

CONTENT MANAGEMENT FROM SCRATCH:  
A STUDY OF FORM, FUNCTION, AND BEST PRACTICES

A CREATIVE PROJECT  
SUBMITTED TO THE GRADUATE SCHOOL  
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS  
FOR THE DEGREE  
MASTER OF ARTS

BY

BRIAN M. BAUMAN

CHRISTOPHER A. FLOOK - CHAIRPERSON

BALL STATE UNIVERSITY

MUNCIE, INDIANA

MAY 2012

## **Introduction**

One thing I noticed while earning my bachelor's degree was that I was regularly receiving more job offers related to my minors of computer science than to my telecommunications major. These employers valued the ability to analyze a project from the design and development perspectives simultaneously, as well as being able to communicate effectively between the groups assigned to those tasks. As the focus of my major was audio production, I wanted to diversify my education and expand my knowledge in the overlap between the two fields.

Enter the *Historic Muncie* project.

Beginning in the fall of 2011, a team of telecommunications students, history students, and news students began the process of cataloguing various historical buildings and architectural styles in Muncie, and producing documentaries and short vignettes of the various districts in the city. The ultimate goal of this project was the creation of a customized content management system (CMS) to house this documentation online. While the content and the general user interface would be designed and produced by teams of students and faculty, the implementation, architecture, and software were my jurisdiction. My tasks were to build a well-structured site with which users could browse or search for the sorts of buildings in which they were interested, and obtain additional information about specific buildings or other related buildings. They could also learn about the development of Muncie as a whole and about the different districts and styles of the city.

## **Architecture**

The architecture of the CMS was designed first around a linchpin of versatility. The AMP stack - consisting of the Apache webserver, the MySQL database, and the

PHP server-side processor - is one of the most commonly available configurations for either Linux or Windows servers. It is versatile, easy to use, and simple to set up and maintain. It is not without weak points, however. The object-oriented implementation in PHP is haphazard, naming conventions are inconsistent, and weak typing allows for many run-time errors to go unnoticed. As the goal was to construct the CMS from scratch, no libraries outside of the basic PHP API were used.

A traditional three-tier architectural approach was used. I knew that effective data querying and packaging would be essential, necessitating clean and robust application and database layers. In previous projects I've worked on, often the view layers can be mixed into the application logic with minimal negative effect, but in this case I knew the viewers would need to display the same data in numerous ways, meaning that separating the viewers into their own layer made a lot of sense. The additional complexity of AJAX clinched this decision - integrating AJAX with a merged application and view layer would quickly become a mess of unmaintainable code.

The file structure was centered around keeping similar parts of the code together. The application and view layers were nested in packages devoted to specific objects, all of which were placed in an "includes" folder in the server document root. For example, all files dedicated to manipulating and displaying districts was placed in "includes/districts". This made it simple for a district file to reference other district files in the same folder as needed, and a touch more difficult to reference files in other packages, such as "user/user.php".

Assets were sorted by type, then by application. For example, "images/district" contained all the district-related images for the site, whereas "video/" contained the various encodings of the documentaries for the project.

## **Development**

Generally, the development cycle was based around the completion of complete features as discrete entities. The user interface for a given feature was usually worked out first, as going through the process of determining how a user would actually make use of a feature firmly solidified what that feature needed to accomplish precisely. Once the skeleton was in place, the back end was written to satisfy the needs of a user interacting with the interface. Finally, the two were stitched together, completing the front end by tying it properly with the back end.

## **The Editing Paradigm**

In building any content management system, it is critical to decide how a user will go about adding, updating, and deleting data. The possible approaches can generally be broken down into two main categories - inline editors and administrator panels.

An administrator panel is the traditional, simplest approach. It presents the editable information succinctly, and is simple to code. Generally it looks nothing like the final result, however. Inline editing has the nice advantage of allowing users to navigate through the website as usual, and simply editing any information they wish. The downside to this approach, however, is that it requires a bit more coding work to accomplish. I decided to give it a try, though, as I had already used administrative pages on previous web pages.

