blend interactive

231 S. Phillips Ave, Suite 200
Sioux Falls, South Dakota 57104

605.334.7077
blendinteractive.com

Managing Content in the Transactional Application

by Deane Barker for Movable Type

February 19, 2014

Abstract

Transactional applications almost always have non-trivial content needs. These needs can often not be met by a traditional content management system (CMS) due to systemic conflicts and the potential for instability in the production environment.

This paper examines the usage of a small footprint, decoupled CMS such as Movable Type to service this content. This paper will explain:

- Why is managing content difficult in transactional applications?
- What solutions exist to manage this content?
- What challenges are presented when integrating managed content into a transactional application and how can they be overcome?

This paper is divided into two parts:

- Part 1 discusses the problem of managed content in transactional applications. This is an overhead view of the problem domain, suitable for product managers.
- Part 2 discusses a specific set of technical challenges and solutions using Movable Type and the MT.Net library. This section is intended for developers or software engineers.

About the Author

Deane Barker is a founding partner in Blend Interactive, a content management consultancy based in Sioux Falls, South Dakota. Deane has been implementing web content management solutions for almost two decades.

Part 1: The Problem

Defining the Transactional Application

In the world of the web, there's an important distinction between a web *site* and a web *application*.

The former is generally focused on providing *content*, which we can define as information specifically created by humans through editorial processes and intended for mass consumption by other humans. Web sites produce and manage content for primarily other ends – either an advertising supported business model, or the promotion of a marketing goal.

Clear examples of content-focused websites would be a corporate marketing site, a blog, a news or gossip site, etc.

Web *applications* exist to manage user-specific data which was generated from some other transaction. In these cases, a user is manipulating data created through some other process and intended specifically for them.

A classic example is an online banking interface. The information contained therein wasn't generated by humans (only indirectly, as a byproduct of a financial transaction), is not designed for mass consumption (indeed, consumption beyond the intended user would be considered a privacy breach), and is not designed to further any other goal.

Applications such as this can be roughly termed "transactional," as they concentrate on the manipulation/reporting of a data transaction, whether it be the transfer of money from one account to another, the registration of a user for an event, or the retrieval and display of the current weather conditions for the user's city.

Applications don't manage "content," so much as they manage "data." And the data they manage is often not subject to any editorial process – there is no creation, no editing, no approval, etc.

The Content Needs of the Transactional Application

While most of the information in a transactional application will be provided by the source domain (the database of banking transactions, for instance), there are invariably more traditional content needs spread throughout the application.

Consider:

- **Help Topics:** the application might require many pages of text- and image-heavy documentation.
- **Marketing Content:** the application might offer subscription options which require marketing content used to upsell users.
- **Label and Instructional Content:** the application likely has introductory headings, form element labels, copyright notices, error messages, etc.

These examples are not transactional data, they are *content*. This information needs to be created, revised, discussed, collaborated on, versioned, approved, and published. Collectively, these are editorial processes.

To manage this content, many applications build in rudimentary content management capabilities (some frameworks even provide this –"flatpages" in Django[1], for instance), but these generally functional poorly and suffer from being viewed as a necessary evil and not the prime focus of the development team. As such, they're almost always missing higher-level content management functionality such as granular permissions, workflow, versioning, localization, etc.

Content management systems exist to solve these exact problems. Why aren't these used to manage the content needs of a transactional application?

Problems with the Traditional CMS Model

Content management systems can be roughly divided into "coupled" and "decoupled" models[2].

- In a "coupled" environment, one CMS environment serves both editors and consumers – the same server on which the content is delivered is the server on which the content is edited. This means the CMS is actively installed and running on the production server.
- In a "decoupled" environment, the CMS exists apart from the production server. Content is edited and managed in one environment, then pushed into another environment for delivery. Indeed, the delivery server may just accept this content as flat HTML files with no knowledge of the CMS that generated it – there is an impermeable line between the CMS and the delivery server, such that the delivery server maintains complete ignorance of the CMS.

