Scripting Intelligence

Web 3.0 Information Gathering and Processing

Mark Watson

Apress®

Scripting Intelligence: Web 3.0 Information Gathering and Processing

Copyright © 2009 by Mark Watson

ISBN-13 (pbk): 978-1-4302-2351-1

ISBN-13 (electronic): 978-1-4302-2352-8

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

To Carol, Julie, David, Josh, Calvin, and Emily

Contents at a Glance

Contents

About the Author

MARK WATSON is the author of 15 books on artificial intelligence (AI), software agents, Java, Common Lisp, Scheme, Linux, and user interfaces. He wrote the free chess program distributed with the original Apple II computer, built the world's first commercial Go playing program, and developed commercial products for the original Macintosh and for Windows 1.0. He was an architect and a lead developer for the worldwide-distributed Nuclear Monitoring Research and Development (NMRD) project and for a distributed expert system designed to detect telephone credit-card fraud. He has worked on the AI for Nintendo video games and was technical lead for a Virtual Reality system for Disney. He currently works on text- and data-mining projects, and develops web applications using Ruby on Rails and server-side Java.

Mark enjoys hiking and cooking, as well as playing guitar, didgeridoo, and the American Indian flute.

About the Technical Reviewer

PETER SZINEK is a freelance software developer. He left his Java job and academic career a few years ago to hack on everything Ruby- and Rails-related, and never looked back. He is the author of Ruby's most popular web-scraping framework, scRUBYt! (`http://scrubyt.org`), which is featured in this book. After founding two startups, he started his own consultancy called HexAgile (`http://hexagile.com`), which offers Ruby, Rails, JavaScript, and web-scraping services. In addition to coding, he also enjoys writing—namely, blogging at `http://www.rubyrailways.com`, working on the "AJAX on Rails" guide for the docrails project, and tweeting too much. As one of the first members of the RailsBridge initiative (`http://railsbridge.com`), he tries to expand and enrich the Rails community, one project at a time. He loves to travel and chill out with his two-and-a-half-year-old daughter.

Acknowledgments

Many people helped me with this book project. I would like to thank all of the open source developers who wrote software that I used both for this book and in my work. My wife Carol supported my efforts in many ways, including reviewing early versions to catch typos and offering comments on general readability. My technical editor Peter Szinek made many useful comments and suggestions, as did my editor Michelle Lowman. Project manager Beth Christmas kept me on schedule and ensured that everything ran smoothly. Copy editor Nina Goldschlager Perry helped me improve the general readability of my text. Production editor Ellie Fountain made the final manuscript look good. I would also like to thank Apress staff who helped, even though I did not directly interact with them: indexer Kevin Broccoli, proofreader Liz Welch, compositor Dina Quan, and artist Kinetic Publishing.

Introduction

This book covers Web 3.0 technologies from a software developer's point of view. While non-techies can use web services and portals that other people create, developers have the ability to be creators and consumers at the same time—by integrating their work with other people's efforts.

The Meaning of Web 3.0

Currently, there is no firm consensus on what "Web 3.0" means, so I feel free to define Web 3.0 for the context of this book and to cover Ruby technologies that I believe will help you develop Web 3.0 applications. I believe that Web 3.0 applications will be small, that they can be constructed from existing web applications, and that they can be used to build new web applications. Most Web 3.0 technologies will be important for both clients and services. Web 3.0 software systems will need to find and "understand" information, merge information from different sources, and offer flexibility in publishing information for both human readers and other software systems. Web 3.0 applications will also take advantage of new "cloud" computing architectures and rich-client platforms.

Web 3.0 also means you can create more powerful applications for less money by using open source software, relying on public Linked Data sources, and taking advantage of third-party "cloud" hosting services like Amazon EC2 and Google App Engine.

Reasons for Using Ruby

This book reflects a major trend in software development: optimizing the process by saving programmer time rather than computing resources. Ruby is a concise and effective programming language that I find ideal for many development tasks. Ruby code will probably never run as fast as natively compiled Common Lisp or server-side Java—both of which I also use for development. Ruby hits a sweet spot for me because much of the software that I write simply does not require high runtime performance: web scrapers, text-handling utilities, natural language processing (NLP) applications, system-administration utilities, and low- or medium-volume web sites and web portals.