The end result was a bit rough - but serviceable - and the process taught me a fair bit about building an AJAX-based update system. One crucial lesson was the necessity for spreading out the process into multiple files with a clear naming convention. Any given AJAX implementation, for example, will start with one core web page, usually written in PHP. The AJAX activation requires a JavaScript function, which

for simplicity's sake should be in its own file. This function must then load a PHP page, which it inserts into the proper spot on the *original* page. Keeping all this straight required a consistent naming convention: all the original web page file names were prepended with "view". The file with the JavaScript was prepended with "update". Finally, the PHP page's filename was appended with "Builder". This tactic simplified the entire process and made the implementation of AJAX less complicated.

### **Designing the Structure Browser**

Early in the design process, I faced an interesting logistical issue. While the initial release would only have a few dozen structures available for viewing, ultimately more than a thousand entries were planned. We needed a method of filtering, searching, and narrowing down those thousand structures in a manner that was intuitive and effective - essentially we needed an easy way to handle large data sets. I felt right from the start that more traditional list-based approaches would fall horrendously short of what we needed. For starters, the site was intended to be media content-heavy. Starting users off with thousands of text entries or a Google-style search was the antithesis of the sort of experience I was attempting to create. Displaying such a vast quantity of results would also require either an unreasonably long page length or splitting up the results into page after page.

We also briefly considered the idea of a browse-based approach, in which we required people to make selections in various categories to "burrow" deeper and deeper into the site until they reached a set of buildings small enough to display. This approach came with a myriad of problems of its own, however. For one, it's a cumbersome method of filtering a data set - requiring multiple pages of user decisions to reach a useful result. Additionally, it completely ignores the needs of the casual user by

discouraging experimentation. The casual user is there to explore the content that is available - requiring them to make an exclusive choice between a dozen different architectural types (which they may or may not have heard of before) is unreasonable.

The design we ended up using was a grid-based, AJAX-driven solution that drew its inspiration primarily from two sources. In the year prior to beginning work on the *Historic Muncie* project, I had done some investigation of Silverlight's PivotViewer<sup>1</sup> interface, a seamless way to work with thousands upon thousands of units in a fluid grid-based environment. As filters and sorting rules were applied, the individual grid items would disappear or reappear as needed, and arrange themselves into various formations to allow the user to easily visualize the data sub-set they had created. The graceful manner in which the system scaled caught my eye, and though the project I was working on at the time ended up using a different system to display its data, the concept stayed embedded in my head.

The second inspiration for the filter design was, unexpectedly enough, the character selection interface from one of my favorite games - *League of Legends*<sup>2</sup> (See Appendix C). Controls on the left allow for a variety of filters to be added and the grid view on the right allowed the 100+ champions to be displayed efficiently in the space.

The display system that resulted from the blending of these two systems has numerous benefits. The most obvious benefit is the improved customizability for the user. Rather than searching for terms or exclusively selecting values for options, it permits complicated mix-and-match queries. The near-instantaneous results from these queries are more useful, as it allows users to gain an intuitive sense of the size of the set - rather than merely an abstracted number, the space itself grows to accommodate the

---

<sup>1</sup> <http://www.microsoft.com/silverlight/pivotviewer/>

<sup>2</sup> <http://na.leagueoflegends.com/>

structures to be displayed. Comparing the relative popularity of two architectural styles, for example, is as simple as checking one box, then unchecking it and checking another. The grid view also allowed us to exploit the beautiful pictures we have of each building, creating a more engaging experience for the user.

### **Designing TinyGallery**

The first library I had to build from scratch was a package to display the variety of photos each structure was sure to accumulate over the course of the project.

Architecturally, I decided to treat TinyGallery as a stand-alone library. The goal was to have a library that I could transfer to various other projects that needed a photo gallery. To this end, I tried to build the library around a very simple interface - simply passing an array of Photo objects to the library would be enough to populate the entire gallery. The first image passed in is displayed in large format, and underneath it a bar of thumbnails stretches across. Upon clicking a thumbnail, the featured image is replaced with a large format of the thumbnail image (See Appendix D).

