SAMPLE CHAPTER

Tika IN ACTION

Chris A. Mattmann
Jukka L. Zitting

FOREWORD BY JÉRÔME CHARRON

MANNING

Tika in Action

by Chris A. Mattmann
Jukka L. Zitting

Chapter 8

brief contents

8 What's in a file?

This chapter covers

- File formats
- Extracting content from files
- How file storage impacts data extraction

By now, your Tika-fu is strong, and you're feeling like there's not much that you can't do with your favorite tool for file detection, metadata extraction, and language identification. Believe it or not, there's plenty more to learn!

One thing we've purposefully stayed away from is telling you what's in those files that Tika makes sense of.[1] That's because files are a source of rich information, recording not only text or metadata, but also things like detailed descriptions of scenery, such as a bright image of a soccer ball on a grass field; waveforms representing music recorded in stereo sound; all the way to geolocated and time-referenced observations recorded by a Fourier Transform Spectrometer (FTS) instrument on a spacecraft. The short of the matter is that their intricacies and complexities deserve treatment in their own right.

[1] We covered some parts of the file contents, for example, we discussed BOM markers in chapter 4 while talking about file detection. In chapter 5, we discussed methods for dealing with file reading via `InputStream`s. In both cases, we stayed away from the *actual contents* of files in particular, since it would receive full treatment in this chapter.

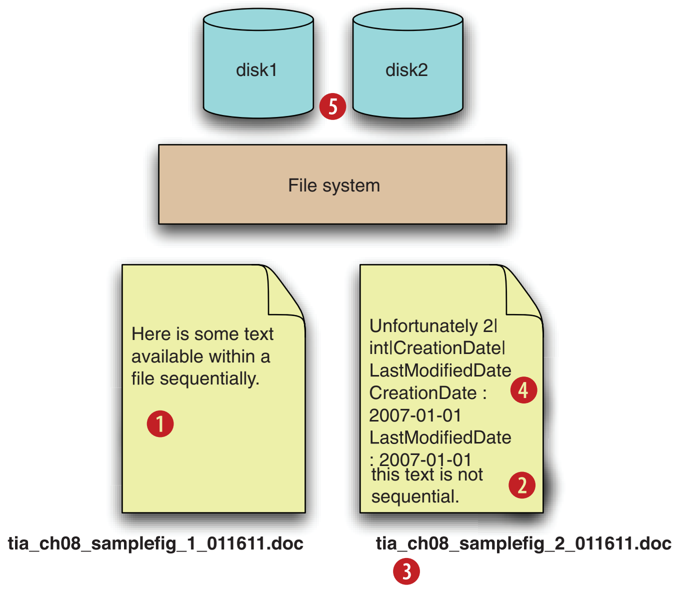

disk1
disk2
5
File system
Here is some text available within a file sequentially.
1
Unfortunately 2| int|CreationDate| LastModifiedDate CreationDate : 2007-01-01 LastModifiedDate : 2007-01-01
4
this text is not sequential.
2
tia_ch08_samplefig_1_011611.doc
tia_ch08_samplefig_2_011611.doc
3

Figure 8.1 Several areas where content can be gleaned from a file

Files store their information using different methodologies as shown in figure 8.1. The information may be available only by sequentially scanning each byte of information recorded in the logical file, as shown at ❶ in the figure, or it may be accessible randomly by jumping around through the file, as ❷ in the figure demonstrates. Metadata information may be available by reading the file's header (its beginning bytes on disk) as shown at ❹, or it could be stored in a file's name or directory structure on disk as shown at ❸. Finally, files may be physically split across multiple parts on disk, or they may be logically organized according to some common collection somehow as shown at ❺. It sounds complex, and it is, but we'll hone in on Tika's ability to exploit these complexities.

In this chapter, we'll cover all internal and external aspects of files, as well as how Tika exploits this information to extract textual content and metadata. Files, their content, their metadata, and their storage representation are all fair game. The big takeaway from this chapter is that it'll show you how to develop your own Tika parsers and methodologies for extracting information from files using Tika, demonstrated by looking at how existing Tika parsers exploit file content for some common file formats like RSS and HDF. Let's dive in!

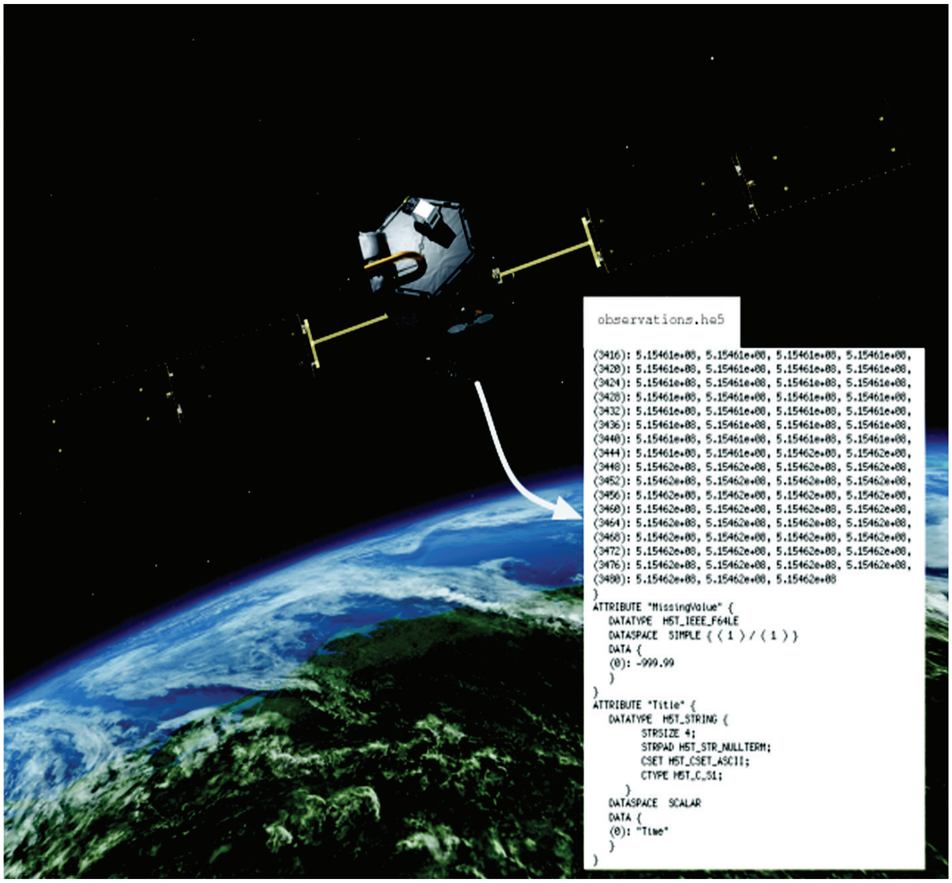
8.1 Types of content

The types of content within files vary vastly. We've picked two sample file format types to examine in this section: the Hierarchical Data Format (HDF), http://www.hdfgroup.org/, a common file format used to capture scientific information, and Really Simple Syndication (RSS), the most commonly used format to spread news and rapidly changing information.

