CHAPTER 17

Mashing Up Desktop and Web-Based Office Suites

I've long been excited about the mashability and reusability of office suite documents (for example, word processor documents, spreadsheets, and slide presentations), the potential of which has gone largely unexploited. There are many office suites, but in this chapter I'll concentrate on the latest versions of OpenOffice.org, often called OO.o (version 2.*x*), and Microsoft Office (2007 and 2003). Few people realize that both these applications not only have programming interfaces but also have XML-based file formats. In theory, office documents using the respective file formats (OpenDocument and Office Open XML) are easier to reuse and generate from scratch than older generations of documents using opaque binary formats. And as you have seen throughout the book, knowledge of data formats and APIs means having opportunities for mashups. For ages, people have been reverse engineering older Microsoft Office documents, whose formats were not publicly documented; however, recombining office suites has been made easier, though not effortless, by these new formats. In this chapter, I will also introduce you to the emerging space of web-based office suites, specifically ones that are programmable. I'll also briefly cover how to program the office suites.

This chapter does the following:

* Shows how to do some simple parsing of the OpenDocument format (ODF) and Office Open XML documents
* Shows how to create a simple document in both ODF and Open XML
* Demonstrates some simple scripting of OO.o and Microsoft Office
* Lays out what else is possible by manipulating the open document formats
* Shows how to program Google Spreadsheets and to mash it up with other APIs (such as Amazon E-Commerce Services)

Mashup Scenarios for Office Suites

Why would mashups of office suite documents be interesting? For one, word processing documents, spreadsheets, and even presentation files hold vast amounts of the information that we communicate to each other. Sometimes they are in narratives (such as documents), and sometimes they are in semistructured forms (such as spreadsheets). To reuse that information, it is sometimes a matter of reformatting a document into another format. Other times, it's about extracting valuable pieces; for instance, all the references in a word processor document might be extracted into a reference database. Furthermore, not only does knowledge of the file formats enable you to parse documents, but it allows you to generate documents.

Some use case scenarios for the programmatic creation and reuse of office documents include the following:

Reusing PowerPoint: Do you have collections of Microsoft PowerPoint presentations that draw from a common collection of digital assets (pictures and outlines) and complete slides? Can you build a system of personal information management so that PPT presentations are constructed as virtual assemblages of slides, dynamically associated with assets?

Writing once, publishing everywhere: I'm currently writing this manuscript in Microsoft Office 2007. I'd like to republish this book in (X)HTML, Docbook, PDF, and wiki markup. How would I repurpose the Microsoft Word manuscript into those formats?

Transforming data: You could create an educational website in which data is downloaded to spreadsheets, not only as static data elements but as dynamic simulations. There's plenty of data out there. Can you write programs to translate it into the dominant data analysis tool used by everyone, which is spreadsheets, whether it is on the desktop or in the cloud?

Getting instant PowerPoint presentations from Flickr: I'd like to download a Flickr set as a PowerPoint presentation. (This scenario seems to fit a world in which PowerPoint is the dominant presentation program. Even if Tufte hates it, a Flickr-to-PPT translator might make it easier to show those vacation pictures at your next company presentation.)

There are many other possibilities. This chapter teaches you what you need to know to start building such applications.

The World of Document Markup

This chapter focuses on XML-based document markup languages in two dominant groups of office suites: Microsoft Office 2007 and OpenOffice.org. There are plenty of other markup languages, which are covered well on Wikipedia:

* `http://en.wikipedia.org/wiki/Document_markup_language`
* `http://en.wikipedia.org/wiki/List_of_document_markup_languages`
* `http://en.wikipedia.org/wiki/Comparison_of_document_markup_languages`

The OpenDocument Format

ODF is "an open XML-based document file format for office applications to be used for documents containing text, spreadsheets, charts, and graphical elements," developed under the auspices of OASIS.[1] ODF is also an ISO/IEC standard (ISO/IEC 206300:2006).[2] ODF is used most prominently in OpenOffice.org (`http://www.openoffice.org/`) and KOffice (`http://www.koffice.org/`), among other office suites. For a good overview of the file format, consult J. David Eisenberg's excellent book on ODF, called *OASIS OpenDocument Essentials*, which is available for download as a PDF (free of charge) or for purchase.[3]

The goal of this section is to introduce you to the issues of parsing and creating ODF files programmatically.

Note For this section, I am assuming you have OpenOffice.org version 2.2 installed.

A good way to understand the essentials of the file format is to create a simple instance of an ODF file and then analyze it:

1. Fire up OpenOffice.org Writer, type **Hello World**, and save the file as `helloworld.odt`.[4]
2. Open the file in a ZIP utility (such as WinZip on the PC). One easy way to do so is to change the file extension from `.odt` to `.zip` so that the operating system will recognize it as a ZIP file. You will see that it's actually a ZIP-format file when you go to unzip it. (See the list of files in Figure 17-1.)

Insert 858Xf1701.tif

Figure 17-1. Unzipping `helloworld.zip`. An OpenDocument file produced by OpenOffice.org is actually in the ZIP format.

1. http://www.oasis-open.org/committees/office/
2. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=43485
3. http://books.evc-cit.info/OD_Essentials.pdf or http://develop.opendocumentfellowship.com/book/
4. http://examples.mashupguide.net/ch17/helloworld.odt

You'll see some of the files that can be part of an ODF file:

* `content.xml`
* `styles.xml`
* `meta.xml`
* `settings.xml`
* `META-INF/manifest.xml`
* `mimetype`
* `Configuration2/accelerator/`
* `Thumbnails/thumbnail.png`

You can also use your favorite programming language, such as Python or PHP, to generate a list of the files. The following is a Python example:

```
import zipfile
z = zipfile.ZipFile(r'[path_to_your_file_here]')
z.printdir()
```

This generates the following:

```
File Name                                        Modified             Size
mimetype                                   2007-06-02 16:10:18          39
Configurations2/statusbar/                 2007-06-02 16:10:18           0
Configurations2/accelerator/current.xml    2007-06-02 16:10:18           0
Configurations2/floater/                   2007-06-02 16:10:18           0
Configurations2/popupmenu/                 2007-06-02 16:10:18           0
Configurations2/progressbar/               2007-06-02 16:10:18           0
Configurations2/menubar/                   2007-06-02 16:10:18           0
Configurations2/toolbar/                   2007-06-02 16:10:18           0
Configurations2/images/Bitmaps/            2007-06-02 16:10:18           0
content.xml                                2007-06-02 16:10:18        2776
styles.xml                                 2007-06-02 16:10:18        8492
meta.xml                                   2007-06-02 16:10:18        1143
Thumbnails/thumbnail.png                   2007-06-02 16:10:18         945
settings.xml                               2007-06-02 16:10:18        7476
META-INF/manifest.xml                      2007-06-02 16:10:18        1866
```

You can get the equivalent functionality in PHP with the PHP `zip` library (see `http://us2.php.net/zip`):

```
<?php

$zip = zip_open('[path_to_your_file]');
while ($entry = zip_read($zip)) {
  print zip_entry_name($entry) . "\t". zip_entry_filesize($entry). "\n";
}
zip_close($zip);
?>
```

This produces the following:

```
mimetype                                39
Configurations2/statusbar/               0
Configurations2/accelerator/current.xml  0
Configurations2/floater/                 0
Configurations2/popupmenu/               0
Configurations2/progressbar/             0
Configurations2/menubar/                 0
Configurations2/toolbar/                 0
Configurations2/images/Bitmaps/          0
content.xml                           2776
styles.xml                            8492
meta.xml                              1143
Thumbnails/thumbnail.png               945
settings.xml                          7476
META-INF/manifest.xml                 1866
```

Generating a simple ODF file using OpenOffice.org gives you a basic file from which you can build. However, it's useful to boil the file down even further because

even the simple ODF generated by OO.o contains features that make it difficult to see what's happening. Let's pare down the "Hello World" ODF document further.

There are at least two ways to figure out a minimalist instance of an ODF document. One is to consult the ODF specification, specifically the ODF schema, to generate a small instance. OO.o 2.2 uses the ODF 1.0 specification.[5] The specification contains a RELAX NG schema for ODF. RELAX NG (`http://relaxng.org/`) is a schema language for XML. That is, you can use RELAX NG to specify what elements and attributes can be used in ODF—and in what combination.

Schemas, stemming from the `http://oasis-open.org` page, include the following:

* The schema for office documents, "extracted from chapter 1 to 16 of the specification" —Version 1.0[6]
* "The normative schema for the manifest file used by the OpenDocument package format" —Version 1.0[7]
* "The strict schema for office documents that permits only meta information and formatting properties contained in this specification itself" —Version 1.0[8]

5. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=office
6. http://www.oasis-open.org/committees/download.php/12571/OpenDocument-schema-v1.0-os.rng
7. http://www.oasis-open.org/committees/download.php/12570/OpenDocument-manifest-schema-v1.0-os.rng
8. http://www.oasis-open.org/committees/download.php/12569/OpenDocument-strict-schema-v1.0-os.rng

Instead of taking this approach here, I will instead show you how to use OO.o and the online ODF Validator (`http://opendocumentfellowship.com/validator`). The basic approach is to use a bit of trial and error to generate an ODF file and add pieces while feeding it to the ODF Validator to see how far you can distill the file. Why should you care about minimal instances of ODF (and later OOXML) documents? ODF and OOXML are complicated markup formats. One of the best ways to figure out how to create formats is to use a tool such as OO.o or Microsoft Office to generate what you want, save the file, unzip the file, extract the section of the document you want, and plug that stuff into a minimalist document that you know is valid. That's why you're learning about boiling ODF down to its essence.

The ODF specification (and its RELAX NG schema) should tell you theoretically how to find a valid ODF instance—but in practice, you need to actually feed a given instance to the applications that are the destinations for the ODF documents. OpenOffice.org is currently the most important implementation of an office suite that interprets ODF, making it a good place to experiment.

J. David Eisenberg's excellent book on ODF, *OASIS OpenDocument Essentials*, provides an answer to the question of which files are actually required by OO.o:

> *The only files that are actually necessary are content.xml and the META-INF/manifest.xml file. If you create a file that contains word processor elements and zip it up and a manifest that points to that file, OpenOffice.org will be able to open it successfully. The result will be a plain text-only document with no styles. You won't have any of the meta-information about who created the file or when it was last edited, and the*

printer settings, view area, and zoom factor will be set to the OpenOffice.org defaults.

