

Managing Web Content with Perl & XML

XML Publishing with

AxKit



O'REILLY®

Kip Hampton

Your First XML Web Site

With AxKit installed, you can begin putting it through its paces. In this chapter, we create a simple XML-based web site. Along the way, I will introduce AxKit's facilities for how to apply stylesheets to transform data marked up in XML into a commonly used delivery format, how to combine XML from different sources, and how to configure an alternate style transformation to deliver the same XML content in a different format in response to data received from the requesting client.

Preparation

By design, XML processing tools are less forgiving about what they accept than the HTML browsers that you may be used to working with. Omitting a closing tag when creating an element in an HTML page, for example, may cause an undesirable result when the page is rendered, but the browser usually tries to recover gracefully and render *something* for you to see. In contrast, omitting an end tag when creating an element in a document that an XML parser will consume results in a fatal well-formedness error, and no such recovery is possible. In the context of AxKit (in which all XML processing happens on the server), this means that if you pass in a bad document, AxKit sends no content to the client. At best, you see an error message that indicates where things went wrong. To avoid frustration, take a little time to familiarize yourself with the XML processing tools available to you. At the very least, investigate how the XML parser you installed can be used from the command line to verify a document's well-formedness and validity. Being able to catch bad documents going in reduces the overall number of potentially user-visible errors. The ability to verify that your content and stylesheets are at least syntactically correct can make finding the cause of an error easier.

Even more than with a static HTML-based site, starting with a good directory structure is key to creating an easy-to-maintain XML-based site. The time and labor-saving benefits of having predictable paths for images, CSS stylesheets, etc. also apply to the files associated with XML processing. It's a good idea to get in the habit of

creating a *stylesheets* (or similarly named) directory at the base of the host's DocumentRoot when you start a new project.

If you installed the AxKit demonstration site or included the sample *axkit.conf* in your main Apache configuration (covered in “Basic Server Configuration” in Chapter 2), you do not need to alter the web server's configuration at all. If not, follow the directions there, or add the following lines to the web server's *httpd.conf*, and stop and restart the server before proceeding:

```
PerlModule AxKit
AddHandler axkit .xml .xsp .dxb .rdf

AxAddStyleMap text/xsl Apache::AxKit::Language::LibXSLT
AxAddStyleMap application/x-xsp Apache::AxKit::Language::XSP
AxAddStyleMap application/x-xpathscript Apache::AxKit::Language::XPathScript
```

These directives load the AxKit core into Apache, set up the required Language transformation processors, and configure Apache to process all files that end in *.xml*, *.xsp*, *.dxb*, and *.rdf* with AxKit. With an appropriate directory structure and configuration in place, you can move on to creating the XML documents that you want to publish.

Creating the Source XML Documents

Often, many benefits of using an XML publishing framework such as AxKit become obvious only later in a project's life (e.g., the ability to easily add new heavy-duty features to an existing site, or the power to completely change the look and feel of an entire site without touching its content). Given this, any examples you may choose for this introduction will surely fall short of illustrating AxKit's real power. Accepting the notion that the task at hand is a bit absurd frees you to have a little fun with it while still learning the basics. Let's run with the absurdity, and imagine that you are charged with the task of publishing a small site on the very silly subject of cryptozoology.

Cryptozoology (literally, *the study of hidden animals*) is concerned with the gathering and analysis of data related to animals that are frequently reported by local residents or found in popular folklore, but whose existence the scientific community has not formally recognized. Familiar examples include the Yeti, Loch Ness Monster, and Mokele-Mbembe.

The first document for your site, *cryptozoo.xml*, contains a list of cryptozoological species (called *cryptids* by insiders). (See Example 3-1.)

