents. This is the case in our example, where id contains a unique identifier for the person being described.

As far as XML is concerned, the order in which attributes are presented in the element start tag doesn't matter. For example, these two elements contain the same information, as far as an XML 1.0–conformant processing application is concerned:

```
<animal name="dog" legs="4"/>
<animal legs="4" name="dog"/>
```

On the other hand, the information presented to an application by an XML processor after reading the following two lines is different for each animal element, because the ordering of elements is significant:

```
<animal><name>dog</name><legs>4</legs></animal>
<animal><legs>4</legs><name>dog</name></animal>
```

XML treats a set of attributes like a bunch of stuff in a bag—there is no implicit ordering—while elements are treated like items on a list, where ordering matters.

New XML developers frequently ask when it is best to use attributes to represent information and when it is best to use elements. As you can see from the authors example, if order is important to you, then elements are a good choice. In general, there is no hard-and-fast "best practice" for choosing whether to use attributes or elements.

The final author described in our document has no information available. All that's known about this person is his or her id, mysteryperson. The document uses the XML shortcut syntax for an empty element. The following is a reasonable alternative:

```
<person id="mysteryperson"></person>
```

Name Syntax

XML 1.0 has certain rules about element and attribute names. In particular:

- Names are case-sensitive: e.g., <person/> isn't the same as <Person/>.
- Names beginning with xml (in any permutation of uppercase or lowercase) are reserved for use by XML 1.0 and its companion specifications.
- A name must start with a letter or an underscore, not a digit, and may continue with any letter, digit, underscore, or period. (Actually, a name may also contain a colon, but the colon is used to delimit a namespace prefix and isn't available for arbitrary use as of the second edition of XML 1.0. Knowledge of namespaces isn't required for understanding RSS, but for more information, see Tim Bray's "XML Namespaces by Example," published at *http://www.xml.com/pub/a/1999/01/namespaces.html*.)

A precise description of names can be found in Section 2.3 of the XML 1.0 specification at *http://www.w3.org/TR/REC-xml#sec-common-syn*.

Well-Formedness

An XML document that conforms to the rules of XML syntax is said to be *well-formed*. At its most basic level, well-formedness means that elements should be properly matched, and all opened elements should be closed. A formal definition of well-formedness can be found in Section 2.1 of the XML 1.0 specification at *http://www.w3.org/TR/REC-xml#sec-well-formed*. Table A-1 shows some XML documents that aren't well-formed.

Table A-1. Examples of poorly formed XML documents

Document	Reason it's not well-formed
<foo> <bar> </foo> </bar>	The elements aren't properly nested, because foo is closed while inside its child element bar.
<foo> <bar> </foo>	The bar element was not closed before its parent, foo, was closed.
<foo baz> </foo>	The baz attribute has no value. While this is permissible in HTML (e.g., <table border>), it is forbidden in XML.
<foo baz=23> </foo>	The baz attribute value, 23, has no surrounding quotes. Unlike HTML, all attribute values must be quoted in XML.

Comments

As in HTML, it is possible to include comments within XML documents. XML comments are intended to be read only by people. With HTML, developers have occasionally employed comments to add application-specific functionality. For example, the server-side include functionality of most web servers uses instructions embedded in HTML comments. XML provides other ways to indicate application-processing instructions. A discussion of processing instructions (PIs) is outside the scope of this book. For more information on PIs, see Section 2.6 of the XML 1.0 specification at *http://www.w3.org/TR/REC-xml#sec-pi*. Comments should not be used for any purpose other than those for which they were intended.

The start of a comment is indicated with <!--, and the end of the comment is indicated with -->. Any sequence of characters, aside from the string --, may appear within a comment. Comments tend to be used more in XML documents intended for human consumption than those intended for machine consumption. Comments aren't widely used in RSS.

Entity References

Another feature of XML that is occasionally useful when writing RSS documents is the mechanism for escaping characters.