Let's verify Eisenberg's assertion. Create an `.odt` file with the same `content.xml` as `helloworld.odt`, listed here:

```
<?xml version="1.0" encoding="UTF-8"?>
<office:document-content
  xmlns:office="urn:oasis:names:tc:opendocument:xmlns:office:1.0"
  xmlns:style="urn:oasis:names:tc:opendocument:xmlns:style:1.0"
  xmlns:text="urn:oasis:names:tc:opendocument:xmlns:text:1.0"
  xmlns:table="urn:oasis:names:tc:opendocument:xmlns:table:1.0"
  xmlns:draw="urn:oasis:names:tc:opendocument:xmlns:drawing:1.0"
  xmlns:fo="urn:oasis:names:tc:opendocument:xmlns:xsl-fo-compatible:1.0"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:meta="urn:oasis:names:tc:opendocument:xmlns:meta:1.0"
  xmlns:number="urn:oasis:names:tc:opendocument:xmlns:datastyle:1.0"
  xmlns:svg="urn:oasis:names:tc:opendocument:xmlns:svg-compatible:1.0"
  xmlns:chart="urn:oasis:names:tc:opendocument:xmlns:chart:1.0"
  xmlns:dr3d="urn:oasis:names:tc:opendocument:xmlns:dr3d:1.0"
  xmlns:math="http://www.w3.org/1998/Math/MathML"
  xmlns:form="urn:oasis:names:tc:opendocument:xmlns:form:1.0"
  xmlns:script="urn:oasis:names:tc:opendocument:xmlns:script:1.0"
  xmlns:ooo="http://openoffice.org/2004/office"
  xmlns:ooow="http://openoffice.org/2004/writer"
  xmlns:oooc="http://openoffice.org/2004/calc"
  xmlns:dom="http://www.w3.org/2001/xml-events"
  xmlns:xforms="http://www.w3.org/2002/xforms"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" office:version="1.0">
  <office:scripts/>
  <office:font-face-decls>
    <style:font-face style:name="Tahoma1" svg:font-family="Tahoma"/>
    <style:font-face style:name="Times New Roman"
      svg:font-family="&apos;Times New Roman&apos;"
      style:font-family-generic="roman"
      style:font-pitch="variable"/>
    <style:font-face style:name="Arial" svg:font-family="Arial"
      style:font-family-generic="swiss"
      style:font-pitch="variable"/>
    <style:font-face style:name="Arial Unicode MS"
      svg:font-family="&apos;Arial Unicode MS&apos;"
      style:font-family-generic="system"
      style:font-pitch="variable"/>
    <style:font-face style:name="MS Mincho" svg:font-family="&apos;MS
Mincho&apos;"
      style:font-family-generic="system" style:font-pitch="variable"/>
    <style:font-face style:name="Tahoma" svg:font-family="Tahoma"
      style:font-family-generic="system"
      style:font-pitch="variable"/>
  </office:font-face-decls>
  <office:automatic-styles/>
  <office:body>
```

```
    <office:text>
      <office:forms form:automatic-focus="false" form:apply-design-
mode="false"/>
      <text:sequence-decls>
        <text:sequence-decl text:display-outline-level="0"
          text:name="Illustration"/>
        <text:sequence-decl text:display-outline-level="0" text:name="Table"/>
        <text:sequence-decl text:display-outline-level="0" text:name="Text"/>
        <text:sequence-decl text:display-outline-level="0" text:name="Drawing"/>
      </text:sequence-decls>
      <text:p text:style-name="Standard">Hello World!</text:p>
    </office:text>
  </office:body>
</office:document-content>
```

Now edit `META-INF/metadata.xml` to reference only `content.xml` and the `META-INF` directory:

```
<?xml version="1.0" encoding="UTF-8"?>
<manifest:manifest
    xmlns:manifest="urn:oasis:names:tc:opendocument:xmlns:manifest:1.0">
  <manifest:file-entry manifest:media-
type="application/vnd.oasis.opendocument.text"
                       manifest:full-path="/"/>
  <manifest:file-entry manifest:media-type="text/xml"
                       manifest:full-path="content.xml"/>
</manifest:manifest>
```

This leaves you with an `.odt` file that consists of only those two files.[9] You will find that such a file will load successfully in OpenOffice.org 2.2 and the OpenDocument Viewer[10]—giving credence to the assertion that, in OO.o 2.2 at least, you don't need any more than `content.xml` and `META-INF/manifest.xml`.

Note You can download and install the OpenDocument Validator[11] or run the online version.[12]

Nonetheless, the OpenDocument Validator doesn't find the file to be valid; it produces the following error message:

```
1. warning
   does not contain a /mimetype file. This is a SHOULD in OpenDocument 1.0
2. error
   styles.xml is missing
3. error
   settings.xml is missing
4. error
   meta.xml is missing
```

Since the OpenDocument Validator dies on one of the Fellowship's test files,[13] you can see there are some unresolved problems with the validator or the test files produced

by the OpenDocument Fellowship. Although there is nothing wrong with our minimalist file, it's a good idea to use a file that has all the major pieces in place.

If you insert skeletal `styles.xml`, `settings.xml`, and `meta.xml` files, you can convince the OpenDocument Validator to accept the resulting `.odt` file as a valid document. Furthermore, you can strip `content.xml` of extraneous declarations. (Strictly speaking, the namespace declarations are extraneous, but they are useful to have once you start plugging in chunks of ODF.) The resulting ODF text document is what you find here:

`http://examples.mashupguide.net/ch17/helloworld_min_odt_2.odt`

9. http://examples.mashupguide.net/ch17/helloworld_min_odt_1.odt
10. http://opendocumentfellowship.com/odfviewer
11. http://opendocumentfellowship.com/projects/odftools
12. http://opendocumentfellowship.com/validator
13. http://testsuite.opendocumentfellowship.com/testcases/General/DocumentStructure/

SingleDocumentContents/testDoc/testDoc.odt via
http://testsuite.opendocumentfellowship.com/testcases/General/DocumentStructure/SingleDocumentContents/TestCase.html

Here are the constituent files:

```
<!-- meta.xml -->

<?xml version="1.0" ?>
<office:document-meta office:version="1.0"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:meta="urn:oasis:names:tc:opendocument:xmlns:meta:1.0"
  xmlns:office="urn:oasis:names:tc:opendocument:xmlns:office:1.0"
  xmlns:ooo="http://openoffice.org/2004/office"
  xmlns:xlink="http://www.w3.org/1999/xlink"/>

<!-- settings.xml -->
<?xml version="1.0" ?>
<office:document-settings office:version="1.0"
  xmlns:config="urn:oasis:names:tc:opendocument:xmlns:config:1.0"
  xmlns:office="urn:oasis:names:tc:opendocument:xmlns:office:1.0"
  xmlns:ooo="http://openoffice.org/2004/office"
  xmlns:xlink="http://www.w3.org/1999/xlink" />

<!-- styles.xml -->
<?xml version="1.0" ?>
<office:document-styles office:version="1.0"
  xmlns:chart="urn:oasis:names:tc:opendocument:xmlns:chart:1.0"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:dom="http://www.w3.org/2001/xml-events"
  xmlns:dr3d="urn:oasis:names:tc:opendocument:xmlns:dr3d:1.0"
  xmlns:draw="urn:oasis:names:tc:opendocument:xmlns:drawing:1.0"
  xmlns:fo="urn:oasis:names:tc:opendocument:xmlns:xsl-fo-compatible:1.0"
  xmlns:form="urn:oasis:names:tc:opendocument:xmlns:form:1.0"
  xmlns:math="http://www.w3.org/1998/Math/MathML"
  xmlns:meta="urn:oasis:names:tc:opendocument:xmlns:meta:1.0"
  xmlns:number="urn:oasis:names:tc:opendocument:xmlns:datastyle:1.0"
  xmlns:office="urn:oasis:names:tc:opendocument:xmlns:office:1.0"
```

```
  xmlns:ooo="http://openoffice.org/2004/office"
  xmlns:oooc="http://openoffice.org/2004/calc"
  xmlns:ooow="http://openoffice.org/2004/writer"
  xmlns:script="urn:oasis:names:tc:opendocument:xmlns:script:1.0"
  xmlns:style="urn:oasis:names:tc:opendocument:xmlns:style:1.0"
  xmlns:svg="urn:oasis:names:tc:opendocument:xmlns:svg-compatible:1.0"
  xmlns:table="urn:oasis:names:tc:opendocument:xmlns:table:1.0"
  xmlns:text="urn:oasis:names:tc:opendocument:xmlns:text:1.0"
  xmlns:xlink="http://www.w3.org/1999/xlink" />

<!-- content.xml -->
<?xml version="1.0" ?>
<office:document-content office:version="1.0"
  xmlns:chart="urn:oasis:names:tc:opendocument:xmlns:chart:1.0"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:dom="http://www.w3.org/2001/xml-events"
  xmlns:dr3d="urn:oasis:names:tc:opendocument:xmlns:dr3d:1.0"
  xmlns:draw="urn:oasis:names:tc:opendocument:xmlns:drawing:1.0"
  xmlns:fo="urn:oasis:names:tc:opendocument:xmlns:xsl-fo-compatible:1.0"
  xmlns:form="urn:oasis:names:tc:opendocument:xmlns:form:1.0"
  xmlns:math="http://www.w3.org/1998/Math/MathML"
  xmlns:meta="urn:oasis:names:tc:opendocument:xmlns:meta:1.0"
  xmlns:number="urn:oasis:names:tc:opendocument:xmlns:datastyle:1.0"
  xmlns:office="urn:oasis:names:tc:opendocument:xmlns:office:1.0"
  xmlns:ooo="http://openoffice.org/2004/office"
  xmlns:oooc="http://openoffice.org/2004/calc"
  xmlns:ooow="http://openoffice.org/2004/writer"
  xmlns:script="urn:oasis:names:tc:opendocument:xmlns:script:1.0"
  xmlns:style="urn:oasis:names:tc:opendocument:xmlns:style:1.0"
  xmlns:svg="urn:oasis:names:tc:opendocument:xmlns:svg-compatible:1.0"
  xmlns:table="urn:oasis:names:tc:opendocument:xmlns:table:1.0"
  xmlns:text="urn:oasis:names:tc:opendocument:xmlns:text:1.0"
  xmlns:xforms="http://www.w3.org/2002/xforms"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <office:body>
    <office:text>
      <text:p>
        Hello World!
      </text:p>
    </office:text>
  </office:body>
</office:document-content>

<!-- manifest.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<manifest:manifest
    xmlns:manifest="urn:oasis:names:tc:opendocument:xmlns:manifest:1.0">
  <manifest:file-entry manifest:media-
type="application/vnd.oasis.opendocument.text"
                       manifest:full-path="/"/>
  <manifest:file-entry manifest:media-type="text/xml"
                       manifest:full-path="content.xml"/>
```

```
  <manifest:file-entry manifest:media-type="text/xml"
                       manifest:full-path="meta.xml"/>
  <manifest:file-entry manifest:media-type="text/xml"
                       manifest:full-path="settings.xml"/>
  <manifest:file-entry manifest:media-type="text/xml"
                       manifest:full-path="styles.xml"/>
</manifest:manifest>
```

You now have a *minimalist* and *valid* ODF document.

As an exercise to the reader, I'll leave it to you to generate minimal instances of the spreadsheet (`.ods`), presentation (`.odp`), graphics (`.odg`), and math (`.odf`) documents. In the rest of the chapter, I'll continue to focus on the text documents (`.odt`)—but what you learn from it applies to the other ODF formats as well.

Learning Basic ODF Tags

With a minimalist ODF text document in hand, let's look at how to generate a small example document that illustrates some of the basic features of ODF. Here we can consult Eisenberg once again on how to proceed:

> *Just start OpenOffice.org or KOffice, create a document that has the feature you want, unpack the file, and look for the XML that implements it. To get a better understanding of how things works, change the XML, repack the document, and reload it. Once you know how a feature works, don't hesitate to copy and paste the XML from the OpenDocument file into your program. In other words, cheat. It worked for me when I was writing this book, and it can work for you too!*

In this section, I'll walk you through how to create the ODF text document (see Figure 17-2) in steps here:

`http://examples.mashupguide.net/ch17/odt_example_4.odt`

You can study the parts of this document by downloading and unzipping it or by looking at the files here:

`http://examples.mashupguide.net/ch17/odt_example_4/`

Insert 858Xf1702.tif

Figure 17-2. The culminating ODF text document generated in this chapter

I will show you how I took the approach advocated by Eisenberg to construct this document. I added a new element in an ODT, unzipped the file, found the corresponding XML fragment, took that fragment, and added it to the document I was building. I consciously stripped out any references to styles to focus first on content. And then I applied styles to achieve the effects I want. I will leave it to you to take this approach on spreadsheets and presentations; the ODF for those files formats have a similar framework as the text documents.