Example 3-1. cryptozoo.xml

```
<?xml version="1.0"?>
<cryptids>
  <species>
```

Example 3-1. *cryptozoo.xml* (continued)

```
<name>Jackalope</name>
<habitat>Western North America</habitat>
<description>
  <para>
    Similar to the Bavarian raurackl (stag-hare), the
    North American Jackalope resembles a large jackrabbit
    with small, deer-like antlers. This vicious
    carnivore is frequently mistaken for common rabbits or hares
    suffering from <italic>papillomatosis</italic> (a condition
    that produces horn-like growths on the head in those species).
  </para>
</description>
</species>
<species>
  <name>Dahut</name>
  <habitat>French Alps</habitat>
  <description>
    <para>
      A shy relative of the Alpine deer, the Dahut has
      adapted to the challenges of its mountainous habitat by
      growing legs that are considerably longer on one side
      of its body. While this asymmetrical limb configuration allows
      for level grazing on steep grades, it leaves the unfortunate
      creature unable to reverse its course. Local hunters exploit
      this weakness by sneaking up behind the Dahut and either
      whistling softly or crying "Dahut!"; when the startled
      creature turns to face its assailant, it finds its
      longer legs on the wrong side and it tumbles to its doom.
    </para>
  </description>
</species>
<!-- ...more species here -->
</cryptids>
```

No cryptozoology site worth its salt is complete without a list of cryptid sightings. Your second XML document, creatively named *cryptid_sightings.xml*, contains just that. (See Example 3-2.)

Example 3-2. *cryptid_sightings.xml*

```
<?xml version="1.0"?>
<sightings>
  <sighting>
    <species>Jersey Devil</species>
    <location>Bordentown, NJ</location>
    <date>Autumn, 1816</date>
    <description>
      <p>
        A Jersey Devil was reportedly seen
        by Joseph Bonaparte, former King of Spain
        and brother of Napoleon, while hunting in
        the woods near Bordentown, New Jersey.
      </p>
    </description>
  </sighting>
</sightings>
```

Example 3-2. *cryptid_sightings.xml* (continued)

```
</p>
</description>
<witnesses>
  <name>Joseph Bonaparte</name>
</witnesses>
</sighting>
<sighting>
  <species>Snipe</species>
  <location>Phelan, CA</location>
  <date>June 2002</date>
  <description>
    <p>
      The Phelan Phine Snipe Hunters Association
      celebrated the opening of this year's Snipe
      season. Unfortunately, the entire
      photographic record of the event was ruined
      during a nasty "keg stand" incident in
      the beer tent after the hunt.
    </p>
  </description>
  <witnesses>
    <name>Jason Nugall</name>
    <name>William Q. Rozborne</name>
  </witnesses>
</sighting>
</sightings>
```

You need one last XML document: a small file that captures the filenames of the two other documents in the site. You will use this bit of metadata to create the navigation in the final result delivered to the requesting HTML browser, so the name *nav.xml* seems appropriate. (See Example 3-3.)

Example 3-3. *nav.xml*

```
<?xml version="1.0"?>
<links>
  <a href="cryptozoo.xml">Species</a>
  <a href="cryptid_sightings.xml">Sightings</a>
</links>
```

Writing the Stylesheet

So far, you have three XML documents that contain three very different, but randomly overlapping, grammars. (The `species` and `name` elements appear in different roles in the two main content documents.) Your goal is to make this information available on the Web to HTML browsers. You want to reach the widest possible audience, and that means maintaining the lowest possible expectations of the requesting client's capabilities. That is, you cannot rely on everyone who wants to read your pages having a thoroughly modern browser capable of doing appropriate

client-side transformations to your XML documents via CSS or XSLT. You must deliver basic HTML if you expect your data to be widely accessible.

With this in mind, you need a way to transform the disparate data structures contained in each of your XML documents into the unified grammar of simple HTML. That's where AxKit's transformational languages and stylesheets enter the picture. AxKit offers many ways to transform XML data. (We will examine the merits of many of these in later chapters.) In this example, we examine how you can transform your cryptozoology documents into HTML using two of the more popular transformation languages: XSLT and XPathScript.

I will save the examination of the lower-level details of these languages for later. At this point, it suffices to understand that both XSLT and XPathScript offer a declarative syntax that provides a way to create new documents by applying transformations to all or some of the elements, attributes, and other content that an existing XML document contains.

Using XSLT

Rather than taking small steps through the XSLT stylesheet, I present it here in one block to give you an idea of what a full, working stylesheet looks like. (See Example 3-4.) Do not worry if much of it seems foreign; we will look at the syntactic elements of XSLT in more detail in Chapter 5.

As you read through the stylesheet, keep in mind that:

- An XSLT stylesheet itself is an XML document.
- Transformation rules are applied based on the properties of the various parts of the source XML document (element and names, relationships between elements, etc.).
- Template rules are created to match all elements of the grammars found in your XML documents, so the same stylesheet can be used to transform both the list of species and the list of sightings.