8.1.1 *HDF: a format for scientific data*

Consider a scenario in which a science instrument flown on a NASA satellite records data, which is then downlinked via one of a number of existing ground stations here on the earth, as shown in figure 8.2. The downlinked data is then transferred via dedicated networks to a science data processing center for data transformation, and ultimately for dissemination to the public.

In this scenario the raw data arriving at the science data processing center represents engineering and housekeeping information, including raw voltages from the instrument and rudimentary location information (such as an orbit number). This information is represented as a series of files, each corresponding to one channel of data from the instrument (three channels in total), and one set of three files per each orbit of the satellite around the earth.



```
(3416): 5.15461e+08, 5.15461e+08, 5.15461e+08, 5.15461e+08,
(3420): 5.15461e+08, 5.15461e+08, 5.15461e+08, 5.15461e+08,
(3424): 5.15461e+08, 5.15461e+08, 5.15461e+08, 5.15461e+08,
(3428): 5.15461e+08, 5.15461e+08, 5.15461e+08, 5.15461e+08,
(3432): 5.15461e+08, 5.15461e+08, 5.15461e+08, 5.15461e+08,
(3436): 5.15461e+08, 5.15461e+08, 5.15461e+08, 5.15461e+08,
(3440): 5.15461e+08, 5.15461e+08, 5.15461e+08, 5.15461e+08,
(3444): 5.15461e+08, 5.15461e+08, 5.15462e+08, 5.15462e+08,
(3448): 5.15462e+08, 5.15462e+08, 5.15462e+08, 5.15462e+08,
(3452): 5.15462e+08, 5.15462e+08, 5.15462e+08, 5.15462e+08,
(3456): 5.15462e+08, 5.15462e+08, 5.15462e+08, 5.15462e+08,
(3460): 5.15462e+08, 5.15462e+08, 5.15462e+08, 5.15462e+08,
(3464): 5.15462e+08, 5.15462e+08, 5.15462e+08, 5.15462e+08,
(3468): 5.15462e+08, 5.15462e+08, 5.15462e+08, 5.15462e+08,
(3472): 5.15462e+08, 5.15462e+08, 5.15462e+08, 5.15462e+08,
(3476): 5.15462e+08, 5.15462e+08, 5.15462e+08, 5.15462e+08,
(3480): 5.15462e+08, 5.15462e+08, 5.15462e+08
}
ATTRIBUTE "MissingValue" {
   DATATYPE  H5T_IEEE_F64LE
   DATASPACE  SIMPLE { ( 1 ) / ( 1 ) }
   DATA {
   (0): -999.99
   }
}
ATTRIBUTE "Title" {
   DATATYPE  H5T_STRING {
         STRSIZE 4;
         STRPAD H5T_STR_NULLTERM;
         CSET H5T_CSET_ASCII;
         CTYPE H5T_C_S1;
      }
   DATASPACE  SCALAR
   DATA {
   (0): "Time"
   }
}
```

Figure 8.2 **A postulated satellite scenario, observing the earth and collecting those observations in data files represented in the Hierarchical Data Format (HDF). HDF stores data in a binary format, arranged as a set of named scalars, vectors, and matrices corresponding to observations over some space/time grid.**

Data within each channel file is stored initially in a binary data format; for the purposes of this example we'll assume the widely used Hierarchical Data Format (HDF), version 5 (HDF5). HDF provides an external user-facing API for writing to and reading from HDF5 files. The HDF5 API allows users to write data using a small canonical set of data constructs, specifically those shown in table 8.1.

Table 8.1 Simplified representation of content within Hierarchical Data Format (HDF) files. HDF represents observational data and metadata information using a small set of constructs: named scalars, vectors, and matrices.

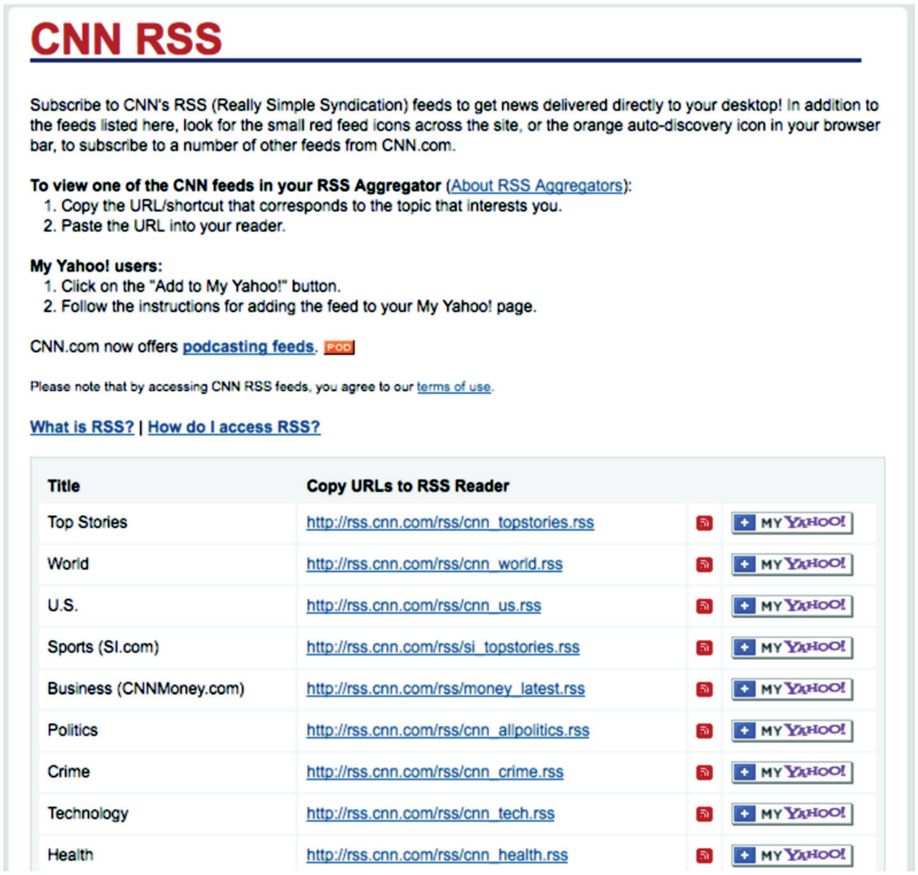
Data type	Description
Scalar	Named scalar data, such as single-valued metadata information, numerical, or string-based. Examples might include Mission Name, with the associated scalar value Orbiting Carbon Observatory, as well as Instrument Type, with an associated scalar value of Spectrometer.
Vector (aka "1-dimensional Arrays" in HDF5)	Named vector data, including multivalued metadata or multivalued numerical arrays of integers and floats. Examples may include a set of latitudes corresponding to the satellite orbit path.
Matrix (aka "2-dimensional Arrays" in HDF5)	Named matrix data, including multidimensional numerical array data such as integers and floats. The information contained inside of these data types may correspond to a pixel matrix of some scene observed by the instrument, such as a 45 x 30 matrix of temperatures stored as float values (measured in some unit, such as kelvins), where each value in the 45 x 30 matrix has a corresponding latitude or longitude, stored in some associated additional matrices in the HDF5 file.

