Efficiency in the XML Era

Pete Ryland <pdr@pdr.cx>

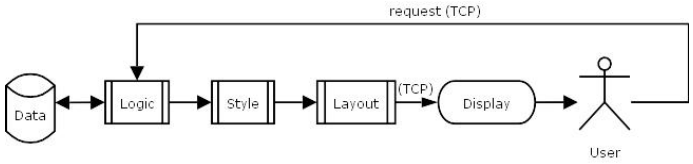
Historically, web content, no matter how it is generated, is sent to a client as HTML. Many sites use XML and XSL to separate content and style, whilst continuing to allow the client to receive HTML. As more people use web browsers capable of interpreting XML and performing the HTML generation at the client end, we are thus approaching the end of this intermediate stage of web content delivery, and it is perhaps now time to re-assess the methods in which we generate said content. This paper serves to analyse how current web languages like Java and PHP will handle this approaching era, and offers a new solution called xmldb.

Introduction

The World-Wide Web (or WWW) is constantly evolving. In the beginning, all web content was static, but then, to our rescue, came server-side includes, allowing us to include common "boiler-plate" code into a number of pages. From there, cgi-bin scripts and scripting languages like mod_perl and PHP have been developed, allowing web pages to be flexible and dynamic, with database backends or streaming news feeds.

The current trend in web development is to have pages produced using a four-layer model. On the lowest layer, we have the Data, be it static or dynamic. Next, there is usually a Logic layer, which takes the requested URI as input and requests the appropriate Data to determine the page's Content. The Content then is sent back to the client for Display.

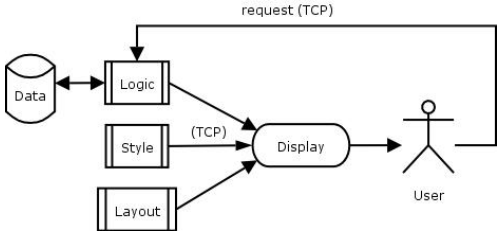
Figure 1. Traditional Content Flow



In days gone by, the client would have received the output of the Layout layer which would often have been intertwined with the Logic and Style layers. Hence, under this model, languages like PHP and mod_perl have thrived.

However, in the next year or so, it is likely that most web content will in fact be using a new, more efficient model, where the Content, Style and Layout are sent directly to the client, like such:

Figure 2. New Content Flow



This has a number of obvious consequences, the biggest of which are reduction of memory and bandwidth requirements and moving much of the processing load off the server and onto the client. Proxying and caching are also more effective under this model, especially for dynamic content. It mostly achieves all this by solving the longest-standing inefficiency with the web: the lack of html to logically include boilerplate markup *from the client end*.

In the former model, our data are generally stored in relational databases and queried from a scripting language using SQL. With such radical evolution of the web, we have to ask ourselves whether this is still the best method for extracting Content.

With this new era of web development in mind, this paper discusses some of the drawbacks of current Web Application frameworks such as Java's JSP/J2EE and PALM (PHP/Apache/Linux/MySQL), and introduces a new framework called xmldb.

Simple Example

For demonstration purposes, let us consider a simple but common web page component: the hierarchical menu. The example data is taken from my current home page at http://pdr.cx/ and looks like such:

Figure 3. Hierarchical Menu Example

```
Name: Home
URL: /
Description: Pete's Home Page

Name: Projects
URL: /projects/
Description: My Projects

   Name: GNetMD
   URL: /projects/gnetmd/main/
   Description: A GNOME NetMD Utility

      Name: FAQ
      URL: /projects/gnetmd/faq/

      Name: Screenshot
      URL: /projects/gnetmd/ss/

   Name: HomeBrew Decompiler
   URL: /projects/hbd/
   Description: My Java Decompiler

[et cetera]
```

With this in mind, let us set out to implement a menu component using current best practices in both PHP and Java. Our goal, for the purposes of this exercise, is to have the client display HTML like the following:

Figure 4. HTML Menu Example

```
<html>
  <head>
    <title>Menu Example: HTML Version</title>
  </head>
```

```
 <body>
  <ul>
   <li>
    <a href="/" title="Pete's Home Page">Home</a>
   </li>
   <li>
    <a href="/projects/" title="My Projects">Projects</a>
    <ul>
     <li>
      <a href="/projects/gnetmd/main/" title="A GNOME NetMD Utility">GNetMD</a>
      <ul>
       <li>
        <a href="/projects/gnetmd/faq/">FAQ</a>
       </li>
       <li>
        <a href="/projects/gnetmd/ss/">Screenshot</a>
       </li>
      </ul>
     </li>
     <li>
      <a href="/projects/hbd/" title="My Java Decompiler">HomeBrew Decompiler</a>
     </li>
    </ul>
   </li>
   [et cetera]
  </ul>
 </body>
</html>
```

Obviously, with appropriate class attributes, CSS and/or javascript we can make this example into a full interactive menu system.

Database Schema

The traditional way to store the menu data would be in a database with a schema like the following:

Figure 5. Possible Database Schema

```
drop table menuitems;
create table menuitems(
 id         integer      primary key,
 parent      integer      references menuitems(id),
 name       varchar(256)   not null,
 url        varchar(256),
 description  varchar(256)
);
```

Traditional PHP Solution

Traditionally, a solution in PHP would simply spit out the necessary HTML, with no intermediate XML. A possible implementation might look like this:

Example 1. Traditional PHP Solution

```
<?
  class menuitem {
    var $name, $url, $desc, $submenu;
    function menuitem($name, $url, $desc, $submenu = null) {
      $this->name = htmlentities($name); $this->url = $url;
      $this->desc = htmlentities($desc); $this->submenu = $submenu;
    }
  }

  class menu {
    var $menuitemarray;
    function menu($menuitemarray) {
      $this->menuitemarray = $menuitemarray;
    }
    function print_as_menulist($depth, $indent) {
      $depth--;
      echo "$indent<ul>\n";
      foreach ($this->menuitemarray as $ind => $i) {
        $desc = ($i->desc == null)? "" : " title=\"{$i->desc}\"";
        echo "$indent  <li>\n$indent   <a href=\"{$i->url}\"$desc>{$i->name}</a>\n";
        if ($depth > 0 && $i->submenu != null) {
          $i->submenu->print_as_menulist($depth-1, $indent . "   ");
        }
        echo "$indent  </li>\n";
      }
      echo "$indent</ul>\n";
    }
  }

  $conn = pg_connect("dbname=example");

  function recursive_fetch($parent_menu_id) {
    $r = pg_query("select * from menuitems where parent $parent_menu_id");
    if (pg_num_rows($r) == 0)
      return null;

    $menuarray = array();

    while ($arr = pg_fetch_array($r)) {
      $menuarray[$arr[id]] = new menuitem($arr[name], $arr[url], $arr[description]);
    }
    foreach($menuarray as $key => $value) {
      $menuarray[$key]->submenu = recursive_fetch("= $key");
    }
    return new menu($menuarray);
  }

  $menu = recursive_fetch('is null');
?>
<html>
  <head>
    <title>Menu Example: PHP Version</title>
  </head>
  <body>
<? $menu->print_as_menulist(7, '   '); ?>
  </body>
</html>
```

This works fine, but it has three major drawbacks:

1. it is difficult to maintain,

2. boilerplate code is transferred to the client on each request, and

3. it doesn't easily allow for generation of other markup languages.

There are a number of reasons why solutions like this are difficult to understand and maintain. For example, the close coupling of the code and the database mean that schema changes are difficult and error-prone. Also, it is easy to write code that produces incorrect HTML unless one is careful to check the output with an HTML verifier. This is mostly due to the complexity involved in this type of solution.