Example 3-4. cryptozoo.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0"
  >
<xsl:output
  method="html" encoding="ISO-8859-1"
/>

<xsl:template match="/">
<html>
  <head><title>Cryptozoology Pages</title>
```

Example 3-4. *cryptozoo.xml* (continued)

```
<link rel="stylesheet" type="text/css" href="crypto.css"/>
</head>
<body>
<div class="header">
<h1>My Cryptozoology Pages</h1>
</div>
<div class="nav">
  <xsl:apply-templates select="document('nav.xml', /)/*/"/>
</div>
<div class="content">
  <xsl:apply-templates/>
</div>
</body>
</html>
</xsl:template>

<!-- top-level templates -->
<xsl:template match="animals">
  <p>
    Today's rural legend is tomorrow's newly
    discovered species.
  </p>
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="sightings">
  <p>
    Here is a list of sightings.
  </p>
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="animals/species">
<div class="species">
  <xsl:apply-templates/>
</div>
</xsl:template>

<xsl:template match="sighting">
<div class="sighting">
  <xsl:apply-templates/>
</div>
</xsl:template>

<xsl:template match="species/name">
<h2>
  <xsl:apply-templates/>
</h2>
</xsl:template>

<xsl:template match="species/habitat">
  <p>
```

Example 3-4. *cryptozoo.xml* (continued)

```
<i>Habitat:</i>
<xsl:text> </xsl:text>
<xsl:value-of select="."/>
</p>
<xsl:apply-templates select="*" />
</xsl:template>

<xsl:template match="sighting/location|sighting/species|sighting/date">
  <div class="subheading">
    <i><xsl:value-of select="name()" /></i>
    <xsl:text> </xsl:text>
    <xsl:value-of select="."/>
  </div>
  <xsl:apply-templates select="*" />
</xsl:template>

<xsl:template match="witnesses">
  <div>
    <i>witnesses:</i>
    <xsl:text> </xsl:text>
    <xsl:for-each select="name">
      <xsl:value-of select="." />
      <xsl:if test="position() != last()">, </xsl:if>
    </xsl:for-each>
  </div>
</xsl:template>

<xsl:template match="witnesses/name">
  <xsl:text> </xsl:text>
  <xsl:value-of select="."/>
</xsl:template>

<xsl:template match="para|p">
<p>
  <xsl:apply-templates/>
</p>
</xsl:template>

<xsl:template match="italic">
<i>
  <xsl:apply-templates/>
</i>
</xsl:template>

<xsl:template match="a">
  <xsl:copy-of select="." /><br/>
</xsl:template>

</xsl:stylesheet>
```

Using XPathScript

Compare the previous XSLT stylesheet, which transforms your cryptozoology documents into HTML, with the one below, written in XPathScript. I will leave the syntactic details for Chapter 6; however, as you read through the stylesheet, note that most moving parts are written in Perl and separated from the larger template document using `<%` and `%>` as delimiters.

```
<%
# declarative templates
$t->{'p'}{pre} = '<p>';
$t->{'p'}{post} = '</p>';

$t->{'para'}{pre} = '<p>';
$t->{'para'}{post} = '</p>';

$t->{'animals'}{pre} = qq|
  <p>
    Today's rural legend is tomorrow's newly
    discovered species.
  </p>
|;

$t->{'sightings'}{pre} = qq|
  <p>
    Here is a list of sightings.
  </p>
|;

$t->{'sighting'}{pre} = '<div class="sighting">';
$t->{'sighting'}{post} = '</div>';

$t->{'habitat'}{pre} = '<p><i>Habitat:</i> ';
$t->{'habitat'}{post} = '</p>';

$t->{'location'}{pre} = '<div class="subheading"><i>location: </i>';

$t->{'location'}{post} = '</div>';

$t->{'date'}{pre} = '<div class="subheading"><i>date: </i>';
$t->{'date'}{post} = '</div>';

$t->{'witnesses'}{pre} = '<div><i>witnesses: </i>';
$t->{'witnesses'}{post} = '</div>';

$t->{'a'}{showtag} = 1;