In today's CMS landscape, coupled options are far more common (especially on the .Net and PHP platforms, where decoupled systems almost unheard of).

Beyond the dearth of options, the coupled model is problematic when trying to manage content for a transactional application. In coupled architectures, the CMS (or at least some components of it) must be installed and run in the production environment.

Given that the modern coupled CMS is complex and computationally-heavy, this presents multiple problems when trying to co-exist in the same environment with another application.

- **Performance:** a coupled CMS introduces considerable processing overhead which is often incurred on every request

[1] "The flatpages app," https://docs.djangoproject.com/en/1.6/ref/contrib/flatpages/
[2] "Decoupled Content Management 101," http://gadgetopia.com/post/7206

- **Stability:** a processing failure of the CMS can bring the entire application down
- **Security:** a security exploit in the CMS can provide a foothold from which an attacker can gain access to the larger application
- **Licensing:** if the application runs across multiple servers, this often requires licensing (and paying for) the CMS multiple times
- **Resource Conflicts:** it's not uncommon for applications to "compete" for the management of inbound requests, database namespaces, file system resources, etc.
- **Complexity:** the introduction of another system to the runtime environment increases the time required for debugging and trouble-shooting issues.

Most transactional applications are complicated works of software engineering, requiring complete control over their processing domain, including the file system, the database server, and the web server itself. *The same can be said for the modern, coupled CMS.*

Clearly, trying to have both systems exist on the same server is problematic.

A Decoupled Solution

A proven method to removing the complication and instability is to manage content in a separate environment, then "inject" that content into the production environment. The goal is to inject it in such an innocuous method that conflicts are avoided.

The simplest, safest and most obvious method is to "bake" content into static, self-contained HTML files then push these files to the production environment (via file copy, FTP, or similar method). The files are then processed as simple HTML by the web server. When content is edited, the affected files are over-written. The production environment remains ignorant of how those files were produced.

Unfortunately, while quite safe, flat HTML has very little presentational agility. The content of each page is fixed at the time it's generated. This presents problems when it becomes necessary to react to situations at request time. Many elements on a page are contextual to both the user's behavior and the location of the content within its larger context.

Examples:

- If a user is in a certain usergroup (they have an active support contract, for example), additional content is available. New navigation options should appear in the menus alongside the default options.

- Based on the user's marketing profile, specific promotional content should display in the sidebars.

- Periodically, downtime notices need to appear above all content in the site for users in a specific geographic region.

Publishing static HTML files requires the publishing of content in *one* format to fulfill multiple presentation profiles.

There's a temptation to add code to these files (inline PHP, for example), which is executed at request time. However, these files might run outside of the core application process, thus excluding them from any information supplied by the application (for example: is the user logged in, in what security groups do they exist, etc.). If they run inside the process, they introduce more complexity and provide a method by which content editors could inject code into the core application. Neither situation is desirable.

The solution is to move from publishing static files to publishing content in a data-centric, presentation-free format that can be absorbed into the core execution process and manipulated for display at request time.

Part 2: The Solution

Enter Movable Type

Movable Type[3] is a well-established storied CMS dating to 2001. In continuous development since, it is an affordable, mature solution for managing decoupled content. Movable Type is written in Perl and can use a variety of databases.

For this paper, a proof-of-concept was created using Movable Type to serialize content into XML data and inject that data into an ASP.Net MVC environment for delivery alongside (presumably) a larger transactional application.

The results of this proof-of-concept have been published as the MT.Net project at Github[4], freely available under an MIT License.

[3] http://movabletype.org/
[4] https://github.com/MTUS/mt-net

Challenges and Solutions

There are dozens of possible ways to architect this solution, but some challenges will be universal, no matter the specific solution architecture. The following challenges were identified and answered during the prototyping process and are intended to be an instructive look at these problems and examples of solutions.

In what data format do you publish content?