Future additions are planned for this library when the need arises. Currently, we don't have enough pictures to necessitate a scrolling feature for the thumbnail bar, but it will be added as more pictures are generated. Unfortunately, it may be necessary to tether the library more closely to the overall application, as the inline editing scheme would require a way to update a new photo directly within the library, requiring interaction between the library and the authentication package, as well as some additional backend tools.

## Designing SoftFocus

The second library I built for the CMS is SoftFocus, a utility that darkens the page and displays various assets on top of the content already there. The library currently accepts video content, but can be extended to accept most anything. The implementation was actually more complicated than I originally thought, however, because there were numerous small bugs that needed to be resolved.

To start, it was necessary to make a translucent black box that covered the entire screen - that wasn't too difficult. Turning SoftFocus on and off is as simple as calling JavaScript functions to show or hide the box. Adding the content, however, turned out to be tricky. Browsers have an interesting design choice in that clicking on a specific element *also* counts as clicking on that element's parents. Put simply, this meant that clicking "play" on a video player inside the translucent black box *also* registered as clicking on the black box itself - which closes SoftFocus. The workaround was a few lines of code to check the precise origin of the mouse click, and only close the black box when the click occurred directly on it.

Once I had fixed that problem, I realized that it was possible to close a SoftFocus containing a video that was playing - and the video would continue playing. This required modifying the code to remove all content upon the closing of SoftFocus.

## Document Rewrite

Document rewriting is a nearly integral part of creating a user-friendly experience these days. This has largely to do with the trade-off between programmatic and user-friendly naming conventions. From a software perspective, an HTTP request to display a specific structure makes the most sense in the following format: "structures.php?id=x". For a user, however, such a URL is unmemorable, if not downright ugly. Far preferable

for a user is something along the lines of “structures/x/{structureAddress}”. Document rewriting, then, is the process of converting URLs of the latter form to URLs of the former.

The most effective method of doing this is by modifying the .htaccess server configuration file, and using a regular expression to filter the modification. For example, the following rule accomplishes the rewrite discussed in the previous chapter:

```
RewriteRule ^structures/([0-9]+)/.+ $ structure.php?id=$1 [NC,L] #Structure URLs
```

The rule is made up of five main portions. The first part, “RewriteRule” specifies that the command is intended to rewrite a URL - as document rewriting isn’t the only configuration that the .htaccess can alter. The next part, “^structures/([0-9]+)/.+” defines the pattern to be matched with a regular expression. The “structure.php?id=\$1” expresses the desired rewrite format. “[NC,L]” is a pair of flags defining how the rule is to be applied. Finally, “#Structure URLs” is a comment reminding me what the rule does.

Overall, about a dozen different rewrite rules were used to improve the display URLs for the site. The rewriting also had secondary, unintended benefits - changing the display URLs caused things like relative CSS paths to fail, forcing me to adopt a more robust “from-site-root” linking scheme. Additionally, as I went through my link generators to change the site links to the more user-friendly versions, I realized that (for example) I had identical links to structure pages being generated in at least three different places. The better approach, which I quickly adopted, was to make the structure object generate its own link upon creation, which could then be easily used by any portion of the site.

The end result is the ability to cut out a lot of code and update what remains quickly and easily.

### **Reskinning the Site**

About a month prior to publication, acting client on the project Chris Flook told me that he wanted to significantly modify the look for the website. While a full redesign can be a daunting task, I was excited to give it a try, because it would be a fantastic test of the code's modularity. A well designed, modular site controls the amount of data flowing back and forth between tasks, allowing different parts of the software to communicate with each other using small, manageable packets of information. By keeping the data packets small, it becomes fairly easy to rip one portion of the software out and replace it with a better implementation that uses the same small data packets.

In the case of CSS, it's a great benefit to be able to completely redesign the website without modifying the constructing code. Ideally, the HTML generated carries few implications regarding how it is to be displayed, and that task is carried out entirely by the CSS. Further, it should be trivial to completely alter the look of the site by swapping out CSS files and modifying nothing else.