# testcode templates for like-named elements
$t->{'name'}{testcode} = sub {
  my ($node, $t) = @_;
  if (findnodes('parent::species', $node)) {
```

```

        $t->{pre} = '<h2>';
        $t->{post} = '</h2>';
    }
    return 1;
};

$t->{'species'}{testcode} = sub {
    my ($node, $t) = @_;
    if (findnodes('parent::animals', $node)) {
        $t->{pre} = '<div class="species">';
        $t->{post} = '</div>';
    }
    else {
        $t->{pre} = '<div class="subheading"><i>species: </i>';
        $t->{post} = '</div>';
    }
    return 1;
};
%>

<html>
  <head><title>Cryptozoology Pages</title></head>
  <link rel="stylesheet" type="text/css" media="screen" href="crypto.css"/>
  <body>
    <div class="header">
      <h1>My Cryptozoology Pages</h1>
    </div>
    <div class="nav">
      <%= apply_templates( "document('nav.xml')" ) %>
    </div>
    <div class="content">
      <%= apply_templates() %>
    </div>
  </body>
</html>

```

Do not be overwhelmed. Remember that most sites typically use *either* XSLT or XPathScript and only rarely both, so you need not try to take in both at once. These duplicated examples only intend to show that with AxKit, as with Perl, there is always more than one way to do it. You are free to choose the tools and techniques that suit you best.

Associating the Documents with the Stylesheet

AxKit offers a variety of configuration options for associating documents with its various language processors. Chapter 4 covers each in detail. In Example 3-5, you create an *.htaccess* file in the same directory as your XML documents. It defines a default style for AxKit to use when processing documents in this directory.

Example 3-5. A simple `.htaccess` file

