



Professional **PHP4 Web Development Solutions**

Raj Kumar Dash, Bryan Waters, Alison Gianotto, Jo Henrik Endrerud,
Matt Anton, Jon Stephens, Daniel Solin



Wrox technical support at: support@wrox.com

Updates and source code at: www.wrox.com

Peer discussion at: p2p.wrox.com

Summary of Contents

Chapter 1:	Introduction	1
Chapter 2:	Association Directory	13
Chapter 3:	Book of Mankind Project	85
Chapter 4:	Store Directory	149
Chapter 5:	Paranormal News Service	191
Chapter 6:	Web Corpus	239
Chapter 7:	Mission Control Wireless Job Board	263
Chapter 8:	Classified Ads Board	307
Chapter 9:	A Simple Content Management System	363
Chapter 10:	Advanced Content Management System	403
Chapter 11:	Simple Search Engine	453
Chapter 12:	Wireless MyStuff Server	495
Chapter 13:	Genealogy Server	527
Chapter 14:	Building Portal Sites	579
Index		603

9

A Simple Content Management System

This case study explores a very basic **Content Management System (CMS)** designed for internal use by a team of technically capable contributors. For those who are looking for a CMS that can be used by administrators and non-technical contributors, it would be advisable to go through this case study first and then proceed to Chapter 10, which deals with an advanced Content Management System (CMS).

In the course of this chapter we will look at:

- ❑ **Specifications**
To begin with, we need to define the features of the CMS.
- ❑ **Designing the Data**
The CMS will be based on a MySQL database. This section describes how the tables in this database are structured.
- ❑ **Managing the Data**
We need some PHP code that manages the data in the database.
- ❑ **Possible Enhancements**
The most obvious enhancements to the core application are presented at the end of the chapter.

This chapter discusses a very simple CMS in which the content of a document takes on several forms while using a single database. This is accomplished by using XML to define each document.

While all documents have the same skeleton set of XML tags, each document type differs in the tags used within the <body> element. This content markup is actually stored as is in the database. Each document type will require its own application component to parse and manipulate the document body.

Specifications

The types of documents that may be stored via this application are limitless, provided that they all follow the same XML skeleton (discussed in the next section). This necessitates the use of an XML authoring tool, whether manual or automated, like XML Spy (see <http://www.xmlspy.com/>) or oXygen (see <http://www.oxygenxml.com/>). The examples in this chapter provide a sample of what can be stored. However, because XML is used, the average application user requires some technical skills. Alternatively, a more user-friendly Graphical User Interface (GUI) can be wrapped around this application engine, but that is beyond the scope of this case study.

Note that the source documents do not necessarily represent web pages although they can. In fact, we could store HTML, WML, CML, SVG, or any other document in the database as well. In the case of WML, however, the `<?xml ... ?>` **Processing Instruction (PI)** does not get stored. It would have to be added by the application component that manipulates WML documents. The same goes for XML documents. The XML PI gets stripped off.

This chapter does not cover XSLT stylesheets. However, they can be used in various application components that manipulate the stored documents. Such stylesheets could be used to export document content in various formats including HTML, WML, RTF, PDF, and so on.

The list below highlights some of the functional specifications and features that can be a part of the application (not all of these items are covered in this chapter):

- ☐ **Private System**
Technically skilled users could use this system. There is no login feature in the present application, but this could be added later.
- ☐ **Variety of document content**
 - ☐ Web pages
 - ☐ Wireless web pages
 - ☐ General XML documents
 - ☐ Articles
 - ☐ Product manuals (print and online versions)
 - ☐ Indexes
 - ☐ Bibliography
 - ☐ Research notes
 - ☐ Non-fiction books
 - ☐ General lists

- ☐ Company records
- ☐ Recipes
- ☐ Letters
- ☐ Poems
- ☐ Short stories
- ☐ Fiction/novels
- ☐ **Variety of export facilities**
 - ☐ Text
 - ☐ XML
 - ☐ WML
 - ☐ HTML
 - ☐ RTF
 - ☐ PDF

As an application for storing a wide variety of XML-based document structures, export facilities are limited to pre-defined formats based on content type and subtype. The only facility for changing the appearance of finished documents is XSLT, which we do not discuss here.

On the surface, this application appears to be an awkward alternative to a word processing package. However, for users who want to store a wide range of documents in XML format, this is an ideal, albeit simplified, application. The advantage is that we now have an abstract model of a document to which we can add interfaces. Using these we can generically search for information regardless of the type of the stored document.

To aid in building a generic search engine, we can go a step further by building a document **corpus**. A corpus is a unique-word list used in the sampling of documents. By temporarily stripping off XML tags for each document, we can add any new unique words, which we encounter, to a web corpus.

A web corpus is the necessary set of database tables and their contents that represent the list of unique words and pointers to their occurrences in documents

For a simple web corpus, we simply log one occurrence of a word in a document. For example, if the word 'science' occurs one or more times in three different imported documents, we would catalog one occurrence in each document by creating three **occurrence** records. Each record would contain a reference to the actual word, as well as a reference to the document where it occurs.

Refer to Chapter 6 for more information about a web corpus and on how to build a search engine based on a web corpus.

A different type of search engine could be designed by parsing an imported document's body markup, recording tag, and attribute names. This would allow users to search for all documents containing a specific tag name or even all tags that have a particular attribute. This is particularly useful if we are designing parsing components for this application on a need-to basis and have recurring tag names that must have the same attributes.

For example, if we are storing company documents that are mostly about employees, some document types might have an <employee> tag with the following attributes:

```
<employee lname="mouse" empid="1234567890"/>
```

Storing occurrences of a tag name and its attribute names in a database gives us the ability to query for specific documents based on the inclusion of this tag and its attributes. In the case of the `<employee>` tag, we can query for a list of documents in which this tag occurs either with or without the attributes. This makes it easy to pinpoint markup errors, for instance in legacy documents that have been converted into XML through brute force methods. The type of search engine mentioned here is briefly touched upon towards the end of this chapter.

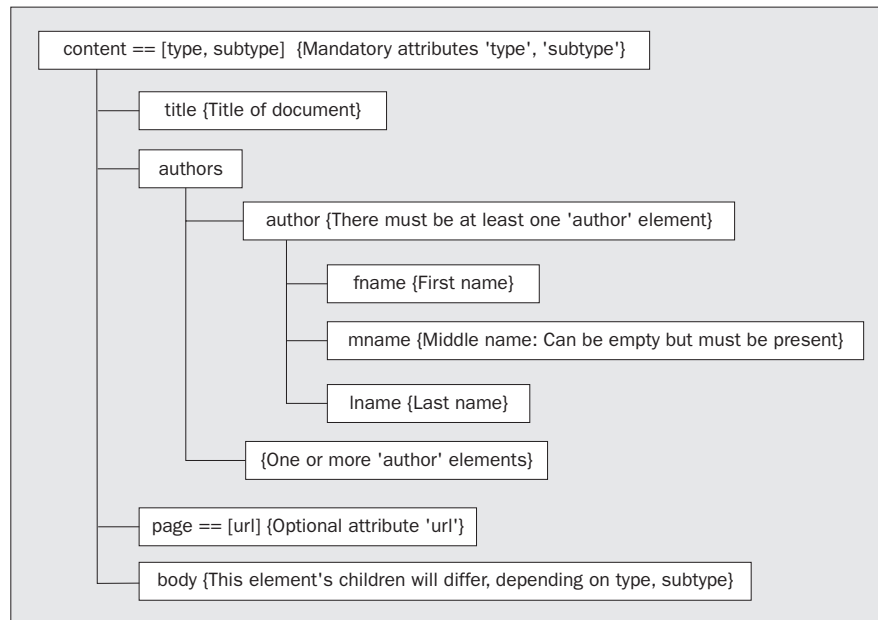
All content blocks are supplied using custom XML tagsets defined by the end user. Documents can be defined in an incomplete manner (provided they are still well-formed XML) and updated later (the application code for update is not covered here). The code that is covered in the rest of this chapter is the basic import code required to catalog each document. Numerous interfaces can be built over the top of the database that results from running the import code.

Several examples for different types of content are detailed after a discussion of the general structure required. As usual, we will use PEAR::DB for communicating with the MySQL database on which this whole application relies. By using PEAR::DB, we also makes it easy to change database server if that ever gets interesting.

Because we are only covering a batch mode document import component, there are no particular web page needs.