All of the data and metadata from our postulated scenario is represented in a set of three HDF5 files (corresponding to each channel of the instrument) for each orbit the satellite makes around the earth. That means that if the instrument is measuring a set of scientific variables, such as air temperature, wind speed, CO_2, or any number of other variables, that information is represented in the HDF5 files as sets of named scalars, vectors, and matrices.

8.1.2 Really Simple Syndication: a format for rapidly changing content

Let's consider another scenario, such as a Really Simple Syndication (RSS) feed file that lists the latest news stories provided by CNN.com, an example of which is provided in figure 8.3.

RSS files are based on a simple but powerful data model. Each RSS file is an XML file adhering to a prescribed XML schema that defines the RSS vocabulary. That vocabulary consists of two main data structures. First, each RSS file typically contains a *channel*, which aggregates a set of associated RSS *items*, each of which typically points to some news story of interest. Every RSS channel has a set of metadata associated with it, such as a URL and description (http://www.cnn.com/sports/ for the URL and "Latest news stories about sports within the last hour" as the description), as does each RSS item tag.



CNN RSS

Subscribe to CNN's RSS (Really Simple Syndication) feeds to get news delivered directly to your desktop! In addition to the feeds listed here, look for the small red feed icons across the site, or the orange auto-discovery icon in your browser bar, to subscribe to a number of other feeds from CNN.com.

To view one of the CNN feeds in your RSS Aggregator (About RSS Aggregators):
1. Copy the URL/shortcut that corresponds to the topic that interests you.
2. Paste the URL into your reader.

My Yahoo! users:
1. Click on the "Add to My Yahoo!" button.
2. Follow the instructions for adding the feed to your My Yahoo! page.

CNN.com now offers podcasting feeds. POD

Please note that by accessing CNN RSS feeds, you agree to our terms of use.

What is RSS? | How do I access RSS?

Title	Copy URLs to RSS Reader		
Top Stories	http://rss.cnn.com/rss/cnn_topstories.rss		+ MY YAHOO!
World	http://rss.cnn.com/rss/cnn_world.rss		+ MY YAHOO!
U.S.	http://rss.cnn.com/rss/cnn_us.rss		+ MY YAHOO!
Sports (SI.com)	http://rss.cnn.com/rss/si_topstories.rss		+ MY YAHOO!
Business (CNNMoney.com)	http://rss.cnn.com/rss/money_latest.rss		+ MY YAHOO!
Politics	http://rss.cnn.com/rss/cnn_allpolitics.rss		+ MY YAHOO!
Crime	http://rss.cnn.com/rss/cnn_crime.rss		+ MY YAHOO!
Technology	http://rss.cnn.com/rss/cnn_tech.rss		+ MY YAHOO!
Health	http://rss.cnn.com/rss/cnn_health.rss		+ MY YAHOO!

Figure 8.3 The CNN Really Simple Syndication (RSS) index page. CNN provides a set of RSS files that users can subscribe to in order to stay up to date on all of their favorite news stories, categorized by the type of news that users are interested in.

In the CNN example, CNN publishes sets of RSS files, each containing an RSS channel, one for each CNN news category (such as Top Stories, World, U.S., or any of the other categories in figure 8.3). Each RSS channel has a corresponding set of latest news stories and links that users can subscribe to via any number of different RSS readers, including most modern web browsers.

Understanding the types of content is the first step toward automatically extracting information from it. We'll go into the details of that in the next section, describing how Tika codifies the process of extracting content.

8.2 How Tika extracts content

By now, you've seen that engineers often must write applications that can understand many different file types, including HDF5 and RSS files as discussed earlier. The organization of content within different file types has a strong effect on the methodology

Tika uses to extract information from them, as well as the overall performance of the extraction process.

In particular, the organization of content within a file impacts Tika's two main approaches to content extraction. The first is Tika's ability to access a file in a streaming fashion, extracting content as it's read, in contrast to reading the whole file at once, extracting the content, and being able to access it randomly. The next section will demonstrate how Tika extracts content no matter how it's organized!

8.2.1 *Organization of content*

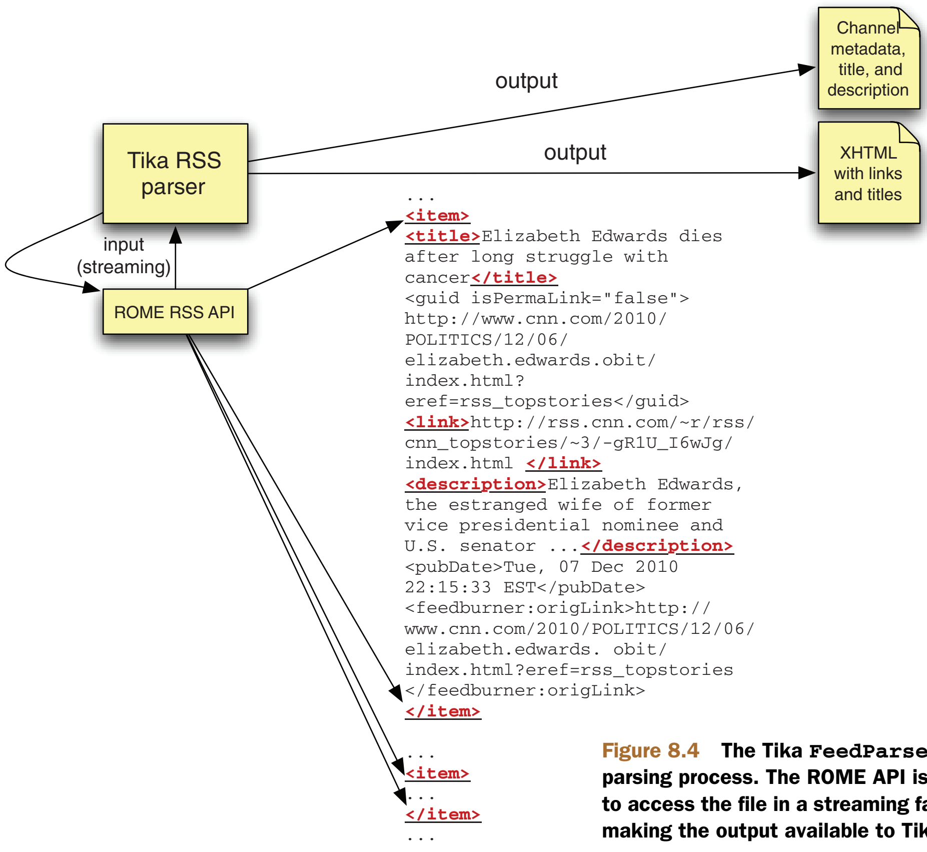
We'll spend this section examining how Tika makes sense of content, whether it supports streaming or random access in the context of RSS files and HDF5 files. We'll use Tika's `FeedParser` and `HDFParser` classes to demonstrate. Onward!