This example text contains some common elements:

* Headings of level 1 and 2

* Several paragraphs
* An ordered and unordered list
* Text that has some italics and bold and a font change
* A table
* An image

I'll show how to build this document in four steps to highlight what's involved in constructing a nontrivial ODF document:

1. Create an ODF text document without any styling of ODF elements.
2. Set the style of the paragraph text.
3. Format lists to distinguish between ordered and unordered lists.
4. Get bold, italics, font changes, and color changes into text spans.

Create an ODF Text Document Without Any Styling of ODF Elements

The first step is to create a document while purposefully eliminating any use of styling. This will let us focus on content-oriented tags, much like studying HTML first without CSS and then applying CSS. When studying ODF (and later Office Open XML), it's useful to keep in mind analogous constructs from HTML and CSS.

When you create an ODF text document with headers, paragraphs, lists, and the other features listed earlier and strip out the style, you get something like this:

`http://examples.mashupguide.net/ch17/odt_example_1.odt`

whose constituent files (once you unzip the document) are here:

`http://examples.mashupguide.net/ch17/odt_example_1/`

Not surprisingly, most of the action is in `content.xml`:

`http://examples.mashupguide.net/ch17/odt_example_1/content.xml`

Remember the overall structure of `content.xml`, a framework in which you can plug in the ODF tags representing various elements:

```
<?xml version="1.0" ?>
<!-- the namespace declarations of office:document-content are omitted -->
<office:document-content>
  <office:body>
    <office:text>
[INSERT CONTENT HERE]
    </office:text>
  </office:body>
</office:document-content>
```

Headers and Paragraphs

There are headers and paragraphs as in HTML—but in HTML, you have h1, h2, . . . , h6. With ODF, you use `<text:h>` with `text:outline-level` to indicate the level of the header. For paragraphs in ODF, you use the `<text:p>` element. Note the `text:` namespace:

```
urn:oasis:names:tc:opendocument:xmlns:text:1.0
```

Here's some ODF you can plug in to create two headers (one of level 1 and the other of level 2) along with a series of paragraphs:

```
<text:h text:outline-level="1">Purpose (Heading 1)</text:h>
<text:p>The following sections illustrate various possibilities in ODF Text.
</text:p>
<text:h text:outline-level="2">A simple series of paragraphs (Heading
2)</text:h>
<text:p>This section contains a series of paragraphs.</text:p>
<text:p>This is a second paragraph.</text:p>
<text:p>And a third paragraph.</text:p>
```

Lists

The following ODF markup creates two lists. The first one will ultimately be an unordered one, and the second one will be an ordered one. Unlike HTML in which you would use `<ul>` and `<ol>`, with ODF you get across the difference between an ordered and unordered list through styling alone, which we'll do in a moment. In the meantime, let's set up the two lists using `<text:list>` and `<text:list-item>`:

```
<text:h text:outline-level="2">A section with lists (Heading 2)</text:h>
<text:p>Elements to illustrate:</text:p>
<text:list>
  <text:list-item>
    <text:p>hyperlinks</text:p>
  </text:list-item>
  <text:list-item>
    <text:p>italics and bold text</text:p>
  </text:list-item>
  <text:list-item>
    <text:p>lists (ordered and unordered)</text:p>
  </text:list-item>
</text:list>
<text:p>How to figure out ODF</text:p>
<text:list>
  <text:list-item>
    <text:p>work out the content.xml tags</text:p>
  </text:list-item>
  <text:list-item>
    <text:p>work styles into the mix</text:p>
  </text:list-item>
  <text:list-item>
    <text:p>figure out how to apply what we learned to spreadsheets and
presentations</text:p>
  </text:list-item>
```

```
</text:list>
```

Using text:a and text:span to Bracket Text Styling

The following markup constructs a paragraph with embedded hyperlinks, indicated by `<text:a>` elements. You also want to eventually mark certain text areas as italics ("URL"), as bold ("API page"), and as Arial and red in color ("Flickr"). It turns out that doing so requires styling, which we'll do later. Here we partition out the regions to which styles can then be applied with `<text:span>`—akin to an HTML span.

```
<text:p>The <text:span>URL</text:span> for Flickr is
  <text:a xlink:type="simple"  xlink:href="http://www.flickr.com/">
    http://www.flickr.com
  </text:a>.
  <text:s/>The <text:span>API page</text:span> is
  <text:a  xlink:type="simple" xlink:href="http://www.flickr.com/services/api/">
    http://www.flickr.com/services/api/
  </text:a>
</text:p>
```

Table

The following markup creates a table with three columns and three rows. Note the use of `<table:table>`, `<table:table-row>`, and `<table:table-cell>` where the table namespace is `urn:oasis:names:tc:opendocument:xmlns:table:1.0`:

```
<text:h text:outline-level="1"> A Table (Heading 1)</text:h>
<table:table table:name="Table1">
  <table:table-column table:number-columns-repeated="3"/>
  <table:table-row>
    <table:table-cell office:value-type="string">
      <text:p>Website</text:p>
    </table:table-cell>
    <table:table-cell office:value-type="string">
      <text:p>Description</text:p>
    </table:table-cell>
    <table:table-cell office:value-type="string">
      <text:p>URL</text:p>
    </table:table-cell>
  </table:table-row>
  <table:table-row>
    <table:table-cell office:value-type="string">
      <text:p>Flickr</text:p>
    </table:table-cell>
    <table:table-cell office:value-type="string">
      <text:p>A social photo sharing site</text:p>
    </table:table-cell>
    <table:table-cell office:value-type="string">
      <text:p>
        <text:a xlink:type="simple" xlink:href="http://www.flickr.com/"
          >http://www.flickr.com</text:a>
      </text:p>
    </table:table-cell>
```

```
  </table:table-row>
  <table:table-row>
    <table:table-cell office:value-type="string">
      <text:p>Google Maps</text:p>
    </table:table-cell>
    <table:table-cell office:value-type="string">
      <text:p>An online map</text:p>
    </table:table-cell>
    <table:table-cell office:value-type="string">
      <text:p>
        <text:a xlink:type="simple" xlink:href="http://maps.google.com/"
          >http://maps.google.com</text:a>
      </text:p>
    </table:table-cell>
  </table:table-row>
</table:table>
```

Footnote

The following ODF shows how to embed a footnote through a `<text:note>` element, which contains `<text:note-citation>` , `<text:note-body>`, and `<text:note>` elements:

```
<text:h text:outline-level="1">Footnotes (Heading 1)</text:h>
<text:p>This sentence has an accompanying footnote.<text:note text:id="ftn0"
  text:note-class="footnote">
  <text:note-citation>1</text:note-citation>
  <text:note-body>
    <text:p text:style-name="Footnote">You are reading a footnote.</text:p>
  </text:note-body>
</text:note>
  <text:s text:c="2"/>Where does the text after a footnote go?
</text:p>
```

Embedded Image

The markup in this section embeds an image that I have shown before:

`http://flickr.com/photos/raymondyee/18389540/`

specifically the original size:

`http://farm1.static.flickr.com/12/18389540_e37cc4d464_o.jpg`

You download the image, rename it to `campanile_fog.jpg`, and insert it into the `Pictures` subdirectory of the ODF structure. (Remember that when you unzip an ODF text document, you can often find a `Pictures` subdirectory. That's where embedded images get placed.) Here's what you have to include in `content.xml`:

```
<text:h text:outline-level="1">An Image</text:h>
<text:p>
  <draw:frame draw:name="graphics1" text:anchor-type="paragraph"
    svg:width="5in" svg:height="6.6665in" draw:z-index="0">
    <draw:image xlink:href="Pictures/campanile_fog.jpg" xlink:type="simple"
      xlink:show="embed" xlink:actuate="onLoad"/>
  </draw:frame>
```

```
</text:p>
```

Note that you need to add the photo to the list of files in `META-INF/manifest.xml` (which keeps track of the files that are zipped up in an ODF text document). See the specific changes:[14]

```
<manifest:file-entry manifest:media-type="image/jpeg"
  manifest:full-path="Pictures/campanile_fog.jpg"/>
<manifest:file-entry manifest:media-type="" manifest:full-path="Pictures/"/>
```

Setting the Paragraph Text to text-body

Now that you have the content elements of the ODF text document in place, you can turn to applying styles. In this section, you'll focus first on styling the paragraphs. The default style for paragraphs in OpenOffice.org make it hard to tell when the paragraphs start and end. Let's apply the Text body style from OO.o to some paragraphs. You can do so in two steps: first define the relevant styles, and then apply the new styles to the relevant paragraphs.

To define the styles, you insert the following definition in `styles.xml`:[15]

```
<?xml version="1.0" ?>
<!-- the namespace declarations of office:document-styles are omitted -->
<office:document-styles office:version="1.0">
  <office:styles>
    <style:style style:name="Standard" style:family="paragraph"
style:class="text"/>
    <style:style style:name="Text_20_body" style:display-name="Text body"
      style:family="paragraph"
      style:parent-style-name="Standard" style:class="text">
      <style:paragraph-properties fo:margin-top="0in" fo:margin-
bottom="0.0835in"/>
    </style:style>
  </office:styles>
</office:document-styles>
```

Then you use `text:style-name` attribute to associate this style with a `<text:p>` in `content.xml` to the relevant paragraphs. For example:[16]

```
<text:p text:style-name="Text_20_body">The following sections illustrate various
possibilities in ODF Text.</text:p>
```

You can see the resulting ODF text file here:

```
http://examples.mashupguide.net/ch17/odt_example_2.odt
```

whose constituent files are here:

```
http://examples.mashupguide.net/ch17/odt_example_2/
```

14. http://examples.mashupguide.net/ch17/odt_example_1/META-INF/manifest.xml
15. http://examples.mashupguide.net/ch17/odt_example_2/styles.xml
16. http://examples.mashupguide.net/ch17/odt_example_2/content.xml

Formatting Lists to Distinguish Between Ordered and Unordered Lists

Let's now style the two lists. Recall that you couldn't make the first list unordered and the second ordered without using styling. Let's now do so by using `<style:style>` again—but this time embedded in an `<office:automat-styles>` element in `content.xml`.

```
<?xml version="1.0" ?>
<!-- the namespace declarations of office:document-content are omitted
<office:document-content>
  <office:automatic-styles>
[INSERT automatic-styles here]
  </office:automatic-styles>
  <office:body>
    <office:text>
[...]
    </office:text>
  </office:body>
</office:document-content>
```

specifically:

```
<office:automatic-styles>
  <style:style style:name="P1" style:family="paragraph"
    style:parent-style-name="Standard"
    style:list-style-name="L1"/>
  <style:style style:name="P6" style:family="paragraph"
    style:parent-style-name="Standard"
    style:list-style-name="L5"/>
  <text:list-style style:name="L1">
    <text:list-level-style-bullet text:level="1"
      text:style-name="Numbering_20_Symbols"
      style:num-suffix="." text:bullet-char="•">
      <style:list-level-properties text:space-before="0.25in"
        text:min-label-width="0.25in"/>
      <style:text-properties style:font-name="StarSymbol"/>
    </text:list-level-style-bullet>
[....]
  </text:list-style>
  <text:list-style style:name="L5">
    <text:list-level-style-number text:level="1"
      text:style-name="Numbering_20_Symbols"
      style:num-suffix="." style:num-format="1">
      <style:list-level-properties text:space-before="0.25in"
        text:min-label-width="0.25in"/>
    </text:list-level-style-number>
[....]
  </text:list-style>
</office:automatic-styles>
```

Note that the styling for levels beyond level 1 are deleted in this excerpt.