General Structure of XML Documents

The following listing shows the general tree structure of a content document for the simple CMS application:



Here is the DTD for the above:

```
<!DOCTYPE CONTENT[
  <!ELEMENT CONTENT (TITLE,AUTHORS,PAGE,BODY)>
  <!ATTLIST CONTENT TYPE CDATA #REQUIRED>
  <!ATTLIST CONTENT SUBTYPE CDATA #REQUIRED>
  <!ELEMENT TITLE (#PCDATA)>
  <!ELEMENT AUTHORS (FNAME,MNAME,LNAME)>
  <!ELEMENT FNAME (#PCDATA)>
  <!ELEMENT MNAME (#PCDATA)>
  <!ELEMENT LNAME (#PCDATA)>
  <!ELEMENT PAGE (#PCDATA)>
  <!-- ATTLIST PAGE URL CDATA #IMPLIED -->
  <!ELEMENT BODY (#PCDATA)>
]>
```

All documents imported into this simple CMS database must follow this skeleton structure; the documents will differ only in the <body> element's subtree structure. The equivalent XML template looks like this:

```
<content type="__type__" subtype="__subtype__">
  <title>__title__</title>
  <authors>
    <author>
      <fname>__firstname__</fname>
      <mname>__middlename__</mname>
      <lname>__lastname__</lname>
    </author>
  </authors>
  <page url="__url__"/> <!-- Attribute 'url' is optional -->
  <body>
    __body_of_content__
  </body>
</content>
```

Values for attributes and tags are shown as field markers, in bold. For example, the value of type is indicated by the marker **__type__**. When a document is added to the database, only the above tags will be parsed. Everything in the <body> element will be stored as a block of XML-like markup. The XML PI that starts every XML file will be stripped off.

The resulting <body> block will be parsed later by an application component that knows how to process the particular document type/subtype. That component is not talked about here though.

Such components are not discussed in this chapter.

Example 1: Non-fiction Book (Beginning Stages)

This example represents a non-fiction book. The following XML listing (nonfic01.xml) is broken up into sections in order to explain the tags:

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

Each document must start with the root element `<content>` and include two attributes – `type` and `subtype` (`subtype` can be a blank string). The resulting combination of attribute values will determine the structure of the sub-document within the `<body> . . . </body>` tag block.

```
<content type="nonfiction" subtype="book">
```

The combination of `type`, `subtype`, and the `title` must be unique. In other words, for a given `type/subtype` combination, all relevant documents must have a unique `title`:

```
<title>Web Development Methodologies</title>
```

Note, however, that titles can be repeated in a different `type/subtype` combination.

Each document may have one or more contributors. For each author, use one `<author> . . . </author>` block. This block defines two authors:

```
<authors>
  <author>
    <fname>mary</fname>
    <mname>elizabeth</mname>
    <lname>smith</lname>
  </author>
  <author>
    <fname>john</fname>
    <mname>q</mname>
    <lname>doe</lname>
  </author>
</authors>
```

Additional authors can be added later.

If we use an automated batch method for updates, we might use code similar to the following:

```
<?selectdoc type="nonfiction" subtype="book"
            title="Web Development Methodologies" ?>
<?addauthor ?>

<author>
  <fname>bernard</fname>
  <mname/>
  <lname>rubble</lname>
</author>
```

The XML PI `<?selectdoc . . . ?>` would be interpreted as a request to retrieve a particular document; the three attributes – `type`, `subtype`, and `title` – would jointly form a unique key. Once the document has been selected, the XML PI `<?addauthor ?>` would be interpreted as a request to add an `<author>...</author>` block to the `<authors>...</authors>` block of the document.

The document's URL is defined next:

```
<page url="test.doc"/>
<body>
```

Some documents will either not have a URL associated with them, or the URL may be presently undefined. When no URL is defined, use the empty form, `<page/>`.

To update the URL (either to define it or to change it), we might use the following code in batch mode:

```
<?selectdoc type="nonfiction" subtype="book"
           title="Pro PHP Web Site Solutions" ?>
<?changeurl pageurl="__newurl__" ?>
```

Finally, the `<body>` element is where the actual document structure is defined. Each unique combination of content type and subtype will likely have a different XML tagset. Further, each combination will need to have a custom XML parser built for it to allow manipulation of documents:

In this example, we are defining a non-fiction book. In particular, we are defining a subset of the information in the book that we are reading. For the first chapter, we are only defining the title, chapter number, and style (the style attribute defines a font) attributes. For now, we shall leave out the content in the chapter:

```
<chap num="1" title="Introduction" style="chapter title"/>
```

In the listing above, "chapter title" represents the font used for chapter titles, not the actual text that constitutes the title. For the second chapter, we shall define the chapter number, title, and style. We are also defining sections in the chapter (and their level, title, and style). However, we are not defining any chapter or section textual content. Notice that the sections are not numbered. Instead, we only define a level. For example, a level 1 section is equivalent to the font style Heading 1. A level 2 section is equivalent to a Heading 2, and is a sub-section of a level 1 section, and so on.

While the level attribute (`lev`) is not strictly necessary, it actually makes the parsing of the markup much easier.

The last of ten design goals in the XML specification (<http://www.w3.org/TR/2000/REC-xml-20001006>) states that, "Terseness of XML markup is of minimal importance". Keeping this in mind, the markup below adds explicit information that can be retrieved easily, to reduce document-parsing time:

```
<chap num="2" title="PHP Overview" style="chapter title">
  <sec lev="1" title="Introduction" style="Heading 1"/>
  <sec lev="1" title="PHP File Extensions" style="Heading 1"/>
  <sec lev="1" title="Comments" style="Heading 1"/>
  <sec lev="1" title="Identifiers + Variables" style="Heading 1"/>
  <sec lev="1" title="Variable Scope" style="Heading 1"/>
```

The above `<sec>` elements use the 'empty' form of tag, since no child elements are defined. The next section contains three level 2 sections, so we cannot use the 'empty' tag form for the parent element's open tag:

```

<sec lev="1" title="Data Types + Casting" style="Heading 1">
  <sec lev="2" title="Miscellaneous Type-Related Functions"
    style="Heading 2"/>
  <sec lev="2" title="Converting Data Types" style="Heading 2">
    <sec lev="3" title="Casting Data Types" style="Heading 3"/>
  </sec>
  <sec lev="2" title="Arithmetic on Strings" style="Heading 2"/>
</sec>

```

The remainder of the `<body>` block simply shows a few more examples of section definition:

```

<sec lev="1" title="Expressions" style="Heading 1">
  <sec lev="2" title="Constants" style="Heading 2">
    <sec lev="3" title="User-defined Constants" style="Heading 3"/>
  </sec>
  <sec lev="2" title="Operators" style="Heading 2">
    <sec lev="3" title="String Operators" style="Heading 3"/>
    <sec lev="3" title="Arithmetic Operators" style="Heading 3"/>
    <sec lev="3" title="Comparison Operators" style="Heading 3"/>
    <sec lev="3" title="Logical Operators" style="Heading 3"/>
    <sec lev="3" title="Reference Operators" style="Heading 3"/>
    <sec lev="3" title="Bitwise Operators" style="Heading 3"/>
    <sec lev="3" title="Assignment Operators" style="Heading 3"/>
    <sec lev="3" title="Miscellaneous Operators"
      style="Heading 3"/>
  </sec>
  <sec lev="2" title="Operator Precedences" style="Heading 2"/>
</sec>

<sec lev="1" title="Statements" style="Heading 1">
  <sec lev="2" title="Assignments" style="Heading 2"/>
  <sec lev="2" title="Input/Output" style="Heading 2">
    <sec lev="3" title="Output" style="Heading 3">
      <sec lev="4" title="echo" style="Heading 4"/>
      <sec lev="4" title="print()" style="Heading 4"/>
      <sec lev="4" title="printf()" style="Heading 4"/>
      <sec lev="4" title="sprintf()" style="Heading 4"/>
    </sec>
    <sec lev="3" title="Output" style="Heading 3"/>
  </sec>
</sec>

</chap>
</body>
</content>

```

Example 2: Online Manual

This example (`onlineman01.xml`) sets up a placeholder document for an online manual:

```

<?xml version="1.0" ?>
<content type="nonfiction" subtype="onlinemanual">
  <title>Guide to the TOOLS Language - Online Edition</title>
  <authors>
    <author>
      <fname>mookie</fname>

```

```

        <mname/>
        <lname>mouse</lname>
    </author>
</authors>
<page/>
<body>
</body>
</content>

```

Currently, there is no content in the `<body>...</body>` block. Content might be added later in batch mode as follows:

```

<?selectdoc type="nonfiction" subtype="onlinemanual" title="Guide to the TOOLS
Language - Online Edition" ?>

<?definebody ?>

<body>
  <toolsmanual>
    <functions>
      <function>
        <name>print()</name>
        <params>
          <param><i>stringexpr</i></param>
        </params>
      </function>
    </functions>
  </toolsmanual>
</body>

```

The application component that parses the above would select the appropriate document from the database and then add the segment of markup code. The component would have to have some scheme for adding functions to the online manual. One possibility, of course, is adding functions in alphabetical order.