Because some characters have special significance in XML, there needs to be a way to represent them. For example, in some cases the < symbol might really be intended to mean "less than" rather than to signal the start of an element name. Clearly, just inserting the character without any escaping mechanism will result in a poorly formed document, because a processing application assumes you are starting another element. Another instance of this problem is the need to include both double quotes and single quotes simultaneously in an attribute's value. Here's an example that illustrates both difficulties:

```
<badDoc>
  <para>
    I'd really like to use the < character
  </para>
  <note title="On the proper 'use' of the " character"/>
</badDoc>
```

XML avoids this problem by using the predefined entity reference. The word *entity* in the context of XML simply means a unit of content. The term *entity reference* means just that: a symbolic way of referring to a certain unit of content. XML predefines entities for the following symbols: left angle bracket (<), right angle bracket (>), apostrophe ('), double quote ("), and ampersand (&).

An entity reference is introduced with an ampersand (&), which is followed by a name (using the word "name" in its formal sense, as defined by the XML 1.0

specification) and terminated with a semicolon (;). Table A-2 shows how the five predefined entities can be used within an XML document.

Table A-2. Predefined entity references in XML 1.0

Literal character	Entity reference
<	<
>	>
'	'
"	"
&	&

Here's the problematic document, revised to use entity references:

```
<badDoc>
  <para>
    I'd really like to use the &lt; character
  </para>
  <note title="On the proper &apos; use &apos;  of the &quot;character"/>
</badDoc>
```

XML 1.0 allows you to define your own entities and use entity references as shortcuts in your document, but the predefined entities are often all you need for RSS or Atom; in general, entities are provided as a convenience for human-created XML. Section 4 of the XML 1.0 specification, available at *http://www.w3.org/TR/REC-xml#sec-physical-struct*, describes the use of entities.

Character References

You may find *character references* in the context of RSS documents. Character references allow you to denote a character by its numeric position in the Unicode character set (this position is known as its *code point*). Table A-3 contains a few examples that illustrate the syntax.

Table A-3. Example character references

Actual character	Character reference
1	0
A	A
~	Ñ
®	®

Note that the code point can be expressed in decimal or, with the use of x as a prefix, in hexadecimal.

Character Encodings

The subject of *character encodings* is frequently a mysterious one for developers. Most code tends to be written for one computing platform and, normally, to run within one organization. Although the Internet is changing things quickly, most of us have never had cause to think too deeply about internationalization.

XML, designed to be an Internet-friendly syntax for information exchange, has internationalization at its very core. One of the basic requirements for XML processors is that they support the Unicode standard character encoding. Unicode attempts to include the requirements of all the world's languages within one character set. Consequently, it is very large!

Unicode encoding schemes

Unicode 3.0 has more than 57,700 code points, each of which corresponds to a character. (You can obtain charts of all these characters online by visiting *http://www.unicode.org/charts/*.) If you were to express a Unicode string using the position of each character in the character set as its encoding (in the same way as ASCII does), expressing the whole range of characters would require four *octets* for each character. (An octet is a string of eight binary digits, or bits. A byte is commonly, but not always, considered the same thing as an octet.) Clearly, if a document is written in 100% American English, it would be four times larger than required—all the characters in ASCII fitting into a 7-bit representation. This places a strain on both storage space and on memory requirements for processing applications.

Fortunately, two encoding schemes for Unicode alleviate this problem: UTF-8 and UTF-16. As you might guess from their names, applications can process documents in these encodings in 8- or 16-bit segments at a time. When code points are required in a document that can't be represented by one chunk, a bit-pattern indicates that the following chunk is required to calculate the desired code point. In UTF-8, this is denoted by the most significant bit of the first octet being set to `1`.

This scheme means that UTF-8 is a highly efficient encoding for representing languages using Latin alphabets, such as English. All of the ASCII character set is represented natively in UTF-8; an ASCII-only document and its equivalent in UTF-8 are identical byte for byte.

This knowledge will also help you debug encoding errors. One frequent error arises because of the fact that ASCII is a proper subset of UTF-8; programmers get used to this fact and produce UTF-8 documents but use them as if they were ASCII. Things start to go awry when the XML parser processes a document containing, for example, characters such as `&Aacute;`. Because this character can't be represented using only one octet in UTF-8, a two-octet sequence is produced in the output document; in a non-Unicode viewer or text editor, it looks like a couple of characters of garbage.