A Better PHP Solution

A more maintainable and flexible solution would be to build an XML document from our data instead. This can then be sent to the client directly, assuming that the client's browser can work with our new model of operation. To achieve this, we can use the experimental PHP extension of Daniel Veillard's excellent libxml C library. Here is an example of how to do this:

Example 2. A Better PHP Solution

```
<?
 class menuitem {
  var $name, $url, $desc, $submenu;
  function menuitem($name, $url, $desc, $submenu = null) {
   $this->name = htmlentities($name); $this->url = $url;
   $this->desc = htmlentities($desc); $this->submenu = $submenu;
  }
 }

 class menu {
  var $menuitemarray;
  function menu($menuitemarray) {
   $this->menuitemarray = $menuitemarray;
  }
  function to_xml_element(&$doc) {
   $node = $doc->create_element("menu");
   foreach ($this->menuitemarray as $ind => $i) {
    $itemnode = $doc->create_element("menuitem");
    $namenode = $doc->create_element("name");
    $namenode->append_child($doc->create_text_node($i->name));
    $itemnode->append_child($namenode);
    if ($i->url) {
     $urlnode = $doc->create_element("url");
     $urlnode->append_child($doc->create_text_node($i->url));
     $itemnode->append_child($urlnode);
    }
    if ($i->desc) {
     $descnode = $doc->create_element("desc");
     $descnode->append_child($doc->create_text_node($i->desc));
     $itemnode->append_child($descnode);
    }
    if ($i->submenu != null) {
     $itemnode->append_child($i->submenu->to_xml_element($doc));
    }
    $node->append_child($itemnode);
   }
   return $node;
  }
 }
```

```
  $conn = pg_connect("dbname=example");

  function recursive_fetch($parent_menu_id) {
    $r = pg_query("select * from menuitems where parent $parent_menu_id");
    if (pg_num_rows($r) == 0)
      return null;

    $menuarray = array();

    while ($arr = pg_fetch_array($r)) {
      $menuarray[$arr[id]] = new menuitem($arr[name], $arr[url], $arr[description]);
    }
    foreach($menuarray as $key => $value) {
      $menuarray[$key]->submenu = recursive_fetch("= $key");
    }
    return new menu($menuarray);
  }

  $menu = recursive_fetch('is null');

  $doc = domxml_new_xmldoc("1.0");
  $doc->append_child($menu->to_xml_element($doc));
  echo $doc->dump_mem(1);
?>
```

This produces the following XML document:

Figure 6. Generated XML Document

```
<?xml version="1.0"?>
<menu>
  <menuitem>
    <name>Home</name>
    <url>/#top</url>
    <desc>Pete's Home Page</desc>
  </menuitem>
  <menuitem>
    <name>Projects</name>
    <url>/projects/#top</url>
    <desc>My Projects</desc>
    <menu>
      <menuitem>
        <name>GNetMD</name>
        <url>/projects/gnetmd/main/#top</url>
        <desc>A GNOME NetMD Utility</desc>
        <menu>
          <menuitem>
            <name>FAQ</name>
            <url>/projects/gnetmd/faq/#top</url>
          </menuitem>
          <menuitem>
            <name>Screenshot</name>
            <url>/projects/gnetmd/ss/#top</url>
          </menuitem>
        </menu>
      </menuitem>
      <menuitem>
        <name>HomeBrew Decompiler</name>
        <url>/projects/hbd/#top</url>
        <desc>My Java Decompiler</desc>
```

```
    </menuitem>
   </menu>
  </menuitem>
  [et cetera]
</menu>
```

This is somewhat more maintainable, and allows for the generation of markup other than HTML. It also allows boilerplate code to exist in the XSL stylesheet, which is much more efficient, and ensures that our output is valid markup. However, it is still not as readable and maintainable as we might like, and still suffers from the close coupling with the database schema. For completeness, here is the XSL I've used to generate the same HTML code as above:

Example 3. XSL for Menu Example

```
<?xml version="1.0" encoding="utf8"?>
<xsl:stylesheet
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     xmlns="http://www.w3.org/1999/xhtml"
     version="1.0">

  <xsl:output indent="yes" omit-xml-declaration="yes"/>

  <xsl:template match="/">
    <html><head><title>Menu Example: PHP Version #2</title></head><body>
      <xsl:apply-templates select="menu"/>
    </body></html>
  </xsl:template>

  <xsl:template match="menu">
    <ul>
      <xsl:apply-templates select="menuitem"/>
    </ul>
  </xsl:template>

  <xsl:template match="menuitem">
    <li>
      <a href="{url}">
        <xsl:if test="desc">
          <xsl:attribute name="title">
            <xsl:value-of select="desc"/>
          </xsl:attribute>
        </xsl:if>
        <xsl:value-of select="name"/>
      </a>
      <xsl:apply-templates select="menu"/>
    </li>
  </xsl:template>
</xsl:stylesheet>
```