Example 3: Geek Poetry

This example (`geekpoetry01.xml`) defines an arbitrary document of the type 'geek poetry':

```

<?xml version="1.0" ?>
<content type="poetry" subtype="geek">
  <title>ODE TO PHP</title>

  <authors>
    <author>
      <fname>golly</fname>
      <mname/>
      <lname>gee</lname>
    </author>
  </authors>

  <page/>

```

```
<body>

  <stanza>
    <line>$str = "How do I love PHP?</line>
    <line>Let me count the ways.</line>
  </stanza>

  <stanza>
    <line>I love thy dynamic typing,</line>
    <line>With options galore.</line>
    <line>Strongly typed languages</line>
    <line>Are such a bore.</line>
  </stanza>

  <stanza>
    <line>My data type mistakes</line>
    <line>Are my own to make</line>
    <line>That's an option</line>
    <line>That I choose to take.</line>
  </stanza>

  <stanza>
    <line>I love thy ease of use,</line>
    <line>And focus on the web.</line>
    <line>I love that I can weave</line>
    <line>Web apps quickly - thanks, Zeev.</line>
  </stanza>

  <stanza>
    <line>I love you PHP</line>
    <line>Because you're so simple.</line>
    <line>Changing you too much too soon</line>
    <line>Would cause such a ripple.</line>
  </stanza>

  <stanza>
    <line>I love you PHP</line>
    <line>And so does the "web"</line>
    <line>Seven million domains</line>
    <line>Just can't be wrong!</line>
    <line>"</line>
    <line>print $str;</line>
  </stanza>

</body>
</content>
```

Example 4: Recipes

This example (`recipes01.xml`) defines a document that will contain one or more recipes. This sample document shows one recipe:

```
<?xml version="1.0" ?>
<content type="nonfiction" subtype="recipes">
  <title>Guide to Spicy Cooking - Online Edition</title>

  <authors>
    <author>
      <fname>mookie</fname>
      <mname/>
      <lname>mouse</lname>
    </author>
  </authors>

  <page/>

  <body>
    <recipes>
      <recipe title="Savoury Curry Minced Lamb" categ="">

        <ingredients>
          <item num="1">
            <ingred>Olive oil</ingred>
            <quantity units="tbsp">2</quantity>
            <notes></notes>
          </item>

          <item num="2">
            <ingred>Curry powder</ingred>
            <quantity units="tbsp">2-3</quantity>
            <notes></notes>
          </item>

          <item num="3">
            <ingred>Haldi (turmeric)</ingred>
            <quantity units="tbsp">2</quantity>
            <notes></notes>
          </item>

          <item num="4">
            <ingred>Ginger powder</ingred>
            <quantity units="tbsp">1</quantity>
            <notes></notes>
          </item>

          <item num="5">
            <ingred>Ground lamb, thawed</ingred>
            <quantity units="oz">16</quantity>
            <notes>(or subst. beef, chicken or turkey)</notes>
          </item>

          <item num="6">
            <ingred>Mushrooms, sliced</ingred>
            <quantity units="">3 med</quantity>
            <notes></notes>
          </item>
        </ingredients>
      </recipe>
    </recipes>
  </body>
</content>
```

```
<item num="7">
  <ingred>Large calamata olives, quartered</ingred>
  <quantity units="">4-6</quantity>
  <notes>optional</notes>
</item>

<item num="8">
  <ingred>Scallions (green onion)</ingred>
  <quantity units="">5-6</quantity>
  <notes>sliced - white portion only</notes>
</item>

<item num="9">
  <ingred>Tomato sauce</ingred>
  <quantity units="cups">1/2</quantity>
  <notes>Dilute with 1/2 cup water or substitute with
    1 cup of tomato or vegetable juice.</notes>
</item>

<item num="10">
  <ingred>Cooked rice</ingred>
  <quantity units="cups">4</quantity>
  <notes>or one cup per person</notes>
</item>

</ingredients>

<instructions>
  <step num="1">Heat olive oil in skillet at med-high (8)
    heat.</step>

  <step num="2">Add dry spices (curry powder, turmeric, ginger
    powder) and roast for less than 30 seconds. Keep
    an eye out, as these spices will burn fairly
    quickly.</step>

  <step num="3">Add thawed or fresh ground meat and break up with
    spatula or chopsticks.</step>

  <step num="4">When meat has nearly browned, add sliced
    mushrooms. Toss with a spatula.</step>

  <step num="5">If desired, add quartered olives now.</step>

  <step num="6">Add tomato sauce and water, or tomato juice or
    vegetable juice. Simmer for 10 minutes.</step>

  <step num="7">Remove from stove; add sliced scallions.</step>
</instructions>

<serving>
<suggestion>
```

```

        Serve on steamed rice, with slices of mango or watermelon.
      </suggestion>
      <serves>Serves 4-6 as a main dish, 8-10 as an appetizer.
    </serves>
  </serving>

</recipe>
</recipes>
</body>
</content>

```

More recipes can be added later using something like the following:

```

<?selectdoc type="nonfiction" subtype="recipes" title="Guide to Spicy Cooking -
Online Edition" ?>
<recipe title="Peace and Harmony">
...
</recipe>

```

The application component that handles recipes would have some scheme for adding `<recipe>...</recipe>` blocks to the current document, for example, alphabetically by title or by category of the dish, if such information is included in each recipe. However, at this point, we could order the output in any way we want.

Example 5: Fiction – Short Story Fragment

This example (`shortstory01.xml`) shows a fragment of a short story:

```

<?xml version="1.0" ?>
<content type="fiction" subtype="short">
  <title>The Great Lake</title>

  <authors>
    <author>
      <fname>aeric</fname>
      <mname>sir stinkybutt</mname>
      <lname>halfacat</lname>
    </author>
  </authors>
  <page url="/stories/fiction/short/thegreatlake.html"/>

  <body>
    <p>The small rowboat rocked violently from the wake of the large
      speedboat that passed. Brent waved with one hand, the other still
      holding his fishing rod. Joe harumpphed. \ "Damn idiots. No
      respect for others.\ "</p>

    <p>\ "What now, Dad?\ " Brent asked.</p>

    <p>\ "Did you see how close they came, son?\ " Joe responded. \ "They were
      too close by at least 15 feet.\ "</p>

    <p>\ "Yeah, I guess you're right,\ " Brent agreed. \ "But what can we do

```

```

        about it?"</p>

<p>"Do? We can't do nothing, boy. If the harbor patrol wasn't so damn
    busy making sure the lake was full of water, they'd be paying
    attention to stuff like this. But we can't do nothing. Not one
    thing.\" A distant look came into Joe's eyes.</p>

<p>"Dad?! What are you up to now?" Brent asked. \"I know that look,
    Dad. You're thinking of some form of revenge.\"</p>

<p>"Nothing that shouldn't be done, son. Did you happen to catch the
    name on that boat? Never mind, I got it. Give me that pen over
    there...\" Just then, Joe's fishing line gave a mighty tug. \"Whoa. I
    think I caught a big one!\" Joe reeled in his line as fast as he
    could, fighting the tugging, only to discover that he'd bagged an
    old tire. \"Just what the hell would a car tire be doing in the
    middle of the lake? I never understand people. It's bad enough
    that they pollute. But some seem to go to great lengths to do
    so.\"</p>

<?timechange ?>

<p>Joe and Brent made their way in to shore. Brent jumped out and
    tied up the boat to their small berth, then helped his father to
    unload the gear. \"Wait a minute, son. I want to talk to the harbor
    patrol.\"</p>

<p>Brent stood with the gear while Joe made his way down the docks to
    the little boathouse office that the harbor patrol maintained. Joe
    had a way with people, despite his general crotchety composure,
    which made people laugh and agree to more things than they might
    normally do. In the distance, he could see the on-duty
    harbormaster <?check term?> laughing away with his father.
    Whatever Joe had said, the harbormaster had given him a salute and
    re-entered the boathouse.</p>

<p>"I know you're up to something, Dad,\" Brent said when his father
    returned.\"</p>
</body>
</content>

```

Note that this is not necessarily the best way to represent the content of a short story. To allow for later update in batch mode, the appropriate `<?processinginstruction ?>` PIs would have to be defined. However, such instructions would need a means to reference paragraphs for adding, editing, or deleting. One possibility is to change the tagset slightly. For example, we could number the paragraphs by adding a `num` attribute to each `<p>` tag. This would provide the short story application component a means of numbering paragraphs. Then we could insert several paragraphs, after the first one above, as follows:

```

<?selectdoc type="fiction" subtype="short" title="The Great Lake" ?>
<?insertpara after="1" ?>
<p num="1.1">...</p>
<p num="1.2">...</p>
<p num="1.3">...</p>
<p num="1.4">...</p>

```

This is simply one possible method. The database is a very simple and open starting point for building a variety of content management systems using XML.