```
<AxStyleName "#default">
  AxAddProcessor text/xsl stylesheets/cryptozoo.xsl
</AxStyleName>
```

Pay attention to the arguments passed to the `AxAddProcessor` directive. The first is the MIME type that AxKit examines to decide which language processor modules to use, and the second is the DocumentRoot-relative path to the stylesheet that will be passed to that language processor to transform your XML documents. If you want to use your XPathScript stylesheet rather than the XSLT, you would use `AxAddProcessor application/x-xpathscript stylesheets/cryptozoo.xps` instead. This processor definition is wrapped in an `AxStyleName` block. This directive block, in turn, combines the processor definitions it contains into a single “named style” that a `StyleChooser` or other plug-in can select at runtime. By giving this style the special name `#default`, you are configuring AxKit to use this style as a fallback if no other style is explicitly selected.

It’s time to fire up a web browser and check the results of your work. A request to `http://myhost.tld/cryptozoo.xml` yields what is shown in Figure 3-1.



Figure 3-1. `cryptozoo.xml` rendered as HTML

Clicking on the Sightings link reveals what is shown in Figure 3-2.

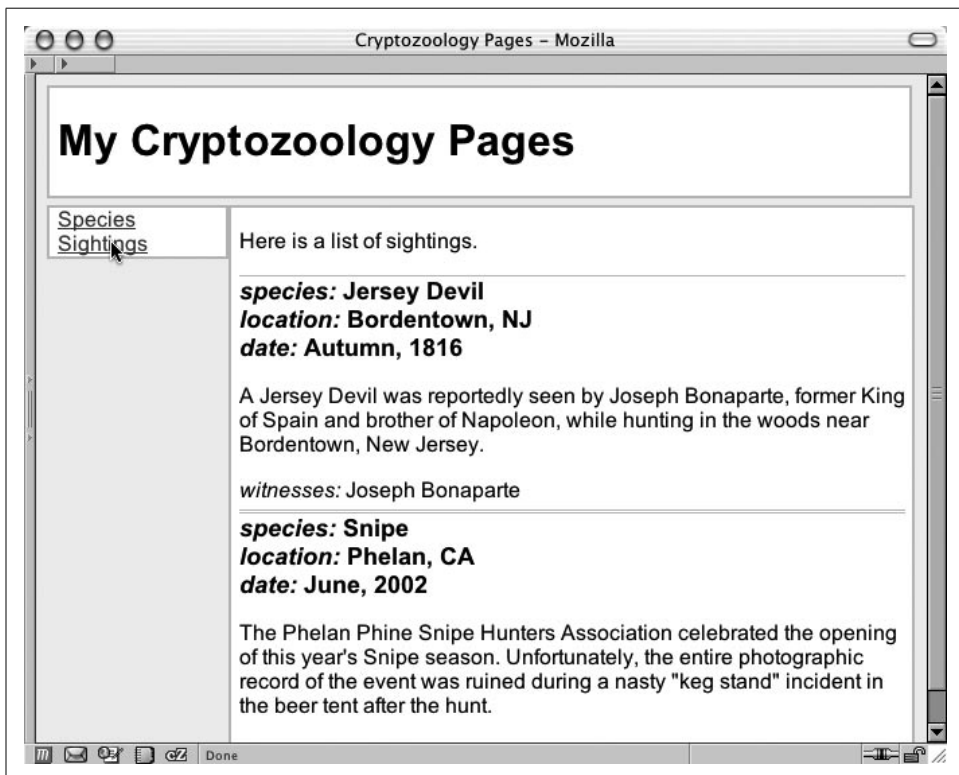


Figure 3-2. *cryptid_sightings.xml* rendered as HTML

A Step Further: Syndicating Content

You have reached your initial goal of publishing your XML documents for consumption by HTML browsers on the Web using AxKit. Even if that were all you ever wanted to do, you still made a clear division between the content you will maintain and the way in which it is presented. Among other benefits, you can now redesign the look and feel of pages sent to the client without touching content documents. Don't worry about clobbering or obscuring essential information just to change the way it renders in a browser. Similarly, using a custom XML grammar for your content means that the documents themselves can unambiguously define the intended roles of the data they contain, rather than the way that data may be represented on the visual medium of an HTML browser. This makes reusing the data for other purposes a lot easier.

To understand the practical benefits of separating content from presentation, suppose that your list of cryptid sightings becomes wildly popular on the Web. People

start asking for a way to put links to the newly reported sighting on their own cryptozoology sites. You could tell them to screen-scrape the HTML list. Instead, you decide to be a good information-sharing citizen and make the list available as an RSS syndication feed. To achieve this, the first thing you need is a stylesheet that transforms the list of cryptid sightings to RSS, in addition to the one you already have that transforms the data into HTML.

For those who may not be familiar with it, RSS (RDF Site Summary, Rich Site Summary, or Really Simple Syndication, depending on whom you ask) is a popular XML grammar used for syndicating online content, especially news headlines. Most weblogs use RSS as the means to both publish content and share links with other bloggers, and many weblog tools store their data natively as RSS. (See Example 3-6.) For more information about RSS and some of its more creative uses, see Ben Hamersley's *Content Syndication with RSS* (O'Reilly).

Example 3-6. cryptidsightings_rss.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns="http://purl.org/rss/1.0/"
  version="1.0"
>

<xsl:variable
  name="this_url"
  select="'http://myhost.tld/cryptosightings.xml'"
/>

<xsl:template match="/">
<rdf:RDF>
  <channel rdf:about="{ $this_url }">
    <title>My Cryptozoology Pages- Sightings</title>
    <link>
      <xsl:value-of select="$this_url"/>
    </link>
    <description>
      The latest sightings of animals that do not officially exist.
    </description>
    <items>
      <rdf:Seq>
        <xsl:for-each select="/sightings/sighting">
          <rdf:li rdf:resource="{ $this_url }#{position()}" />
        </xsl:for-each>
      </rdf:Seq>

      </items>
    </channel>
```

Example 3-6. *cryptidsightings_rss.xml* (continued)

```
<xsl:apply-templates select="/sightings/sighting"/>
</rdf:RDF>
</xsl:template>

<xsl:template match="sighting">
<item rdf:about="{ $this_url }#{position()}">
  <link>
    <xsl:value-of select="concat($this_url,'#', position())"/>
  </link>

  <title>
    <xsl:value-of select="species"/> Sighting - <xsl:value-of select="date"/>
  </title>
  <description>
    <xsl:apply-templates select="description"/>
  </description>
</item>
</xsl:template>

<xsl:template match="description|p">
  <xsl:value-of select="normalize-space(.)" />
</xsl:template>

</xsl:stylesheet>
```