STREAMING

Content that's organized as a set of discrete, independent chunks within a file can be interpreted in a *streaming* fashion. Those independent chunks can be read in or out of order, and the entire file isn't required to make sense of those chunks—they make sense on their own. RSS is an XML-based file format that's amenable to streaming.

We'll start off by putting an RSS file under the microscope and by inspecting its organization:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<rss version="2.0">
  <channel>
    <title>CNN.com</title>
    <link>http://www.cnn.com/?eref=rss_topstories</link>
    <description>
      CNN.com delivers up-to-the-minute news and information ...
    </description>
    <language>en-us</language>
    <copyright>© 2010 Cable News Network LP, LLLP.</copyright>
    <pubDate>Tue, 07 Dec 2010 22:25:36 EST</pubDate>
    <ttl>5</ttl>
    <image>...</image>
    <item>
      <title>Elizabeth Edwards dies ...</title>
      <guid isPermaLink="false">...</guid>
      <link>http://rss.cnn.com/...</link>
      <description>Elizabeth Edwards, the ...</description>
      <pubDate>Tue, 07 Dec 2010 22:15:33 EST</pubDate>
    </item>
    <item>
      <title>Obama slams GOP, ...</title>
      ...
    </item>
    <item>
      <title>WikiLeaks founder sent to jail</title>
      ...
    </item>
    ...
  </channel>
</rss>
```


output
output
Channel metadata, title, and description
XHTML with links and titles
Tika RSS parser
input (streaming)
ROME RSS API

```
...
<item>
<title>Elizabeth Edwards dies
after long struggle with
cancer</title>
<guid isPermaLink="false">
http://www.cnn.com/2010/
POLITICS/12/06/
elizabeth.edwards.obit/
index.html?
eref=rss_topstories</guid>
<link>http://rss.cnn.com/~r/rss/
cnn_topstories/~3/-gR1U_I6wJg/
index.html </link>
<description>Elizabeth Edwards,
the estranged wife of former
vice presidential nominee and
U.S. senator ...</description>
<pubDate>Tue, 07 Dec 2010
22:15:33 EST</pubDate>
<feedburner:origLink>http://
www.cnn.com/2010/POLITICS/12/06/
elizabeth.edwards. obit/
index.html?eref=rss_topstories
</feedburner:origLink>
</item>

...
<item>
...
</item>
...
```

Figure 8.4 The Tika `FeedParser`'s parsing process. The ROME API is used to access the file in a streaming fashion, making the output available to Tika.

One advantage of RSS is that it's implemented using a specific XML dialect as mentioned earlier. An RSS document consists of a single `channel` tag, wrought with `item` tags with descriptive information such as links to the actual content (in this case, the news story), along with information about when the story was published, who published it, an optional media file, and so on.

Tika's `org.apache.tika.parser.feed.FeedParser` class exploits the underlying content structure of the RSS file to extract its text and metadata information, as depicted in figure 8.4. The open source RSS parser library ROME is used for handling the nitty-gritty details of the RSS format.

The following listing starts off with the gory details. We'll discuss the listing shortly after.

Listing 8.1 Tika's RSS feed parser exploiting RSS's XML-based content structure

```
public void parse(
        InputStream stream, ContentHandler handler,
        Metadata metadata, ParseContext context)
        throws IOException, SAXException, TikaException {
    try { //
```

(1) **Leverage ROME API for parsing**

```
        SyndFeed feed = new SyndFeedInput().build(
               new InputSource(stream)); //            ❷ Parse using SAX parser

        String title = stripTags(feed.getTitleEx()); //
        String description = stripTags(feed.getDescriptionEx());

        metadata.set(Metadata.TITLE, title);
        metadata.set(Metadata.DESCRIPTION, description);   ❸ Extract core channel metadata

        ... //                             See next listing

        xhtml.endDocument();
    } catch (FeedException e) {
        throw new TikaException("RSS parse error", e);
    }
}
```

As should be second nature by now (if not, head back over to chapter 5), Tika parsers implement the `parse(...)` method defined in the `org.apache.tika.parser.Parser` interface. The `FeedParser` begins by leveraging the ROME API for RSS feed processing, as shown in ❶. ROME allows for stream-based XML parsing via its SAX-based parse interface, as shown in ❷. In doing so, Tika is able to exploit the SAX parsing model for XML and take advantage of a number of its emergent properties, including low memory footprint and faster result processing. Once Tika hands off the RSS input stream to ROME, ROME provides methods, as shown in the bottom portion of the listing, that allow extraction of information from the RSS channel, which Tika's `FeedParser` flows into its extracted metadata, as shown in ❸.

> **WHEN IN ROME** The Java ROME API (humorously subtitled *All feeds lead to ROME*) is the most widely developed and most actively used Java API for RSS feed parsing. ROME handles a number of the modern RSS formats in development including RSS 2.0 and ATOM. Tika uses ROME's RSS parsing functionality because, well, it rocks, and there's no reason to write it again.

Here's the second half of the `parse` method.

Listing 8.2 The latter half of the `FeedParser`'s parse method: extracting links

```
XHTMLContentHandler xhtml =
    new XHTMLContentHandler(handler, metadata);
xhtml.startDocument();

xhtml.element("h1", title);
xhtml.element("p", description);

xhtml.startElement("ul");                          ❶ Use ROME to extract feed items
for (Object e : feed.getEntries()) {
    SyndEntry entry = (SyndEntry) e;
    String link = entry.getLink();                 ❷ Obtain item links
    if (link != null) {
        xhtml.startElement("li");
        xhtml.startElement("a", "href", link);

        xhtml.characters(stripTags(entry.getTitleEx()));   ❸ Output Tika XHTML links and title text
```

