Tutor.com

The Tutor.com Website Content Management System

Concepts, Introduction, and Implementation

Gabe Hollombe, Lead Web Developer
3/16/2009

Contents

I Can Has CMS?

The Tutor.com Website Content Management System (CMS) is a custom-built system designed to power the Tutor.com website (http://www.tutor.com) and allow users outside the development team to manage the site's content without the need for development tools like Microsoft Visual Studio.

Why Roll Our Own?

We knew that we wanted CMS-like capabilities in our new website, so we started by looking at what was already on the market.

We found a pair of what seemed to be very full-featured ASP.NET- based CMSes: Ektron (http://www.ektron.com/) and SiteCore (http://www.sitecore.net/). However, both were relatively expensive. The cheapest option was Ektron at around $80k, and I don't even remember SiteCore's price, except that it was also very expensive. Costs aside, we could have definitely built our new website using Sitecore or Ektron. But, once we outlined the features we were actually going to use, we realized we could probably build something in-house to suit our needs.

It should also be mentioned that when we were researching ASP.NET-based CMSes, we also discovered Umbraco (http://www.umbraco.org/). Umbraco seemed to be almost exactly what we needed and wanted, but it had one major shortfall: you had to edit the page templates (think 'Master Pages'), inside a text area control in a web browser; using Visual Studio to author your page templates was impossible except in a copy/paste sort of way. Apparently, this is no longer the case in the latest version of Umbraco, but it was a deal-breaker back when we were evaluating it.

Ultimately, we decided to create our own CMS because the big, commercial CMSes were too heavy (we would only use an extremely small subset of the features they offered) and too expensive. After a short prototyping phase, we had a working proof of concept custom-build CMS that looked like it would handle our needs. The Tutor.com CMS is still doing a great job at the time of this writing.

Templates and Pages and Content Items, Oh My!

The CMS creates web pages works by combining three simple concepts: **Templates, Pages, and Content Items**. Templates describe which types of content go where on a page. Pages are instantiations of a template and have one or more content items associated with them. Content items are the blocks of content that get inserted into the named placeholders of a page's template.

Templates

Templates are just *.aspx pages that live in ~/Templates and contain one or more content user controls which serve as placeholders for content when the page is rendered. Each placeholder (user control) has a friendly name attribute that is used when associating content items with the named place holders in the CMS Admin app and a content type which describes the type of content that will be inserted into the placeholder (html/test/image). The names that you give to placeholders should describe the content type they'll be hosting and not their position on the page. Some example placeholder names might be 'Headline', 'Call to Action', and 'Page Body'. Template data lives in the CMS_Templates table, and the mapping of templates to content placeholders lives in CMS_TemplateContentBlocks.

CMS_Templates – Column Guide

ViewName – the file name of the aspx file in ~/Templates (without the .aspx extension)

Description – a friendly description of the template so you'll recognize it in the CMS Admin

Pages

Pages don't exist on the filesystem of the web server at all, but rather as rows in the database. Each page has one or more placeholder-name / content-item-mappings in the CMS_PageContentBlockItems table, and each page has a URL value. Together, this content mapping and URL is how the routing system and rendering engine (described below) is able to look up and render the appropriate content items for the requested page.

CMS_Pages – Column Guide

Url – the url that this page 'lives at' – no beginning / needed, so 'home' maps to http://www.tutor.com/home
Title – the page title rendered in the <title> html tag
PageMetaDescription – the text to be rendered inside the <meta name="description"> tag
IsPublished – a Boolean value to indicate if this page should be accessible when requested
IsDeletable – a Boolean value to indicate if a CMS user can delete this page
TemplateId – maps to the CMS_Templates table to identify the template that this page will use when it gets rendered