There are other fine scripting languages. Python in particular is a widely used and effective scripting language that, like Ruby, also finds use in medium- and large-scale systems. The choice of using Ruby for this book is a personal choice. I actually started using Python before Ruby (and used ABC, a precursor to Python, back in ancient history). But once I started using Ruby, I felt that I had happily concluded my personal search for a lightweight scripting language to augment and largely replace the use of Common Lisp and Java in my day-to-day development.

Motivation for Developing Web 3.0 Applications

The world of information will continue to catch up in importance with the physical world. While food, shelter, family, and friends are the core of our existence, we're seeing tighter coupling between the world of information and the physical aspects of our lives. As developers of Web 3.0 technologies and beyond, we have the opportunity to help society in general by increasing our abilities to get the information we need, make optimal decisions, and share with the world both raw information and information-aggregation resources of our own creation.

I consider Web 3.0 technologies to be an evolutionary advance from the original Web and Web 2.0. The original Web is characterized by linked pages and other resources, whereas Web 2.0 is commonly defined by supporting social networks and web-based systems that in general utilize contributions from active users (I would also add the slow integration of Semantic Web technologies to this definition). Only time will tell how Web 3.0 technologies evolve, but my hope is that there will be a balance of support for both human users and software agents—for both consuming and generating web-based information resources.

Evolution of the Web

The evolution of the Web has been greatly facilitated by the adoption of standards such as TCP/IP, HTML, and HTTP. This success has motivated a rigorous process of standardization of Semantic Web technologies, which you will see in Part 2 of this book. Examples from Part 3 take advantage of information resources on the Web that use standards for Linked Data. You will also see the advantages of using standard web-service protocols in Part 4, when we look at techniques for publishing information for both human readers and software agents.

The first version of the Web consisted of hand-edited HTML pages that linked to other pages, which were often written by people with the same interests. The next evolutionary step was database-backed web sites: data in relational databases was used to render pages based on some interaction with human readers. The next evolutionary step took advantage of user-contributed data to create content for other users. The evolution of the Web 3.0 platform will support more automation of using content from multiple sources and generating new and aggregated content.

I wrote this book specifically for software developers and not for general users of the Web, so I am not going to spend too much time on my personal vision for Web 3.0 and beyond. Instead, I will concentrate on practical technologies and techniques that you can use for designing and constructing new, useful, and innovative systems that process information from different sources, integrate different sources of information, and publish information for both human users and software agents.

Book Contents

The first part of this book covers practical techniques for dealing with and taking advantage of rich-document formats. I also present some of the techniques that I use in my work for determining the "sentiment" of text and automatically extracting structured information from text. Part 2 covers aspects of the Semantic Web that are relevant to the theme of this book: discovering, integrating, and publishing information.

Part 3 covers techniques for gathering and processing information from a variety of sources on the Web. Because most information resources do not yet use Semantic Web technologies, I discuss techniques for automatically gathering information from sources that might use custom or ad-hoc formats.

Part 4 deals with large-scale data processing and information publishing. For my own work, I use both the Rails and Merb frameworks, and I will show you how to use tools like Rails and Hadoop to handle these tasks.

Ruby Development

I am assuming that you have at least some experience with Ruby development and that you have a standard set of tools installed: Ruby, `irb`, `gem`, and Rails. Currently, Ruby version 1.8.6 is most frequently used. That said, I find myself frequently using JRuby either because I want to use existing Java libraries in my projects or because I want to deploy a Rails web application using a Java container like Tomcat, JBoss, or GlassFish. To make things more confusing, I'll point out that Ruby versions 1.9.x are now used in some production systems because of better performance and Unicode support. I will state clearly if individual examples are dependent on any specific version of Ruby; many examples will run using any Ruby version.