Example 6: Resume

This example (resume01.xml) defines the XML skeleton for a resume:

```
<?xml version="1.0" ?>
<content type="nonfiction" subtype="resume">
  <title>RESUME: Bam Bam Rubble</title>

  <authors>
    <author>
      <fname>bam</fname>
      <mname>bam</mname>
      <lname>rubble</lname>
    </author>
  </authors>

  <page/>

  <body>

    <dateofbirth>
      <year>2000 AD</year>
      <month>May</month>
      <day>29</day>
    </dateofbirth>

    <education>
      <school>kindergartrock</school>
      <training>
        <program>
        </program>
      </training>
    </education>

    <experience>
    </experience>

    <hobbies>
      <hobby>Breaking rocks</hobby>
    </hobbies>

    <references>
      <reference>Frederick Flintstone</reference>
      <reference>Mr. Slate</reference>
    </references>

  </body>
</content>
```

A set of PIs will have to be defined for manipulating a resume document. The instruction targets (names) are arbitrary, but necessary functionality includes the update of most of the tags used above.

The above examples mainly show the starting points of various types of documents. The document types that can be stored are limitless, provided they are marked up in XML.

Each type of document, after import, requires an application component that allows for batch or live update of documents. Each component must be coded around the implicit needs of the tagset used for the document type. Each component, out of necessity, must have some form of XML parsing code. A parser could be defined using the PHP XML/Expat functions or from scratch. Each parser would have to know how to interpret PIs and then manipulate the stored document as necessary. This could be done through XQuery (XML Query).

Unfortunately, at the time of writing, PHP does not support this XML technology. One alternative is to write the necessary code from scratch. Another alternative is to send the relevant XML markup and the processing request to some other scripting/programming language that does support XQuery. Alternatively, one can use XPath. PHP has some functions for using XPath in the DOM XML module. (This would have to be done through system calls, as PHP's pipe/socket functions are still in an experimental stage.)

The bulk of the PHP code in this case study focuses only on the import of XML documents. Additional application components for manipulating the content require more space than we have here for discussion.

The types of users for this application would be technically-skilled contributors. However, given the right interface, less technical contributors might also use the database to store and update content. All content stored in the database will be either plain text or XML-like markup.

Designing the Database

The database required for storing a wide variety of XML documents is relatively simple.

Every document stored by this application contains certain common information:

- ☐ Content Type
- ☐ Content Subtype
- ☐ Title
- ☐ Author(s)
 - ☐ First name
 - ☐ Middle name, if any
 - ☐ Last name
- ☐ Virtual URL associated with the document, if any
- ☐ Body of the document (XML-like markup)

Any differences in the document structure are retained in an XML markup form.

Database `simplecms`

The above information can be represented in two tables, for a non-normalized database:

Table Name	Alias	Description
<code>simplecms_contents</code>	<code>co</code>	This table stores meta-information about each document, as well as the actual content in pseudo-XML form
<code>simplecms_authors</code>	<code>au</code>	This table stores information about the author of each document

To normalize the database, we need two lookup tables:

Table Name	Alias	Description
<code>simplecms_doctypes</code>	<code>dt</code>	This table stores combinations of type and subtype. It could also store an XML DTD, which could be used to validate an imported XML document claiming to be of a certain type and subtype. <i>This is left as an exercise for the reader.</i>
<code>simplecms_authordocs</code>	<code>ad</code>	This table stores reference information about author(s) who contributed to a particular document.

Database Table Specifics

The database tables discussed above are described in more detail in the sections below.

Doctypes

The `simplecms_doctypes` lookup table stores a list of document type/subtype combinations:

Column Name	Data Type	Description
<code>doctype_typeid</code>	INT	Unique integer, generated by MySQL that identifies this type/subtype combination.
<code>doctype_type</code>	CHAR	Type of document, for instance, 'nonfiction', 'fiction', 'poetry'. If we knew what document types we were planning to store in the database, we could make an ENUM of this column. This would increase performance quite a bit.

Column Name	Data Type	Description
doctype_subtype	CHAR	Subtype of document, for instance, short, onlinemanual, geek, book. As for previous column, we could make an ENUM of this if we had a predefined number of subtypes. Note that type and subtype values are rather arbitrary. What matters is that for each unique combination of type and subtype, a different tagset should be used within the <body>...</body>. Adding a DTD column to this table could enforce the correct document structure. This column, for a particular type/subtype combination would store an appropriate XML DTD. This is left as an exercise for the reader.

If type/subtype combinations are known beforehand, the `simplecms_doctypes` table can be pre-populated before document import. Otherwise, a record is inserted each time a combination that is not stored in the database is encountered.

We could further normalize the table by adding two separate lookup tables – one each of type and subtype information. However, this is unnecessary and complicates the queries.

Authors

The `simplecms_authors` lookup table stores information about document authors. This table could very well be expanded at a later stage. If the table gets expanded, this must also be reflected in the XML markup.

Column Name	Data Type	Description
author_authorid	INT	Unique integer generated by MySQL that identifies this author
author_firstname	CHAR	First name of this author
author_middlename	CHAR	Middle name of this author, if any
author_lastname	CHAR	Last name of this author

As with the `simplecms_doctypes` table, the table can be pre-populated if some or all potential authors are known beforehand. Otherwise authors are added once to this table as they are encountered.

Contents

The `simplecms_contents` table stores most of the information extracted from each imported XML document. One record is used for each document. The author information is stored in the `simplecms_authors` table:

Column Name	Data Type	Description
content_contentid	INT	A unique integer, generated by MySQL that identifies this record.
content_title	CHAR	Title of the document. This field can be empty.
content_pageurl	CHAR	A URL associated with this document. This field can be empty.
content_body	CHAR	The XML content extracted from between the <body>...</body> block, that is, the <block> and </block> tags are not included. This field can be empty.
content_typeid	INT	This is a foreign key that points to the appropriate record in the simplecms_doctypes table, and thus indirectly identifies the type/subtype of this document.

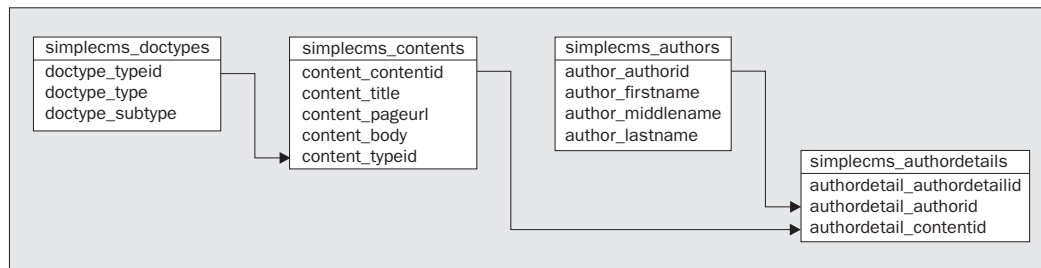
Authordocs

The simplecms_authordocs table stores reference information indicating which authors contributed to a particular document. For each document, one record is added to this table for each author:

Column Name	Data Type	Description
authordoc_id	INT	A unique integer, generated by MySQL that identifies this record.
authordocs_authorid	INT	This is a foreign key that points to the appropriate record in the simplecms_authors table, and thus indirectly identifies an author of the document pointed to by authordocs_contentid.
authordocs_contentid	INT	This is a foreign key that identifies the document for which the author pointed to by authordetail_authorid is a contributor for. This column points to the appropriate record in the simplecms_contents table.

