

Tika IN ACTION

Jukka L. Zitting
Chris A. Mattmann





**MEAP Edition
Manning Early Access Program
Tika in Action production version**

Copyright 2011 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

Table of Contents

Part 1: Getting Started

- Chapter 1 The case for the digital Babel fish
- Chapter 2 Getting started With Tika
- Chapter 3 The information landscape

Part 2: Tika in Detail

- Chapter 4 Document type detection
- Chapter 5 Content extraction
- Chapter 6 Understanding metadata
- Chapter 7 Language detection
- Chapter 8 What's in a file?

Part 3: Integration and Advanced Use

- Chapter 9 The big picture
- Chapter 10 Tika and the Lucene search stack
- Chapter 11 Extending Tika

Part 4: Case Studies

- Chapter 12 Powering NASA science data systems
- Chapter 13 Content management with Apache Jackrabbit
- Chapter 14 Curating cancer research data with Tika
- Chapter 15 The classic search engine example

Appendixes

- Appendix A Tika quick reference
- Appendix B Supported metadata keys

The case for the digital Babel fish



“The Babel fish,” said The Hitchhiker’s Guide to the Galaxy quietly, “is small, yellow and leech-like, and probably the oddest thing in the Universe. It feeds on brainwave energy not from its carrier but from those around it. It absorbs all unconscious mental frequencies from this brainwave energy to nourish itself with. It then excretes into the mind of its carrier a telepathic matrix formed by combining the conscious thought frequencies with nerve signals picked up from the speech centers of the brain which has supplied them. The practical upshot of all this is that if you stick a Babel fish in your ear you can instantly understand anything said to you in any form of language.”

Douglas Adams: *The Hitchhiker’s Guide to the Galaxy*

In this chapter

- Understanding documents
- Parsing documents
- Introducing Apache Tika

The Babel fish in Douglas Adams’ book *The Hitchhiker's Guide to the Galaxy* is a universal translator that allows you to understand all the languages in the world. It feeds on data that would otherwise be incomprehensible, and produces an understandable translation. This is essentially what Apache Tika, a nascent technology available from the Apache Software Foundation, does for digital documents. Just like the protagonist Arthur Dent, who after inserting a Babel fish in his ear could understand Vogon poetry, a computer program that uses Tika can extract text and objects from Microsoft Word documents and all sorts of other files.

Our goal in this book is to equip you with enough understanding of Tika's architecture, implementation, extension points, and philosophy that the process of making your programs file-agnostic is equally simple.

In the remainder of this chapter, we'll familiarize you with the importance of understanding the vast array of content that has sprung up as a result of the information age. PDF files, Microsoft Office files (including Word, Excel, PowerPoint, and so on), images, text, binary formats, and more are a part of today's digital lingua franca, as are the application tasked to handle such formats. We'll discuss this issue and modern attempts to classify and understand these file formats (such as those from the Internet Assigned Numbers Authority, IANA) and the relationships of those frameworks to Tika. After motivating Tika, we'll discuss its core Parser interface and its use in obtaining text for processing. Beyond the nuts and bolts of this discussion, we'll provide a brief history of Tika, along with an overview of its architecture, when and where to use Tika, and a brief example of Tika's utility.

In the next section, we'll introduce you to the existing work by IANA on classifying all the file formats out there and how Tika makes use of this classification to easily understand those formats.

1.1 Understanding digital documents

The world of digital documents and their file formats is like a universe where everyone speaks a different language. Most programs only understand their own file formats or a small set of related formats, as depicted in figure 1.1. Translators such as import modules or display plugins are usually needed when one program needs to understand documents produced by another program.

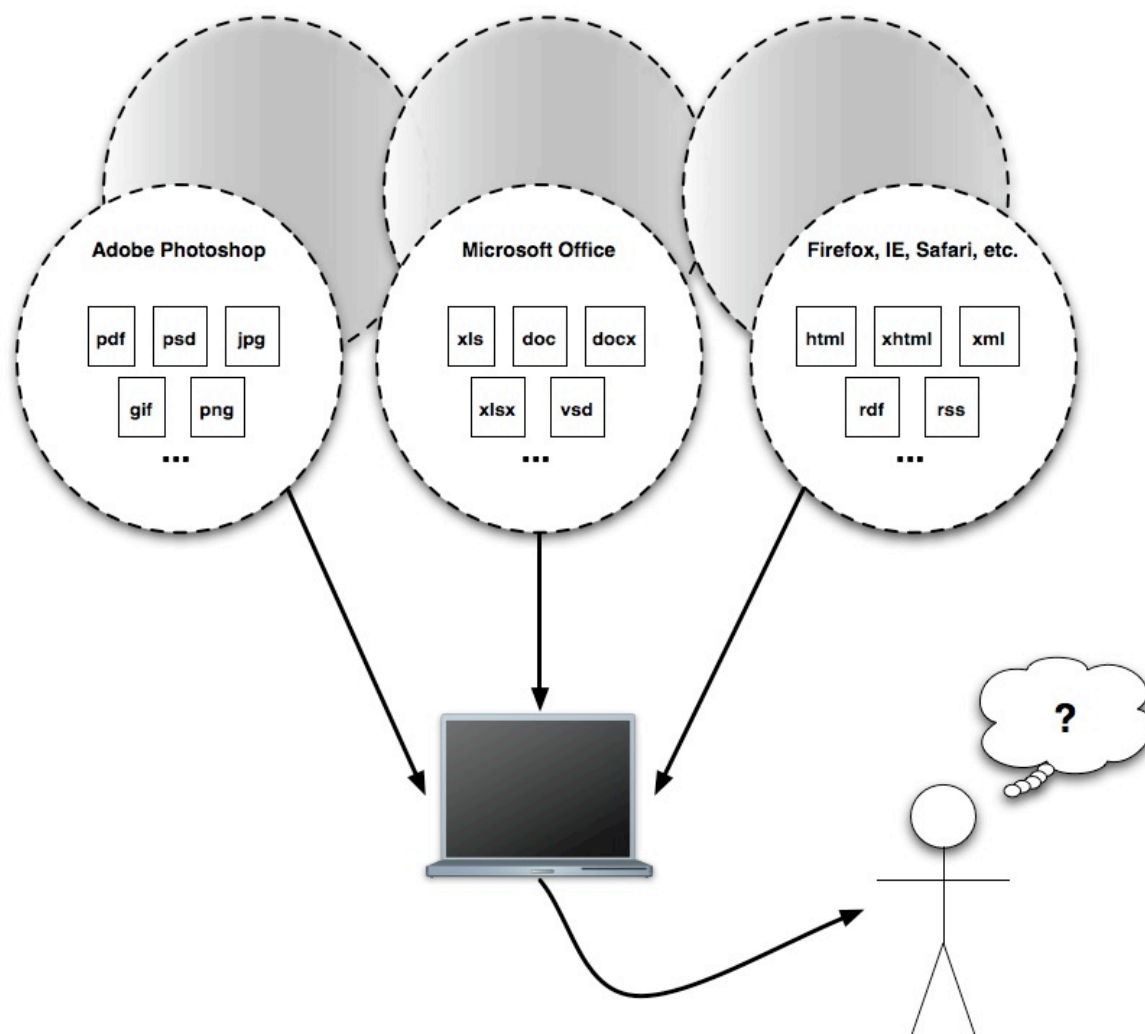


Figure 1.1 Computer programs usually specialize in reading and interpreting only one file format (or family of formats). To deal with .pdf files, .psd files, and the like, you'd purchase Adobe products. If you needed to deal with Microsoft Office files (.doc, .xls, and so on), you'd turn to Microsoft products or other office programs that support these Microsoft formats. Few programs can understand all of these formats.