Ruby provides a standard format for writing and distributing libraries: gems. I strongly encourage you to develop the good habit of packaging your code in gem libraries. Because a strong advantage of the Ruby language is brevity of code, I encourage you to use a "bottom-up" style of development: package and test libraries as gems, and build up domain-specific languages (DSLs) that match the vocabulary of your application domain. The goal is to have very short Ruby applications with complexity hidden in DSL implementations and in tested and trusted gem libraries.

I will use most of the common Ruby programming idioms and assume that you are already familiar with object modeling, using classes and modules, duck typing, and so on.

Book Software

The software that I have written for this book is all released under one or more open source licenses. My preference is the Lesser General Public License (LGPL), which allows you to use my code in commercial applications without releasing your own code. But note that if you improve LGPL code, you are required to share your improvements. In some of this book's examples, I use other people's open source projects, in which cases I will license my example code with the same licenses used by the authors of those libraries.

The software for this book is available in the Source Code/Download area of the Apress web site at `http://www.apress.com`. I will also maintain a web page on my own web site with pointers to the Apress site and other resources (see `http://markwatson.com/books/web3_book/`).

To make it easier for you to experiment with the web-service and web-portal examples in this book, I have made an Amazon EC2 machine image available to you. You'll learn more about this when I discuss cloud services. Appendix A provides instructions for using my Amazon Machine Image (AMI) with the book examples.

Development Tools

I use a Mac and Linux for most of my development, and I usually deploy to Linux servers. I use Windows when required by customers. On the Mac I use TextMate for writing small bits of Ruby code, but I prefer IDEs such as RubyMine, NetBeans, and IntelliJ IDEA, all of which offer good support for Ruby development. You should use your favorite tools—there are no examples in this book that depend on specific development tools. It is worth noting that Microsoft is making Ruby a supported language on the .NET Framework.

PART 1

Text Processing

Part 1 of this book gives you the necessary tools to process text in Web 3.0 applications. In Chapter 1, you'll learn how to parse text from common document formats and convert complex file types to simpler types for easier processing. In Chapter 2, you'll see how to clean up text, segment it into sentences, and perform spelling correction. Chapter 3 covers natural language processing (NLP) techniques that you'll find useful for Web 3.0 applications.

CHAPTER 1

Parsing Common Document Types

Rich-text file formats are a mixed blessing for Web 3.0 applications that require general processing of text and at least some degree of semantic understanding. On the positive side, rich text lets you use styling information such as headings, tables, and metadata to identify important or specific parts of documents. On the negative side, dealing with rich text is more complex than working with plain text. You'll get more in-depth coverage of style markup in Chapter 10, but I'll cover some basics here.

In this chapter, I'll introduce you to the `TextResource` base class, which lets you identify and parse a text resource's tagged information such as its title, headings, and metadata. Then I'll derive several subclasses from it to help you parse text from common document formats such as plain-text documents, binary documents, HTML documents, RSS and Atom feeds, and more. You can use the code as-is or modify it to suit your own needs. Finally, I'll show you a couple command-line utilities you can use to convert PDF and Word files to formats that are easier to work with.

Representing Styled Text

You need a common API for dealing with text and metadata from different sources such as HTML, Microsoft Office, and PDF files. The remaining sections in this chapter contain implementations of these APIs using class inheritance with some "duck typing" to allow the addition of plug-ins, which I'll cover in Chapters 2 and 3. If certain document formats do not provide sufficient information to determine document structure—if a phrase is inside a text heading, for example—then the API implementations for these document types simply return no information.

You want to identify the following information for each input document:

- Title
- Headings
- URI

You use the Ruby class `TextResource` to extract information from any text resource that possibly has its title, headings, and metadata tagged. Here is the complete source listing for the `TextResource` class:

```
class TextResource
  attr_accessor :source_uri
  attr_accessor :plain_text
  attr_accessor :title
  attr_accessor :headings_1
  attr_accessor :headings_2
  attr_accessor :headings_3
  attr_accessor :sentence_boundaries
  attr_accessor :categories
  attr_accessor :place_names
  attr_accessor :human_names
  attr_accessor :summary
  attr_accessor :sentiment_rating # [-1..+1]  positive number
                                  # implies positive sentiment

  def initialize source_uri=''
    @source_uri = source_uri
    @title = ''
    @headings_1 = []
    @headings_2 = []
    @headings_3 = []
  end
  def cleanup_plain_text text # just a placeholder until chapter 2
    text
  end
  def process_text_semantics! text # a placeholder until chapter 3
  end
 end
```

The most important things for you to notice are the attributes and the two placeholder methods. I'll introduce the attributes in this chapter and delve into them further in Chapters 2 and 3, and I'll implement the `cleanup_plain_text` and `process_text_semantics` methods in Chapters 2 and 3, respectively.