```
            xhtml.endElement("a");
            SyndContent content = entry.getDescription();
            if (content != null) {
                xhtml.newline();
                xhtml.characters(content.getValue());
            }
            xhtml.endElement("li");
        }
    }
    xhtml.endElement("ul");
```

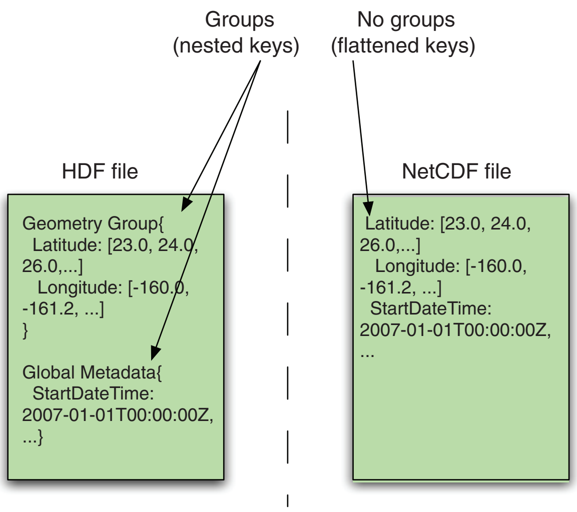
After the channel `Metadata` has been extracted, the `FeedParser` proceeds to iterate over each `Item` in the feed as shown in ❶. The first step is to use ROME's `SyndEntry` class, which represents a single `Item` from the `Channel`. For each `Item`, its links and metadata are extracted as shown in ❷. Once the information has been extracted, it's output as XHTML in the final step ❸ of listing 8.2.

Tika was able to exploit the underlying content organization of an RSS file and the associated ROME library's easy API. ROME provided access to RSS file information to extract both RSS metadata and link text in a streaming fashion, sending the information to Tika's `FeedParser` class.

Now that we've seen streaming, let's take another example, this time looking at an HDF5 file and how Tika's `HDFParser` is affected by the underlying file content organization. The HDF5 file format prohibits random access of information, requiring the user to have an API which loads the entire file into memory before accessing it.

RANDOM ACCESS

Tika's `HDFParser` builds on top of the NetCDF Java API. NetCDF is a popular binary scientific format, similar to HDF5 except for how data and metadata are stored, as shown in figure 8.5. In HDF5 (and prior versions), data and metadata can be grouped into different associations, connected by a common group name. In NetCDF, all of the data and metadata within the file is assumed to be "flat," and all within the global group.



Groups (nested keys)
No groups (flattened keys)
HDF file
Geometry Group{ Latitude: [23.0, 24.0, 26.0,...] Longitude: [-160.0, -161.2, ...] }
Global Metadata{ StartDateTime: 2007-01-01T00:00:00Z, ...}
NetCDF file
Latitude: [23.0, 24.0, 26.0,...] Longitude: [-160.0, -161.2, ...] StartDateTime: 2007-01-01T00:00:00Z, ...

Figure 8.5 A side-by-side comparison of HDF and NetCDF. HDF supports grouping of keys like putting Latitude and Longitude inside of the Geometry group. NetCDF doesn't support grouping, and the keys are all flattened and ungrouped.

As it turns out, the underlying content model of scalars, vectors, and matrices (remember table 8.1?) is so similar for HDF5 and NetCDF4 that we can leverage the same Java API (originally intended for NetCDF4) to read HDF5. Let's take a look at Tika's `HDFParser` and see.

Listing 8.3 The Tika `HDFParser`'s `parse` method

```
public void parse(InputStream stream, ContentHandler handler,
        Metadata metadata, ParseContext context) throws IOException,
        SAXException, TikaException {
    ByteArrayOutputStream os = new ByteArrayOutputStream();
    this.writeStreamToMemory(stream, os);                    <-- Must read entire
                                                                 file in memory

    NetcdfFile ncFile = NetcdfFile.openInMemory("", os.toByteArray());  <-- Use NetCDF API
                                                                            to parse file
                                                   Helper function to
                                                   extract string met
    this.unravelStringMet(ncFile, null, metadata);  <--
}
```

Much of the magic of the `HDFParser` lies in the `unravelStringMet` function which we'll look at shortly. But there's one important aspect of the parser to point out, and it has to do directly with the way that the HDF5 content is organized. HDF5 does *not* support random access (in contrast to RSS, which as we saw in the prior section, *does* support random access). Because of this limitation, the API used to read the HDF5 file must be given the *entire* file contents as a `ByteArray` as shown in upper portion of the following listing.

Listing 8.4 The Tika `HDFParser`'s `unravelStringMet` method

```
protected void unravelStringMet(NetcdfFile ncFile, Group group,
  Metadata met)
{
    if (group == null) {
        group = ncFile.getRootGroup();
    }

    // unravel its string attrs                               Only consider
    for (Attribute attribute : group.getAttributes()) {  <-- HDF scalars
        if (attribute.isString()) {
            met.add(attribute.getName(), attribute.getStringValue());
        } else {
            // try and cast its value to a string
            met.add(attribute.getName(), String.valueOf(attribute
                    .getNumericValue()));                <-- Typecast values
        }                                                    to Strings
    }

    for (Group g : group.getGroups()) {                      Flatten and
        unravelStringMet(ncFile, g, met);                <-- unpack recursively
    }
}
```

Luckily Tika supports both types of files: those that support random access, and those that don't. We should note that file formats that support random access typically also support file streaming, as was the case with the `FeedParser` example from listing 8.1.

Now that we've seen how the organization of a file's content can influence the way that Tika extracts information from it, we'll focus on how a file's header structure and naming conventions can also play a big role in how Tika extracts metadata information. File creators codify information in all sorts of different ways; we'll have to do some detective work for the upcoming sections, but luckily Tika is like your Watson to help unravel the mystery!

8.2.2 File header and naming conventions

A file's metadata information can take many forms and be locked away in a number of places. In some cases, you can examine the first few bytes of information (sometimes called a *file header*) and obtain rich semantic information. In other cases, you're forced to look elsewhere to find metadata to extract from a particular file, including the file's name and (in some cases) its directory structure.

In this section, we'll examine both areas of metadata that a file presents, and we'll show you how Tika is the right tool for the job no matter which one the file uses to codify its metadata.

FILE HEADERS

Depending on the file type, it's possible to focus on just the file header information in order to extract useful information. Let's take a real HTML page as an example:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<!--  -->
<html><!-- InstanceBegin template="/Templates/book.dwt"
codeOutsideHTMLIsLocked="false" -->
<head>
<!-- InstanceBeginEditable name="doctitle" -->
<title>Manning: Tika in Action</title>
<!-- InstanceEndEditable -->
<link href="../styles/main.css" rel="stylesheet" type="text/css" />
<!-- InstanceBeginEditable name="head" --><!--
InstanceEndEditable -->
<meta name = "keywords" content = "Apache, Tika, content analysis,
language
identification, mime detection, file format, Lucene, Solr, Nutch,
search engine,
indexing, full text, parser, MIME-INFO, freedesktop.org, Office Open
XML, PDF,
Zitting, Mattmann, metadata, Dublin Core, XMP, ISO 11179, MIME type,
media type,
magic bytes, IETF" />
</head>

<body>
<!-- ... -->
</body>
</html>
```

In the sample HTML file, note the `<meta>` tag and its attribute `name`, which lists a set of keywords about the HTML page. In this particular example, we've omitted much of the actual page in between the `<body>` tags, focusing only on the *header* of the file.

HEAD HUNTING The HTML file format uses the tag `<head>` to denote the HTML file header, a pointer to the location of header information. This is an area where meta information about the page is placed, including stylesheets, JavaScript, base links, and other global page information.

Most HTML parsers provide a mechanism to extract header information fron the HTML file, and Tika's `HtmlParser` leverages this functionality to pull out the file metadata as shown next.

Listing 8.5 Snippet of Tika's `HtmlHandler` class that deals with meta tags

```
if ("META".equals(name) && atts.getValue("content") != null) {

    if (atts.getValue("http-equiv") != null) {
        metadata.set(
                atts.getValue("http-equiv"),
                atts.getValue("content"));
    } else if (atts.getValue("name") != null) {
        metadata.set(
                atts.getValue("name"),
                atts.getValue("content"));
    }
}
```

Parse and add meta keywords

Other file formats have similar notions of file header information. JPEG and other image formats are good examples of this behavior. For example, most image formats encode the size, color depth, and other similar facts about a image within the first few hundred bytes of the file.