In making the modifications necessary to implement the new design scheme, I was pleased to note that in the vast majority of cases, only alterations to the CSS were necessary. This, I felt was evidence that I'd done a good job separating the form of the CMS from the function (See Appendix E).

## APPENDICES

### Appendix A: Physical Equipment

- “Melkor” custom workstation
  - Intel Core i7 920 @ 2.67 GHz
  - 6 GB memory
  - 2x 1TB drives, RAID 1

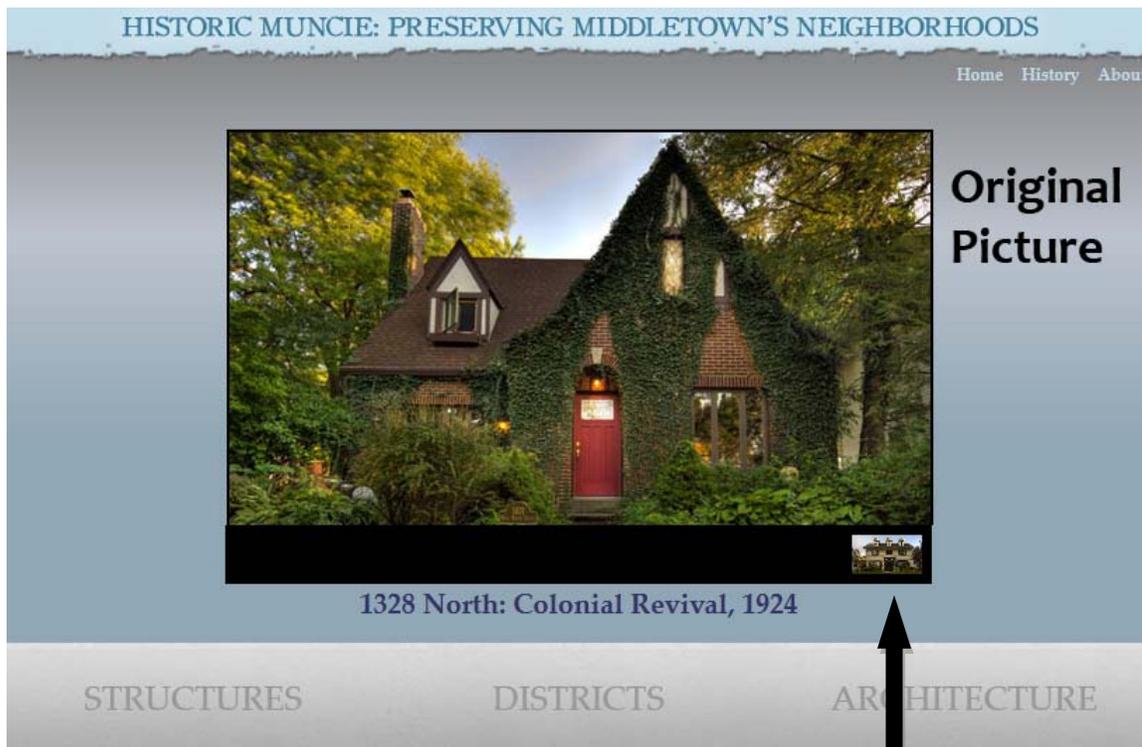
### Appendix B: Software

- XAMPP 1.7.7 - AMP Package
  - Apache 2.2 - HTTP Server
  - PHP 5.3.8 - Script Parser
  - MySQL 5.5 - SQL Server
  - FileZilla 0.9.93 beta - FTP Server
  - PHPmyAdmin 3.4.5 - Web-based SQL Front End
- Adobe Dreamweaver CS 5 - IDE
- Adobe Photoshop CS 5 - Image Manipulation
- Miro Video Converter 2.5 - Video Conversion
- SimpleTest - PHP Testing

