Design Proposal for a Meta-Data-Driven Content Management System

Andreas Krennmair
ak@synflood.at

15th August 2005

Contents

1 Basic Idea

The idea of this new meta-data-driven content management system (CMS), which will be codenamed "priamus", is to store content objects (i.e. files), make them retrievable, keep their revision record, and associate a number of required and/or required meta-data fields, so-called "attributes" with them.

Every content object is identified by its object ID and its revision number. The latest revision always has the highest revision number.

The attributes can be optional or required, and can be of different data types, which depend on the underlying storage system's support. Data types that must be supported are e.g. text, integer or date. Additional data types like BLOBs depend on the underlying storage system.

In addition, every content object is of a certain content type, like "static web page" or "dynamically scripted page". A set of predefined system types will be set up by default (which also give certain pages a special meaning), additional user-defined types can be added later by the user.

The following attributes are mandatory:

- content type
- object ID
- object title
- revision number
- filename in the filesystem

All other attributes are optional.

2 Services

A service provides means to display a content object (optionally including its attributes) via HTTP. This can be simply printing out the static content of the page or transforming or interpreting the file content in some way (see "Programmability" below).

When a service is called, it is provided with the object ID and optionally, the revision number, which it then uses to retrieve the correct object and display it.

By default, two services will be provided:

- show page: displays static pages, and interprets dynamically scripted pages.
- show page source: displays the static content, even if the page is a dynamically scripted page.

Additional services could be "render XML page", which takes the content, transforms it according to a XSL template (whose filename is taken from an attribute) and then displays it, or an "inspect content object", which displays the content object's attributes in a pretty form.

If the page to be displayed is a dynamically scripted page, then any additional parameters that was provided to the service will be passed to this page.

3 Programmability

As mentioned above, on of the special content types will be "dynamically scripted page". When a content object of this page is shown via the "show page" service (see above), then the content of this file will be executed and its output will be displayed. A language or system such as JSP, ePerl, eRuby or PHP where code is embedded with HTML/XML source seems to be advantageous as it combines embedded code with powerful languages. Which language or system will be used depends on the implementation language of the CMS.

eRuby example:

```
<html>
  <head>
    <title><%= params["title"] %></title>
  </head>
  <body>
  <h1>Example page</h1>
  <p>
  <% if params["count"].to_i <= 2 %>
    <%= params["count"] %>
  <% else %>
    many
  <% endif %>
</html>
```

In addition, a mechanism (e.g. CMS-specific library functions) will be provided to include other content objects of the same or similar types (i.e. static pages or dynamically scripted pages) within a page, and to query the CMS for content objects after different criterias. The result of such queries will be available as some kind of result set over which the programmer can iterate.

Example:

```
<ul>
<% cms_query("ObjectTitle,Filename","ObjectId = '#{params["id"]}'") do |cobj| %>
<li><%= cobj.ObjectTitle %>: <%= cobj.Filename %></li>
<% end %>
</ul>
```

In addition, a mechanism for querying XML files that are checked in into the CMS using XPath[1] needs to be provided.

```
<%
 nodes = cms_count_xml_nodes("XML_CONTENT","codecs/codec")
 if nodes > 0 then
   1.upto(nodes) do |i|
     codecname = cms_query_xml("XML_CONTENT","codecs/codec[#{i}]/name/node()")
   %>
   <%= codecname %><br />
   <%
   endif
 endif
%>
```

4 Storage

The storage of the attributes will be implemented by directly mapping the attributes to the fields of a table in an SQL database. Currently, PostgreSQL is the preferred system, as it is licensed under a free license, and it provides very powerful mechanisms that compete with commercial, expensive databases. It is recommended to use PostgreSQL's object-relational database features.

The content type will be kept in a separate type table, and will be associated via the ID with the content objects.

The content itself will be stored to a special location in the filesystem, where each revision has its own, unique filename. Special measures should be taken that even when a lot of content and/or a lot of revisions will be added to the CMS, no directory shall be filled up with a lot of files that could slow down reading the directory content significantly.