But it's not always possible to get all the metadata information present in a file by examining its header. In other cases, the entire file needs to be parsed. (Remember the `HDFParser` from the previous section?) In some other cases, we don't even need to crack open the file to get the metadata information. We'll specifically look at those cases in the next section, where we examine file naming conventions as a means of metadata extraction.

FILE NAMING CONVENTIONS

File naming conventions sometimes convey metadata. Many people name their files intuitively based on some hierarchy of how the files should be organized in their mind, or attributing to some other criteria.

For example, let's look at the following output from a UNIX /bin/ls command run on a local machine of ours. Note that this command was run on Mac OS X 10.6, with the -FG option also provided as an alias to ls (in other words the command is actually ls -lFG). We've redacted the host names and other identifying information to protect the innocents.

```
[host:~/src/tikaInAction] unixuser% ls -l
total 25944
-rw-r--r--@  1 unixuser  unixgrp      5590 Sep 30 14:07 Tika-in-Action.xml
-rw-r--r--@  1 unixuser  unixgrp       585 Jun 18 08:53 assembly.xml
-rw-r--r--@  1 unixuser  unixgrp    268853 Sep 30 14:07 cover.jpg
drwxr-xr-x  13 unixuser  unixgrp       442 Nov 22 20:42 figs/
drwxr-xr-x   4 unixuser  unixgrp       136 Apr 19  2010 misc/
-rw-r--r--@  1 unixuser  unixgrp      3373 Sep 30 14:07 pom.xml
drwxr-xr-x   8 unixuser  unixgrp       272 Sep 30 22:07 src/
drwxr-xr-x  11 unixuser  unixgrp       374 Nov 22 16:33 target/
-rw-r--r--@  1 unixuser  unixgrp     73088 Nov 22 12:53 tia-ch01.xml
-rw-r--r--@  1 unixuser  unixgrp     50181 Nov 22 12:56 tia-ch02.xml
-rw-r--r--@  1 unixuser  unixgrp     49612 Sep 30 14:07 tia-ch03.xml
-rw-r--r--@  1 unixuser  unixgrp     71947 Nov 22 13:02 tia-ch04.xml
-rw-r--r--@  1 unixuser  unixgrp     77175 Nov 22 13:50 tia-ch05.xml
-rw-r--r--@  1 unixuser  unixgrp     63988 Nov 22 13:45 tia-ch06.xml
-rw-r--r--@  1 unixuser  unixgrp     30700 Nov 22 13:27 tia-ch07.xml
-rw-r--r--@  1 unixuser  unixgrp      7076 Nov 22 21:18 tia-ch08.xml
-rw-r--r--@  1 unixuser  unixgrp      1917 Nov 21 20:21 tia-ch09.xml
-rw-r--r--@  1 unixuser  unixgrp       223 Sep 30 14:07 tia-ch10.xml
-rw-r--r--@  1 unixuser  unixgrp       167 Sep 30 14:07 tia-ch11.xml
-rwx------@  1 unixuser  unixgrp  12085010 Nov 22 21:20 tika.pdf*
```

This listing output holds a lot of information. First, we can gather information such as who created the file, when it was created, how big it is, and the permissions for reading, writing, and executing the file. We'll see later that the actual file path, including both its directory path and filename, provide metadata that we can extract.

For the first part of information available from the /bin/ls output, we can leverage Tika's `Parser` interface to write a `Parser` implementation which will allow us to extract `Metadata` from the /bin/ls output. Let's cook up the example next.

Listing 8.6 Leveraging directory information to extract file metadata

```
public void parse(InputStream is, ContentHandler handler, Metadata metadata,
    ParseContext context) throws IOException, SAXException, TikaException {

  List<String> lines = FileUtils.readLines(TikaInputStream.get(is).
          getFile());
  for (String line : lines) {
    String[] fileToks = line.split("\s+");
    if (fileToks.length < 8)                          ❶ Ignore
      continue;                                         nonlisting entries
    String filePermissions = fileToks[0];
    String numHardLinks = fileToks[1];                 ❷ Parse line cols
    String fileOwner = fileToks[2];                       from /bin/ls
    String fileOwnerGroup = fileToks[3];
    String fileSize = fileToks[4];
    StringBuffer lastModDate = new StringBuffer();
    lastModDate.append(fileToks[5]);
    lastModDate.append(" ");
    lastModDate.append(fileToks[6]);
    lastModDate.append(" ");
    lastModDate.append(fileToks[7]);
    StringBuffer fileName = new StringBuffer();
    for (int i = 8; i < fileToks.length; i++) {
        fileName.append(fileToks[i]);
        fileName.append(" ");
      }
      fileName.deleteCharAt(fileName.length() - 1);
      this
          .addMetadata(metadata, filePermissions, numHardLinks,
                  fileOwner,
              fileOwnerGroup, fileSize, lastModDate.toString(),
              fileName
                  .toString());                    ❸ Add extracted file meta
    }
  }


  private void addMetadata(Metadata metadata, String filePerms,
      String numHardLinks, String fileOwner, String fileOwnerGroup,
      String fileSize, String lastModDate, String fileName) {
    metadata.add("FilePermissions", filePerms);
    metadata.add("NumHardLinks", numHardLinks);        ❹ Add scalar meta
    metadata.add("FileOwner", fileOwner);
    metadata.add("FileOwnerGroup", fileOwnerGroup);
    metadata.add("FileSize", fileSize);
    metadata.add("LastModifiedDate", lastModDate);
    metadata.add("Filename", fileName);

    if (filePerms.indexOf("x") != -1 &&
            filePerms.indexOf("d") == -1) {
      if (metadata.get("NumExecutables") != null) {
        int numExecs = Integer.valueOf(
                metadata.get("NumExecutables"));
        numExecs++;
        metadata.set("NumExecutables", String.valueOf(numExecs));
      } else {
        metadata.set("NumExecutables", "1");      ❺ Add derived meta
      }
    }
  }
```

In effect, our `Parser` implementation is a glorified streaming line tokenizer, pulling out the relevant pieces of the /bin/ls output as shown in ❷ and ❸. The `Parser` implementation tokenizes each line on whitespace, and ignores lines such as the first line where the information provided is summary information—specifically the total size of the directory as shown in ❶.

One nice feature of the provided `Parser` implementation is that it not only extracts scalar metadata (shown in ❹) from the /bin/ls output, but it extracts derived metadata (shown in ❺). In this case, it extracts the number of executables it could find by counting the number of files that contain the `x` permission in their `FilePermissions` extracted metadata field.

We can see the output of listing 8.6 by running a command similar to the following:

```
ls -l | java -classpath \
tika-app-1.0.jar:tika-in-action-SNAPSHOT.jar:commons-io-1.4.jar \
tikainaction.chapter8.DirListParser
```

Or, you can also use Maven to execute the command:

```
ls -l | mvn exec:java -Dexec.mainClass="tikainaction.chapter8.DirListParser"
```

Now that we've seen how to examine basic file and directory information available as output from a listing command,[2] let's look at what metadata information we can extract if we examine the full file path including both its directory and filename components. Figure 8.6 magnifies a particular file from the directory listing output and demonstrates the file's conceptual parts.