There are literally thousands of different file formats in use, and most of those formats come in various different versions and dialects. For example the widely used PDF format has evolved through eight incremental versions and various extensions over the past 18 years. Even the adoption of generic file formats such as XML has done little to unify the world of data. Both the Office Open XML format used by recent versions of Microsoft Office and the OpenDocument format used by OpenOffice.org are XML-based formats for office documents, but programs written to work with one of these formats still need special converters to understand the other format.

Luckily most programs never need to worry about this proliferation of file

formats. Just like you only need to understand the language used by the people you speak with, a program only needs to understand the formats of the files it works with. The trouble begins when you're trying to build an application that's supposed to understand most of the widely used file formats.

For example, suppose you've been asked to implement a search engine that can find any document on a shared network drive based on the file contents. You browse around and find Excel sheets, PDF and Word documents, text files, images and audio in a dozen different formats, PowerPoint presentations, some OpenOffice files, HTML and Flash videos, and a bunch of Zip archives that contain more documents inside them. You probably have all the programs you need for accessing each one of these file formats, but when there are thousands or perhaps millions of files, it's not feasible for you to manually open them all and copy-paste the contained text to the search engine for indexing. You need a program that can do this for you, but how would you write such a program?

The first step in developing such a program is to understand the properties of the proliferation of file formats that exist. To do this we'll leverage the taxonomy of file formats specified in the Multipurpose Internet Mail Extensions (MIME) standard and maintained by the IANA.

1.1.1 A taxonomy of file formats

In order to write the aforementioned search engine, you must understand the various file formats and the methodologies that they employ for storing text and information. The first step is being able to identify and differentiate between the various file types. Most of us understand commonly used terms like *spreadsheet* or *web page*, but such terms aren't accurate enough for use by computer programs. Traditionally extra information in the form of file name suffixes such as .xls or .html, resource forks in Mac OS, and other mechanisms have been used to identify the format of a file. Unfortunately these mechanisms are often tied to specific operating systems or installed applications, which makes them difficult to use reliably in network environments such as the Internet.

The MIME standard was published in late 1996 as the Request for Comment (RFC) documents 2045-2049. A key concept of this standard is the notion of *media types*¹ that uniquely name different types of data so that receiving applications can "deal with the data in an appropriate manner." Section 5 of RFC 2045 specifies that a media type consists of a `type/subtype` identifier and a set of optional `attribute=value` parameters. For example the default media type

`text/plain; charset=us-ascii` identifies a plain-text document in the US-ASCII character encoding. RFC 2046 defines a set of common media types and their parameters, and since no single specification can list all past and future document types, RFC 2048 and the update in RFC 4288 specified a registration procedure by which new media types can be registered at IANA. As of early 2010, the official registry ² contains more than a thousand media types such as `text/html`, `image/jpeg`, and `application/msword`, organized under the eight top-level types shown in figure 1.2. Thousands of unregistered media types such as `image/x-icon` and `application/vnd.amazon.ebook` are also being used.

Footnote 1 Though often referred to as MIME type, the MIME standard in reality was focused on extending email to support different extensions, including non-text attachments and multipart requests. The use of MIME has since grown to cover the array of media types, including PDF, Office, and non-email-centric extensions. Although the use of MIME type is ubiquitous, in this book, we use *MIME type* and the more historically correct *media type* interchangeably.

Footnote 2 The official MIME media type registry is available at <http://www.iana.org/assignments/media-types/>.

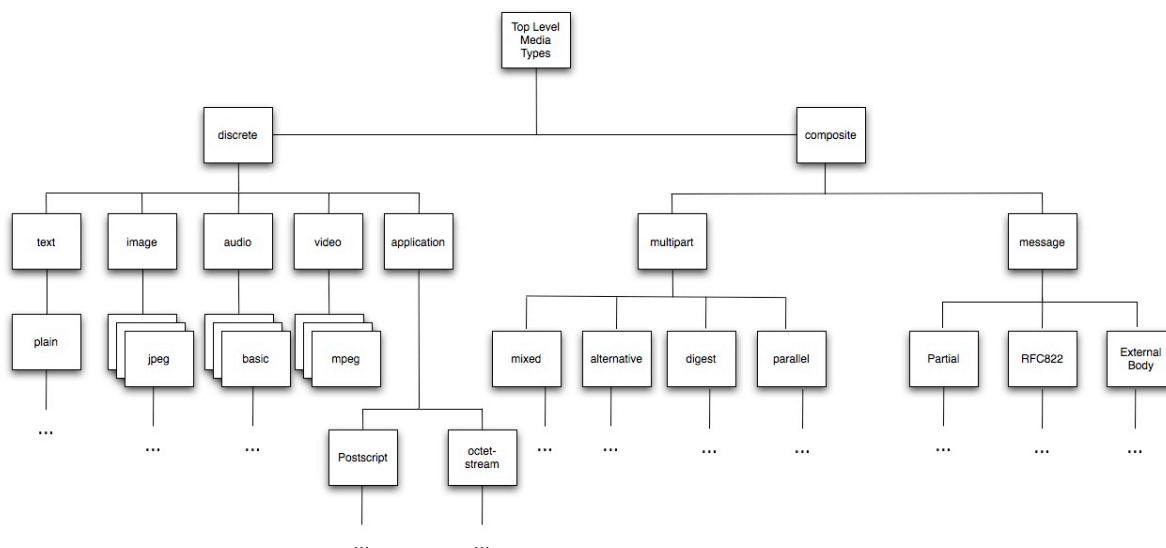


Figure 1.2 Seven top-level MIME type hierarchy as defined by IANA’s RFC 2046 (the eighth type model was added later in RFC 2077). Top-level types can have subtypes (children), and so on, as new media types are defined over the years. The use of the multiplicity denotes that multiple children may be present at the same level in the hierarchy, and the ellipses indicates that the remainder of the hierarchy has been elided in favor of brevity.

Given that thousands of media types have already been classified by IANA and others, programmers need the ability to automatically incorporate this knowledge into their software applications (imagine building a collection of utensils in your

kitchen without knowing that pots were used to cook sauces, or that kettles brewed tea, or that knives cut meat!). Luckily Tika provides state-of-the-art facilities in automatic media type detection. Tika takes a multipronged approach to automatic detection of media types as shown in table 1.1.

Table 1.1 Tika's main methods of media type detection. These techniques can be performed in isolation or combined together to formulate a powerful and comprehensive automatic file detection mechanism.