Once these styles are defined, you then use `text:style-name` attributes to associate the L1/ P1 and L5/P6 styles to the unordered and ordered lists, respectively:

```
<text:p>Elements to illustrate:</text:p>
<text:list text:style-name="L1">
  <text:list-item>
    <text:p text:style-name="P1">hyperlinks</text:p>
  </text:list-item>
  <text:list-item>
    <text:p text:style-name="P1">italics and bold text</text:p>
  </text:list-item>
  <text:list-item>
    <text:p text:style-name="P1">lists (ordered and unordered)</text:p>
  </text:list-item>
</text:list>
<text:p>How to figure out ODF</text:p>
<text:list text:style-name="L5">
  <text:list-item>
    <text:p text:style-name="P6">work out the content.xml tags</text:p>
  </text:list-item>
  <text:list-item>
    <text:p text:style-name="P6">work styles into the mix</text:p>
  </text:list-item>
```

Getting Bold, Italics, Font Changes, and Color Changes into Text Spans

The final changes to make are to define and apply the relevant styles to introduce a number of text effects (bold, italics, font changes, and color changes). Remember that you have the `<text:span>` in place already in `content.xml`. In `content.xml`, you need to do the following:

* Add an `<office:font-face-decls>` element containing a `<style:font-face>` that declares an Arial style.
* Create `<style:style>` elements named T1, T2, T5, respectively, to express the styles of the three `<text:span>` elements to which you are applying styling.
* Associate the T1, T2, and T5 styles with the `<text:span>` elements using `text:style-name` attributes.

Concretely, this means the following:[17]

```
<?xml version="1.0" ?>
<!-- the namespace declarations of office:document-content are omitted
<office:document-content office:version="1.0">
  <office:font-face-decls>
    <style:font-face style:name="Arial" svg:font-family="Arial"
      style:font-family-generic="swiss"
      style:font-pitch="variable"/>
  </office:font-face-decls>
  <office:automatic-styles>
[....]
    <style:style style:name="T1" style:family="text">
      <style:text-properties fo:font-style="italic" style:font-style-
asian="italic"
```

```
        style:font-style-complex="italic"/>
    </style:style>
    <style:style style:name="T2" style:family="text">
      <style:text-properties fo:font-weight="bold" style:font-weight-
asian="bold"
        style:font-weight-complex="bold"/>
    </style:style>
    <style:style style:name="T5" style:family="text">
      <style:text-properties fo:color="#ff0000" style:font-name="Arial"/>
    </style:style>
  </office:automatic-styles>
  <office:body>
    <office:text>
[...]
      <text:p>The <text:span text:style-name="T1">URL</text:span> for <text:span
~CCC
text:style-name="T5">Flickr</text:span> is <text:a xlink:type="simple"
        xlink:href="http://www.flickr.com/"
        >http://www.flickr.com</text:a>. <text:s/>
        The <text:span text:style-name="T2">API page</text:span> is <text:a
          xlink:type="simple" xlink:href="http://www.flickr.com/services/api/"
          >http://www.flickr.com/services/api/</text:a></text:p>
[....]
    </office:text>
  </office:body>
</office:document-content>
```

This series of changes brings you to the completed ODF text document here:

`http://examples.mashupguide.net/ch17/odt_example_4.odt`

There are obviously many other features in ODF that are not demonstrated here. But these examples should give you a good idea of how to learn about the other elements.

17. http://examples.mashupguide.net/ch17/odt_example_4/content.xml

API Kits for Working with ODF

In the previous sections, I showed the approach of working directly with the ODF specification and the validator and using trial and error to generate valid ODF files. In this section, you'll move up the abstraction ladder and look at using libraries/API kits/wrapper libraries that work with ODF. Such libraries can be a huge help if they are implemented well and reflect conscientious effort on the part of the authors to wrestle with some of the issues I discussed in the previous section.

You can find a good list of tools that support ODF here:

`http://en.wikipedia.org/wiki/OpenDocument_software`

You can find another good list here:

`http://opendocumentfellowship.com/development/tools`

In this chapter, I'll cover two API kits:

* I'll cover Odfpy (http://opendocumentfellowship.com/projects/odfpy). According to documentation for Odfpy: "Odfpy aims to be a complete API for OpenDocument in Python. Unlike other more convenient APIs, this one is essentially an abstraction layer just above the XML format. The main focus has been to prevent the programmer from creating invalid documents. It has checks that raise an exception if the programmer adds an invalid element, adds an attribute unknown to the grammar, forgets to add a required attribute or adds text to an element that doesn't allow it."
* I'll cover OpenDocumentPHP (http://opendocumentphp.org/), which is in the early stages of development.

In the next two subsections, I will show you how to use Odfpy and OpenDocumentPHP.

Odfpy

I'll first use Odfpy to generate a minimalist document and then to re-create the full-blown ODF text document from earlier in the chapter. To use it, follow the documentation here:

```
http://opendocumentfellowship.com/files/api-for-odfpy.odt
```

You can access the code via Subversion:

```
svn export http://opendocumentfellowship.com/repos/odfpy/trunk odfpy
```

To generate a "Hello World" document, use this:

```
from odf.opendocument import OpenDocumentText
from odf.text import P

textdoc = OpenDocumentText()
p = P(text="Hello World!")
textdoc.text.addElement(p)
textdoc.save("helloworld_odfpy.odt")
```

This code will generate helloworld_odfpy.odt with the following file structure:

File Name	Modified	Size
mimetype	2007-12-03 15:06:20	39
styles.xml	2007-12-03 15:06:20	403
content.xml	2007-12-03 15:06:20	472
meta.xml	2007-12-03 15:06:20	426
META-INF/manifest.xml	2007-12-03 15:06:20	691

But the generated instance doesn't validate (according to the ODF Validator), even though OO.o 2.2 has no problem reading the file. For many practical purposes, this may be OK, though it'd be nice to know that a document coming out of Odfpy is valid since that's the stated design goal of Odfpy.