Taking an example from the /bin/ls command output, note the different components of the full file path. Information including who created the file, which book the file is associated with, and the chapter number are all available by examining the file and directory naming conventions.

The file includes information such as its creator (denoted by the particular user ID part of the /Users/ folder on the file system), the book that the file is associated with (the first set of information *after*/src/ and before tikaInAction), as well as the chapter number (the information available after the hyphen in tia-ch and before .xml in the filename). These rules that we've codified in parentheses provide a recipe for exploiting filename and directory information to extract useful and relevant metadata that we may use in processing the associated file.

There's one other major place where metadata information may lie. Files often point to *other files* which may themselves have metadata associated with *both files* or all files in a particular collection. Let's see how we can extract and leverage this link information using Tika.

LINKS TO OTHER FILES

Often files, in order to cut down on the amount of direct metadata and text they capture, will reference other files. In the MS Office suite, you can create explicit hyperlinks between Word documents and Excel files, or Excel files and PowerPoint files, and so on. In HTML files, you can create <a> tags with an `href` attribute, a hyperlink pointing at related content from the origin HTML file. In content management

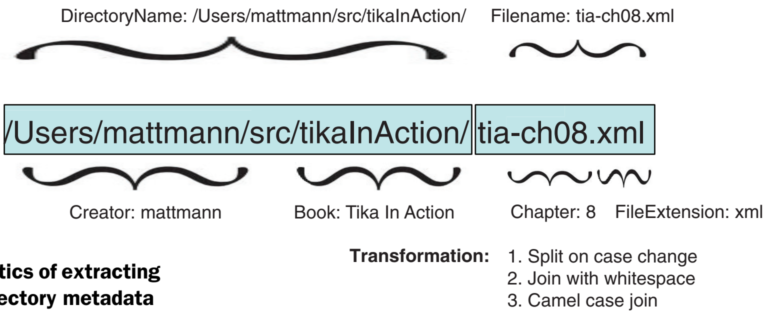


Figure 8.6 The semantics of extracting file and directory metadata

[2] We used /bin/ls, a basic UNIX utility. Similar information would have been available if we used the Windows dir command.

systems, you can also explicitly create links between web pages, documents, and other forms of media as part of the content metadata information. There are numerous examples of file-based linking; these are only a representative few.

File links themselves are valuable metadata information because they may point us to other forms of rich associated content, ripe for extraction. So, how does Tika help you with deciphering things such as links to other files?

Tika uses a SAX `ContentHandler` interface mechanism to allow output from its XHTML extraction step to be customized in some specific way. One of the useful `ContentHandler` implementations included in Tika is the `LinkContentHandler` class. This class is responsible for taking document link information extracted by the underlying parser, and making it easily available to downstream Tika API users. The main snippet of the `LinkContentHandler` class is highlighted next.

Listing 8.7 Tika's `LinkContentHandler` class makes extracting file links a snap

```
public void startElement(
        String uri, String local, String name, Attributes attributes) {
    if (XHTML.equals(uri)) {
        if ("a".equals(local)) {                                    ❶ Detected <a> tag
            LinkBuilder builder = new LinkBuilder("a");
            builder.setURI(attributes.getValue("", "href"));
            builder.setTitle(attributes.getValue("", "title"));
            builderStack.addFirst(builder);                         ❷ Cache extracted link
        } else if ("img".equals(local)) {
            LinkBuilder builder = new LinkBuilder("img");
            builder.setURI(attributes.getValue("", "src"));
            builder.setTitle(attributes.getValue("", "title"));
            builderStack.addFirst(builder);                               Cache extracted
                                                                        ❸ image link
            String alt = attributes.getValue("", "alt");
            if (alt != null) {
                char[] ch = alt.toCharArray();
                characters(ch, 0, ch.length);
            }
        }
    }
}

public void endElement(String uri, String local, String name) {
    if (XHTML.equals(uri)) {
        if ("a".equals(local) || "img".equals(local)) {
            links.add(builderStack.removeFirst().getLink());
        }                                                   Commit extracted
    }                                                     ❹ links to link set
}
```

The `LinkContentHandler` first determines whether it's encountered an <a> tag within the XHTML as shown in ❶. If it has found an <a> tag, the `LinkContentHandler` extracts its `href` and `title` attributes as shown in ❷. The `LinkContentHandler` also inspects <img> tags and extracts their relevant links as demonstrated in ❸. All of the extracted links are then passed onto the downstream handler, shown in ❹.

The last section of the chapter is up next. In it, we'll explain how the physical and logical representation of how a file is stored affects methods for information extraction, and throws off ordinary toolkits that purport to do text and metadata extraction. Good thing you have Tika, and good thing it's no ordinary toolkit!

8.2.3 Storage affects extraction

The mechanism by which a file is stored on media may transmit useful information worthy of Tika's extraction. These mechanisms include the logical representation of files via storage, such as through links (such as symbolic links), as well as the notion that files can be sets of independent physical files linked together somehow.

Files can be physically stored on a single disk or via the network. Sometimes files may be physically distributed—as in the case of networked file systems like Google File System (GFS) or Hadoop Distributed File System (HDFS)—but centrally represented via a collection of network data blocks or some other higher-order structure. We'll discuss how Tika's use of the `InputStream` abstraction hides some of this complexity and uniqueness.

Individual files may be stored on disk as part of a larger whole of logically or physically linked files via some mechanism such as a common collection label, or a unique directory to collect the files. Tika doesn't care because it can exploit information from either case. Madness, you say? Read on!

LOGICAL REPRESENTATION

Let's postulate a simple example of software deployment to illustrate how logical representation of files and directories may convey otherwise-hidden meaning that we'll want to bring out in the open using Tika. Take, for example, the software deployment scenario in figure 8.7.

In our postulated scenario, software is extracted from a configuration management system—let's say Apache Subversion—and then run through a deployment process which installs the latest and greatest version of the software into the /deploy directory, giving the installed software a unique version number. A symbolic link, titled *current*, is also updated to point to the most recent installed version of the software as a result of this process.

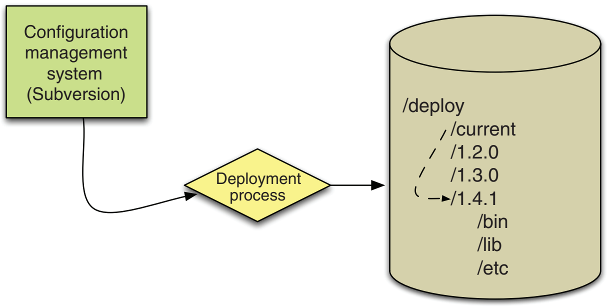


Figure 8.7 A software deployment scenario in which the system is pulled out of configuration management, run through a deployment process that copies and installs the software to a directory path, and codified with the unique software version number. A symlink titled *current* points to the latest and greatest installed version of the software.