Content Items

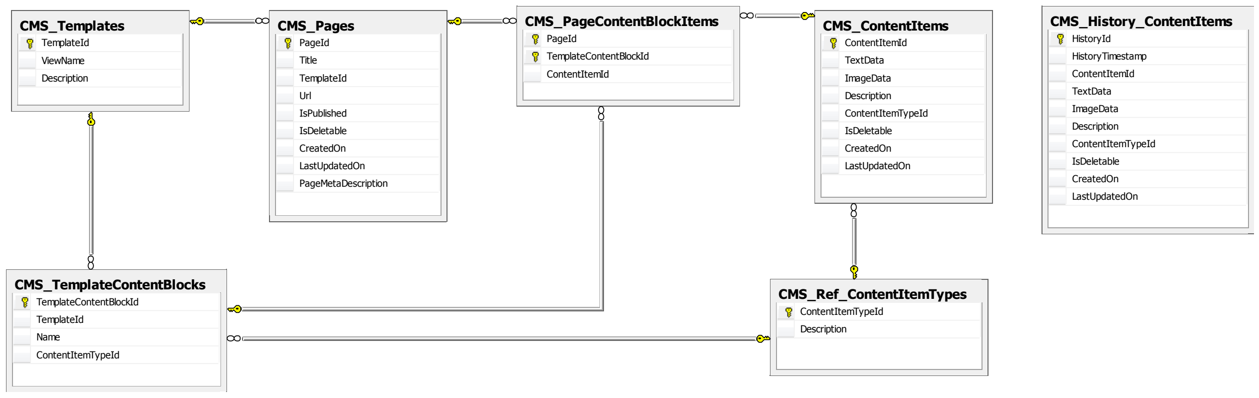
Content Items contain the individual pieces of content that the rendering engine slots in to the content placeholders on a page's template when the page is rendered. These records live in the database and are associated with a specific content type (html/text/image).

CMS_ContentItems – Column Guide

Description – a friendly description of the content, displayed to users in the CMS Admin app
TextData – the content data, if the content type id (below) is html or plain text
ImageData – the binary image data if the content type id (below) is an image type
ContentItemTypeId – maps to the CMS_ContentTypes table to identify the type of content that this content item contains
IsDeletable – a Boolean value indicating a CMS user can delete this content item

Database Diagram

This diagram is zoom-able; please zoom in if you want to read it without pressing your face up to the page.



Routing 101

A core component of the Tutor CMS is the ability to map any URL to a CMS rendered page. We wanted to be able to support URLS like "/pricing" and "/subjects/algebra", without requiring a .aspx file extension. There seemed to be two ways we could support a flexible URL structure like this: either create an ISAPI extension for IIS or take advantage of IIS's ability to execute a URL as a 404 error handler. We opted for the latter and, while it's a bit of a clever hacky solution, it's simple and gets the job done with a minimal amount of fuss.

One consequence of using the 404Handler approach is that when 404Handler.aspx gets called by IIS, HttpContext.Current.Request.Url.ToString() contains the original requested URL but it prefixes it with the path to the 404Handler file. As an example, a request to /subjects?foo=bar would looks something like *http://host/404Handler.aspx?404;http://host:80/subjects?foo=bar.* So, the requested URL and the query string vars need to get parsed off of the modified request URL as part of the routing process.

The CMS routing engine is responsible for mapping a requested URL to a PageId in the database. Here's the flow:

1. IIS gets a request for a page that's not on the file system
2. IIS fires /404Handler.aspx because the website has this configured as an ExecuteURL for 404 errors
3. 404Handler.aspx calls WWW.Global.DoRouting()
4. DoRouting() is where the magic happens

The website application keeps a dictionary of all the URL that live in the CMS. When a request comes in, we check to see if the requested URL is a CMSed page and, if so, we prepare the necessary environment variables for the CMS to render and then run a Server.Execute on the appropriate CMS template (based on the requested URL's parameters in the CMS database tables).

If a requested URL doesn't live in the CMS, we return a 404 (since pages that exist on the file system instead of the CMS will load directly and DoRouting() won't ever fire for requested paths that exist.