5 Interface

5.1 XML-RPC

The CMS will not provide a user interface by itself, but instead only comes with an interface that allows access to all the content management functionality from other programs. This interface will be implemented using the XML-RPC[2] protocol.

This lack of user interface gives the advantage that the developers can fully concentrate on the CMS' core functionality without having to cope with details of the user interface. A comfortable user interface can be implemented later, e.g. as web interface or as a Java rich client. Today, most modern programming languages provide XML-RPC language binding in one form or another, so interfacing with the CMS shouldn't be much of a problem.

5.2 Facilities

Facilities represent the logical separation between the different parts of the CMS. The following facilities are defined:

- Service Execution Facility (SEF): runs services (see above). This is the only read-only facility within the CMS.
- Revision Management Facility (RMF): the XML-RPC interface for adding and deleting content objects respectively their revisions. This is a read-write facility which requires authentication.
- Administration and Configuration Facility (ACF): the XML-RPC interface to remotely manage users and CMS configurations. This is a read-write facility which requires authentication.

Example URLs for the facilities:

```
http://host:7780/mdcms/sef?service=SHOW_PAGE&oId=4711
http://host:7780/mdcms/rmf
http://host:7780/mdcms/acf
```

Examples XML-RPC calls for *RMF* and *ACF*:

```
# RMF XML-RPC interface (incomplete):
mdrmf.login(user : string, password : string)
        returns string (sessionid)
mdrmf.createContentObject(sessionid : string, attribs : struct)
mdrmf.addRevision(sessionid : string, contentid : string, file : base64)
        returns string (revision)
mdrmf.deleteRevision(sessionid : string, contentid : string, revision : string)
mdrmf.listRevisions(sessionid : string, contentid : string)
        returns struct (list of revisions + creation dates et al)
mdrmf.deleteContentObject(sessionid : string, contentid : string)
mdrmf.updateAttributes(sessionid : string, contentid : string)
mdrmf.getAttributes(sessionid : string, contentid : string, revision : string)
        return struct (attributes)
mdrmf.getAvailableAttributes(sessionid : string)
```

```
        return struct (available attribute names and their type)

# ACF XML-RPC interface (incomplete):
mdacf.login(user : string, password : string)
        returns string (sessionid)
mdacf.setConfigurationValue(sessionid : string, key : string, value : string)
mdacf.getConfigurationValue(sessionid : string, key : string)
        returns strings (value)
mdacf.getAvailableConfigurationValues(sessionid : string)
        return struct (list of available configuration values)
mdacf.getUsers(sessionid : string)
        returns struct (users + permissions)
mdacf.addUser(sessionid : string, name : string, perms : struct)
mdacf.setUserPassword(sessionid : string, name : string, password : string)
mdacf.deleteUser(sessionid : string, name : string)
```

6 Crawling

To transform the (dynamically generated) content of the CMS to static content that can then be delivered to the end-user with the highest speed available, a so-called “crawler” will be provided.

The crawler is fed with one or more “crawler overview pages” (COPs). Such a COP contains links to all pages that need to be crawled. The crawler then follows these links and downloads and stores these pages. The resulting filename of the stored pages needs to be configurable, depending on the dynamic URL of the crawled page.

The reason for creating a static version of (parts of) the content from the CMS is that delivering static files via HTTP can be highly optimized for speed, including clustering techniques.

7 Conclusions

This design document gives an overview about the ideas of a new concept of content management that has never been implemented before in the open source world. It reduces content management to its core, using a simply and modular design, making it easy to implement the needs of high-quality content management.

This CMS is *not designed to make content management easy* like other open source CMS, but instead is here to satisfy the needs for professional, large-scale content management. In fact, the ideas presented here are vaguely based on the concepts found in a large-scale CMS that was used for the implementation of a major music download platform, which shows that the basic ideas are proven to work in the real world. The major difference although is that this CMS is designed for simplicity.

References

[1] Clark, James and DeRose, Steve, *XML Path Language (XPath) Version 1.0*, `http://www.w3.org/TR/xpath`

[2] Winer, Dave, *XML-RPC Specification*, `http://www.xmlrpc.com/spec`