Detection mechanism	Description
<i>File extension, file name, or alias</i>	Each media type in Tika has associated with it a <i>glob</i> pattern, which can be a Java regular expression or a simple file extension, such as *.pdf or *.doc (see http://mng.bz/pNgw).
<i>Magic bytes</i>	Most files belonging to a media type family have a unique signature associated with them in the form of a set of control bytes in the file header. Each media type in Tika defines different sequences of these control bytes, as well as offsets used to define scanning patterns to locate these bytes within the file.
<i>XML root characters</i>	XML files, unique as they are, include hints that suggest their true media type. Outer XML tags (called <i>root elements</i>), namespaces, and referenced schemas are all part of the clues that Tika uses to determine an XML file's real type (RDF, RSS, and so on).
<i>Parent and children media types</i>	By leveraging the hierarchy shown in figure 1.2, Tika can determine the most accurate and precise media type for a piece of content, and fall back on parent types if the precise child isn't detectable.

We're only scratching the surface of Tika's MIME detection patterns here. For more information on automatic media type detection, jump to chapter 4 for more details.

Now that you're familiar with differentiating between different file types, how do you make use of a file once you've identified it? In the next section, we'll describe parser libraries, used to extract information from the underlying file types. There are a number of these parser libraries, and as it turns out, Tika excels (no pun

intended) at abstracting away their heterogeneity, making them easy to incorporate and use in your application.

1.1.2 Parser libraries

To be able to extract information from a digital document, you need to understand the document format. Such understanding is built into the applications designed to work with specific kinds of documents. For example the Microsoft Office suite is used for reading and writing Word documents, whereas Adobe Acrobat and Acrobat Reader do the same for PDF documents. These applications are normally designed for human interaction and usually don't allow other programs to easily access document content. And even if programmatic access is possible, these applications typically can't be run in server environments.

An alternative approach is to implement or use a parser library for the document format. A *parser library* is a reusable piece of software designed to enable applications to read and often also write documents in a specific format (as will be shown in figure 1.3, it's the software that allows text and other information to be extracted from files). The library abstracts the document format to an API that's easier to understand and use than raw byte patterns. For example, instead of having to deal with things such as CRC checksums, compression methods, and various other details, an application that uses the `java.util.zip` parser library package included in the standard Java class library can simply use concepts such as `ZipFile` and `ZipEntry`, as shown in the following example that outputs the names of all of the entries within a zip file:

```
public static void listZipEntries(String path) throws IOException {
    ZipFile zip = new ZipFile(path);
    for (ZipEntry entry : Collections.list(zip.entries())) {
        System.out.println(entry.getName());
    }
}
```

In addition to Zip file support, the standard Java class library and official extensions include support for many file formats, ranging from plain text and XML to various image, audio, video, and message formats. Other advanced programming languages and platforms have similar built-in capabilities. But most document formats aren't supported, and even APIs for the supported formats are often designed for specific use cases and fail to cover the full range of features required by many applications. Many open source and commercial libraries are available to address the needs of such applications. For example the widely used

Apache PDFBox (<http://pdfbox.apache.org/>) and POI (<http://poi.apache.org/>) libraries implement comprehensive support for PDF and Microsoft Office documents.

SIDEBAR **The wonderful world of APIs**

APIs, or *application programming interfaces* are interfaces that applications use to communicate with each other. In object-oriented frameworks and libraries, APIs are typically the recommended means of providing functionality that clients of those frameworks can consume. For example, if you're writing code in Java to read and/or process a file, you're likely using `java.io.*` and its set of objects (such as `java.io.File`) and its associated sets of methods (`canWrite`, for example), that together make up Java's IO API.

Thanks to parser libraries, building an application that can understand multiple different file formats is no longer an insurmountable task, lots of complexity is still to be covered, starting with understanding the variety of licensing and patent constraints on the use of different libraries and document formats. The other big problem with the myriad available parser libraries is that they all have their own APIs designed for each individual document format. Writing an application that uses more than a few such libraries requires a lot of effort learning how to best use each library. What's needed is a unified parsing API to which all the specific parser APIs could be adapted. Such an API would essentially be a universal language of digital documents.

In the ensuing section, we'll make a case for that universal language of digital documents, describing the lowest common denominator in that vocabulary: structured text.

1.1.3 Structured text as the universal language

Though the number of multimedia documents is rising, most of the interesting information in digital documents is still numeric or textual. These are also the forms of data that current computers and computing algorithms are best equipped to handle. The known search, classification, analysis, and many other automated processing tools for numeric and textual data are far beyond our current best understanding of how to process audio, image, or video data. Since numbers are also easy to express as text, being able to access any documents as a stream of text is probably the most useful abstraction that a unified parser API could offer. Though plain text is obviously close to a least common denominator as a document abstraction, it still enables a lot of useful applications to be built on top of it. For example a search engine or a semantic classification tool only needs access to the text content of a document.

A plain text stream, as useful as it is, falls short of satisfying the requirements of many use cases that would benefit from a bit of extra information. For example all the modern Internet search engines leverage not only the text content of the documents they find on the net but also the links between those documents. Many modern document formats express such information as hyperlinks that connect a specific word, phrase, image or other part of a document to another document. It'd be useful to be able to accurately express such information in a uniform way for all documents. Other useful pieces of information are things such as paragraph boundaries, headings, and emphasized words and sentences in a document.

Most document formats express such structural information in one way or another (an example is shown in figure 1.3), even if it's only encoded as instructions like "insert extra vertical space between these pieces of text" or "use a larger font for that sentence." When such information is available, being able to annotate the plain text stream with semantically meaningful structure would be a clear improvement. For example, a web page such as ESPN.com typically codifies its major news categories using instructions encoded via HTML list (``) tags, along with Cascading Style Sheets (CSS) classes to indicate their importance as top-level news categories.

The screenshot shows the ESPN.com home page as of April 1, 2010. A black rectangular box highlights the top navigation menu, which includes links for ALL SPORTS, COMMENTARY, PAGE 2, FANTASY, VIDEO, SPORTSNATION, CITIES, and THE LIFE. Below the screenshot, a code block displays the HTML structure for these links, showing that each link is enclosed in an `` tag with a specific class name (e.g., `t-allsports`, `t-commentary`, `t-page2`).

```

...
<ul class="top">
<li class="t-allsports"><a href="http://espn.go.com/sports/"
name="&pos=sitenav&lid=sitenav_sports">ALL SPORTS</a>
<div>
<ul>
...
<li class="t-commentary"><a href="http://espn.go.com/espn/commentary/"
name="&pos=sitenav&lid=sitenav_columnists">COMMENTARY</a>
<div>
<ul class="last">
<li><a href="http://sports.espn.go.com/espn/blog/main"
name="&pos=sitenav&lid=sitenav_columnists_blogs">Blogs</a></li>
...
<li class="t-page2"><a href="http://espn.go.com/espn/page2/"
name="&pos=sitenav&lid=sitenav_page2">PAGE 2</a>
<div>
...

```

Figure 1.3 A snippet of HTML (at bottom) for the ESPN.com home page. Note the top-level category headings for sports (All Sports, Commentary, Page 2) are all surrounded by `` HTML tags that are styled by a particular CSS class. This type of structural information about a content type can be exploited and codified using the notion of structured text.