■ **Note** The source code for this book contains a single gem library called `text-resource` that contains all the code for the `TextResource` class and other examples developed in Chapters 1 through 3. You can find the code samples for this chapter in the Source Code/Download area of the Apress web site (`http://www.apress.com`).

You will never directly create an instance of the `TextResource` class. Instead, you will use subclasses developed in the remainder of this chapter for specific document formats (see Figure 1-1). In Chapters 2 and 3, you will "plug in" functionality to the base class `TextResource`. This functionality will then be available to the subclasses as well.

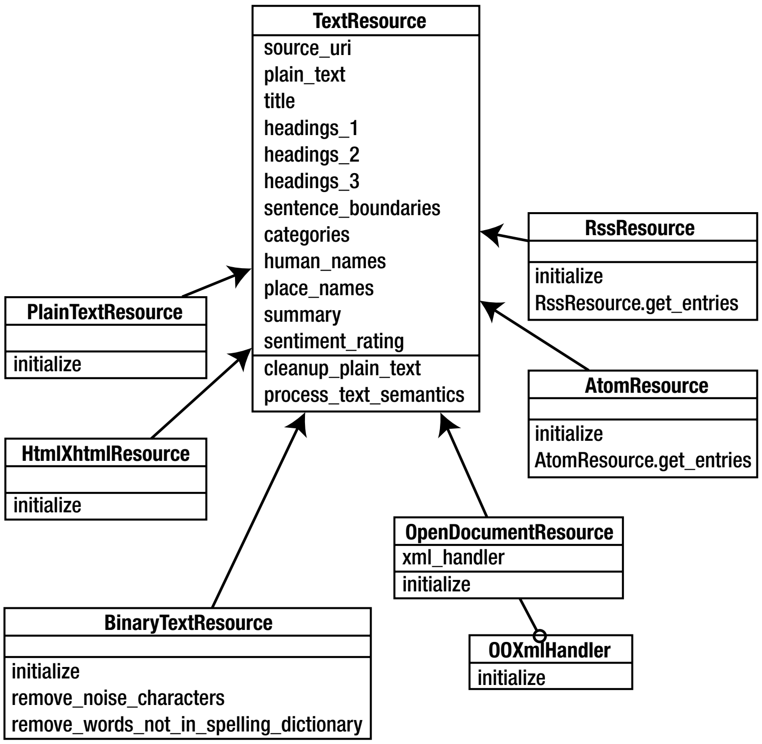

Figure 1-1. *TextResource base class and derived classes*

The `RssResource` and `AtomResource` classes (see Figure 1-1) have static class factories for creating an array of text-resource objects from RSS and Atom blog feeds. (You'll learn more about `RssResource` and `AtomResource` in the corresponding subsections under the section "Implementing Derived Classes for Different Document Types.")

As a practical software developer, I consider it to be a mistake to reinvent the wheel when good open source libraries are available for use. If existing libraries do not do everything that you need, then consider extending an existing library and giving your changes back to the community. I use the following third-party gem libraries in this chapter to handle ZIP files and to parse RSS and Atom data:

- `gem install rubyzip`
- `gem install simple-rss`
- `gem install atom`
- `gem install nokogiri`

These libraries all work with either Ruby 1.8.6 or Ruby 1.9.1.

Implementing Derived Classes for Different Document Types

In this section, I'll show you the implementations of classes that I'll derive from Ruby's `TextResource` class, each of which is shown in Figure 1-1. You can use these derived classes to parse data from the corresponding document types.