Content should be published in a presentation-neutral format which allows for easy parsing and manipulation with supported tools. For this prototype, XML was selected as a serialization language as it's well-supported in the .Net framework through both traditional DOM parsing and LINQ-based querying.

Movable Type's templating architecture will render any text format, and tag modifiers are available to enable the secure formatting of XML content.

There's a temptation to simply serialize the entire content database as XML and inject a single monolithic file into the production environment. In some situations – perhaps those with smaller amounts of content – this might be desirable.

However, for the sake of processing efficiency, we elected to publish each content item (page or blog entry) as a single file, along with a single manifest file listing all the available content. The benefit of this model is that publishing new content becomes incurs overhead of N+1 – the number content objects changed, plus the manifest file.

How will content be transferred from the management environment to the delivery environment?

A number of methods could be used. Some examples:

- Movable Type supports multiple direct methods, including FTP, SFTP, and rsync.
- File system assets can be synchronized outside of the Movable Type process using common tools such as Microsoft's Distributed File System[5] or Unison[6].
- Movable Type is able to "notify" the publishing environment that new content has been published (via HTTP ping; see below), thus enabling a "ping and retrieve" scenario where the publishing environment proactively pulls new content through a web service, file copy, or other method. (There are some drawbacks to this method; again, see below.)

[5] "Distributed File System (Microsoft)," http://en.wikipedia.org/wiki/Distributed_File_System_(Microsoft)
[6] "Unison: File Synchronizer," http://www.cis.upenn.edu/~bcpierce/unison/

- Content doesn't need to be bound to the file system. A process on the Movable Type server could insert changed content into a remote database accessible to the publishing environment, for instance.

For our proof-of-concept, we left this question open by simply publishing directly into a file location accessible to the publishing environment.

How is an inbound request identified as a request for content, rather than application functionality?

This question lays bare the fact that two domains of information are sharing the same URL namespace. Which functionality "owns" a specific inbound request? Should the transactional application respond to the request, or should it be fulfilled from the editorial content?

If publishing flat HTML, it would be enough to simply allow the web server to fulfill requests normally. However, to deliver parsed data in-process, we need to find a way for the application to identify and respond to content requests.

Our solution was to implement a "filter" through which all inbound requests pass. Requests that are intended for content are re-routed to an MVC controller for handling, while other requests simply pass through to be handled normally.

ASP.Net provides an event-based request lifecycle which allows us to hook into the inbound request before any other processing takes place by attaching a handler to the appropriately-named "BeginRequest" event.[7]

A data structure holds an index of all valid content paths. Our request filter checks this structure for content which matches the inbound path. If matching content exists, the request is rewritten to a controller/action which retrieves and delivers the content. Requests for which no matching path can be found simply pass through the filter to be handled as normal.

To keep overhead as minimal as possible, we elected to cache the path database in memory, rather than do a lookup from any persistent datastore. A C# Dictionary object was used to store all available content paths as indexed keys. This Dictionary was cleared and re-loaded whenever new content was published.

Another benefit of the filtering method is instant "detachability." Removing all content functionality (for example, to improve performance, remove a security threat, or simplify debugging) simply means disabling the filter. With nothing evaluating inbound requests

[7] "Walkthrough: Creating and Registering a Custom HTTP Module," http://msdn.microsoft.com/en-us/library/ms227673(v=vs.100).aspx

for matching content, this functionality is removed instantly and completely without any other changes required to the underlying application.

How are bootstrapping activities initiated when content is published?
A content publishing event might require bootstrap activities, such clearing of caches (in the example above), re-indexing content for search, resetting analytics, etc.

To capture the content publishing event, Movable Type has a "ping" feature which will make an empty HTTP request to a specified URL after publishing is complete. To use this, we can simply specify this URL as one fielded by a controller in our application which initiates all necessary bootstrapping.

A disadvantage of this method is that, depending on your Movable Type configuration, the ping might not be a reliable publishing indicator. Movable Type has an alternate publishing method called the Publish Queue[8]. When using this method, the ping takes place when items are queued rather than published.