Java Best Practices

Before I move on to describe how to go about solving the issues raised above, I'd like to briefly talk about how this same problem might be attacked from a Java developer's point of view. Whilst it is well beyond the scope of this paper to explain the intricacies of the EJB model, suffice it to say that we can use the same database

schema outlined above for this example. An Entity Bean for each menu item might be created with the following interfaces:

Example 4. Java Solution - MenuItemOperations Interface

```
public interface MenuItemOperations {
  public long getId() throws RemoteException;
  public LongPK getPK() throws RemoteException;
  public long getParent() throws RemoteException;
  public void setParent(long _parent) throws RemoteException;
  public MenuItem getParent()
     throws RemoteException, NamingException, FinderException;
  public String getName() throws RemoteException;
  public void setName(String _name) throws RemoteException;
  public String getUrl() throws RemoteException;
  public void setUrl(String _url) throws RemoteException;
  public String getDescription() throws RemoteException;
  public void setDescription(String _description) throws RemoteException;
}
```

Example 5. Java Solution - MenuItemHome Interface

```
public interface MenuItemHome extends EJBHome {
  public MenuItem create(long _parent, String _name, String _url,
                  String _description)
     throws CreateException, EJBException, RemoteException;
  public MenuItem findByPrimaryKey(LongPK primaryKey)
     throws RemoteException, FinderException, EJBException;
  public Collection findAll()
     throws RemoteException, FinderException, EJBException;
  public Collection findByParent(long id)
     throws RemoteException, FinderException, EJBException;
}
```

The implementation classes would then contain a lot of boiler-plate code that fetches the relevant data from the database - for brevity I won't repeat my example code here. A session bean might then be written to amalgamate all the MenuItems into a complete menu system. This might be instantiated by a JSP or a servlet which may expect the result back in the form of an XML document to be output back to the client.

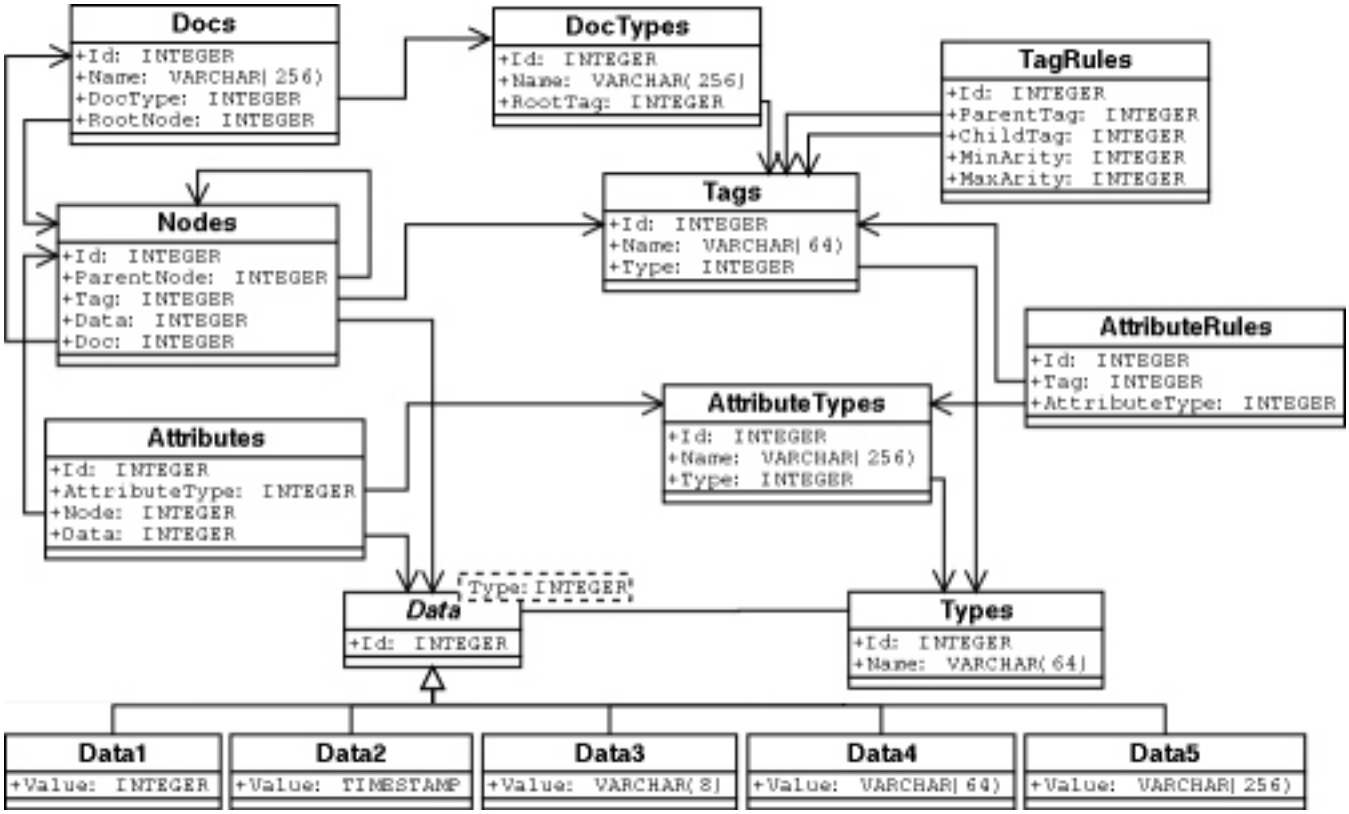
By using this approach, we have gained an important advantage: namely, we now have a scalable model which can be deployed on a number of servers (this may indeed be quite valuable if we were dealing with data that were more sensitive to changes, which our menu example doesn't exemplify). This scalability comes at a price, however, since every single method invocation on a Java Bean has to be serialised and marshaled. This is a very hefty penalty, especially when dealing with hierarchical data. We also haven't solved any of the issues raised with the PHP solution above; we are still closely coupled with the database schema, and the Java solution is actually more difficult to maintain than our PHP example.