Plain Text

The base class `TextResource` is abstract in the sense that it provides behavior and class attribute definitions but does not handle any file types. In this section, I implement the simplest derived class that you will see in this chapter: `PlainTextResource`. The implementation of this class is simple because it only needs to read raw text from an input URI (which can be a local file) and use methods of the base class:

```
class PlainTextResource < TextResource
  def initialize source_uri=''
    super(source_uri)
    file = open(source_uri)
    @plain_text = cleanup_plain_text(file.read)
    process_text_semantics(@plain_text)
  end
end
```

Except for reading text from a URI (web or local file), all other class behavior is implemented from the `TextResource` superclass. You can use this `PlainTextResource` class for any information sources in plain text or for structured data that is externally converted to plain text.

Binary Document Formats

The class you use to parse binary documents differs from the class you use to parse plain-text documents because you need to remove unwanted characters and words. The strategy is to read a binary file as if it were text and then discard nonprinting ("noise") characters and anything that is not in a spelling dictionary. (You'll learn more about noise characters in Chapter 2.)

Here's the code for the `BinaryTextResource` class, which is also derived from the base class `TextResource`:

```
class BinaryTextResource < TextResource
  def initialize source_uri=''
    puts "++ entered BinaryPlainTextResource constructor"
    super(source_uri)
    file = open(source_uri)
    text = file.read
    text = remove_noise_characters(text)
    text = remove_words_not_in_spelling_dictionary(text)
    @plain_text = cleanup_plain_text(text)
    process_text_semantics(@plain_text)
  end
  def remove_noise_characters text
    text # stub: will be implemented in chapter 2
  end
  def remove_words_not_in_spelling_dictionary text
    text # stub: will be implemented in chapter 2
  end
end
```

I'll implement the two stub methods (`remove_noise_characters` and `remove_words_not_in_spelling_dictionary`) in Chapter 2 when I discuss strategies for cleaning up data sources. (You'll also find the complete implementation in the code samples for this book on the Apress web site.)

HTML and XHTML

There are several gem libraries for parsing HTML. I use the Nokogiri library in this chapter because it also parses XML, which means it supports Extensible Hypertext Markup Language (XHTML). So the example code in this section works for both HTML and XHTML. I will discuss only the processing of "clean" HTML and XHTML here; Part 3 of the book covers how to process information from web sites that contain advertisements, blocks of links to other web sites that are not useful for your application, and so on. For those cases, you need to use custom, site-specific web-scraping techniques.

Before showing you the derived class for parsing HTML and XHTML, I'll give you a quick introduction to Nokogiri in which I use Nokogiri's APIs to fetch the HTML from my web site (I'll remove some output for brevity). Here's a snippet from an interactive `irb` session:

```
irb(main):001:0>   require 'nokogiri'
=> true
irb(main):002:0>   require 'open-uri'
=> true
irb(main):003:0> doc = Nokogiri::HTML(open('http://markwatson.com'))
=> ...
>> doc.class
=> Nokogiri::HTML::Document
irb(main):004:0> (doc.public_methods - Object.public_methods).sort
=> ["/", "<<", "[]", "[]=", "add_child", "add_next_sibling",
"add_previous_sibling", "after", "at", "attributes", "before", "blank?",
"cdata?", "child", "children", "collect_namespaces", "comment?",
"content", "content=", "css", "css_path", "decorate", "decorate!", "decorators",
"document", "document=", "encode_special_chars", "get_attribute",
"has_attribute?", "html?", "inner_html", "inner_text", "internal_subset",
"key?", "name=", "namespaces", "next", "next_sibling", "node_cache",
"parent=", "path", "pointer_id", "previous_sibling", "remove", "remove_attribute",
"replace", "root", "root=", "search", "serialize", "set_attribute", "slop!",
"text", "to_html", "to_xml", "traverse", "unlink", "xml?", "xpath"]
```

I suggest that you try the preceding example yourself and experiment with the methods for the `Nokogiri::HTML::Document` class listed at the end of the snippet. (I'll show you portions of `irb` sessions throughout this book.)