Other character encodings

Unicode, in the context of computing history, is a relatively new invention. Native operating system support for Unicode is by no means widespread. For instance, although Windows NT offers Unicode support, Windows 95 and 98 don't.

XML 1.0 allows a document to be encoded in any character set registered with the Internet Assigned Numbers Authority (IANA). European documents are commonly encoded in one of the ISO Latin character sets, such as ISO-8859-1. Japanese documents normally use Shift-JIS, and Chinese documents use GB2312 and Big 5.

A full list of registered character sets can be found at *http://www.iana.org/assignments/character-sets*.

XML processors aren't required by the XML 1.0 specification to support any more than UTF-8 and UTF-16, but most commonly support other encodings, such as US-ASCII and ISO-8859-1. Although most RSS transactions are currently conducted in ASCII (or the ASCII subset of UTF-8), there is nothing to stop RSS documents from containing, say, Korean text. However, you will probably have to dig into the encoding support of your computing platform to find out if it is possible for you to use alternate encodings.

Validity

In addition to well-formedness, XML 1.0 offers another level of verification called *validity*. To explain why validity is important, let's take a simple example. Imagine you invented a simple XML format for your friends' telephone numbers:

```
<phonebook>
  <person>
    <name>Albert Smith</name>
    <number>123-456-7890</number>
  </person>
  <person>
    <name>Bertrand Jones</name>
    <number>456-123-9876</number>
  </person>
</phonebook>
```

Based on your format, you also construct a program to display and search your phone numbers. This program turns out to be so useful, you share it with your friends. However, your friends aren't so hot on detail as you are, and they try to feed your program this phone book file:

```
<phonebook>
  <person>
    <name>Melanie Green</name>
    <phone>123-456-7893</phone>
  </person>
</phonebook>
```

Note that, although this file is perfectly well-formed, it doesn't fit the format you prescribed for the phone book, and you find you need to change your program to cope with this situation. If your friends had used number as you did to denote the phone number, and not phone, there wouldn't have been a problem. However, as it is, this second file isn't a valid phonebook document.

Document type definitions (DTDs)

For validity to be a useful general concept, we need a machine-readable way of saying what a valid document is—that is, which elements and attributes must be present and in what order. XML 1.0 achieves this by introducing *document type definitions* (DTDs). For the purposes of RSS, you don't need to know much about DTDs. Rest assured that RSS does have a DTD, and it spells out in detail exactly which combinations of elements and attributes make up a valid document.

The purpose of a DTD is to express the allowed elements and attributes in a certain document type and to constrain the order in which they must appear within that document type. A DTD is generally composed of one file, which contains declarations defining the element types and attribute lists. (In theory, a DTD may span more than one file; however, the mechanism for including one file inside another—*parameter entities*—is outside the scope of this book.) It is common to mistakenly conflate element and element types. The distinction is that an element is the actual instance of the structure as found in an XML document, whereas the instance's kind of element is the element type.

Putting It Together

If you want to validate RSS against a DTD, you need to know how to link a document to its defining DTD. This is done with a document type declaration, `<!DOCTYPE ...>`, inserted at the beginning of the XML document, after the XML declaration in our fictitious example:

```
<?xml version="1.0" encoding="us-ascii"?>
<!DOCTYPE authors SYSTEM "http://example.com/authors.dtd">
<authors>
    <person id="lear">
        <name>Edward Lear</name>
        <nationality>British</nationality>
    </person>
    <person id="asimov">
        <name>Isaac Asimov</name>
        <nationality>American</nationality>
    </person>
    <person id="mysteryperson"/>
</authors>
```

This example assumes the DTD file has been placed on a web server at *example.com*. Note that the document type declaration specifies the root element of the document, not the DTD itself. You can use the same DTD to define person, name, or nationality

as the root element of a valid document. Certain DTDs, such as the DocBook DTD for technical documentation (see *http://www.docbook.org*), use this feature to good effect, allowing you to provide the same DTD for multiple document types.