Relationships Between Tables

The diagram below shows the relationship between the tables in the simplecms database:



Database User

First create a MySQL database for this application:

```
> mysqladmin create simplecms
```

The tables for this case study can all be placed within a single MySQL database, `simplecms`. We only need one database user, say `ADMIN`. We will have to grant privileges to the tables, as well as create the tables. Once the database is created, we use the `mysql` client utility, which comes with MySQL, in the following command line to grant privileges:

```
mysql> GRANT SELECT, INSERT, UPDATE, DELETE ON simplecms.*  
      TO 'ADMIN@localhost' IDENTIFIED BY "ADMIN";
```

Creating Tables

Run the following SQL script, `crdbs_simplecms.sql`, on the command line to create the `simplecms` database and tables:

```
> mysql simplecms < crdbs_simplecms.sql
```

```
# FILE: crdbs_simplecms.sql  
# USE:  create tables for database simplecms  
  
CREATE TABLE simplecms_doctypes (  
    doctype_typeid INT UNSIGNED NOT NULL PRIMARY KEY AUTO_INCREMENT,  
    doctype_type   VARCHAR(28) NOT NULL,  
    doctype_subtype VARCHAR(28) NOT NULL  
);  
  
CREATE TABLE simplecms_authors (  
    author_authorid INT UNSIGNED NOT NULL PRIMARY KEY AUTO_INCREMENT,  
    author_firstname VARCHAR(40) NOT NULL,  
    author_middlename VARCHAR(40),  
    author_lastname  VARCHAR(40) NOT NULL  
);  
  
CREATE TABLE simplecms_contents (  
    content_contentid INT UNSIGNED NOT NULL PRIMARY KEY AUTO_INCREMENT,  
    content_title     VARCHAR(128) NOT NULL,  
    content_pageurl   VARCHAR(255),  
    content_body      MEDIUMTEXT NOT NULL,  
    content_typeid    INT UNSIGNED NOT NULL  
);  
  
CREATE TABLE simplecms_authordocs (  
    authordocs_authordid INT UNSIGNED NOT NULL PRIMARY KEY AUTO_INCREMENT,  
    authordocs_authorid  INT UNSIGNED NOT NULL,  
    authordocs_contentid INT UNSIGNED NOT NULL  
);
```

Building Indexes

Indexes will be created for most columns in the database, to speed up common searches. The SQL commands for creating indexes are in a file called `mkidx_simplecms.sql`.

Run the following SQL script, `mkidx_simplecms.sql`, on the command line to build the indexes for the `simplecms` database and tables. Note that MySQL creates indexes automatically on primary keys, so creating indexes on the ID-columns of our tables may seem a little redundant. However, they are included in the listing below for clarity reasons:

```
> mysql simplecms < mkidx_simplecms.sql
```

```
# FILE: mkidx_simplecms.sql
# USE: create indexes for tables in database simplecms

# Indexes for table simplecms_doctypes
#
CREATE INDEX dt_idxOnTypeid ON simplecms_doctypes(doctype_typeid);
CREATE INDEX dt_idxOnType ON simplecms_doctypes(doctype_type);
CREATE INDEX dt_idxOnSubtype ON simplecms_doctypes(doctype_subtype);

# Indexes for table authors
#
CREATE INDEX au_idxOnAuthid ON simplecms_authors(author_authoid);
CREATE INDEX au_idxOnFname ON simplecms_authors(author_firstname);
CREATE INDEX au_idxOnMname ON simplecms_authors(author_middlename);
CREATE INDEX au_idxOnLname ON simplecms_authors(author_lastname);

# Indexes for table simplecms_contents
# (No index for column 'body', because of MySQL indexing limitations for
# type 'text' and its variations)
#
CREATE INDEX co_idxOnContid ON simplecms_contents(content_contentid);
CREATE INDEX co_idxOnTitle ON simplecms_contents(content_title);
CREATE INDEX co_idxOnPageurl ON simplecms_contents(content_pageurl);
CREATE INDEX co_idxOnTypeid ON simplecms_contents(content_typeid);

# Indexes for table authordetails
#
CREATE INDEX ad_idxOnAuthdetid
ON simplecms_authordetails(authordetail_authordetailid);
CREATE INDEX ad_idxOnAuthid
ON simplecms_authordetails(authordetail_authoid);
CREATE INDEX ad_idxOnContid
ON simplecms_authordetails(authordetail_contentid);
```

Managing the Data

Let's now take a look at the import component of the simple CMS application. It is this component that we are focusing on in the following sections.

Populating the Database

To initially populate the database, we need to import some of the XML-based documents discussed in the previous examples. These are the steps we need to take to populate the database:

1. Create database tables
2. Build table indexes
3. Import documents
 - a. Import `nonfic01.xml`
 - b. Import `onlineman01.xml`
 - c. Import `geekpoetry01.xml`
 - d. Import `recipes01.xml`
 - e. Import `shortstory01.xml`
 - f. Import `resume01.xml`

Refer to the sections *Creating Tables* and *Building Indexes* to perform steps 1 and 2. Next we will look into step 3.

Importing Documents

A single script that scans for the basic tagset and creates a database record can perform all of the import sub-steps in step 3. A point to be made clear here is that we are not using the PHP XML/Expat functions to parse the documents. The reason we do not is because we only want to parse a part of each document's tag blocks. We want to keep each `<body>...</body>` block intact. Since we cannot do this easily with the PHP XML functions, it is best that we write a relatively simple script that uses Perl-Compatible Regular Expressions (PCRE).

Algorithm

The following pseudo-code presents a general algorithm for importing the sample XML files:

```
1.0 Create a list of XML document files to be imported
2.0 Build the regexp filters for each 'common' tag pair with XPath
3.0 For each document to be imported
{
    3.1 Read in the entire XML file
    3.2 Apply the regex filter to extract the 'type' and 'subtype'
        attributes
    3.3 If the combination exists in the 'simplecms_doctypes' table
        {
            3.3.1 Retrieve the 'doctype_typeid' from the
                ' simplecms_doctypes' table
        }
    3.4 else
        {
```

```

        3.4.1 Insert the record into the 'simplecms_doctypes' table
        3.4.2 Retrieve the new 'doctype_typeid'
    }
3.5 Apply the regex filter to extract the document title
3.6 For each author
    {
        3.6.1 Apply the regex filter to extract the author info (one or
            more)
        3.6.2 If the author exists in the 'simplecms_authors' table
            {
                3.6.2.1 Retrieve the 'author_authorid' from the
                    'simplecms_authors' table
            }
        3.6.3 else
            {
                3.6.3.1 Insert the record into the
                    'simplecms_authors' table
                3.6.3.2 Retrieve the new 'author_authorid'
            }
    }
3.7 Apply the regex to extract the page URL, if any
3.8 Apply the regex to extract the contents of "<body>...</body>"
    block
    (without start/end 'body' tags)
3.9 Insert the document information as a record into the
    'simplecms_contents' table
3.10 For each author of this document
    {
        3.10.1 Insert the author and the document details as a record
            into the 'simplecms_authordetails' table
    }
}

```

This is only a generic guideline. The actual PHP code may differ slightly in structure.

PHP Code

The following code loosely follows the above algorithm to import any of the XML document files described early. Document information is stored in the `simplecms` database, described earlier. Note that the debugging code has been left out of the following listing. If one finds problems while importing the XML files, it might indicate that the document does not have well-formed XML markup. Let's first define some global constants for later database access:

```

<?php

// FILE: importxmldocs.php
// USE: Import a content document and store it in the simplecms database
// Define information for database connection
$host = "localhost";    // MySQL server
$user = "admin";        // Username
$dbns = "simplecms";     // Database name
$pass = "";             // Password

```

Next import the PEAR::DB class:

```
require_once 'DB.php';
```

The following class that will be used to import XML documents and define data members:

```
class ImportXMLDocument
{
    var $filename;        // Document to parse
    var $db;
    var $type;            // Document type
    var $subtype;         // Document subtype
    var $title;           // Document title
    var $authors;         // Array of author information
    var $pageurl;         // The URL, if any, associated with this document
    var $body;            // The XML-like markup of the <body>...</body>
                        // block
    var $doc;             // The parsed representation of the XML document
}
```

The constructor will receive information about the database to use and connect to the database using PEAR::DB, and `$this->db` will keep the connection to the database that will be used in this class:

```
function ImportXMLDocument($host,$user,$pass,$dbs)
{
    $dsn = "mysql://$user:$pass@$host/$dbs";
    $this->db = DB::connect($dsn);
    if (DB::isError($db)) {
        die ($this->db->getMessage());
    }
}
```

To parse the XML documents and extract content information we'll use XPath; the XPath `eval()` functions are part of the DOM XML extension in PHP. We'll need PHP 4.0.4 or later compiled with the DOM extension enabled in order to use this code.

XPath is a widely-used language to query XML documents. For more information on XPath and the DOM extension refer to Chapter 6 and 7 in Professional PHP4 XML from Wrox Press (ISBN 1-861007-21-3).