In order to extract all of the plain text, you can use the `inner_text` method:

```
irb(main):005:0> doc.inner_text
=> "Mark Watson, Ruby and Java Consultant and Author\n … "
```

The plain text contains new-line characters and generally a lot of extra space characters that you don't want. In the next chapter, you'll learn techniques for cleaning up this text; for now, the `TextResource` base class contains a placeholder method called `cleanup_plain_text` for cleaning text. Nokogiri supports XML Path Language (XPath) processing, DOM-style processing, and Cascading Style Sheets (CSS) processing. I'll start with the DOM (Document Object Model) APIs. I am assuming that you are also using `irb` and following along, so I am showing only the output for the first child element and the inner text of the first child element:

```
irb(main):006:0> doc.root.children.each {|node| pp node; pp node.inner_text }
#<Nokogiri::XML::Element:0x32f618
 @document=
  #<Nokogiri::HTML::Document:0x104e4c0
   @decorators=nil,
   @node_cache=
    {23674064=>
      #<Nokogiri::XML::Element:0x32f2f8
       @document=#<Nokogiri::HTML::Document:0x104e4c0 ...>>,
     23673296=>#<Nokogiri::XML::Element:0x32f618 ...>,
     23672576=>
      #<Nokogiri::XML::Element:0x32fba4
       @document=#<Nokogiri::HTML::Document:0x104e4c0 ...>>}>>
"Mark Watson, Ruby and Java Consultant and Author\n"
```

As you can see, dealing with HTML using DOM is tedious. DOM is appropriate for dealing with XML data that has a published schema, but the free-style nature of HTML (especially "handwritten" HTML) makes DOM processing difficult.

Fortunately, the XPath APIs are just what you need to selectively extract headings from an HTML document. You use XPath to find patterns in nested elements; for example, you'll use the pattern `'//h3'` to match all HTML third-level heading elements. Combine XPath with the `inner_text` method to extract headings:

```
 irb(main):007:0> doc.xpath('//h1')
=> <h1 class="block" align="center">
     Mark Watson: Ruby and Java Consultant and Author
   </h1>
irb(main):008:0> doc.xpath('//h1').inner_text.strip
=> "Mark Watson: Ruby and Java Consultant and Author"
```

By substituting `'//h2'`, `'//h3'`, and `'//h4'` for the XPath expression, you can collect the page headers. As another example, here's how you would collect all of the headings of level `h3`:

```
irb(main):009:0> doc.xpath('//h3').collect {|h| h.inner_text.strip}
=> ["I specialize in Java, Ruby, and Artificial Intelligence (AI) technologies",
"Enjoy my Open Content Free Web Books and Open Source Software", "Recent News"]
```

Now you're ready to use the `HtmlXhtmlResource` class, which is derived from `TextResource` and included in the `text-resource` gem library. This is the code for processing HTML and XHTML resources:

```
  doc = Nokogiri::HTML(open(source_uri))
  @plain_text = cleanup_plain_text(doc.inner_text)
  @headings_1 = doc.xpath('//h1').collect {|h| h.inner_text.strip}
  @headings_2 = doc.xpath('//h2').collect {|h| h.inner_text.strip}
  @headings_3 = doc.xpath('//h3').collect {|h| h.inner_text.strip}
```

The `TextResource` class's `cleanup_plain_text` utility method is currently a placeholder; I'll implement it in Chapter 2. Running the preceding code yields these extracted headers from my web site:

```
 @headings_1=["Mark Watson: Ruby and Java Consultant and Author"],
 @headings_2=["Blogs", "Fun stuff"],
 @headings_3=
  ["I specialize in Java, Ruby, and Artificial Intelligence (AI) technologies",
   "Enjoy my Open Content Free Web Books and Open Source Software",
   "Recent News"],
```

Here is the complete class implementation:

```
class HtmlXhtmlResource < TextResource
  def initialize source_uri=''
    super(source_uri)
    # parse HTML:
    doc = Nokogiri::HTML(open(source_uri))
    @plain_text = cleanup_plain_text(doc.inner_text)
    @headings_1 = doc.xpath('//h1').collect {|h| h.inner_text.strip}
    @headings_2 = doc.xpath('//h2').collect {|h| h.inner_text.strip}
    @headings_3 = doc.xpath('//h3').collect {|h| h.inner_text.strip}
    process_text_semantics(@plain_text)
  end
end
```

This code extracts headings based on the heading level of HTML tags.