A Better Approach

Over the past several years, I've worked on a number of web sites, both small and large, and these same issues keep cropping up, most notably the issue of maintainability. I decided that what was most needed was to de-couple the database from the code. This required the design of a *generic database schema*. Since most data on the web

is naturally hierarchical in structure, and since XML is an already well-understood way to represent hierarchical data, I based the schema on XML. It looks like this:

Figure 7. Generic Database Schema for Hierarchical Data



This schema obviously requires an object-relational database like PostgreSQL.

Whilst this might look wasteful, it is actually as efficient as our former schema; algorithmicly speaking, the number of database searches required is of the same order as before. So for large sets of data, the number of database operations is approximately the same.

Some of the advantages of using a schema like the above are:

1. data can be represented in a more natural hierarchical form, rather than munged into relational form,
2. no tight coupling is necessary between the code and the structure of the data,
3. data structure planning and modification are much simpler,
4. database access can be completely abstracted, making the code more maintainable, and allowing easy transition from one database vendor to another if necessary, and
5. as before, the data can be dynamic, but in fact now the *structure* of the data can be dynamic as well.

XMLDB

I have implemented a C library on top of the above database schema, and called it xmldb, because its structure is heavily based on XML. There is not an exact one-to-one mapping between XML and the structures that xmldb can store, but the current implementation creates these structures in memory using the previously-mentioned libxml. This means that the DOM functions present in libxml can be used (from PHP or C) to extract and/or modify the data. Using xmldb, our menu example is a simple matter of importing the data into the database from an XML representation, and then either using the PHP or Apache plugins to recall the document when requested. A more complicated example might require some manipulation, or so called business logic, but our simple menu requires absolutely no code whatsoever.

The current version of xmldb is quite stable, but still has plenty of room to grow when it comes to features. There are a number of features currently being planned, including an implementation of XPath to query the database, improved ability to update data, better schema support, and symlink and/or overlay support within the database. A GUI tool for browsing the database is also in development. It can be found at http://xmldb.sourceforge.net/ and any questions or suggestions regarding xmldb can be directed to pdr@pdr.cx.