A validating XML processor is obligated to check the input document against its DTD. If it doesn't validate, the document is rejected. To return to the phone book example, if your application validated its input files against a phone book DTD, you would have been spared the problems of debugging your program and correcting your friend's XML, because your application would have rejected the document as being invalid. While some of the programs that read RSS files do worry about validation, most don't.

XML Namespaces

XML 1.0 lets developers create their own elements and attributes, but it leaves open the potential for overlapping names. "Title" in one context may mean something entirely different than "Title" in a different context. The "Namespaces in XML" specification (which can be found at *http://www.w3.org/TR/REC-xml-names*) provides a mechanism developers can use to identify particular vocabularies using URIs.

RSS 1.0 uses the URI *http://purl.org/rss/1.0/* for its base namespace. The URI is just an identifier; opening that page in a web browser reveals some links to the RSS, XML 1.0, and Namespaces in XML specifications. Programs processing documents with multiple vocabularies can use the namespaces to figure out which vocabulary they are handling at any given point in a document.

Namespaces are very simple on the surface but are a well-known field of combat in XML arcana. For more information on namespaces, see O'Reilly's *XML in a Nutshell* or *Learning XML*. The use of namespaces in RSS is discussed in much greater detail in Chapters 6 and 7.

Tools for Processing XML

While RSS can be parsed directly using text-processing tools, XML parsers are often more convenient. Many parsers exist for using XML with many different programming languages. Most are freely available, and the majority are open source.

Selecting a Parser

An XML parser typically takes the form of a library of code that you interface with your own program. The RSS program hands the XML over to the parser, and the parser hands back information about the contents of the XML document. Typically, parsers do this either via events or via a document object model.

With event-based parsing, the parser calls a function in your program whenever a *parse event* is encountered. Parse events include things like finding the start of an

element, the end of an element, or a comment. Most Java event-based parsers follow a standard API called SAX, which is also implemented for other languages such as Python and Perl. You can find more about SAX at *http://www.saxproject.org*.

Document object model (DOM)-based parsers work in a markedly different way. They consume the entire XML input document and hand back a tree-like data structure that the RSS software can interrogate and alter. The DOM is a W3C standard; documentation is available at *http://www.w3.org/DOM*.

Choosing whether to use an event- or DOM–based model depends on the application. If you have a large or unpredictable document size, it is better to use event-based parsing for reasons of speed and memory consumption (DOM trees can get very large). If you have small, simple XML documents, using the DOM leaves you less programming work to do. Many programming languages have both event-based and DOM support.

As XML matures, hybrid techniques that give the best of both worlds are emerging. If you're interested in finding out what's available and what's new for your favorite programming language, keep an eye on the following online sources:

- XML.com Resource Guide (*http://xml.com/pub/resourceguide*)
- XMLhack XML Developer News (*http://xmlhack.com*)
- Free XML Tools Guide (*http://www.garshol.priv.no/download/xmltools*)

XSLT Processors

Many XML applications can transform one XML document into another or into HTML. The W3C has defined a special language, called XSLT, for doing transformations. XSLT processors are becoming available for all major programming platforms.

XSLT works using a stylesheet, which contains templates that describe how to transform elements from an XML document. These templates typically specify what XML to output in response to a particular element or attribute. Using a W3C technology called XPath gives you the flexibility not only to say "do this for every person element," but also to give instructions as complex as "do this for the third person element, whose `name` attribute is `Fred`."

Because of this flexibility, some applications have sprung up for XSLT that aren't really transformation applications at all but take advantage of the ability to trigger actions on certain element patterns and sequences. Combined with XSLT's ability to execute custom code via extension functions, the XPath language has enabled applications such as document indexing to be driven by an XSLT processor.

The W3C specifications for XSLT and XPath can be found at *http://w3.org/TR/xslt* and *http://w3.org/TR/xpath*, respectively. For more information on XSLT, see *XSLT* (O'Reilly); for more on XPath, see *XPath and XPointer* (O'Reilly).