Here's the relevant part of DoRouting():

```
// Do we have a url for the matching request in our cms?
if (routing_dict.ContainsKey(url.ToLower()))
{
  DataRow page = (DataRow)routing_dict[url.ToLower()];
  string view_name = page["ViewName"].ToString();
  long page_id = Convert.ToInt64(page["PageId"]);


  // Stick the stuff that our template needs to render into the HttpContext and tell the template to
render
  if (page_content_block_items_dict.ContainsKey(page_id))
  {
    HttpContext.Current.Items.Add("content_assignments", (Dictionary<string, long>)
      page_content_block_items_dict[page_id]);
  }
  else
  {
    //no content items for this page were set in the cms, so pass an empty dict into the context
    HttpContext.Current.Items.Add("content_assignments", new Dictionary<string, long>());
  }

  //Also, remember the title value for the requested page so we can display it in the html title element
  HttpContext.Current.Items.Add("page_title", page["Title"].ToString());

  //Even more also, remember the page's meta description
  HttpContext.Current.Items.Add("page_metadesc", page["PageMetaDescription"].ToString());

  //NOTE: Using Server.Execute() instead of Server.Transfer because Transfer() throws a ThreadAbord
exception and we don't want that.
  HttpContext.Current.Server.Execute("~/Templates/" + view_name + ".aspx" + query_vars, true);
  return true;
}
```

Rendering First Steps – The Template

Once the routing engine has fired off a Server.Execute to a CMS template, the server gets to work rendering that page, slotting in the appropriate CMS content for each named content placeholder in the template.

CMS templates are pure ASP.NET Master Page files that contain some CMS Content controls. The simplest example would be a page like this:

```
<%@ Page Language="C#" AutoEventWireup="true" MasterPageFile="~/WWW.master"
CodeFile="SingleColumn.aspx.cs" Inherits="Templates_SingleColumn" %>
<%@ Register src="~/controls/HtmlContent.ascx" tagname="HtmlContent" tagprefix="uc1" %>

<asp:Content ID="CMSTemplateContent" ContentPlaceHolderID="phContentMain" Runat="Server">
        <uc1:HtmlContent ID="HtmlContentPageBody" runat="server" name="Page Body" />
```

```
</asp:Content>
```

This CMS template contains one HtmlContent control named 'Page Body.' This means that any page in the CMS that is configured to use this template will have a mapping in the database, giving us a ContentItemId to slot in to the 'Page Body' container.

Also, it's worth noting that the CMS templates have nothing in their code-behind files but do inherit from a CMSPageTemplate class. The CMSPageTemplate parent class takes care of setting the page title and the meta-description, and handles the important task of assigning the appropriate content item id to each named content control that it contains. Then, the content gets inserted by the CMS controls on the page, since now they know what content item ids they need to render the values from. Here's the relevant code from CMSPageTemplate.cs that assigns the content item ids to the controls. Note that this code fires OnInit so that the content item ids get assigned to the controls before they look for them in their OnLoad event.

```
// Set the controls on this template's page to match what's in the db...
Dictionary<string, long> content_assignments = (Dictionary<string,
long>)HttpContext.Current.Items["content_assignments"];
foreach (Control control in Util.CollectAllControls(Page.Master, new List<Control>()))
{
        if (control is CMSContent)
        {
                CMSContent cms_control = control as CMSContent;
                if (cms_control.Name != null && content_assignments.ContainsKey(cms_control.Name))
                {
                        cms_control.ContentItemId =
Convert.ToInt64(content_assignments[cms_control.Name]);
                }
        }
}
```

Rendering Next Steps – The Content

The two CMS Content controls to know about are: HtmlContent (HTML goes in to the database and gets rendered, unescaped, by the control) and ImageContent (the image data is stored in the database and rendered by a companion page called ShowImage.aspx). We also have a TextContent control that we basically never use, so don't worry about it.