Before you can see this stylesheet in action, you need to add a couple of configuration directives to the *.htaccess* file you created earlier:

```
<Files cryptid_sightings.xml>
  <AxStyleName rss1>
    AxAddProcessor text/xsl stylesheets/cryptidsightings_rss.xml
  </AxStyleName>
  AxAddPlugin Apache::AxKit::StyleChooser::QueryString
</Files>
```

The `AxStyleName` block creates a named style called `rss1`. The `AxAddProcessor` it contains associates that named style with the RSS stylesheet you just created. The `AxAddPlugin` directive, in this case, tells `AxKit` to use additional logic to examine the query string sent from the requesting client for a parameter named `style`. If it finds one, and the value of that parameter matches one of the named styles configured for that URI, it uses the stylesheets contained in that named style to transform the XML source document. Here, this means that a request to `http://myhost.tld/cryptid_sightings.xml?style=rss1` returns the list of species sightings processed by your RSS output stylesheet, not the default style you created earlier. (See Figure 3-3.)

Example 3-7 shows the result for a request for the *cryptid_sightings.xml* file as an RSS document.

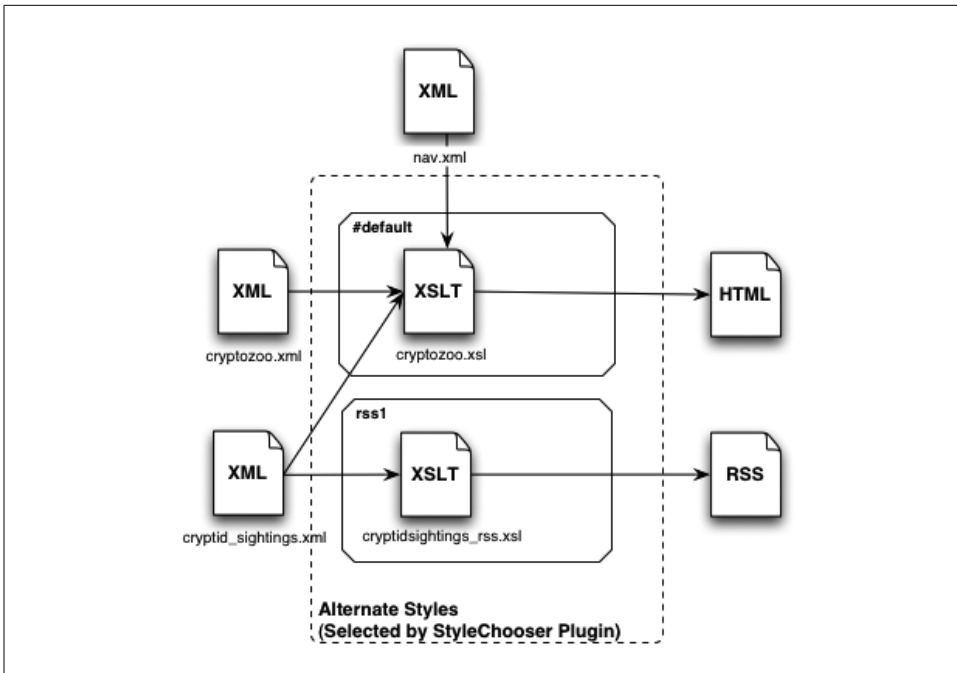


Figure 3-3. Cryptozoology site-processing diagram

Example 3-7. Cryptid sightings delivered as an RSS feed

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns="http://purl.org/rss/1.0/">
  <channel rdf:about="http://myhost.tld/cryptosightings.xml">
    <title>My Cryptozoology Pages- Sightings</title>
    <link>http://myhost.tld/cryptosightings.xml</link>
    <description>
      The latest sightings of animals that do not officially exist.
    </description>
    <items>
      <rdf:Seq>
        <rdf:li rdf:resource="http://myhost.tld/cryptosightings.xml#1"/>
        <rdf:li rdf:resource="http://myhost.tld/cryptosightings.xml#2"/>
      </rdf:Seq>
    </items>
  </channel>
  <item rdf:about="http://myhost.tld/cryptosightings.xml#1">
    <link>http://myhost.tld/cryptosightings.xml#1</link>
    <title>Jersey Devil Sighting - Autumn, 1816</title>
    <description>
      A Jersey Devil was reportedly seen
      by Joseph Bonaparte, former King of Spain
      and brother of Napoleon, while hunting in
      the woods near Bordentown, New Jersey.
```

Example 3-7. Cryptid sightings delivered as an RSS feed (continued)