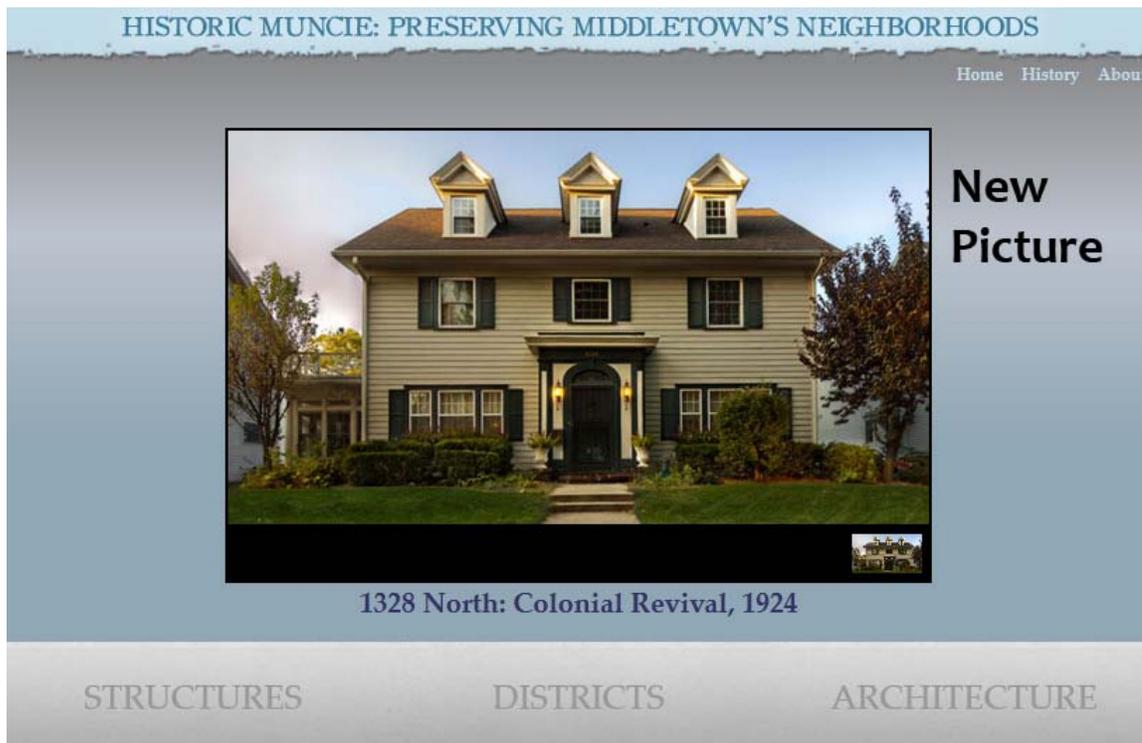
## Appendix C: League of Legends Champion Selection Screen