Such structural annotations should ideally be well known and easy to understand, and it should be easy for applications that don't need or care about the extra information to focus on just the unstructured stream of text. XML and HTML are the best-known and most widely used documents formats that satisfy all these requirements. Both support annotating plain text with structural information, and whereas XML offers a well-defined and easy-to-automate processing model, HTML defines a set of semantic document elements that almost everyone in the computing industry knows and understands. The XHTML standard combines these advantages, and thus provides an ideal basis for a universal document language that can express most of the interesting information from a majority of the

currently used document formats. XHTML is what Tika leverages to represent structured text extracted from documents.

1.1.4 Universal metadata

Metadata, or “data about data” as it’s commonly defined, provides information that can aid in understanding documents independent of their media type. Metadata includes information that’s often pre-extracted, and stored either together with the particular file or available externally to it (when the file has an entry associated with it in some external registry). Since metadata is almost always less voluminous than the data itself (by orders of magnitude in most cases), it’s a preferable asset in making decisions about what to do with files during analysis. The actionable information in metadata can range from the mundane (file size, location, checksum, data provider, original location, version) to the sophisticated (start/end data range, start/end time boundaries, algorithm used to process the data, and so forth) and the richness of the metadata is typically dictated by the media type and its choice of *metadata model(s)* that it employs.

One widely accepted metadata model is the Dublin Core standard (<http://dublincore.org/>) for the description of electronic resources. Dublin Core defines a set of 15 data elements (read *attributes*) that are said to sufficiently describe any electronic resource. These elements include attributes for data format (HDF, PDF, netCDF, Word 2003, and so on), title, subject, publisher language, and other elements. Though a sound option, many users have felt that Dublin Core (which grew out of the digital library/library science community) is too broad and open to interpretation to be as expressive as it purports.

Metadata models can be broad (as is the case for Dublin Core), or narrow, focused on a particular community—or some hybrid combination of the two. The *Extensible Metadata Platform (XMP)* defined by Adobe is a combined metadata model that contains core elements (including those defined by Dublin Core), domain-specific elements related to Photoshop files, images, and more, as well as the ability for users to use their own metadata schemas. As another example, the recently developed *Climate Forecast (CF)* metadata model describes climate models and observational data in the Earth science community. CF, though providing limited extensibility, is primarily focused on a single community (climate researchers and modelers) and is narrowly focused when compared with the likes of Dublin Core or XMP.

Most times, the metadata for a particular file format will be influenced by existing metadata models, likely starting with basic file metadata and then getting

more specific, with at least a few instances of metadata pertaining to that type (Photoshop-specific, CF-specific, and so on). This is illustrated in figure 1.4, where three example sets of metadata driven by three metadata models are used to describe an image of Mars.

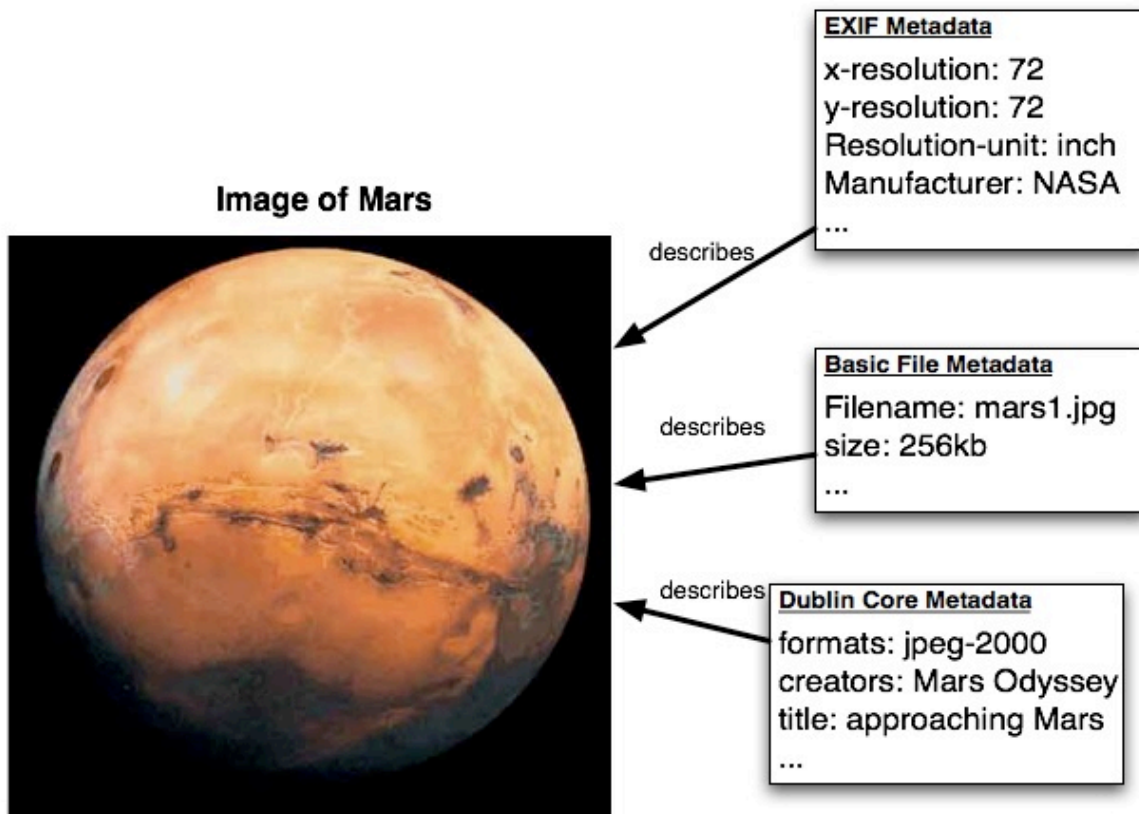


Figure 1.4 An image of Mars (the data), and the metadata (data about data) that describes it. Three sets of metadata are shown, and each set of metadata is influenced by metadata models that prescribe what vocabularies are possible, what the valid values are, what the definitions of the names are, and so on. In this example, the metadata ranges from basic (file metadata like filename) to the image-specific (EXIF metadata like resolution-unit).

In order to support the heterogeneity of metadata models, their different attributes, and different foci, Tika has evolved to allow users to either accept default metadata elements conforming to a set of core models (Dublin Core, models focused on document types such as Microsoft Word models, and so forth) supported out of the box, or to define their own metadata schema and integrate them into Tika seamlessly. In addition, Tika doesn't dictate how or where metadata is extracted within the overall content understanding process, as this decision is typically closely tied to both the metadata model(s) employed and the overall analysis workflow, and is thus best left up to the user.

Coupled with the ability to flexibly extract metadata come the realization that

not all content on the web, or in a particular software application, is of the same language. Consider a software application that integrates planetary rock image datasets from NASA's Mars Exploration Rover (MER) mission with data from the European Space Agency's Mars Express orbiter and its High Resolution Stereo Camera (HRSC) instrument, which captures full maps of the entire planet at 10m resolution. Consider that some of the earliest full planet data sets are directly available from HRSC's principal investigator—a center in Berlin—and contains information encoded in the German language. On the other hand, data available from MER is captured in plain English. To even determine that these two data sets are related, and that they can be correlated, requires reading lengthy abstracts describing the science that each instrument and mission are capturing, and ultimately understanding the languages in which each dataset is recorded. Tika again comes to the rescue in this situation, as it provides a language identification component that implements sophisticated techniques including N-grams that assist in language detection.