What if we wanted to write a quick software program that would roll back the software to the last prior working version when there's some critically identified bug as a result of the latest software deployment process? Let's whip something up with Tika that could address this problem for us!

Listing 8.8 A sample program to roll back a software version using Tika

```
public void rollback(File deployArea) throws IOException, SAXException,
    TikaException {
  LinkContentHandler handler = new LinkContentHandler();
  Metadata met = new Metadata();
  DeploymentAreaParser parser = new DeploymentAreaParser();
  parser.parse(IOUtils.toInputStream(deployArea.getAbsolutePath()),
          handler,
      met);                                            <-- Extract versions
  List<Link> links = handler.getLinks();                   from deploy area
  if (links.size() < 2)
    throw new IOException("Must have installed at least 2 versions!");
  Collections.sort(links, new Comparator<Link>() {          <-- Sort by
    public int compare(Link o1, Link o2) {                      version desc
      return o1.getText().compareTo(o2.getText());
    }
  });

  this.updateVersion(links.get(links.size() - 2).getText());   <-- Roll back to
}                                                                  prior version
```

The example program from listing 8.8 first passes along the path of the deployment area to the `DeploymentAreaParser` class, whose `parse` is shown in listing 8.9. The `DeploymentAreaParser` reads the underlying logical file structure, determining which files are actual deployed software versions in contrast to symlink files pointing to the current version. The returned software version directories are made available by calling `LinkContentHandler`'s `getLinks` method, and then the directories are sorted in descending order. To roll back, we pass along the second-to-last version directory to a function called `updateVersion`, where we update the version to the prior stable software. Not too shabby, huh?

Listing 8.9 A custom Tika `Parser` implementation for our deployment area

```
  public void parse(InputStream is, ContentHandler handler,
      Metadata metadata, ParseContext context) throws IOException,
      SAXException, TikaException {

    File deployArea = new File(IOUtils.toString(is));     <-- Obtain deploy
    File[] versions = deployArea.listFiles(new                area path
            FileFilter() {

      public boolean accept(File pathname) {
          return !pathname.getName().startsWith("current");
      }
    });
    XHTMLContentHandler xhtml = new
      XHTMLContentHandler(handler, metadata);
    xhtml.startDocument();                              Iterate over
                                                        deployed
    for (File v : versions) {                        <- versions
      if(isSymlink(v)) continue;
      xhtml.startElement("a", "href", v.toURL().toExternalForm());
      xhtml.characters(v.getName());                <- Extract file info,
      xhtml.endElement("a");                           ignore symlink
    }

}
```

PHYSICAL REPRESENTATION

If we expand our focus beyond the logical links between files and consider how those files are actually represented on disk, we arrive at a number of interesting information sources ripe for extraction. For example, considering that more and more file systems are moving beyond simple local disks to farms of storage devices, we're faced with an interesting challenge. How do we deal with the extraction of information from a file if we only have available to us a small unit of that file? Even worse, what do we do if that small unit available to us is *not* a "power" unit like the file header?

The reality is that we need a technology that can abstract away the mechanism by which the file is actually stored. If the storage mechanism and physical file representation were abstracted away, then the extraction of text and agglomeration of metadata derived from a file could easily be fed into Tika's traditional extraction processes that we've covered so far.

This is *precisely* why Tika leverages the `InputStream` as the core data passing interface to its `Parser` implementations via the `parse(...)` method. `InputStreams` obfuscate the underlying storage and protocol used to physically represent file contents (or sets of files). Whether it's a GFS URL pointer to a file that's distributed as blocks over the network, or a URL pointer to a file that's locally on disk, Tika still deals with the information as an `InputStream` via a call to `URL.openStream`. And URLs aren't the only means of getting `InputStreams`—`InputStreams` can be generated from `Files`, `byte[]` arrays, and all sorts of objects, making it the right choice for Tika's abstraction for the file physical storage interface.

8.3 Summary

Bet you never thought files had such an influence over how their information is consumed! This chapter served as a wake-up call to the reality that a file's content organization, naming conventions, and storage on disk can greatly influence the way that meaning is derived from them.

- *File content and organization*—We started out by showing you how file content organization can affect performance and memory properties, and influence how Tika parses out information and metadata. In the case of RSS, its content organization (based on XML) allows for easy streaming and random access, whereas in the case of HDF5, the entire file had to be read into memory, precluding streaming, but supporting random access.

- *Extracting file header information and exploiting naming conventions*—The middle portion of the chapter focused on file header metadata and file naming conventions, showing how in many cases metadata can be extracted from a file without even having to open the file. This feature can greatly affect the ability to easily and quickly catalog metadata about files, using Tika as the extractor.
- *File storage and how it affects extraction*—The last important aspect of files is the physical location of a file (or set of associated files) on disk. In many cases, individual files are part of some larger conglomerate, as in the case of directories and split files generated by archive/compression utilities. We examined how this information can be exploited by Tika to extract text and metadata that would normally be impossible to extract when considering each file in isolation.

The next fork in the road will take us to the advanced use and integration of Tika into the larger search ecosystem. You should be well prepared for this journey by now and hopefully eager to see where Tika fits with other information technologies!

JAVA/SEARCH

Tika IN ACTION

C. A. Mattmann • J. Zitting

Free eBook
SEE INSERT

Tika is an Apache toolkit that has built into it everything you and your app need to know about file formats. Using Tika, your applications can discover and extract content from digital documents in almost any format, including exotic ones.

Tika in Action is the ultimate guide to content mining using Apache Tika. You'll learn how to pull usable information from otherwise inaccessible sources, including internet media and file archives. This example-rich book teaches you to build and extend applications based on real-world experience with search engines, digital asset management, and scientific data processing. In addition to architectural overviews, you'll find detailed chapters on features like metadata extraction, automatic language detection, and custom parser development.

What's Inside

- Crack MS Word, PDF, HTML, and ZIP
- Integrate with search engines, CMS, and other data sources
- Learn through experimentation
- Many examples

This book requires no previous knowledge of Tika or text mining techniques. It assumes a working knowledge of Java.

Chris Mattmann is an information architect experienced in the construction of large data-intensive systems. Jukka Zitting is a core Tika developer, a member of the JCR expert group, and chairman of the Apache Jackrabbit project.

"By Tika's two main creators and maintainers."
—From the Foreword by Jérôme Charron, WebPulse

"Easily the most definitive guide to this great new text analysis toolkit."
—John Guthrie, SAP

"An easy-to-read guide—plenty of technical content."
—Rick Wagner, Red Hat

"There's not a single page of 'inaction' in the entire book!"
—Sean Kelly
Technologist, NASA

"Complete, practical, accurate."
—Julien Nioche
DigitalPebble Ltd

For access to the book's forum and a free eBook for owners of this book, go to manning.com/TikainAction

ISBN 13: 978-1-935182-85-6
ISBN 10: 1-935182-85-4
54499
9 781935 182856

MANNING $44.99 / Can $47.99 [INCLUDING eBOOK]