First we define a function to get an array of strings when an XPath expression is executed. The function can be used to get the value of attribute nodes (`//foo/@some`) or text node (`//some/text()`). Note that for PHP versions prior to 4.0.4, the array returned by `xpath_eval()` looks a little different:

```
function XpathGetTextNodes($doc,$expr)
{
    $data = Array();
    $xpath=$doc->xpath_init();
    $ctx = $doc->xpath_new_context();
    $result=$ctx->xpath_eval($expr);
    $nodes=$result->nodeset;
    foreach($nodes as $a_node) {
        if($a_node->node_type() == XML_ATTRIBUTE_NODE) {
```

```

        $value = $a_node->value;
    } elseif ($a_node->node_type() == XML_TEXT_NODE) {
        $value = $a_node->content;
    }
    $data[] = $value;
}
return $data;
}

```

The next function returns an XML fragment for the element selected by an XPath expression; it will be used to get an XML fragment for all the authors in an XML document:

```

function XpathGetXmlNodes($doc,$expr)
{
    $xpath=$doc->xpath_init();
    $ctx = $doc->xpath_new_context();
    $result=$ctx->xpath_eval($expr);
    $nodes=$result->nodeset;
    $data = Array();
    foreach($nodes as $a_node) {
        $data[] = $a_node->dump_node($a_node);
    }
    return $data;
}

```

With these functions in mind, we can write the `ParseDocument()` function that will parse an XML document and store the information in the object's data members so we can then store the information in the database. The function will receive the name of the filename to parse. The first part parses the XML document using the DOM extension, and if there is an error we exit:

```

function ParseDocument($filename)
{
    $this->filename = $filename;
    // We parse the XML document here
    $this->doc = xmldocfile($this->filename);
    if(!$this->doc) {
        die("XML parser error the document cannot be found or it is
            not well-formed");
    }
}

```

Now we can use the `XpathGetTextNodes()` function defined earlier to get the content type, the XPath expression that will return the content of the type attribute in the `<content>` element is `/content/@type`. Here is the code:

```

$aux = $this->XpathGetTextNodes($this->doc,"/content/@type");
$this->type = $aux[0];

```

The same type of code is needed for the subtype, pageurl, and title:

```
$aux = $this->XpathGetTextNodes($this->doc, "/content/@subtype");
$this->subtype = $aux[0];

$aux = $this->XpathGetTextNodes($this->doc,
                                "/content/title/text()");
$this->title = $aux[0];

$aux = $this->XpathGetTextNodes($this->doc, "/content/page/@url");
$this->pageurl = $aux[0];
```

Now we will get all the XML fragments for each author in the authors element:

```
$authors = $this->XpathGetXmlNodes ($this->doc,
                                     "/content/authors/author");
$this->authors = Array();
```

And for each author fragment, we parse the fname, mname, and lname using the same approach we used for the title of the page:

```
foreach($authors as $author) {
    $author_xml = xmldoc($author);
    if(!$author_xml) {
        die("Author portion is not well formed!");
    }
    $aux = $this->XpathGetTextNodes($author_xml,
                                    "/author/fname/text()");
    $aut["fname"] = $aux[0];
    $aux = $this->XpathGetTextNodes($author_xml,
                                    "/author/mname/text()");
    $aut["mname"] = $aux[0];
    $aux = $this->XpathGetTextNodes($author_xml,
                                    "/author/lname/text()");
    $aut["lname"] = $aux[0];
    $this->authors[] = $aut;
}
```

Finally we get the XML content for the <body> element:

```
$aux = $this->XpathGetXmlNodes($this->doc, "/content/body");
$this->body = $aux[0];
} // end of ParseDocument() method
```

Now let's define a member function to save all the collected information for the current document to the simplecms database:

```
function AddToDbs()
{
    // Add the collected information about the current document to
    // the 'simplecms' database.

    // Lookup type/subtype combination in table 'simplecms_doctypes'
    $typeid = $this->LookupTypes();
```

```

        // Insert an appropriate record into table 'simplecms_contents'
        $contid = $this->InsertContentRecord($typeid);

        for ($i=0; $i < count($this->authors); $i++)
        {
            // Lookup author in table 'simplecms_authors'. Add author to
            // table if they are not already in it.
            $authid = $this->LookupAuthor($this->authors[$i]);

            // Insert author details record into table
            // 'simplecms_authordetails'
            $this->LookupAuthorDetail($authid, $contid);
        }
    } // end of AddToDbs() method

```

The following member function has two related purposes. The first purpose is to check if the current document's type and subtype combination already exist in the database (table `simplecms_doctypes`). If so, the record ID (column `doctype_typeid`) for this combination is returned. If the combination does not exist, it is added to the database and the new record's `typeid` value is returned:

```

function LookupTypes()
{
    // Determine if the current document type/subtype combination
    // exists in the 'simplecms_doctypes' table. If so, just retrieve
    // the 'doctype_typeid' field.
    // If not, insert a record into the table and retrieve the new
    // doctype_typeid

    $typeid = 0;

    $type    = $this->type;
    $subtype = $this->subtype;

```

Build the query string to determine if the current type/subtype combination exists in the database:

```

$sqlquery = "SELECT * FROM simplecms_doctypes ";
$sqlquery .= "WHERE doctype_type = '$type'
              AND doctype_subtype = '$subtype' ";
#print " [dbg(lookupTypes)]: qry=[$sqlquery]<br>\n";

$results = $this->db->query($sqlquery);
if (DB::isError($results))
{
    $msg = "Could not query table simplecms_doctypes: ";
    $msg .= $results->getMessage(). "<br>\n";
    die($msg);
}
$numrows = $results->numRows();

```

If the combination exists then get the value of the column `doctype_typeid` for this combination:

```
if ($nrows != 0) {
    // This combination exists (assume one resultset row).
    // Retrieve field 'doctype_typeid'
    $aaRow = $results->fetchRow(DB_FETCHMODE_ASSOC);
    $typeid = $aaRow["doctype_typeid"];
    #print "dbg[doctype]: combo exists ($typeid)<br>\n";
}
```

If the combination does not exist then build an INSERT query string and add a record:

```
else {
    // This combination does not exist. Insert record.
    $insquery = "INSERT INTO simplecms_doctypes
                (doctype_type,doctype_subtype) VALUES ";
    $insquery.= "('$type', '$subtype') ";
    #print "[dbg]: qry=($insquery)<br>\n";

    $insres = $this->db->query($insquery);
    if (DB::isError($insres)) {
        // Problems accessing database.
        $errmsg = "MySQL Error: Cannot access table
                  simplecms_doctypes in ";
        $errmsg .= "database $dbs: ";
        $results->getMessage()."<br>\n ";
        die($errmsg);
    } else {
        // Check how many rows were "affected" by the insert
        $naff = $this->db->AffectedRows();
        if ($naff != 1) {
            // Problems accessing database.
            $errmsg = "Problems with database. ";
            $errmsg .= "Doctype record not inserted.<br>\n";
            die($errmsg);
        }
    }
}
```

Now a new type/subtype combination record has been added to the database. Next we retrieve the value of the `doctype_typeid` column:

```
else {
    // Doctype record inserted
    $typeid = $this->db->getOne("SELECT doctype_typeid
                              FROM simplecms_doctypes
                              WHERE doctype_type='$type'
                              AND doctype_subtype='$subtype'");
    #print "dbg[simplecms_doctypes]: created new
          ($typeid)<br>\n";
}
}
}
return($typeid);
} // end of LookupTypes() method
```

The following member function checks to see if the current document has already been added to the database. If it hasn't been added, a new record is added to the `simplecms_contents` table:

```
function InsertContentRecord($typeid)
{
    // Insert an appropriate record into 'simplecms_contents'. However,
    // there is a rule that for a given document type/subtype
    // combination, only one use of a particular title is allowed.
    // We need to check for duplicate titles first.
```

Here we copy some of the class properties to local variables:

```
$title    = $this->title;
$pageurl  = $this->pageurl;
$body     = addslashes($this->body);
```

Build a query string to determine if this document has already been recorded. The criteria for this check is that for a given `type/subtype` combination, there must be only one document with a particular title. However, there is no comparison of document content:

```
$sqlquery = "SELECT content_contentid FROM simplecms_contents ";
$sqlquery .= "WHERE content_typeid = '$typeid' ";
$sqlquery .= "AND content_title = '$title' ";

#print "[dbg(lookup title)]: qry=$sqlquery<br>\n";

$results = $this->db->query($sqlquery);
if (DB::isError($results)) {
    $msg = "Could not query table content: ";
    $msg .= $results->getMessage()."<br>\n";
    die($msg);
}
$numrows = $results->numRows();
```