Re-creating the Example ODF Text Document

Let's now use Odfpy to generate a more substantial document. The following code demonstrates how you can use Odfpy to re-create the full-blown ODF text document from earlier in the chapter. The code is a rather literal translation of the markup to the corresponding object model of Odfpy—and should give you a feel for how to use Odfpy.

```
# odfpy_gen_example.py

"""

  Description:  This program used odfpy to generate a simple ODF text document
  odfpy:  http://opendocumentfellowship.com/projects/odfpy
  documentation for odfpy:  http://opendocumentfellowship.com/files/api-for-
~CCC
odfpy.odt

"""

from odf.opendocument import OpenDocumentText
from odf.style import Style, TextProperties, ParagraphProperties, ~CCC
ListLevelProperties, FontFace
from odf.text import P, H, A, S, List, ListItem, ListStyle,
ListLevelStyleBullet, ~CCC
ListLevelStyleNumber, ListLevelStyleBullet, Span
from odf.text import Note, NoteBody, NoteCitation
from odf.office import FontFaceDecls
from odf.table import Table, TableColumn, TableRow, TableCell
from odf.draw import Frame, Image

# fname is the path for the output file
fname= '[PATH-FOR-OUTPUT-FILE]';
#fname='D:\Document\PersonalInfoRemixBook\examples\ch17\odfpy_gen_example.odt'

# instantiate an ODF text document (odt)
textdoc = OpenDocumentText()

# styles
"""
<style:style style:name="Standard" style:family="paragraph" style:class="text"/>
<style:style style:name="Text_20_body" style:display-name="Text body"
 style:family="paragraph"
 style:parent-style-name="Standard" style:class="text">
 <style:paragraph-properties fo:margin-top="0in" fo:margin-bottom="0.0835in"/>
</style:style>
"""

s = textdoc.styles

StandardStyle = Style(name="Standard", family="paragraph")
StandardStyle.addAttribute('class','text')
s.addElement(StandardStyle)
```

```
TextBodyStyle = Style(name="Text_20_body",family="paragraph", ~CCC
parentstylename='Standard', displayname="Text body")
TextBodyStyle.addAttribute('class','text')
TextBodyStyle.addElement(ParagraphProperties(margintop="0in", ~CCC
marginbottom="0.0835in"))
s.addElement(TextBodyStyle)

# font declarations
"""
 <office:font-face-decls>
    <style:font-face style:name="Arial" svg:font-family="Arial"
      style:font-family-generic="swiss"
      style:font-pitch="variable"/>
  </office:font-face-decls>
"""

textdoc.fontfacedecls.addElement((FontFace(name="Arial",fontfamily="Arial",
~CCC
fontfamilygeneric="swiss",fontpitch="variable")))

# Automatic Style

# P1
"""
<style:style style:name="P1" style:family="paragraph"
      style:parent-style-name="Standard"
      style:list-style-name="L1"/>
"""
P1style = Style(name="P1", family="paragraph", parentstylename="Standard", ~CCC
liststylename="L1")
textdoc.automaticstyles.addElement(P1style)

# L1
"""
<text:list-style style:name="L1">
  <text:list-level-style-bullet text:level="1"
    text:style-name="Numbering_20_Symbols"
    style:num-suffix="." text:bullet-char="•">
    <style:list-level-properties text:space-before="0.25in"
      text:min-label-width="0.25in"/>
    <style:text-properties style:font-name="StarSymbol"/>
  </text:list-level-style-bullet>
</text:list-style>
"""
L1style=ListStyle(name="L1")
# u'\u2022' is the bullet character
(http://www.unicode.org/charts/PDF/U2000.pdf)
bullet1 = ListLevelStyleBullet(level="1", stylename="Numbering_20_Symbols",
~CCC
numsuffix=".", bulletchar=u'\u2022')
L1prop1 = ListLevelProperties(spacebefore="0.25in", minlabelwidth="0.25in")
bullet1.addElement(L1prop1)
L1style.addElement(bullet1)
textdoc.automaticstyles.addElement(L1style)
```

```
# P6
"""
  <style:style style:name="P6" style:family="paragraph"
      style:parent-style-name="Standard"
      style:list-style-name="L5"/>
"""

P6style = Style(name="P6", family="paragraph", parentstylename="Standard", ~CCC
liststylename="L5")
textdoc.automaticstyles.addElement(P6style)

# L5
"""
<text:list-style style:name="L5">
  <text:list-level-style-number text:level="1"
    text:style-name="Numbering_20_Symbols"
    style:num-suffix="." style:num-format="1">
    <style:list-level-properties text:space-before="0.25in"
      text:min-label-width="0.25in"/>
  </text:list-level-style-number>
</text:list-style>
"""

L5style=ListStyle(name="L5")
numstyle1 = ListLevelStyleNumber(level="1", stylename="Numbering_20_Symbols",
~CCC
numsuffix=".", numformat='1')
L5prop1 = ListLevelProperties(spacebefore="0.25in", minlabelwidth="0.25in")
numstyle1.addElement(L5prop1)
L5style.addElement(numstyle1)
textdoc.automaticstyles.addElement(L5style)

# T1
"""
   <style:style style:name="T1" style:family="text">
      <style:text-properties fo:font-style="italic" style:font-style-
asian="italic"
        style:font-style-complex="italic"/>
    </style:style>
"""
T1style = Style(name="T1", family="text")
T1style.addElement(TextProperties(fontstyle="italic",fontstyleasian="italic",
~CCC
fontstylecomplex="italic"))
textdoc.automaticstyles.addElement(T1style)

# T2
"""
 <style:style style:name="T2" style:family="text">
      <style:text-properties fo:font-weight="bold" style:font-weight-
asian="bold"
        style:font-weight-complex="bold"/>
    </style:style>
```

```
"""
T2style = Style(name="T2", family="text")
T2style.addElement(TextProperties(fontweight="bold",fontweightasian="bold",
~CCC
fontweightcomplex="bold"))
textdoc.automaticstyles.addElement(T2style)

# T5
"""
   <style:style style:name="T5" style:family="text">
      <style:text-properties fo:color="#ff0000" style:font-name="Arial"/>
    </style:style>
"""
T5style = Style(name="T5", family="text")
T5style.addElement(TextProperties(color="#ff0000",fontname="Arial"))
textdoc.automaticstyles.addElement(T5style)

# now construct what goes into <office:text>

h=H(outlinelevel=1, text='Purpose (Heading 1)')
textdoc.text.addElement(h)
p = P(text="The following sections illustrate various possibilities in ODF
Text", ~CCC
stylename='Text_20_body')
textdoc.text.addElement(p)

textdoc.text.addElement(H(outlinelevel=2,text='A simple series of paragraphs
~CCC
(Heading 2)'))
textdoc.text.addElement(P(text="This section contains a series of paragraphs.",
~CCC
stylename='Text_20_body'))
textdoc.text.addElement(P(text="This is a second paragraph.", ~CCC
stylename='Text_20_body'))
textdoc.text.addElement(P(text="And a third paragraph.",
stylename='Text_20_body'))

textdoc.text.addElement(H(outlinelevel=2,text='A section with lists (Heading
2)'))
textdoc.text.addElement(P(text="Elements to illustrate:"))

# add the first list (unordered list)
textList = List(stylename="L1")
item = ListItem()
item.addElement(P(text='hyperlinks', stylename="P1"))
textList.addElement(item)

item = ListItem()
item.addElement(P(text='italics and bold text', stylename="P1"))
textList.addElement(item)

item = ListItem()
item.addElement(P(text='lists (ordered and unordered)', stylename="P1"))
textList.addElement(item)
```

```
textdoc.text.addElement(textList)

# add the second (ordered) list

textdoc.text.addElement(P(text="How to figure out ODF"))

textList = List(stylename="L5")
#item = ListItem(startvalue=P(text='item 1'))
item = ListItem()
item.addElement(P(text='work out the content.xml tags', stylename="P5"))
textList.addElement(item)

item = ListItem()
item.addElement(P(text='work styles into the mix', stylename="P5"))
textList.addElement(item)

item = ListItem()
item.addElement(P(text='figure out how to apply what we learned to spreadsheets
and ~CCC
presentations', stylename="P5"))
textList.addElement(item)

textdoc.text.addElement(textList)

# A paragraph with bold, italics, font change, and hyperlinks
"""
      <text:p>The <text:span text:style-name="T1">URL</text:span> for <text:span
          text:style-name="T5">Flickr</text:span> is <text:a xlink:type="simple"
          xlink:href="http://www.flickr.com/"
          >http://www.flickr.com</text:a>. <text:s/>The <text:span
          text:style-name="T2"
          >API page</text:span> is <text:a xlink:type="simple"
          xlink:href="http://www.flickr.com/services/api/"
        >http://www.flickr.com/services/api/</text:a></text:p>
"""
p = P(text='The ')
# italicized URL
s = Span(text='URL', stylename='T1')
p.addElement(s)
p.addText(' for ')
# Flickr in red and Arial font
p.addElement(Span(text='Flickr',stylename='T5'))
p.addText(' is ')
# link
link = A(type="simple",href="http://www.flickr.com",
text="http://www.flickr.com")
p.addElement(link)
p.addText('.  The ')
# API page in bold
s = Span(text='API page', stylename='T2')
p.addElement(s)
p.addText(' is ')
```

```
link = A(type="simple",href="http://www.flickr.com/services/api", ~CCC
text="http://www.flickr.com/services/api")
p.addElement(link)

textdoc.text.addElement(p)

# add the table
"""
<table:table-column table:number-columns-repeated="3"/>
"""

textdoc.text.addElement(H(outlinelevel=1,text='A Table (Heading 1)'))

table = Table(name="Table 1")

table.addElement(TableColumn(numbercolumnsrepeated="3"))

# first row
tr = TableRow()
table.addElement(tr)
tc = TableCell(valuetype="string")
tc.addElement(P(text='Website'))
tr.addElement(tc)
tc = TableCell(valuetype="string")
tc.addElement(P(text='Description'))
tr.addElement(tc)
tc = TableCell(valuetype="string")
tc.addElement(P(text='URL'))
tr.addElement(tc)

# second row
tr = TableRow()
table.addElement(tr)
tc = TableCell(valuetype="string")
tc.addElement(P(text='Flickr'))
tr.addElement(tc)
tc = TableCell(valuetype="string")
tc.addElement(P(text='A social photo sharing site'))
tr.addElement(tc)
tc = TableCell(valuetype="string")

link = A(type="simple",href="http://www.flickr.com",
text="http://www.flickr.com")
p = P()
p.addElement(link)
tc.addElement(p)

tr.addElement(tc)

# third row
tr = TableRow()
table.addElement(tr)
tc = TableCell(valuetype="string")
tc.addElement(P(text='Google Maps'))
```

```
tr.addElement(tc)
tc = TableCell(valuetype="string")
tc.addElement(P(text='An online map'))
tr.addElement(tc)
tc = TableCell(valuetype="string")

link = A(type="simple",href="http://maps.google.com",
text="http://maps.google.com")
p = P()
p.addElement(link)
tc.addElement(p)
tr.addElement(tc)

textdoc.text.addElement(table)

# paragraph with footnote

"""
   <text:h text:outline-level="1">Footnotes (Heading 1)</text:h>
      <text:p>This sentence has an accompanying footnote.<text:note
text:id="ftn0"
          text:note-class="footnote">
          <text:note-citation>1</text:note-citation>
          <text:note-body>
            <text:p text:style-name="Footnote">You are reading a
footnote.</text:p>
          </text:note-body>
        </text:note>
        <text:s text:c="2"/>Where does the text after a footnote go?</text:p>
"""

textdoc.text.addElement(H(outlinelevel=1,text='Footnotes (Heading 1)'))
p = P()
textdoc.text.addElement(p)
p.addText("This sentence has an accompanying footnote.")
note = Note(id="ftn0", noteclass="footnote")
p.addElement(note)
note.addElement(NoteCitation(text='1'))
notebody = NoteBody()
note.addElement(notebody)
notebody.addElement(P(stylename="Footnote", text="You are reading a footnote."))
p.addElement(S(c=2))
p.addText("Where does the text after a footnote go?")

# Insert the photo

"""
     <text:h text:outline-level="1">An Image</text:h>
      <text:p>
        <draw:frame draw:name="graphics1" text:anchor-type="paragraph"
          svg:width="5in"
          svg:height="6.6665in" draw:z-index="0">
          <draw:image xlink:href="Pictures/campanile_fog.jpg"
xlink:type="simple"
```

```
            xlink:show="embed"
            xlink:actuate="onLoad"/>
        </draw:frame>
      </text:p>
"""

textdoc.text.addElement(H(outlinelevel=1,text='An Image'))
p = P()
textdoc.text.addElement(p)
# add the image
# img_path is the local path of the image to include
img_path = '[PATH-FOR-IMAGE]';
#img_path = 'D:\Document\PersonalInfoRemixBook\examples\ch17\campanile_fog.jpg'
href = textdoc.addPicture(img_path)
f = Frame(name="graphics1", anchortype="paragraph", width="5in",
height="6.6665in", ~CCC zindex="0")
p.addElement(f)
img = Image(href=href, type="simple", show="embed", actuate="onLoad")
f.addElement(img)

# save the document
textdoc.save(fname)
```

You can examine the output from this code:

```
http://examples.mashupguide.net/ch17/odfpy_gen_example.odt
```

OpenDocumentPHP

OpenDocumentPHP (http://opendocumentphp.org/) is a PHP API kit for ODF in its early stages of development.

In this section, I'll show how to use OpenDocumentPHP version 0.5.2, which you can get from here:

```
http://downloads.sourceforge.net/opendocumentphp/OpenDocumentPHP-0.5.2.zip
```

Alternatively, you can install OpenDocumentPHP using PEAR:

```
http://opendocumentphp.org/index.php/home/11-new-pear-server-for-opendocumentphp
```

Some autogenerated documentation of the API is available here:

```
http://opendocumentphp.org/static/apidoc/svn/
```

Unzip the file in your PHP library area. To see a reasonably complicated example of what you can do, consult the samples in `OpenDocumentPHP/samples`.

Here I will write a simple `helloworld`-generated document to demonstrate how to get started with the library:

```
<?php
require_once 'OpenDocumentPHP/OpenDocumentText.php';
$text = new
OpenDocumentText('D:\Document\PersonalInfoRemixBook\examples\ch17\~CCC
helloworld_opendocumentphp.odt');
$textbody = $text->getBody();
$paragraph = $textbody->nextParagraph();
```

```
$paragraph->append('Hello World!');
$text->close();
?>
```

Note You need ZipArchive to be enabled in PHP to run OpenDocumentPHP. On Linux systems use the `--enable-zip` option at compile time. On Windows systems, enable `php_zip.dll` inside `php.ini`.

The following is a more elaborate example using OpenDocumentPHP to generate a couple of headers and several paragraphs. The paragraphs are associated with a Text body style.

```
<?php

require_once 'OpenDocumentPHP/OpenDocumentText.php';

$fullpath =
'D:\Document\PersonalInfoRemixBook\examples\ch17\odp_gen_example.odt';

/*
 * If file exists, remove it first.
 */
if (file_exists($fullpath)) {
    unlink($fullpath);
}

$text = new OpenDocumentText($fullpath);

# set some styles

/**

<style:style style:name="Standard" style:family="paragraph" style:class="text"/>
<style:style style:name="Text_20_body" style:display-name="Text body"
 style:family="paragraph"
 style:parent-style-name="Standard" style:class="text">
 <style:paragraph-properties fo:margin-top="0in" fo:margin-bottom="0.0835in"/>
</style:style>

**/

$Standard_Style = $text->getStyles()->getStyles()->getStyle();
$Standard_Style->setStyleName('Standard');
$Standard_Style->setFamily('paragraph');
$Standard_Style->setClass('text');

$textBody_Style = $text->getStyles()->getStyles()->getStyle();
$textBody_Style->setStyleName('Text_20_body');
$textBody_Style->setDisplayName('Text body');
$textBody_Style->setFamily('paragraph');
$textBody_Style->setClass('text');
```

```
$pp = $textBody_Style->getParagraphProperties();
$pp->setMarginTop('0in');
$pp->setMarginBottom('0.0835in');

# write the headers and paragraphs

$textbody = $text->getBody()->getTextFragment();

$heading = $textbody->nextHeading();
$heading->setHeadingLevel(1);
$heading->append('Purpose (Heading 1)');

$paragraph = $textbody->nextParagraph();
$paragraph->setStyleName('Text_20_body');
$paragraph->append('The following sections illustrate various possibilities in
ODF
Text');

$heading = $textbody->nextHeading();
$heading->setHeadingLevel(2);
$heading->append('A simple series of paragraphs (Heading 2)');

$paragraph = $textbody->nextParagraph();
$paragraph->setStyleName('Text_20_body');
$paragraph->append('This section contains a series of paragraphs.');
$paragraph = $textbody->nextParagraph();
$paragraph->setStyleName('Text_20_body');
$paragraph->append('This is a second paragraph.');
$paragraph = $textbody->nextParagraph();
$paragraph->setStyleName('Text_20_body');
$paragraph->append('And a third paragraph.');

$text->close();

?>
```

You can examine the output from this script here:

`http://examples.mashupguide.net/ch17/odp_gen_example.odt`

Leveraging OO.o to Generate ODF

If you are willing and able to have OpenOffice.org installed on your computer, it is possible to use OO.o itself as a big library of sorts to parse and generate your ODF documents and to convert ODF to and from other formats. Libraries/tools that use this approach include the following:

* The JODConverter Java Library (`http://www.artofsolving.com/opensource/jodconverter`)
* OOoLib, Perl and Python libraries that use OO.o (`http://sourceforge.net/projects/ooolib/`)

On Win32-oriented systems, you can access OpenOffice.org via a COM interface. For instance, the following Python code running the `win32all` library will generate a new `.odt` document by scripting OO.o:

```
import win32com.client

objServiceManager = win32com.client.Dispatch("com.sun.star.ServiceManager")
objServiceManager._FlagAsMethod("CreateInstance")
objDesktop = objServiceManager.CreateInstance("com.sun.star.frame.Desktop")
objDesktop._FlagAsMethod("loadComponentFromURL")

args = []
objDocument = objDesktop.loadComponentFromURL("private:factory/swriter",
"_blank", ~CCC 0, args)
objDocument._FlagAsMethod("GetText")
objText = objDocument.GetText()
objText._FlagAsMethod("createTextCursor","insertString")
objCursor = objText.createTextCursor()
objText.insertString(objCursor, "The first line in the newly created text~CCC
document.\n", 0)
```

ECMA Office Open XML (OOXML)

Now we turn to a competing file format: Office Open XML. Wikpedia provides a good overview of the specification that underpins Microsoft Office 2007:

`http://en.wikipedia.org/wiki/Office_Open_XML`

The Office Open XML specification has been made into an ECMA standard (ECMA-376). You can find the specification here:

`http://www.ecma-international.org/publications/standards/Ecma-376.htm`

Note that the standard is 6,000 pages—in case you want to read it. ECMA provides an overview white paper:

`http://www.ecma-international.org/news/TC45_current_work/OpenXML%20White%20Paper.pdf`

Getting hard, easy-to-digest information on OOXML is challenging. I recommend the following, more colloquial overviews that you might find useful:

* The "5 Cool Things You Must Know About the New Office 2007 File Formats" article might prove helpful (`http://www.devx.com/MicrosoftISV/Article/30907/2046`).
* `http://openxmldeveloper.org/default.aspx` has some useful tutorials on the subject.