More information on structured text, metadata extraction, and language identification is given in chapter 6. Now that we've covered the complexity of dealing with the abundance of file formats, identifying them, and doing something with them (such as parsing them and extracting their metadata), it's time to bring Tika to the forefront and show you how it can alleviate much or all of the complexity induced by the modern information landscape.

1.1.5 The program that understands everything

Armed with the knowledge that Tika can help us navigate the modern information ecosystem, let's revisit the search engine example we considered earlier, depicted graphically in figure 1.5. Imagine that you're tasked with the construction of a local search application, whose responsibility is to identify PDF, Word, Excel, and audio documents available via a shared network drive, and to index those documents locations and metadata for use in a web-based company intranet search appliance.

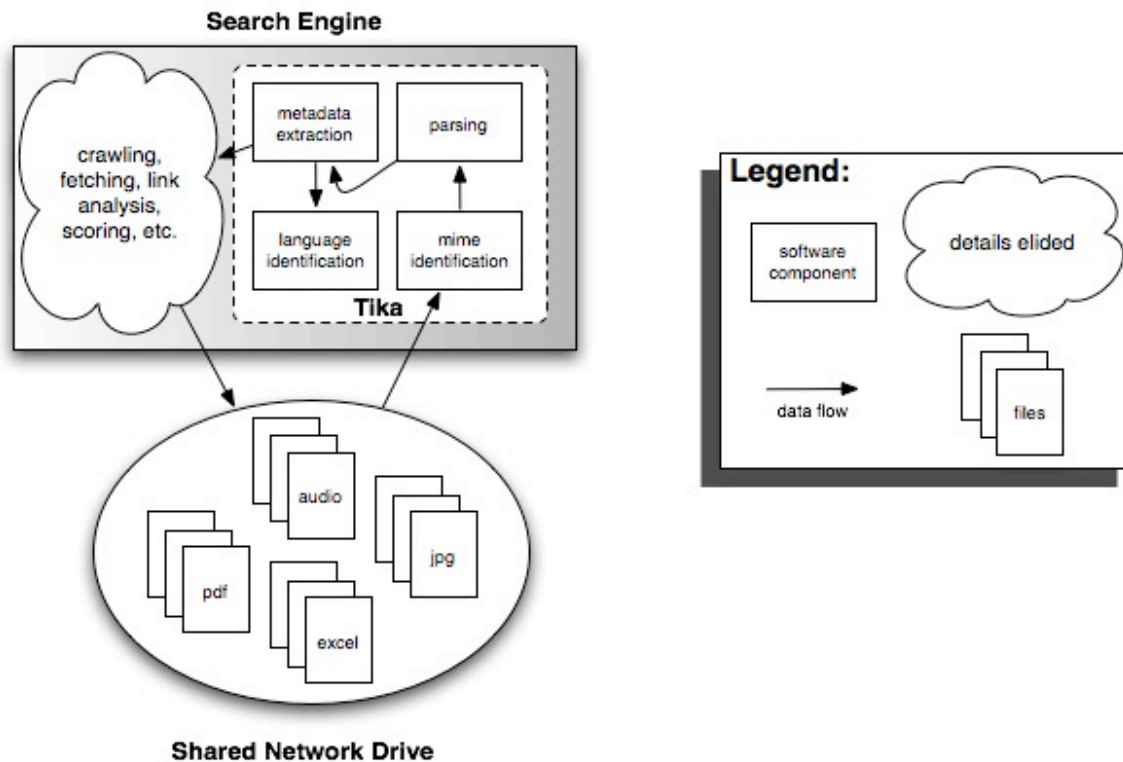


Figure 1.5 Revisiting the search engine example armed with Tika in tow. Tika provides the four canonical functions (labeled as software components in the figure) necessary for content detection and analysis in the search engine component. The remainder of the search engine’s functions (crawling, fetching, link analysis, scoring) are elided in order to show the data flow between the search engine proper, the files it crawls from the shared network drive, and Tika.

Knowing what you know now about Tika, the steps required to construct this search engine may go something like the following. First, you leverage a crawling application that gathers the pointers to the available documents on the shared network drive (depending on your operating system, this may be as simple as a fancy call to `ls` or `find`). Second, after collecting the set of pointers to files of interest, you iterate over that set and then determine each file’s media type using Tika (as shown in middle-right portion of figure 1.5). Once the file’s media type is identified, a suitable parser can be selected (in the case of PDF files, Apache’s PDFBox), and then used by Tika to provide both the extracted textual content (useful for keyword search, summarizing and ranking, and potentially other search functions such as highlighting), as well as extracted metadata from the underlying file (as shown in the upper middle portion of figure 1.5). Metadata can be used to provide additional information on a per-media-type basis—for example, for PDF files, display a lock icon if a metadata field for `locked` is set, or for Excel files, listing the number of cells in the document or the number of rows and columns in a

sheet. From there, you'd decide whether to display additional icons that link to services that can further process the file pointed to by each search result returned from a query. The final step is language identification, used to annotate whether a document is provided in a desired language (such as English), or whether the search engine should provide a link allowing to service that can translate the file from its original language. This process is summarized in figure 1.5.

As can be gleaned from the discussion thus far, Tika strives to offer the necessary functionality required for dealing with the heterogeneity of modern information content. Search is only one application domain where Tika provides necessary services. Chapters 12–15 describe other domain examples, including content management, data processing at NASA, and grid systems at the National Cancer Institute.

Before getting any further down the rabbit hole, it's worth providing a bit of history on Tika's inception, discussing its design goals, and describing its relationship to its parent project, Apache Lucene, and other related technologies.

1.2 What is Apache Tika?

We've talked a lot about Tika so far, but much like a new friend at school, you probably are still lacking a bit of context, and some of the background details (what city was that friend from; how many brothers does she have; or sisters?) that would make you feel better about continuing the relationship. We'll begin with some details on Tika's grandparents and parents: technologies originating in parts from the world of search engines at Apache, from work in XML parsing at Sourceforge.net, and from origins in document management. After introducing you to Tika's predecessors, you'll want some information on Tika's key philosophies, its goals and where it wants to be in five years. Is Tika your friend for life, or simply filling a hole until you meet the next great technology? Read on and we'll give you that information on your new pal Tika.

1.2.1 A bit of history

Figure 1.6 shows a timeline of Tika's development, from proposal to top-level project.