Note For JRuby developers, I provide example code in the next section for using the pure Ruby REXML library to grab all text (attributes and element text). For processing HTML, you can use a pure Ruby library such as ymHTML (included in the source code for this book on the Apress web site).

OpenDocument

Now I'll discuss the `OpenDocumentResource` class, which lets you parse text from documents in OpenOffice.org's OpenDocument format. You won't find many web resources in this document format, but it's an international standard that's used by at least five word processors. I include support for OpenDocument in this chapter because this format is ideal for maintaining document repositories. OpenOffice.org offers batch-conversion utilities for converting various Microsoft Office formats and HTML to the OpenDocument format. You can select directories of files for conversion using the application's menus.

The OpenDocument format is an easy-to-read, easy-to-parse XML format that is stored in a ZIP file. First use the standard Ruby ZIP library to extract the ZIP entry named `content.xml`. Then use the REXML XML parser by providing a Simple API for XML (SAX) XML event handler as a nested class inside the implementation of the `OpenDocumentResource` class:

```
class OpenDocumentResource < TextResource
  class OOXmlHandler
    include StreamListener
    attr_reader :plain_text
    attr_reader :headers
```

REXML calls the `tag_start` method for each new starting XML tag:

```
    def tag_start name, attrs
      @last_name = name
    end
```

You need to save the element name so you know what the enclosing element type is when the `text` method is called. REXML calls the `text` method whenever text is found in the input stream. The XML for the document content has many elements starting with `text:`. You'll collect all the inner text from any element whose name contains `text:h` and save it in an array of header titles. You'll also collect the inner text from any element whose name contains `text` and save it in the plain-text buffer:

```
    def text s
      s.strip!
      if @last_name.index('text:h')
        @headers << s if s.length > 0
      end
      if @last_name.index('text')
        if s.length > 0
          @plain_text << s
          @plain_text << "\n"
        end
      end
    end
   end # ends inner class StreamListener
```

The OpenDocumentResource class constructor uses the internal SAX callback class to parse the XML input stream read from the ZIP file entry content.xml:

```
  def initialize source_uri=''
    Zip::ZipFile.open(source_uri) {
      |zipFile|
      xml_h = OOXmlHandler.new
      Document.parse_stream((zipFile.read('content.xml')), xml_h)
      @plain_text = cleanup_plain_text(xml_h.plain_text)
      @headers_1 = xml_h.headers
    }
   process_text_semantics(@plain_text)
  end
end
```

The OpenDocument standard, which is implemented by many word-processing systems, is ideal for creating and maintaining document repositories. Here, you only collected the headers and plain text from OpenDocument files, but the format is richer than the simple Ruby code in the OpenDocumentResource class indicates. If you are interested, I recommend that you try unzipping any OpenDocument file and examine both the metadata and contents-file entries.

I am using OpenOffice.org to write this book. You might find this amusing: I used the OpenDocument file for this chapter as my test data for writing the OpenDocumentResource class.

RSS

Another useful source of information on the Web is web blogs that use either RSS or Atom XML-syndication formats. I originally considered not supporting web blogs as a subclass of TextResource because a single blog URI refers to many blog entries, but I decided to implement RSS and Atom classes with factories for returning an array of blog entries for a single blog URI. These derived classes are called RssResource and AtomResource (see Figure 1-1). This decision makes sense: a static class-factory method returns a collection of TextResource instances, each with the semantic processing performed by the code that you will see in Chapter 3.

The implementation of RSS-feed reading is simple using Lucas Carlson's simple-rss gem library. The simple-rss library handles both RSS 1.0 and RSS 2.0. The RssResource constructor calls the TextResource constructor to initialize instance data to default empty strings and empty lists. The static class method get_entries is a factory that creates an array of RssResource objects from a blog URL:

```
class RssResource < TextResource
   def initialize
    super('')
  end
  def RssResource.get_entries source_uri = ''
    entries = []
```