When working with Office Open XML, it's good to heed the following warning: "Open XML is a new standard. So new, in fact, that the schemas are still being edited and haven't been published by ECMA yet. And there are no books out on Open XML development, although that will surely change in the next year."[18]

The Office Open XML format has a predecessor in the Microsoft Office 2003 XML format. In the book *Office 2003 XML* (O'Reilly Media, 2004), the following was given as a minimalist Office 2003 XML document:[19]

18. http://openxmldeveloper.org/articles/LearningOnline.aspx (accessed on June 5, 2007)

19. http://examples.mashupguide.net/ch17/helloworld_onedoc_2003.xml

```
<?xml version="1.0"?>
<?mso-application progid="Word.Document"?>
<w:wordDocument
  xmlns:w="http://schemas.microsoft.com/office/word/2003/wordml">
  <w:body>
    <w:p>
      <w:r>
        <w:t>Hello, World!</w:t>
      </w:r>
    </w:p>
  </w:body>
</w:wordDocument>
```

This document is actually readable by Microsoft Office 2007, though in "compatibility mode." Can you get a valid document by using the Microsoft Office 2003 document and updating the namespace of the document? That is, can you just update the namespace for `w`?

```
xmlns:w="http://schemas.openxmlformats.org/wordprocessingml/2006/main"
```

You therefore have the following:

```
<?xml version="1.0"?>
<?mso-application progid="Word.Document"?>
<w:wordDocument
  xmlns:w="http://schemas.openxmlformats.org/wordprocessingml/2006/main">
  <w:body>
    <w:p>
      <w:r>
        <w:t>Hello, World!</w:t>
      </w:r>
    </w:p>
  </w:body>
</w:wordDocument>
```

Unfortunately this document as an Office Open XML instance to Microsoft Office 2007 causes an error. You can certainly keep pushing in this direction by looking through the specification and schema. However, a more promising lead right now is to see what file gets written out by a simple little C# script aimed at generating a simple `.docx` file:

```
http://blogs.msdn.com/dmahugh/archive/2006/06/27/649007.aspx
```

I downloaded the Microsoft Visual Studio C# Express Edition to run the script and made a small change to update the namespace from this:

```
http://schemas.openxmlformats.org/wordprocessingml/2006/3/main
```

to this:

```
http://schemas.openxmlformats.org/wordprocessingml/2006/main
```

With that change, you can generate a simple Office Open XML document file (http://examples.mashupguide.net/ch17/helloworld_simple.1.docx) that is acceptable by Microsoft Office 2007. (This doesn't prove that the file is valid but only that you are on the right track in terms of generating OOXML.)

Unzipping and studying the file gives you insight into what goes into a minimalist instance of OOXML. The list of files is as follows:

File Name	Modified	Size
word/document.xml	2007-06-04 16:43:44	246
[Content_Types].xml	2007-06-04 16:43:44	346
_rels/.rels	2007-06-04 16:43:44	285

Let's look at the individual files. The first is the document.xml file in the word directory, which holds the content of the document and corresponds most closely to content.xml in ODF.

```
<?xml version="1.0" encoding="utf-8"?>
<w:document
    xmlns:w="http://schemas.openxmlformats.org/wordprocessingml/2006/main">
  <w:body>
    <w:p>
      <w:r>
        <w:t>Hello World!</w:t>
      </w:r>
    </w:p>
  </w:body>
</w:document>
```

The .rels file in the rels directory contains information about relationships among the various files that make up the package of files (a bit like the METAINF/meta.xml file in ODF):

```
<?xml version="1.0" ?>
<Relationships
xmlns="http://schemas.openxmlformats.org/package/2006/relationships">
  <Relationship Id="rId1" Target="/word/document.xml"

Type="http://schemas.openxmlformats.org/officeDocument/2006/~CCC
relationships/officeDocument"/>
</Relationships>
```

The final file in the package is [Content_Types].xml:

```
<?xml version="1.0" ?>
<Types xmlns="http://schemas.openxmlformats.org/package/2006/content-types">
  <Default ContentType="application/vnd.openxmlformats-officedocument. ~CCC
wordprocessingml.document.main+xml" Extension="xml"/>
  <Default ContentType="application/vnd.openxmlformats-
package.relationships+xml"
           Extension="rels"/>
</Types>
```

These files should give you a feel of what's in OOXML. To learn more, take a look at the following resources:

* The "Ecma Office Open XML Format Guide" is an official high-level conceptual/marketing overview of OOXML.
* http://openxmldeveloper.org/articles/directory.aspx lists tutorial articles that are gathered by the OOXML community.
* http://openxmldeveloper.org/articles/OpenXMLsamples.aspx has sample OOXML documents.
* http://msdn2.microsoft.com/en-us/library/bb187361.aspx gives the object model of Microsoft Office 2007.
* http://en.wikipedia.org/wiki/User:Flemingr/Microsoft_Office_2003_XML_formats documents the older Office 2003 XML format, which has some family resemblance to OOXML—though an unclear one to me.
* Brian Jones of Microsoft has written some clear tutorials on generating spreadsheets in OOXML: http://blogs.msdn.com/brian_jones/archive/2007/05/29/simple-spreadsheetml-file-part-3-formatting.aspx.

Viewers/Validators for OOXML

A big point of OOXML is being able to read and generate documents that are readable in the latest versions of Microsoft Office without having to directly manipulate the object models of Microsoft Office. Yet, it's always helpful to have tools that view and validate OOXML documents—other than Microsoft Office 2007 itself. Some promising tools are as follows:

* Open XML Package Explorer, which lets you browse and edit Open XML packages and validate against the ECMA final schemas (http://www.codeplex.com/PackageExplorer).
* If you are using Microsoft Office XP and 2003, you can download a Microsoft Office compatibility pack for the Word, Excel, and PowerPoint 2007 file formats to read and write OOXML.[20] This will also enable you to use the free Microsoft Office Word Viewer 2003 and Microsoft Office Excel Viewer 2003 to view Word 2007 and Excel 2007 files.[21]

Comparing ODF and OOXML

I will not get into surveying the complicated and often-heated comparisons made between ODF and OOXML other than to refer you to the following articles, which in turn provide more references:

* http://en.wikipedia.org/wiki/Comparison_of_OpenDocument_and_Office_Open_XML_formats compares formats.

* http://weblog.infoworld.com/realitycheck/archives/2007/05/odf_vs_openxml.html gives a flavor of the conflation of political, economical, PR, and technical issues.

20. http://www.microsoft.com/downloads/details.aspx?FamilyId=941b3470-3ae9-4aee-8f43-c6bb74cd1466&displaylang=en

21. http://support.microsoft.com/kb/925180

Online Office Suites

Web-based offices suites are emerging in addition to the traditional desktop office suites and their respective file formats. Prominent examples of such applications include the Zoho Office Suite (http://zoho.com/) and Google Docs and Spreadsheets (http://docs.google.com/). There are others, of course. You can see a list of online spreadsheets, for instance, here:

```
http://en.wikipedia.org/wiki/List_of_online_spreadsheets
```

I will focus on using a programmable online spreadsheet, specifically Google Spreadsheets, in this section. Google Spreadsheets has an API (which we will use in a mashup later in the chapter):

```
http://code.google.com/apis/spreadsheets/overview.html
```

Usage Scenarios for Programmable Online Spreadsheets

What might one want to do with an online spreadsheet? Here are a few examples I brainstormed:

* Tracking one's weight, finances, or time and sharing that information with your family and friends—or not.
* Having bots calculate data that they put into your spreadsheets that you can then analyze. For instance, if you wanted to track your stock portfolio, you could use the StrikeIron fee-based real-time stock quote service to calculate the value of your portfolio. (You might think twice before storing that portfolio information online, but this is feasible in principle.)
* Build an application to track and disseminate grades.
* Manage a wedding database.
* Build a project management tool that you can update and read with the API.
* Back up a list of your del.icio.us bookmarks in a spreadsheet form.
* Track your library books.
* Build online charts (see http://imagine-it.org/google/spreadsheets/makechart.htm).

There are many other applications. Consider StrikeIron SOA Express for Excel (http://www.strikeiron.com/tools/tools_soaexpress.aspx) as a source of hints about what people might do with the Google Spreadsheets API; that is, start to think of

Google Spreadsheets as Excel in the cloud, but account for its lack of some of Excel's current internal extensibility such as macros. (There is no equivalent to Google Mapplets for the spreadsheets or VBA macros—yet.)

The application I will demonstrate in detail is copying my Amazon.com wishlist and prices to a spreadsheet to more easily take that information with me (say to a real-life bookstore or library).

Google Spreadsheets API

Let's figure out how to use the Google Spreadsheets API, focusing specifically on PHP and Python wrapper libraries. You can also directly manipulate the feed protocol:

```
http://code.google.com/apis/spreadsheets/developers_guide_protocol.html
```

There are also a number of API kits available for Google Spreadsheets. I will first demonstrate how to use the Python API kit by creating a mashup involving the Amazon.com web services. I'll then show you a simple example of using the Zend PHP API kit to read the spreadsheets generated in the mashup.