```
</description>
</item>
<item rdf:about="http://myhost.tld/cryptosightings.xml#2">
  <link>http://myhost.tld/cryptosightings.xml#2</link>
  <title>Snipe Sighting - June, 2002</title>
  <description>
    The Phelan Phine Snipe Hunters Association
    celebrated the opening of this year's Snipe
    season. Unfortunately, the entire
    photographic record of the event was ruined
    during a nasty "keg stand" incident in
    the beer tent after the hunt.
  </description>
</item>
</rdf:RDF>
```

Figure 3-4 shows a request to that same URL rendered in the NetNewsWire RSS client.

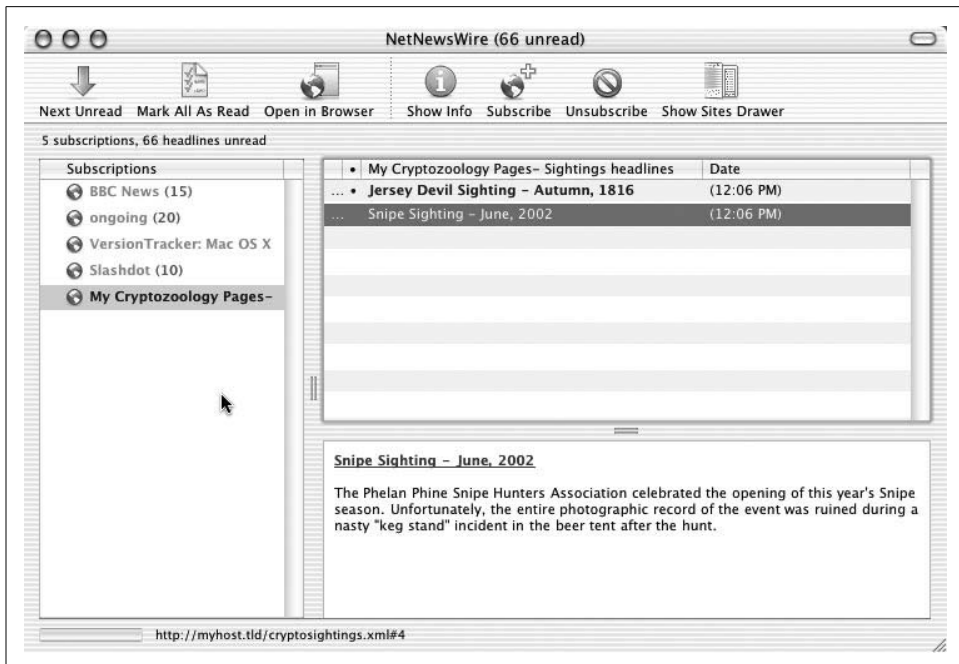


Figure 3-4. Information rendered as RSS in NetNewsWire

Now your friends in the cryptozoology community can add your list of sightings to their list of daily news feeds by pointing their RSS viewer or other client to `http://myhost.tld/cryptid_sightings.xml?style=rss1`, while casual web surfers get the default HTML version. Keep in mind, too, that the query string `StyleChooser` is only one

way to dynamically select the preferred style. Using other plug-ins, you could just as easily examine the connecting client's User-Agent header, or another aspect of the request, to get the same effect. Remember, too, that XSLT and XPathScript are only two transformative Language modules that AxKit supports; there are several others, each with its own unique strengths and weaknesses.

As promised, your first example site is a bit silly (Dahuts, indeed), but let's examine exactly what you have done:

- You used stylesheets to transform data marked up in custom XML grammars into a commonly used delivery format (HTML).
- You combined XML from different resources (the content documents and the navigation metadata) to meet your application's needs.
- You mixed standard Apache configuration blocks with AxKit's custom directives to select an alternative style transformation that delivered the same XML content in a different format in response to data received from the requesting client.

Taken together, these points represent the basic foundation of publishing XML documents with AxKit. You can apply the patterns you learned here to most, if not all, of your future XML publishing projects. However, your little cryptids' site offers only a glimpse of the flexibility and power that AxKit offers. The following chapters build on these basic concepts to show how you can use AxKit to create sophisticated, dynamic, and feature-rich sites.