After the CMSPageTemplate class has taken care of getting the content item ids set for each content control on the template, the content controls go to work rendering their content. Each CMS Control inherits from a CMSContent base class (which lives inside App_Code) that does all the real work of setting up variables for the child controls to use. The CMSContent class exposes a TextData property that contains the contents of the content item from the database (except for images, which live in the ImageData field and the TextData field is used to store the image type extension); the child class just uses this value to inject into a content literal or, in the case of the image control, into the image byte stream. The content control child classes just use their TextData and ImageData properties to get their content into the rendered output stream.

Don't Write Pages Your Website Can't Cache

Another item worthy of mention in the Tutor CMS is the caching ability we built into it. We get a lot of traffic, and we certainly don't want to be hitting the database every time DoRouting() checks to see if the requested URL exists in the CMS, or every time a CMSContent control needs to get its data to render. So, we created a simple caching strategy that's easy to understand and easy to work with.

When the website starts up (from an IISReset or a Web.config edit), we store load the CMS URL/Page/Template mappings and the Page Content Block Names/ID mappings in two static dictionaries at the application level. The routing and rendering system uses these static dictionaries unless preview=1 comes in on the query string of the requested URL, in which case we make a call to the database to get all those values.

Each rendered CMS Content control places the data it needs for rendering into the application cache the first time it renders so that subsequent renders will pull from the cache instead of the database. The image control behaves a little bit differently and takes advantage of ASP.NET's built-in output caching varied by the ContentItemId parameter, but the end result is the same.

All cached data gets reset at 6AM every morning.

When Is a Postback Not a Postback?

There's one funny behavioral side effect of our CMS: **Page.IsPostback doesn't work properly on pages rendered by the CMS**. This isn't usually an issue because we implement any pages that have processing logic in them as standard aspx pages, not as CMSed pages. However, there are a few pages that need to live in the CMS but contain controls on them that need to respond appropriately to postback events, like the SignInOrSignUp control. So, in these cases, you just have to get a little hacky and look at Request.Form.Keys to detect a postback from the Page_Load event, like this:

```
//IsPostBack doesnt work for url-rewritten cms pages, so we check for postback like this
if (!Page.IsPostBack && Request.Form.HasKeys())
{
        foreach (string key in Request.Form.Keys)
        {
                if (key.Contains(this.UniqueID))
                {
                        btnSubmit_Click(this, null);
                        break;
                }
        }
}
```

Along these same lines, we take advantage of ASP.NET's control rewriting ability to set the default action attribute of form elements that appear on CMS pages to the requested URL (instead of the rendered template URL, which would be used by default if we didn't take advantage of control rewriting). This is easily accomplished with a Form.browser file in the App_Browsers directory of the website and the corresponding HtmlTextWriter classes in Global.asax.

301 Redirect Flexibility

Another component that we built into the routing engine of the Tutor CMS is the ability to conduct arbitrary 301 redirects for any pair of URLs. For example, we may discover that a client is linking to their cobranded Live Homework Help page on our server using the wrong URL. Instead of waiting for them to correct the problem on their end, we can just create a new mapping in the WWW_Redirects table to tell our redirect system to respond to requests for /old/path with a 301 redirect header response to /new/path instead.

The redirect system looks at a table named WWW_Redirects which contains a simple two column structure of OldUrl and NewUrl strings to describe the mapping. Here's the part of DoRouting() that handles the 301 redirects:

```
if (WWW.Global.Redirects.ContainsKey(redirectUrl))
{
  //clear server error in case we're mid-error handler
  HttpContext.Current.Server.ClearError();

  //redirect and include input querystring
  //rmg: using HttpContext.Current.Response.Redirect() sends 302, not 301!
  HttpContext.Current.Response.StatusCode = 301;
  HttpContext.Current.Response.Status = "301 Moved Permanently";
  HttpContext.Current.Response.AddHeader("Location", WWW.Global.Redirects[redirectUrl] + query_vars);
  return true;
}
```

It's worth noting above that ASP.NET's default Response.Redirect() method uses a 302 HTTP response code, which isn't what we want, so we're writing the 301 response header explicitly for the browser.

The dictionary of redirect URL mappings is also cached at the application level and is reset every day at 6AM.