Python API Kit

Google provides a Python GData library and sample code to access the Google spreadsheet. You can either download specific releases (from http://code.google.com/p/gdata-python-client/downloads/list) or access the SVN repository:

```
svn checkout http://gdata-python-client.googlecode.com/svn/trunk/~CCC
gdata-python-client
```

Note the dependencies on other libraries, especially ElementTree, which was not part of the standard Python libraries until version 2.5.[22]

I highly recommend reading the documentation on the Google site specific to the Python library:

```
http://code.google.com/apis/spreadsheets/developers_guide_python.html
```

Once you have the Python GData library installed, you can try some code samples, using the Python interpreter, to teach yourself how it works. First here are the obligatory imports:

```
import gdata.spreadsheet.service
```

Let's then declare some convenience functions and variables:

```
GoogleUser = "[your Google email address]"
GooglePW = "[your password]"
```

Define the following convenience function:

```
def GSheetService(user,pwd):
    gd_client = gdata.spreadsheet.service.SpreadsheetsService()
    gd_client.email = user
    gd_client.password = pwd
    gd_client.source = 'amazonWishListToGSheet.py'
    gd_client.ProgrammaticLogin()
```

```
    return gd_client
```

22. http://code.google.com/p/gdata-python-client/wiki/DependencyModules

Instantiate a Google data client for your spreadsheet:

```
gs = GSheetService(GoogleUser,GooglePW)
sheets = gs.GetSpreadsheetsFeed()
```

To get a list of the spreadsheets, their titles, and their IDs, use this:

```
print map(lambda e: e.title.text +~CCC
" : "  + e.id.text.rsplit('/', 1)[1],sheets.entry)
```

This yields something like the following (which is based on my own spreadsheets):

```
['My Amazon WishList : o06341737111865728099.3585145106901556666', 'Udell Mini-
Symposium May 1, 2007 : o06341737111865728099.1877210150658854761',
'weight.journal
: o06341737111865728099.6289501454054682788', 'Plan : o10640374522570553588.
5762564240835257179']
```

Note the key for the spreadsheet "My Amazon WishList." The spreadsheet that the code I write here will be reading from and writing to is as follows:

```
o06341737111865728099.3585145106901556666
```

You will need to create your own Google spreadsheets to work with since you won't be able to write to mine. Note the ID of your spreadsheet, which you will use here.

In the browser, if I'm logged in as the owner of the spreadsheet, I can access this:

```
http://spreadsheets.google.com/feeds/spreadsheets/private/full/{GSheetID}
```

For example:

```
http://spreadsheets.google.com/feeds/spreadsheets/private/full/o0634173711186572
809~CCC
.3585145106901556666
```

Otherwise, I get a 404 error. Now I need to get the ID of the one *worksheet* in the "My Amazon Wishlist" *spreadsheet.* First use this:

```
gs.GetWorksheetsFeed(key="[GSheetID]").entry[0].id.text
```

For example, I use this:

```
gs.GetWorksheetsFeed(key="o06341737111865728099.3585145106901556666").entry[0].
~CCC
id.text
```

to return the URL whose last segment is a worksheet ID—that is, a URL of the following form:

```
http://spreadsheets.google.com/feeds/worksheets/{GSheetID}/private/full/~CCC
{worksheetID}
```

For example:

```
http://spreadsheets.google.com/feeds/worksheets/o06341737111865728099.3585145106
9015~CCC
56666/private/full/od6
```

(in which case the worksheet ID is od6).

Now you can get the worksheet ID:

```
gs.GetWorksheetsFeed(key="[GSheetID]").entry[0].id.text.rsplit('/', 1)[1]
```

For example:

```
gs.GetWorksheetsFeed(key="o06341737111865728099.3585145106901556666").entry[0].
~CCC
id.text.rsplit('/', 1)[1]
```

There are two ways to get at the data in a worksheet—either in a list-based way that gets you rows or in a cell-based way that gets you a range of cells. I will show the list-based approach here, which depends on the assumption that the first row is the header row.

For testing purposes, I created a spreadsheet with the header row and one line of data that I entered, as shown in Table 17-1.

Table 17-1. The Sample Spreadsheet

ASIN	DetailPageURL	Title	Author	Date Added	Price	Quantity Desired
1590598385	http://www.amazon.com/gp/product/1590598385/	*Smart and Gets Things Done: Joel Spolsky's Concise Guide to Finding the Best Technical Talent* (Hardcover)	Joel Spolsky	6/5/2007	13.25	1

The following returns a feed for the rows (there's only one):

```
lfeed = gs.GetListFeed(key="[GSheetID]",wksht_id="[worksheetID]")
```

For example:

```
lfeed =~CCC
gs.GetListFeed(key="o06341737111865728099.3585145106901556666",wksht_id="od6")
```

You can see the content of the row with the following:

```
lfeed.entry[0].content.text
```

This results in the following:

```
'ASIN: 1590598385, DetailPageURL:
http://www.amazon.com/gp/product/1590598385/ref=wl_it_dp/103-8266902-5986239?
ie=UTF8&coliid=I1AOWT8LH796DN&colid=1U5EXVPVS3WP5, Author: Joel Spolsky, Date
Added:
6/5/2007, Price: 13.25, Quantity Desired: 1'
```

The following holds the data that has been mapped from namespace-extended elements in the entry (see http://code.google.com/apis/spreadsheets/developers_guide_protocol.html#listFeedExample):

```
lfeed.entry[0].custom
```

Let's see this in action:

```
map(lambda e: (e[0],e[1].text), lfeed.entry[0].custom.items())
```

Running this returns the following:

```
[('asin', '1590598385'), ('dateadded', '6/5/2007'), ('detailpageurl',
'http://www.amazon.com/gp/product/1590598385/ref=wl_it_dp/103-8266902-5986239?
ie=UTF8&coliid=I1AOWT8LH796DN&colid=1U5EXVPVS3WP5'), ('author', 'Joel
Spolsky'), ('quantitydesired', '1'), ('price', '13.25'), ('title', "Smart and
Gets
Things Done: Joel Spolsky's Concise Guide to Finding the Best Technical Talent
(Hardcover) ")]
```

Now let's look at adding another row of data. Let's see whether you can just duplicate the row by creating a dictionary of the first row and stick it into the second row:

```
h = {}
for (key,value) in lfeed.entry[0].custom.iteritems():
  h[key] = value.text
```

`h` now is as follows:

```
{'asin': '1590598385', 'dateadded': '6/5/2007', 'detailpageurl':
'http://www.amazon.com/gp/product/1590598385/ref=wl_it_dp/103-8266902-5986239?
ie=UTF8&coliid=I1AOWT8LH796DN&colid=1U5EXVPVS3WP5', 'author': 'Joel
Spolsky', 'quantitydesired': '1', 'price': '13.25', 'title': "Smart and Gets
Things
Done: Joel Spolsky's Concise Guide to Finding the Best Technical Talent
(Hardcover)
"}
```

To add the new row, use this:

```
gs.InsertRow(row_data=h,key="[GSheetID]",wksht_id="[worksheetID]")
```

For example:

```
gs.InsertRow(row_data=h,key="oO63417371118657280 99.3585145106901556666", ~CCC
wksht_id="od6")
```

To clear the second row you just added, you need to get an update `lfeed` that reflects the current state of the spreadsheet/worksheet and then issue a delete command:

```
lfeed = gs.GetListFeed(key="[GSheetID]",
wksht_id="[worksheetID]")
gs.DeleteRow(lfeed.entry[1])
```

Note The Google Spreadsheets API is under active development and is still in the process of maturation.

Mashup: Amazon Wishlist and Google Spreadsheets Mashup

To demonstrate how to use Google Spreadsheets for a simple mashup, I will show you how to write code that will transfer the contents of an Amazon.com wishlist to a Google Spreadsheets spreadsheet. Why do that? I use my wishlist to keep track of books and other stuff that I find interesting. If the wishlist belonged to someone else, I might want to download it into a spreadsheet to make it easier to generate a hard-copy shopping list I could use.

Accessing the Wishlist Through the Amazon.com ECS Web Service

First, a word about how you can use Awszone.com to help you formulate the right Amazon.com ECS query to get the information you want to find. I figured out that I wanted to use the `ListLookup` query by using this:

```
http://www.awszone.com/scratchpads/aws/ecs.us/ListLookup.aws
```

Furthermore, I am using a `ListType=WishList` and the `ListID=1U5EXVPVS3WP5`. The URL for web interface to an Amazon.com wishlist is this:

```
http://www.amazon.com/gp/registry/wishlist/{ListID}/
```

Substituting your own `AccessKeyId`, you can get information about the list as a whole:

```
http://ecs.amazonaws.com/onca/xml?Service=AWSECommerceService&Version=2007-05-
14&~CCC
AWSAccessKeyId=[YourAccessKeyID]&Operation=ListLookup&ListType=WishList&~CCC
ListId=1U5EXVPVS3WP5http://ecs.amazonaws.com/onca/xml?Service=AWSECommerceServic
e&~CCC
Version=2007-10-
29&AWSAccessKeyId=[YourAccessKeyID]&Operation=ListLookup&ListType=~CCC
WishList&ListId=1U5EXVPVS3WP5
```

To get a page of the individual items, use the following URL:

```
http://ecs.amazonaws.com/onca/xml?Service=AWSECommerceService&Version=2007-10-
29&~CCC
AWSAccessKeyId=[YourAccessKeyID]&Operation=ListLookup&ListType=WishList&ListId=1
U5~CCC
EXVPVS3WP5&ResponseGroup=ListItems,Medium&ProductPage=2
```