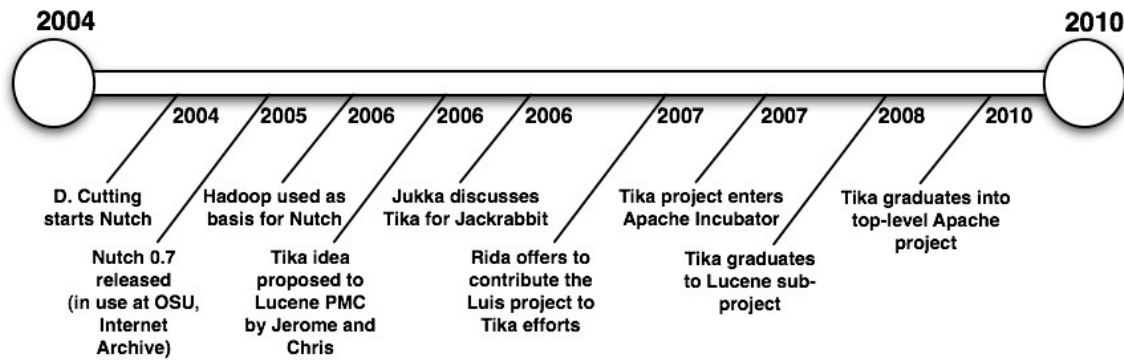


Figure 1.6 A visual timeline of Tika's history. Its early beginnings formed from the Apache Nutch project, which itself spawned several children and grandchildren, including Apache Hadoop and its subprojects. After some steps along the way, as well the work of a few individuals who kept the fire lit, Tika eventually moved into its current form as a top-level Apache project.

As the figure depicts, the idea for Tika was originally proposed in the Apache Nutch project. Nutch is best described as an open source framework for large-scale web search. The project commenced as the brainchild of Doug Cutting (the father of the Lucene and Hadoop projects, a general wizard of open source search), who was frustrated with commercial search companies and the proprietary nature of their ranking algorithms and features. As the original Nutch website at Sourceforge.net stated:

Nutch provides a transparent alternative to commercial web search engines. Only open source search results can be fully trusted to be without bias. (Or at least their bias is public.) All existing major search engines have proprietary ranking formulas, and will not explain why a given page ranks as it does. Additionally, some search engines determine which sites to index based on payments, rather than on the merits of the sites themselves. Nutch, on the other hand, has nothing to hide and no motive to bias its results or its crawler in any way other than to try to give each user the best results possible.

Nutch rapidly grew from a nascent effort into an established framework, with community involvement spanning academia (the Central Web Services department at Oregon State); industry, for example at the Internet Archive (a nonprofit focused on digitally archiving the web); government (with some of the search efforts in planetary science and cancer research performed by yours truly at NASA); and dozens of other commercial entities and efforts. Eventually, Nutch reached its upper limits in scalability, around 100 million web pages, a factor of 40 less than that of the commercial search engines such as Google. Around the same time, the

grid computing team at Yahoo! came into the picture and began to evaluate Nutch, but the scalability limitation was a problem that needed to be solved.

The most promising approach for obviating the scalability problem came when Google published its seminal papers describing its MapReduce and Google File System (GFS) technologies, and when Doug Cutting ran across these papers. Doug decided to implement the software and algorithms described therein in the open source community at Apache, along with Mike Cafarella. Nutch quickly moved from a technology that ran on a single node and maxed out around 100 million web pages, to a technology that could run on 20 nodes and scale to billions of web pages. Once the initial prototype was demonstrated, Yahoo! jumped in with engineers and resources, and eventually the Apache Hadoop project was born. Apache Hadoop was the result of an effort to generalize the Map Reduce and Distributed File System portions of Nutch implemented by Cutting and Cafarella, and to port them to a standalone project, making it easier to inherit their capabilities in isolation, without pulling all of Nutch in.

Around the same time, we and others (including Jerome Charron) saw the value in doing *the exact same thing* for the parsing code in Nutch, and for its MIME detection capabilities. Jerome and Chris sent a proposal to the Apache Lucene Project Management Committee (PMC), but despite positive feedback, the idea gained little momentum until later in the year when Jukka came along with the parsing and content-detection needs of the Apache Jackrabbit community, and others including Rida Benjelloun offered to donate the Lius framework he developed at Sourceforge (a set of parsers and utilities for indexing various content types in Lucene). Critical mass was achieved, and the Tika idea and project were born into the Apache Incubator. After a successful incubating release, and a growing community, Tika graduated as a full-fledged Lucene subproject, well on its way to becoming the framework that you're reading about today.

NOTE**What's a Tika?**

Tika's name followed the open source baby naming technique du jour circa 2005—naming the project after your child's stuffed toy. No, we're not kidding. Doug Cutting, the progenitor of Apache Lucene, Apache Nutch, and Apache Hadoop, had a pension for naming open source projects after his children's favorite stuffed animals. So, when Jerome and Chris Mattmann were discussing what to call their proposed text analysis project, *Tika* seemed a perfect choice—it was Jerome's son's stuffed animal!

Now that you've heard about Tika's ancestors and heritage, let's familiarize you with Tika's current state of mind and discuss its design goals, now and going forward.

1.2.2 Key design goals

A summary of Tika's overall architecture is provided in figure 1.7. Throughout this section, we'll describe the key design goals that influenced Tika's architecture and its key components: a parser framework (middle portion of the diagram), a MIME detection mechanism (right side of the diagram), language detection (left side of the diagram), and a facade component (middle portion of the diagram) that ties all of the components together. External interfaces including the command line (upper left portion of the diagram) and a graphical user interface (discussed in chapter 2 and shown in the upper right portion of the diagram), allow users to integrate Tika into their scripts and applications and to interact with Tika visually. Throughout its architecture, Tika leverages the notion of repositories: areas of extensibility in the architecture. New parsers can be easily added and removed from the framework, as can new MIME types and language detection mechanisms, using the repository interface. Hopefully, this terminology has become second nature to you by now!