An alternate option would be to "watch" a file which is republished every time content changes (the manifest file, for instance). When this file changes, either initiate bootstrapping without a pending request, or clear a flag which is detected on the next inbound request, and initiate bootstrapping then.

(ASP.Net even supports this through its built-in Cache object. Items cached in memory can be bound to a file and invalidated when the file changes.)

How do you avoid path conflicts between content and application functionality?
Under the filtering architecture described, editors can create content at any path, which effectively gives editors the power to usurp any URL path accidentally. This re-surfaces the question who "owns" the URL namespace – the content or the application?

For instance, if the transactional application provides major account functionality through a controller mapped to "/account," an editor could inadvertently cripple the application by publishing content to that path. Since the request filter runs before any other processing, it will match content at that path and reroute the request before the application was aware of it.

The solution is to disallow the indexing content belonging to specified path patterns.

Configuration options were added to allow application developers to "protect" specified URL prefixes. Thus, the "/account" URL prefix can be reserved for the application alone.

[8] "Using Movable Type's Publishing Queue," http://movabletype.org/documentation/administrator/publishing/publish-queue.html

Editors can still publish content to conflicting paths, but this content would never be indexed, and would therefore be invisible to the request filter.

(This pattern matching was implementing by simple checking the beginning of the path string, but more sophisticated methods – such as regex parsing – are available. As discussed below, the paths were cached at publish time, meaning the frequency and consequent overhead of this operation is very low.)

How do you create easy, consistent, and manageable content handling at the template layer?

Template designers need to be abstracted away from the serialized content as much as possible, to avoid introducing any dependencies on serialization, and to ensure content is parsed and handled consistently.

Parsing of the content was moved "backwards" away from the template as much as possible. It can be done in the controller, but can also be moved even further backward into the request filter itself. As content has to be parsed to populate the path index, it introduces little overhead to simply store the parsed content object as the value of the path index.

The result is that the controller which displays the content receives a strongly-typed C# object representing the parsed content. This content object is injected directly into the template (the MVC View), ensuring consistent handling and ease of content manipulation by template designers. This also ensures the serialization and parsing format can be changed with no impact to existing template code.

How do you query the repository of content?

While templating individual content objects is important, at some point you need to query the repository to determine more global, contextual issues, especially when rendering navigation.[9]

This includes queries such as:

- What content is available?
- What content is spatially related to Content X? (For example: parent, child, or sibling content.)
- What content is available "below" a specific path?

[9] "The Necessity of a Content Index," http://gadgetopia.com/post/8041

This is addressed by creating a “repository” object which stores a strongly-typed reference to every content item. Using .Net’s LINQ query facilities, we are able to build methods that allow code to traverse and freely querying the “tree” of content in order to do such things as render navigation. The content repository contains a reference to the hierarchy of content based on its path, and can answer global questions (the retrieval of all content at the top level, for instance).

Properties were added to the individual content item class to allow the navigation to spatially-related content – its parents, direct children, siblings, ancestors, and descendants.

Conclusion

Decoupling the management of editorial content from your transactional application provides clear benefits to stability, security, and performance. Unfortunately, decoupled CMS solutions tend to be expensive, and running the resulting content as part of an integrated whole is difficult.

Movable Type and MT.Net combine to present a cost-effective solution for deeply integrating managed content into a transactional application in such a way that the content doesn’t compromise the runtime performance and appears to be completely native to the application.

MT.Net

The proof-of-concept as described is freely available as “MT.Net” at Github:

https://github.com/MTUS/mt-net/

Included is the core C# library for managing the generated XML, sample templates for the Movable Type installation, a sample website implementation, and sample data to test with. Upon download, it should run directly out of Visual Studio.

This code is open-source under the MIT license.

It is to be considered concept code only. It has not been deployed to a production environment and should be formally tested before use.