If the content record exists, get the value of the record ID and return it to the calling code:

```
if ($numrows != 0) {
    // This content record exists (assume one resultset row).
    // Retrieve field 'content_contentid'

    $aaRow = $results->fetchRow(DB_FETCHMODE_ASSOC);
    $contid = $aaRow["content_contentid"];
    print "[dbg(content)]: content record exists ($contid)<br>\n";
}
```

If no document with this title and document type exists in the database, add a new record:

```
else {
    // This content record does not exist. Insert record.
    $insquery = "INSERT INTO simplecms_contents
        (content_typeid,content_title,
         content_pageurl,content_body) VALUES ";
    $insquery.="($typeid, '$title', '$pageurl', '$body') ";
    print "[dbg]: qry=($insquery)<br>\n";
```

```

    $insres = $this->db->query($insquery);

    if (DB::isError($insres)) {
        // Problems accessing database.
        $errmsg = "MySQL Error:Cannot access table content in ";
        $errmsg .= "database $dbs: ".$this->db->getMessage().
            "<br>\n ";
        die($errmsg);
    } else {
        // Check how many rows were "affected" by the insert
        $naff = $this->db->affectedRows();
        if ($naff != 1) {
            // Problems accessing database.
            $errmsg = "Problems with database. ";
            $errmsg .= "Content record not inserted.<br>\n";
            die($errmsg);
        }
    }
}

```

Once the record is inserted, retrieve the ID of the new record:

```

        else {
            // Content record inserted
            $contid = $this->db->getOne("SELECT content_contentid
                                     FROM simplecms_content
                                     WHERE content_title='$title'
                                     AND content_pageurl='$pageurl'
                                     AND content_body='$body'");
            print "dbg[content]: created new ($contid)<br>\n";
        }
    }
}
return($contid);
} // end of InsertContentRecord() method

```

The following member function checks to see if an author already exists in the `simplecms_authors` table. If the author exists, the appropriate record ID is returned, else they are added to the database and the ID of the new record is returned:

```

function LookupAuthor($author)
{
    // Check whether this author is already in table
    'simplecms_authors'.
    // If it is, retrieve the 'author_authorid' field.
    // Otherwise add the author and retrieve the new
    'author_authorid' field.

    // Uncomment the following code to get the author printed out.
    #print "<pre>\n";
    #print "[dbg(lookup)]: author\n";
    #var_dump($author);
    #print "</pre>\n";
}

```

```

$fname = $author["fname"];
$mname = $author["mname"];
$lname = $author["lname"];

$sqlquery = "SELECT author_authorid FROM simplecms_authors ";
$sqlquery .= "WHERE author_firstname = '$fname' ";
$sqlquery .= "AND author_middlename = '$mname' ";
$sqlquery .= "AND author_lastname = '$lname' ";

#print "[dbg(lookupAuthor)]: qry=[$sqlquery]<br>\n";

$results = $this->db->query($sqlquery);
if (DB::isError($results)) {
    $msg = "Could not query table simplecms_authors: ";
    $msg .= $results->getMessage()."<br>\n";
    die($msg);
}
$nrows = $results->numRows();
if ($nrows != 0) {
    // This combination exists (assume one resultset row).
    // Retrieve field 'author_authorid'

    $aaRow = $results->fetchRow(DB_FETCHMODE_ASSOC);
    $authid = $aaRow["author_authorid"];
    print "dbg[simplecms_authors]: author exists
        ($authid)<br>\n";
}

else {
    // This author does not exist. Insert record.
    $insquery = "INSERT INTO simplecms_authors
        (author_firstname, author_middlename,
         author_lastname) VALUES ";
    $insquery.= "('$fname', '$mname', '$lname') ";
    print "[dbg]: qry=($insquery)<br>\n";

    $insres = $this->db->query($insquery);
    if (DB::isError($insres)) {
        // Problems accessing database.
        $errmsg = "MySQL Error: Cannot access table
            simplecms_authors in ";
        $errmsg .= "database $dbs: ".
            $insres->getMessage()."<br>\n ";
        die($errmsg);
    } else {
        // Check how many rows were "affected" by the insert
        $naff = $this->db->affectedRows();
        if ($naff != 1) {
            // Problems accessing database.
            $errmsg = "Problems with database. ";
            $errmsg .= "Author record not inserted.<br>\n";
            die($errmsg);
        }
    }
}

```

```

        } else {
            // Author record inserted
            $authid = $this->db->getOne("SELECT author_authorid
                                      FROM simplecms_authors
                                      WHERE author_firstname='$fname'
                                      AND author_middlename='$mname'
                                      AND author_lastname = '$lname'");
            print "dbg[author]: created new ($authid)<br>\n";
        }
    }
    return($authid);
} // end of LookupAuthor() method

```

The following member function checks to see whether the author details for a particular document have been added. Author details simply refer to a particular author's record ID from the `simplecms_authors` table. For example, if an author's record ID (`author_authorid`) in the `simplecms_authors` table is 5, and the record ID (`authordetail_contentid`), of the document in question is 17, then we are checking to see if the `simplecms_authordetails` table has a record with these two values. If there is a record, then the author details for this document have already been added, else we need to add a new record:

```

function LookupAuthorDetail($authid, $contid)
{
    // Check to see whether this author's contribution to the
    // indicated document has already been recorded. If so, retrieve
    // the 'authordetail_authordetailid' field. Otherwise, insert an
    // 'author detail' record and retrieve the new 'authordetailid'.

    $authdetid = 0;

    $sqlquery = "SELECT authordetail_authordetailid
                 FROM simplecms_authordetails ";
    $sqlquery .= "WHERE authordetail_authorid = $authid ";
    $sqlquery .= "AND authordetail_contentid = $contid ";

    print "[dbg(lookupAuthDet)]: qry=[$sqlquery]<br>\n";

    $results = $this->db->query($sqlquery);
    if (DB::isError($results)) {
        $msg = "Could not query table simplecms_authordetails: ";
        $msg .= $results->getMessage()."<br>\n";
        die($msg);
    }
    $nrows = $results->numRows();

    if ($nrows != 0) {
        // This author detail record exists
        // (assume one resultset row).
        // Retrieve field 'authordetail_authordetailid'

        $aaRow = $results->fetchRow(DB_FETCHMODE_ASSOC);
        $authdetid = $aaRow["authordetail_authordetailid"];
        print "dbg[simplecms_authordetails]: author detail record

```

```

        exists $authdetid)<br>\n";
    } else {
        // This author detail record does not exist. Insert record.
        $insquery = "INSERT INTO simplecms_authordetails
                    (authordetail_authordid, authordetail_contentid)
                    VALUES ";
        $insquery.= "($authid, $contid) ";
        print "[dbg]: qry=($insquery)<br>\n";

        $insres = $this->db->query($insquery);
        if (DB::isError($insres)) {
            // Problems accessing database.
            $errmsg = "MySQL Error: Cannot access table
                    simplecms_authordetails in ";
            $errmsg .= "database $dbs: ";
            $insres->getMessage()."<br>\n ";
            die($errmsg);
        } else {
            // Check how many rows were "affected" by the insert
            $naff = $this->db->affectedRows();
            if ($naff != 1) {
                // Problems accessing database.
                $errmsg = "Problems with database. ";
                $errmsg .= "Author detail record not
                        inserted.<br>\n";
                die($errmsg);
            } else {
                // Author detail record inserted
                $authdetid = $this->db->getOne("SELECT
                        authordetail_authordetailid
                        FROM simplecms_authordetails
                        WHERE authordetail_authordid='$authid'
                        AND authordetail_contentid='$contid'");
                print "dbg[authordet]: created new
                        ($authdetid)<br>\n";
            }
        }
    }
    return($authdetid);
} # end of LookupAuthorDetail() method

} # end of ImportXMLDocument class

```

The main code can be written to parse one or many XML documents. This is an example of how to parse one document:

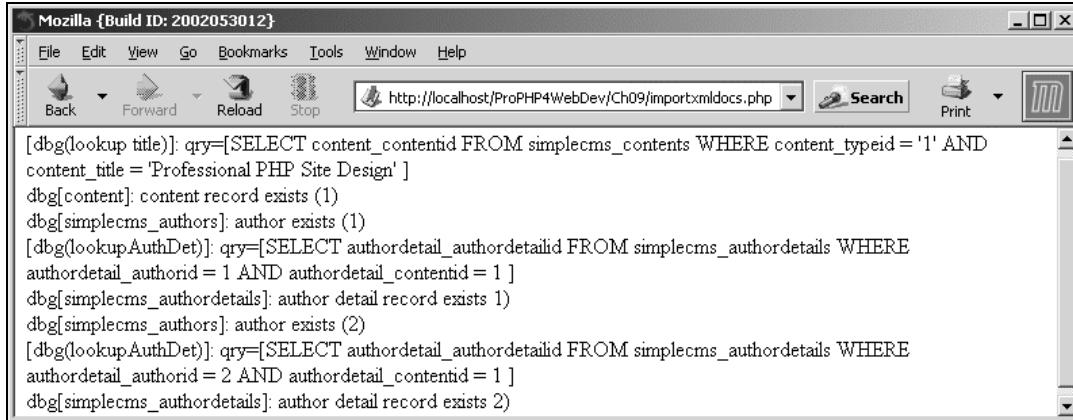
```

$d1 = new ImportXMLDocument($host,$user,$pass,$dbs);
$d1->ParseDocument("C:\Program Files\Apache Group\Apache2\htdocs\
                  ProPHPSiteDesign\Ch09\resume01.xml");
$d1->AddToDbs();
?>

```

Of course we can define a list of documents and parse all of them, if necessary, in a loop.