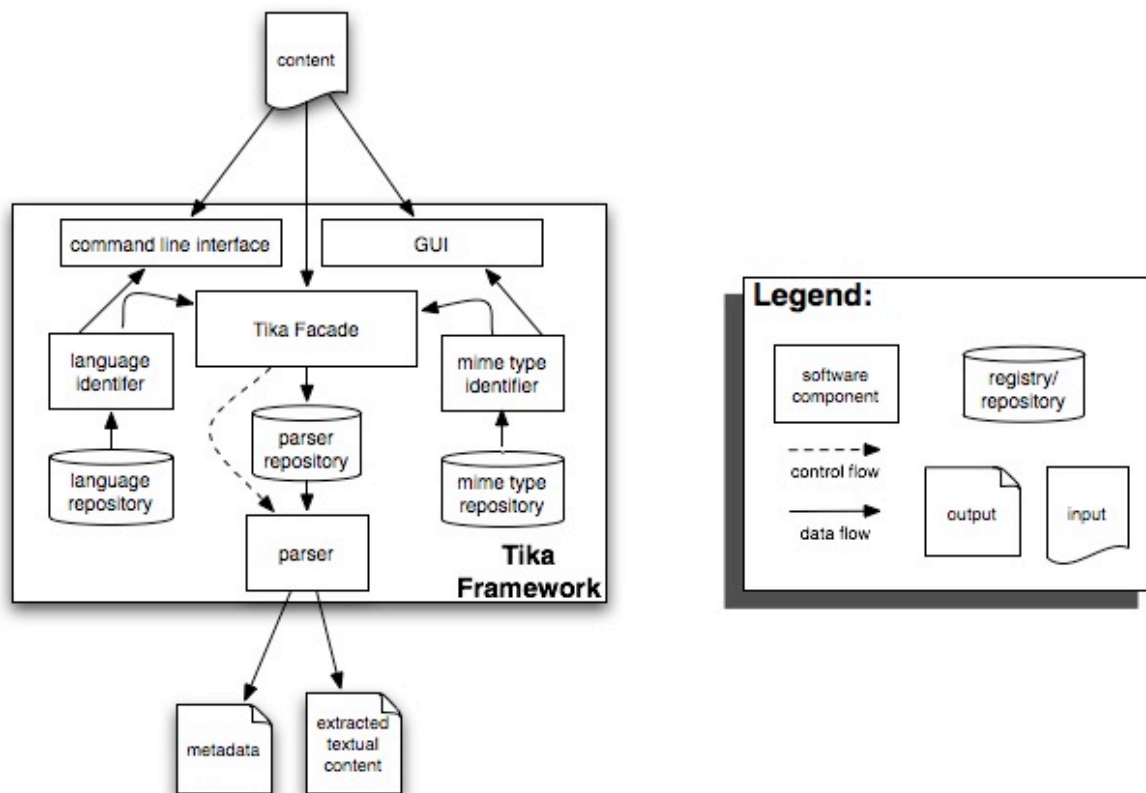


Figure 1.7 High level Tika architecture. Note explicit components exist that deal with

MIME detection (understanding how to identify the different file formats that exist), language analysis, parsing, and structured text and metadata extraction. The Tika facade (center of the diagram) is a simple, easy to use front-end to all of Tika's capabilities.

Strong early interest in Tika paved the way for discussions on the mailing lists, for birds-of-a-feather (BOF) meetings at ApacheCon (Apache's flagship conference), and for other public forums where much of the original design and architecture for Tika were fleshed out. Several of the concepts we've already discussed: providing a means to extract text in XHTML format; allowing for flexible metadata models, and explicit interfaces for its extraction; and support for MIME-type detection were all identified as necessary first-order features and input accordingly into Tika's JIRA issue tracking system. The key design goals are summarized in table 1.2, and further discussed in the remainder of this section.

Table 1.2 Tika's key design goals, numbered and described briefly for reference. Each design goal is elaborated upon in detail in this section, and ties back to the overall necessity for Tika.

Design goal	Description
<i>G1: Unified parsing</i>	Provide a single uniform set of functions and a single Java interface to wrap around heterogeneous third-party parsing libraries.
<i>G2: Low memory footprint</i>	Tika should be embeddable within Java applications at low memory cost so that it's as easy to use Tika in a desktop-class environment with capacious network and memory as it is within a mobile PDA with limited resources on which to operate.
<i>G3: Fast processing</i>	The necessity of detecting file formats and understanding them is pervasive within software, and thus we expect Tika to be called all the time, so it should respond quickly when called upon.
<i>G4: Flexible metadata</i>	There are many existing metadata models to commonly describe files, and Tika has the burden of understanding all of the file formats that exist, so it should in turn understand the formats' associated metadata models.
<i>G5: Parser Integration</i>	Just as there are many metadata models per file format, there are also many parsing libraries. Tika should make it easy to use these within an application.
<i>G6: MIME database</i>	MIME types provide an easy-to-use, understandable classification of file formats and Tika should leverage these classifications.
<i>G7: MIME detection</i>	There are numerous ways to detect MIME types based on their existing IANA classifications (recall table 1.1), and Tika should provide a means for leveraging all or some combination of these mechanisms.
<i>G8: Language detection</i>	Understanding what language a document's content is in is one of the cornerstones of extracting metadata from it and its textual information, so Tika should make language identification a snap.

UNIFIED PARSING INTERFACE

One of the main early discussions was regarding the creation of the `org.apache.tika.Parser` interface. The choices were many: parse content in one fell swoop or parse the content incrementally? How should the parsed text be provided—via the return of the method signature or via reference? What was the relationship of the parsers to media types?

LOW MEMORY FOOTPRINT AND FAST PROCESSING

After lengthy discussions, the decision was made to parse text incrementally and output it as SAX-based XHTML events. SAX, the Simple API for XML processing, is *the* primary alternative to parsing XML using the Document Object Model (DOM), which loads the entire XML document into memory and then makes it available via an API. SAX, on the other hand, parses tags incrementally, causing a low memory footprint, allowing for rapid processing times, and ultimately providing the functionality required by the Tika architecture. After that, you may be wondering, why does DOM even exist? DOM exists because it provides a more conceptually understandable API than SAX, where you have to be cognizant of state (if you're parsing a complex XML model with many tags) as you iterate over the XML document. SAX parsers by their nature attach “handler” code to callback functions (agglomerated as `org.xml.sax.ContentHandler` implementations) that implement the workflow of processing an XML document. SAX callback functions include `startDocument` (called when the SAX parser begins parsing), `endDocument` (called when the SAX parser is finished), `startElement` (called when an XML open tag is encountered for a tag with a given name, such as `<book>`), and `endElement` (called when the SAX parser encounters an end XML tag such as `</book>`), to name a few. Developers fill in the body of these functions to tell the SAX parser what to do as it parses the XML document piecemeal. In DOM, these details are obfuscated from the user and handled by the DOM implementation provider, at the cost of memory footprint and overall speed.

By adopting the SAX model, Tika allows developers and those wishing to customize how Tika's `Parser` deals with extracted information to define custom `org.xml.sax.ContentHandlers` that describe what to do: pass along a subset of the extracted XHTML tags; pass along all of the tags, discard others; and so on.

FLEXIBLE METADATA

The next major question to answer in Tika's `Parser` was determining how extracted metadata should be provided. Earlier versions of Tika focused on modifying a passed-in `org.apache.tika.metadata.Metadata` object instance, and adding the extracted metadata to that object. Modern and future versions of Tika have moved in the direction of a `org.apache.tika.parser.ParseContext` object, containing the returned state from the parser, including the extracted text and metadata. The decision for how to deal with extracted metadata boils down to the metadata's lifecycle. Questions include what should Tika do with existing metadata keys (overwrite or keep)? Should Tika return a completely new `Metadata` object instance during each parse? There are benefits of allowing each scenario. For example, MIME detection can benefit from a provided metadata "hint"—whereas creating a new `Metadata` object and returning it per parse simplifies the key management and merge issues during metadata extraction.

EASY-TO-INTEGRATE NEW PARSER LIBRARIES

Some other early design considerations in Tika's parsing framework were focused on exactly how third-party parsing libraries should be provided. Should Tika developers become experts in the underlying parsing libraries, and as such, should Tika be in the business of providing parser library implementations? The resounding community consensus was *no*, and fittingly, as each parsing library can be the result of many years of work from hundreds of developers and users. The consensus from a design perspective was that Tika should look to virtualize underlying parser libraries, and ensure their conformance to Tika's `org.apache.tika.parser.Parser` interface. But much complexity is hidden in that simple sentence. Dealing with underlying parser exceptions, threads of control, delegation, and nuances in each of these libraries has been a large effort in its own right. But this effort is a cost well-spent, as it opens the door to cross-document comparison, uniformity, standardized metadata and extracted text, and other benefits we're hopefully starting to ingrain in your mind.