Python Code to Mash Up Amazon.com and Google Spreadsheets

Now you can stitch all of this together with the following code, called `amazonWishListtoGSheet.py`. (Remember to substitute your own parameters into this code.)

```
"""
an example to copy over a public Amazon wishlist to a Google Spreadsheet
owned by the user based on code at
http://code.google.com/apis/spreadsheets/developers_guide_python.html
"""

GoogleUser = "[GoogleUSER]"
```

```
GooglePW = "[GooglePASSWORD]"
GSheet_KEY = "[GSheetID]"
# GSheet_KEY = "o06341737111865728099.3585145106901556666"
GWrkSh_ID = "[worksheetID]"
#GWrkSh_ID = "od6"

AMAZON_LIST_ID = "[LIST_ID_FOR_WISHLIST]"
# AMAZON_LIST_ID = "1U5EXVPVS3WP5"
AMAZON_ACCESS_KEY_ID = "[AMAZON_KEY]"

from xml.dom import minidom

import gdata.spreadsheet.service

def getText(nodelist):
    """
    convenience function to return all the text in an array of nodes
    """
    rc = ""
    for node in nodelist:
        if node.nodeType == node.TEXT_NODE:
            rc = rc + node.data
    return rc

# a sample row for testing the insertion of a row into the spreadsheet~CCC
GS_Example_Row = {'asin': '1590598385', 'dateadded': '6/5/2007',
'detailpageurl': ~CCC
'http://www.amazon.com/gp/product/1590598385/ref=wl_it_dp/103-8266902-5986239?
~CCC
ie=UTF8&coliid=I1AOWT8LH796DN&colid=1U5EXVPVS3WP5', 'author': 'Joel Spolsky',
~CCC
'quantitydesired': '1', 'price': '13.25', 'title': "Smart and Gets Things Done:
Joel~CCC
Spolsky's Concise Guide to Finding the Best Technical Talent (Hardcover) "}

GS_HEADER = ['ASIN', 'DetailPageURL', 'Title', 'Author', 'Date Added', 'Price',
~CCC
'Quantity Desired']

GS_KEYS = ['asin', 'detailpageurl', 'title', 'author', 'dateadded', 'price',
~CCC
'quantitydesired']

class GSheetForAmazonList:
    def __init__(self,user=GoogleUser,pwd=GooglePW):
        gd_client = gdata.spreadsheet.service.SpreadsheetsService()
        gd_client.email = user
        gd_client.password = pwd
        gd_client.source = 'amazonListToGsheet.py'
        gd_client.ProgrammaticLogin()
        self.gd_client = gd_client
    def setKey(self,key):
        self.key = key
    def setWkshtId(self,wksht_id):
```

```
        self.wksht_id = wksht_id
    def listSpreadsheets(self):
        """
        return a list with information about the spreadsheets available to the
user
        """
        sheets = self.gd_client.GetSpreadsheetsFeed()
        return map(lambda e: (e.title.text , e.id.text.rsplit('/', 1)[1]), ~CCC
sheets.entry)
    def listWorkSheets(self):
        wks = self.gd_client.GetWorksheetsFeed(key=self.key)
        return map(lambda e: (e.title.text , e.id.text.rsplit('/',
1)[1]),wks.entry)
    def getRows(self):
        return
self.gd_client.GetListFeed(key=self.key,wksht_id=self.wksht_id).entry
    def insertRow(self,row_data):
        return
self.gd_client.InsertRow(row_data,key=self.key,wksht_id=self.wksht_id)
    def deleteRow(self,entry):
        return self.gd_client.DeleteRow(entry)
    def deleteAllRows(self):
        entrylist = self.getRows()
        i = 0
        for entry in entrylist:
            self.deleteRow(entry)
            i += 1
            print "deleted row ", i

class amazonWishList:

# we can use Python and WSDL
# http://webservices.amazon.com/AWSECommerceService/AWSECommerceService.wsdl?

# I've been wondering how to introspect using WSDL -- Mark Pilgrim has some
answers:
# http://www.diveintopython.org/soap_web_services/introspection.html
# well -- the introspection of the input parameters doesn't seem to yield the
useful
# stuff. I was hoping for more info

    def
__init__(self,listID=AMAZON_LIST_ID,amazonAccessKeyId=AMAZON_ACCESS_KEY_ID):
        self.listID = listID
        self.amazonAccessKeyId = amazonAccessKeyId
        self.getListInfo()

    def getListInfo(self):

        aws_url =
"http://ecs.amazonaws.com/onca/xml?Service=AWSECommerceService&~CCC
Version=2007-10-
29&AWSAccessKeyId=%s&Operation=ListLookup&ListType=WishList&ListId~CCC
=%s" % (self.amazonAccessKeyId, self.listID)
```

```
        import urllib
        f = urllib.urlopen(aws_url)
        dom = minidom.parse(f)
        self.title = getText(dom.getElementsByTagName('ListName')[0].childNodes)
        self.listLength = int(getText(dom.getElementsByTagName('TotalItems')[0].
~CCC
childNodes))
        self.TotalPages = int(getText(dom.getElementsByTagName('TotalPages')[0].
~CCC
childNodes))
        return(self.title, self.listLength, self.TotalPages)

    def ListItems(self):
        """
        a generator for the items on the Amazon list
        """

        import itertools
        for pageNum in xrange(1,self.TotalPages):
            aws_url =
"http://ecs.amazonaws.com/onca/xml?Service=AWSECommerce~CCC
Service&Version=2007-10-
29&AWSAccessKeyId=%s&Operation=ListLookup&ListType=Wish~CCC
List&ListId=%s&ResponseGroup=ListItems,Medium&ProductPage=%s" %
(self.amazon~CCC
AccessKeyId,self.listID,pageNum)
            import urllib
            f = urllib.urlopen(aws_url)
            dom = minidom.parse(f)
            f.close()
            items = dom.getElementsByTagName('ListItem')
            for c in xrange(0,10):
                yield items[c]

    def parseListItem(self,item):
        from string import join
        from decimal import Decimal

        itemDict = {}

        itemDict['asin'] =
getText(item.getElementsByTagName('ASIN')[0].childNodes)
        itemDict['dateadded'] =
getText(item.getElementsByTagName('DateAdded')[0]. ~CCC
childNodes)
        itemDict['detailpageurl'] = getText(item.getElementsByTagName(~CCC
'DetailPageURL')[0].childNodes)

        # join the text of all the author nodes, if they exist
        authorNodes = item.getElementsByTagName('Author')
        # blank not allowed
        itemDict['author'] = join(map(lambda e: getText(e.childNodes), ~CCC
authorNodes), ", ") or ' '
```

```
        itemDict['quantitydesired'] = getText(item.getElementsByTagName(~CCC
'QuantityDesired')[0].childNodes)

        titleNodes = item.getElementsByTagName('Title')
        # blank title not allowed
        itemDict['title'] = join(map(lambda e: getText(e.childNodes), ~CCC
titleNodes), ", ") or ' '

        # to fix -- not all things have a LowestNewPrice
        itemDict['price'] = str(Decimal(getText(item.getElementsByTagName(~CCC
'LowestNewPrice')[0].getElementsByTagName('Amount')[0].childNodes))/100) or ' '

        return itemDict

def main():

    gs = GSheetForAmazonList(user=GoogleUser,pwd=GooglePW)
    gs.setKey(GSheet_KEY)
    gs.setWkshtId(GWrkSh_ID)

    aWishList = amazonWishList(listID=AMAZON_LIST_ID,amazonAccessKeyId=~CCC
AMAZON_ACCESS_KEY_ID)
    items = aWishList.ListItems()
    print "deleting all rows..."
    gs.deleteAllRows()
    for item in items:
        try:
            h = aWishList.parseListItem(item)
            print h['asin']
        except Exception, e:
            print "Error %s parsing %s" % (e, item.toprettyxml("  "))
        try:
            gs.insertRow(h)
        except Exception, e:
            print "Error %s inserting %s" % (e, h['asin'])

if __name__ == '__main__':
    main()
```

Here are some things to note about this code:

* The `GSheetForAmazonList` class provides convenience methods for the Google GData library.
* The error handling is essential since not all wishlist items necessarily have all the pieces of data requested. It's important for the code to keep moving even if data is missing.
* At least in the Python GData interface to Google Spreadsheets, you can't insert blank cells.

* `amazonWishList.ListItems` is a Python generator, which creates an iterator to parcel out the Amazon items one at a time. See `http://www.ibm.com/developerworks/library/l-pycon.html?t=gr,lnxw16=PyIntro` for a tutorial on Python generators.
* You can speed up the operation of this code through batch operations (`http://code.google.com/apis/gdata/batch.html`), which are currently supported in the GData interface and in the Java API kit (but not Python).

Zend PHP API Kit for Google Spreadsheets

In this section, I'll show you how to use the PHP API kit for Google Spreadsheets to read the contents of the Google Spreadsheets that we'll generate in the previous mashup. You can download the Zend Framework from here:

`http://framework.zend.com/`

and read about how to use the Zend Framework to access Google Spreadsheet here:

`http://framework.zend.com/manual/en/zend.gdata.spreadsheets.html`

and here:

`http://code.google.com/apis/spreadsheets/developers_guide_php.html`

The following code first lists your spreadsheets and then the rows and named columns of the spreadsheet containing items from your Amazon.com wishlist:

```
<?php

# user and password for google spreadsheet
$user = "[GoogleUSER]";
$pass = "[GooglePASSWORD]";

# set parameters for your version of "My Amazon WishList" Google Spreadsheet
$GSheetID = "[GSheetID]";
$worksheetID="[worksheetID]";
#$GSheetID = "o06341737111865728099.3585145106901556666";
#$worksheetID="od6";

# list entries from a spreadsheet

require_once('Zend/Loader.php');
Zend_Loader::loadClass('Zend_Gdata');
Zend_Loader::loadClass('Zend_Gdata_ClientLogin');
Zend_Loader::loadClass('Zend_Gdata_Spreadsheets');
Zend_Loader::loadClass('Zend_Http_Client');

$service = Zend_Gdata_Spreadsheets::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$spreadsheetService = new Zend_Gdata_Spreadsheets($client);

#  the following printFeed shows how to parse various types of feeds
#  coming from Google Spreadsheets API
#  function is extracted from
#  http://code.google.com/apis/spreadsheets/developers_guide_php.html
```

```
function printFeed($feed)
{
  $i = 0;
  foreach($feed->entries as $entry) {
    if ($entry instanceof Zend_Gdata_Spreadsheets_CellEntry) {
      print $entry->title->text .' '. $entry->content->text . "\n";
    } else if ($entry instanceof Zend_Gdata_Spreadsheets_ListEntry) {
      print $i .' '. $entry->title->text .' '. $entry->content->text . "\n";
    } else {
      print $i .' '. $entry->title->text . "\n";
    }
    $i++;
  }
}

# figuring out how to print rows

function printWorksheetFeed($feed)
{
  $i = 2;  # the first row of content is row 2
  foreach($feed->entries as $row) {
    print "Row " . $i .' '. $row->title->text . "\t";
    $i++;
    $rowData = $row->getCustom();
    foreach($rowData as $customEntry) {
      print $customEntry->getColumnName() . " = " . $customEntry->getText().
"\t";
    }
    print "\n";
  }
}

# first print a list of your Google Spreadsheets

$feed = $spreadsheetService->getSpreadsheetFeed();
printFeed($feed);

# Print the content of a specific Spreadsheet/Worksheet
# set a query to return a worksheet and print the contents of the worksheet

$query = new Zend_Gdata_Spreadsheets_ListQuery();
$query->setSpreadsheetKey($GSheetID);
$query->setWorksheetId($worksheetID);
$listFeed = $spreadsheetService->getListFeed($query);
printWorksheetFeed($listFeed);

?>
```

A Final Variation: Amazon Wishlist to Microsoft Excel via COM

With code to access the Amazon.com wishlist in hand, you can use COM programming to generate an Excel spreadsheet with the same information. To learn more about the details about how to do so, consult Chapter 12 of *Python Programming on Win32*.[23]

```
from amazonListToGSheet import GS_HEADER, amazonWishList, AMAZON_LIST_ID, ~CCC
AMAZON_ACCESS_KEY_ID, GS_KEYS
from win32com.client import Dispatch

# fire up the Excel application
xlApp = Dispatch("Excel.Application")
xlApp.Visible = 1
xlApp.Workbooks.Add()

# write the headers
col = 1

def insertRow(sheet,row,data,keys):
    col = 1
    for k in keys:
        sheet.Cells(row,col).Value = data[k]
        col += 1

for h in GS_HEADER:
    xlApp.ActiveSheet.Cells(1,col).Value = h
    col +=1
# now loop through the amazon wishlist

aWishList =
amazonWishList(listID=AMAZON_LIST_ID,amazonAccessKeyId=AMAZON_ACCESS_KEY_ID)
items = aWishList.ListItems()

row = 2
for item in items:
    try:
        p = aWishList.parseListItem(item)
        print p['asin']
    except Exception, e:
        print "Error %s parsing %s" % (e, item.toprettyxml("  "))
    try:
        insertRow(xlApp.ActiveSheet,row,p,GS_KEYS)
        row += 1
    except Exception, e:
        print "Error %s inserting %s" % (e, p['asin'])
```

23. http://www.oreilly.com/catalog/pythonwin32/chapter/ch12.html

Zoho APIs

Zoho (http://zoho.com) has been generating a good amount of attention for its online office suite, which is the most comprehensive one available right now.[24] Among its

offerings are Zoho Writer (a word processor), Zoho Sheet (a spreadsheet), and Zoho Show.

There are currently APIs for Writer, Sheet, and Show:

```
http://writer.zoho.com/public/help/zohoapi/fullpage
```

Currently, the APIs do not talk deeply to pieces of documents. There are storage APIs that let you upload and download documents. To access pieces, you have to use the techniques shown in the rest of this chapter to parse and write documents (for instance: `http://writer.zoho.com/public/help/userView.AddWorkbook/noband`). I suspect that API calls to access parts of documents will follow at some point in the future.

Summary

This chapter focused on two aspects of desktop and web-based office suites. First, you examined two major XML-based document formats: OpenDocument format (used in such applications as OpenOffice.org) and Open Office XML (used by Microsoft Office 2007). Then, you studied how to figure out how to generate some basic ODF text documents (through writing ODF directly and through using API kits) and how to generate a rudimentary OOXML document. You can apply the techniques shown here to deepen your understanding of ODF and Office Open XML.

Second, you studied a major instance of a programmable online office application: Google Spreadsheets. You studied how to use the Python API kit to create a Google spreadsheet from an Amazon.com wishlist using the APIs of their respective services. You also looked briefly at how to use the PHP API kit.

24. http://www.technologyreview.com/Biztech/18816/ provides a useful analysis of the Zoho vs. Google Docs battle and how they compare.