This is the output of the `importxmldocs.php` script on the browser:



Delivering the Content

As stated earlier in this chapter, the focus of the case study is the addition of tags to and import of the document content. How the stored content is used depends on the needs of the end user. How the content of a particular document is manipulated and delivered depends on the type/subtype combination (that is, depends on the inherent tagset and document structure for a particular document type).

Since there are virtually unlimited types of XML document structures that can be stored in the `simplecms` database, we cannot get into a detailed discussion of writing application components for the manipulation and delivery of the content of each. However, we will look at algorithms for some example components.

List all Documents

A report that is very likely to be used frequently is a list of documents that are currently in the database. The algorithm below lists all documents ordered by type, subtype, and then title. Depending on functional requirements, the output might be plain text, XML, or some other format:

1. Retrieve all documents in the database by selecting from the content-table, and join it together with the doctypes-table using the typeid-key.
2. Process retrieved data and output list of documents by type, subtype, and title.

List Documents of a Specific Type and Subtype

What might be more useful is a list of document titles for a specific type and/or subtype. The algorithm below shows the necessary steps. Once again, the output might be plain text, XML, or some other format:

1. Specify document type
2. Specify document subtype
3. Retrieve all relevant documents by selecting from the content-table, defining a specific type and subtype, and grouping by both type and subtype.
4. Process retrieved data and output list of documents by type, subtype, and title.

Note the way in which the end user will convey the necessary information (type, subtype) depends on the functional requirements. The information might be conveyed via an HTML form or in batch mode as:

```
<?listdocs_by_type type="fiction" ?>
```

or:

```
<?listdocs_by_subtype subtype="resume" ?>
```

or:

```
<?listdocs_by_doctype type="nonfiction" subtype="book" ?>
```

We would then have to write an application component that could interpret these XML PIs.

Display Table of Contents for Non-Fiction Book

Finally, we give the algorithm, for one example, of how an application-parsing component would behave. If we want to see the Table of Contents for a non-fiction book document, we might use the following algorithm:

1. Specify document type
2. Specify document subtype
3. Specify document title
4. Retrieve document content by selecting from the content-table, and define a specific type, a specific subtype, and a specific title.
5. Parse the retrieved XML content:
 - 5.1 Create a 'doctree' object
 - 5.2 For each chapter tag, <chap>, encountered
 - 5.2.1 Create a 'chap' object and add it to the doctree
 - 5.2.2 Extract the 'num', 'title', and 'style' attributes of the <chap> tag and save them in the chap object.

```
5.2.3 For each section tag, <sec>, encountered
{
    5.2.3.1 Create an 'sec' object and it to the currently
        open chap or parent sec object.
    5.2.3.2 Extract lev, title, style attributes from the
        <sec> tag and save them to the sec object.
    5.2.3.3 For each paragraph tag (not shown in example) or
        other sectional document content
        {
            Ignore the content
        }
}

6. Output the TOC in the desired format.
```

We now have a document tree in the form of a complex object data structure. Depending on functional needs, there are many ways in which the table of contents can be displayed.

Tagname Search Engine

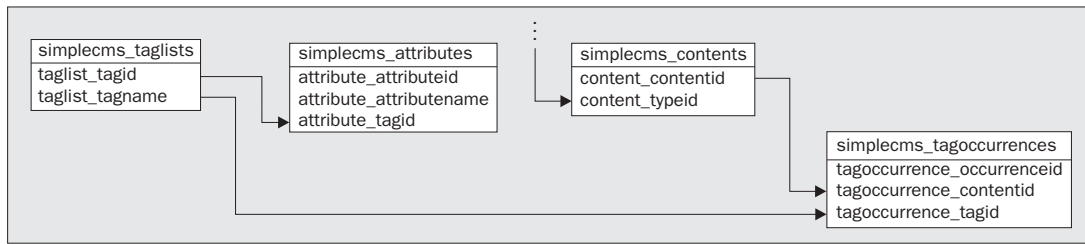
Earlier we have discussed two types of search engines that might be built for retrieving the content. The second type discussed was the **tag name** search engine. Given a specific tag name and optional attribute names, a list of documents could be produced which contain the specified tag in the body. This is a useful search engine, particularly if we are developing application components to parse different document types.

To implement such an engine, we would need to record the occurrences of each and every tag and all the attribute names that occur in each imported XML document's body (that is, between the <body> and </body> tags). Since this step would be best performed during import, we have two choices for implementation:

- ❑ Toss out the custom parser defined earlier for the common tags (that is, <content>, <title>, <authors>, <author>, <fname>, and so on) and replace it with code built around PHP's XML/Expat functions
- ❑ Add some new code that uses the XML/Expat functions to just parse the document body markup

Whichever choice we make, recording tags and attribute occurrences is relatively simple. We need one database table to store a list of unique tag names. A second table would store attribute names and a pointer to the tag name they belong to. A third table would record occurrences of tags. Such a scheme would also implicitly define where the attributes occur. This allows an end user to search for documents containing tags that have a specific attribute name.

The tables we can use are `simplecms_taglist`, `simplecms_attributes`, and `simplecms_tagoccurrences`, shown in the relationship diagram below. Note that the `simplecms_content` table is repeated here (with just the ID columns) to show the relationship with the previously discussed database tables:



Here is the general algorithm for populating the three new tables:

```

For each tag encountered in the current document's body
{
    Check to see if the current tag has already been recorded in the tag
    list. If it has not, add it:
    If the tagname is already in the "taglist" table {
        Retrieve the record id, tagid
    } else {
        Add the tagname to table "taglist"
        Retrieve id, tagid, of new record
    }
}

```

Now we can store even a single occurrence of a particular tagname and its attributes. Keep in mind that some tags may have optional attributes. Even though the tag occurs several times, the attributes used may be in different combinations. The code snippet below stores multiple occurrences, even of the same combination in a single document, as it is easy to do, and provides the user a quick method of determining how many times a particular tag occurred in a document:

```

// Store an "occurrence" record for this tag in the current document:

Add a record to table "tagocc" containing the id of the current
document, contentid, and the current tag, tagid.

```

Now we store attribute names for the current tag. Since we are storing multiple occurrences of a tag within the current document, we will want to know which attributes were used in each occurrence. So we simply add a record to the attribs table for each attribute used with the current tag:

```

Add a record to table "attribs" (use tagid from above)

```

Note that more complex queries can be performed than what we have discussed above. For example, a feature that allows recording the occurrences of tag and attribute names allows searching for tag names or attribute names based on the document type. This can be used to enforce certain tagging requirements for specific document types.

Possible Enhancements

A couple of suggestions for the application in this case study are suggested below:

- ❑ Ensure database integrity; if the author detail records (the `simplecms_authordetails` table) for a particular document are not successfully added then there should be a provision to delete the appropriate record from table content
- ❑ Build a web corpus and set up a regular search engine
- ❑ Add a web interface to the import application, particularly for manipulating common information for each document
- ❑ Add an application parsing component for each document type/subtype combination

Summary

This case study looks at a relatively simple method of storing textual content with different document structures. Content is marked up in a custom XML tagset that follows the same general structure and can thus be stored in a single database. XML parsers are then defined for each combination of content type and subtype and are tied to the application's features.

While XML's predecessor SGML was used mostly for technical documentation and XML is often used for representing structured data, there are still numerous advantages to marking up various types of textual documents in XML. Note, however, that although we are not converting the body of each document into appropriate database records (beyond the basic tags common to each type of document), the application could work very well for a production web site, maintaining a database from which you could draw content.

For those who are interested in a CMS for web pages, with multiple contributors both technical and non-technical contributors, please consult Chapter 10.