MIME DATABASE

Several design considerations in Tika's MIME framework pervade its current reification in the Tika library. First and foremost, we wanted Tika to support a flexible mechanism to define media types, per the discussion on IANA and its rich repository and media type model discussed earlier. Because the IANA MIME specification and the aforementioned RFCs were forward-looking, they defined a mechanism procedurally for adding additional media types as they're created—we desired this same flexibility for Tika's MIME repository. In addition, we wanted Tika to provide an easy, XML-based mechanism (similar to Freedesktop.org, Apache Nutch, and other projects) for adding media types, their magic character patterns, regular expressions, and glob patterns (such as *.txt) for identifying filename patterns and extensions.

Besides ensuring that the definition of media types in Tika is user-friendly and easy, we also wanted to support as many of the existing IANA types as possible. One of our design considerations was the creation of *the comprehensive* media type repository, akin to the MIME information used by Apache's HTTPD web server, or by Freedesktop.org's shared MIME-info database. With more than 1276 defined MIME types and relationships captured, Tika is well on its way in this regard.

PROVIDE FLEXIBLE MIME DETECTION

To expose the MIME information programatically, we decided to expose as many MIME detection mechanisms (via `byte[]` arrays, `java.io.Files`, filenames and `java.net.URLs` pointing to the files, and so forth) as possible to end-users of the Tika API. Tika's `org.apache.tika.mime.MimeTypes` class was designed to act as this honest broker of functionality. The class loads up a Tika XML MIME repository file, and then provides programmatic access to detection mechanisms and allows users to obtain `org.apache.tika.mime.MimeTypes` (encapsulating not only the name, but other MIME information such as the parents, magic characters, patterns, and more), or simply the names of the detected type for the provided file.

Another important consideration for Tika's MIME repository was tying the MIME information to that of `org.apache.tika.parser.Parsers` that deal with extracting text content and metadata. The main details to flush out in this arena were whether Tika Parsers should deal with only single media types, or handle many per Parser. This issue dictates whether specific Parser implementations are allowed to be complex (dealing with multiple media types), or

whether each supported `Tika Parser` should be more canonical, dealing with a single media type, and increasing the number of parsers in the Tika framework. Beyond that detail (Tika opted to allow one to many types per `Parser`, achieving the greatest flexibility and decreasing the overall number of parsers), the exchange of MIME information between `Parser` and `Metadata` object was another important consideration, as the detected media type can be useful information to return as extracted metadata along with the parsing operation.

PROVIDE LANGUAGE DETECTION

Language identification, though a newer feature in Tika, fits into the overall Tika framework, as it's simply another piece of information that can be fed into the `Parser` and leveraged (similar to the media type information) during the parsing process. Much of the design discussion to date in this area is centered on mechanisms to improve language-specific charset detection, and to inject that information into the overall Tika lifecycle (for example, make it present in the `Metadata` object, make it available during parsing, and so on).

In the following section, we'll detail more on the best places to use Tika, provide some example domains, and set the stage for advanced discussion of Tika features in the forthcoming chapters.

1.2.3 When and where to use Tika

Now that you have an idea of what Tika is and what it does, the next question is where and when it's best used, and more importantly, is it of any use to you? This section covers some of the more prominent use cases and domains where Tika is now being used.

SEARCH ENGINES AND CONTENT REPOSITORIES

The main use case for which Tika was originally conceived is supporting a search engine to index the full text contents of various kinds of digital documents. A search engine typically includes a crawler component that repeatedly traverses a set of documents and adds them to a search index. Since the search index normally only understands plain text, a parser is needed to extract the text contents from the documents. Tika fits this need perfectly, and we'll cover this use case in more detail in chapter XREF ch10 where we discuss integration with the various search engine components of the Apache Lucene project. The case study in chapter XREF ch15 takes a more practical view on how Tika fits into such a search engine.

A related use case includes different kinds of document and content repositories that make all contained documents searchable. Whenever a document is added or

modified in the repository, its content is extracted and indexed. A generic and extensible parsing tool such as Tika allows the repository to support virtually any kinds of documents, and Tika's metadata extraction capabilities can be used to automatically classify or annotate the documents stored in the repository. The case study in chapter XREF ch13 shows how the Apache Jackrabbit project uses Tika for such purposes.

DOCUMENT ANALYSIS

The field of artificial intelligence is often associated with large promises and poor results, but the decades of research have still produced some impressive tools for automatically analyzing documents on a semantic level and extracting all sorts of interesting information. Some of the simpler practical applications are the ability to extract key terms such as people and places and their relationships from normal written text, and the ability to automatically classify documents based on the key topics covered. Projects such as Apache UIMA and Mahout provide open source tools for such applications, and Tika can be used to easily extend the scope of the applications from plain text to any kinds of digital documents.

DIGITAL ASSET MANAGEMENT

The key assets of many organizations are increasingly digital. CAD drawings, book manuscripts, photographs, music, and video are just some examples of digital assets with high value. Instead of storing such documents on a disk or backup tape somewhere, organizations are increasingly using more sophisticated digital asset management (DAM) applications to keep track of these assets and to guide related processes. A DAM system often categorizes tracked documents by type, annotates them with rich metadata, and makes them easily searchable, all of which can easily be implemented with support from Tika.

1.3 Summary

We've introduced you to Apache Tika, an extensible Java-based framework for content analysis and detection. First, we explained the motivation for Tika by describing the proliferation of content types such as PDFs, Word, Excel, and HTML, and tools associated with performing functions on those types. The sad fact is that the typical pattern involves specializing knowledge of these types to particular applications, and needing to maintain a set of applications to deal with each type of file (the Microsoft Office suite, Adobe Photoshop, XML editors, and so forth). Beyond applications, many application programming interfaces (APIs) exist that handle these document types, but they're highly heterogeneous—they make different assumptions, provide different interfaces, and support varying qualities of service. Enter Apache Tika, a mechanism to bridge the content type diversity that exists and to deal with file types in a uniform way.

The nuances and complexity in writing an application (such as a search engine) that must deal with all of these content types at once quickly grow to be untenable without a technology like Tika. Clearly, the problem of obtaining information from these file types is centered around the ability to identify the type of file automatically (especially when dealing with large numbers of such files), extract the textual information in batch, and extract common metadata (such as Dublin Core) useful for quickly comparing and understanding the file types. Language detection is also a needed feature (similar to MIME detection) to determine means for extracting out the textual information from each file type. As it turns out, Apache Tika (big surprise!) provides simple mechanisms to address these functions in modern software.

To familiarize you more with Tika, we provided some history, explaining how and why certain design decisions and goals were arrived upon. Tika's modular design and its assumptions were detailed, hopefully providing more intimate understanding of the existing framework and its high-level benefits, strengths, and weaknesses.

We gave you some tips and high-level advice on where to leverage Tika in your applications: where it works, and where you shouldn't even think of putting it (Hint: it doesn't cook your breakfast for you!). We wrapped up the chapter by grounding our discussion in some real-world domains, using the discussion to describe Tika's utility clearly and concisely.

In the next chapter, we'll get you familiar with how to obtain Tika from the Apache Software Foundation (ASF), how to begin to construct your application

(including Tika's code, its distribution jar files, and so on), and travel further down the path of automatic content detection and analysis engendered by the Tika technology.