### Appendix D: TinyGallery Implementation



Click Here



Appendix E: Reskinning



Figure 1 - Version 3 Design

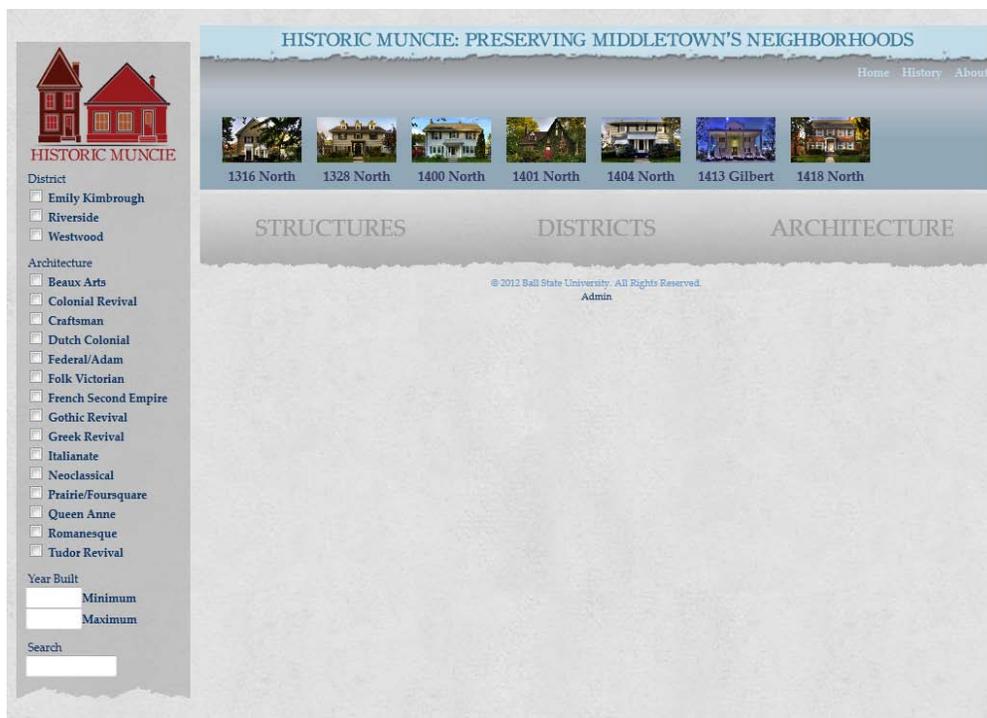


Figure 2 - Version 4 Design - Same Code

**Appendix F: Administrator Guide**

Managing assets as an administrator is a straightforward task in the CMS. At the bottom of every page is a link to an administrator log in page. Once a correct username and password are entered, a log out button appears in the upper right of the main header on each page. The page also refreshes to allow the administrator to modify any global settings permissible. To edit content on the site, a user need only navigate through the site as usual while logged in. A new placeholder element will appear in any list of structures, districts, etc. Clicking this element will allow the administrator to add a new content item of that type.

To modify an element, navigate to it as normal. Any aspects that can be edited will appear within input boxes, and a feedback display will appear above the content. Modifying an item in one of the input boxes will send a change request to the server, and update the feedback display with the success (or failure) of the change.

## GLOSSARY

*Apache*: The most popular HTTP server in the world, a software package that delivers web pages in response to requests from browsers.

*AMP Package*: A collection of utilities containing a distribution of Apache, MySQL, and PHP (or Python or PERL), preconfigured for easy setup and maintenance, especially in a testing environment. An AMP will usually be referred to as a WAMP (Windows), a LAMP (Linux), or a MAMP (Mac OS), depending on the underlying operating system it is configured for.

*Asynchronous JavaScript And XML (AJAX)*: A technique that utilizes JavaScript to send data to a server and update a portion of the page with new information.

*Back End*: The portion of software devoted to manipulating information and updating records.

*Content Management System (CMS)*: A web-based software package dedicated to updating, editing, and displaying content quickly and easily.

*Cascading Style Sheet (CSS)*: A set of instructions an HTTP server sends to a browser to allow it to properly display the HTML sent by the same server.

*Child:* An element that exists inside another element.

*Element:* A single item in a web page.

*File Transfer Protocol:* A method of communication that FTP servers and clients use to send files to each other.

*Front End:* The portion of software devoted to creating the user interface, and controlling how the user can interact with it.

*Hypertext Markup Language (HTML):* A language of descriptors used to label content and allow browsers to display it a way that makes sense.

*Hypertext Transfer Protocol (HTTP):* A method of communication that HTTP servers and web browsers use to send web pages and data to each other.

*Integrated Development Environment (IDE):* A software program that provides multiple tools to facilitate and expedite the programming process.

*JavaScript:* A scripting language that runs on a web browser.

*Library:* A collection of code to complete a specific function, built to be easily transferred and integrated into any program.

*MySQL:* A common, lightweight database used for storing information.

*Object-Oriented:* A programming paradigm that emphasizes modeling programmatic elements after their real-world counterparts.

*Package:* A collection of code grouped together for easy reference, maintenance, and modularity.

*Parent:* An element that contains other elements inside of it.

*PHP Hypertext Preprocessor (PHP):* A scripting language that runs on an HTTP server.

*Regular Expression:* A syntax for describing patterns of text to be compared to text samples.

*Run-time Error:* An error that manifests when a program is run, rather than when the program is written.

*Structured Query Language (SQL):* A language for requesting specific data sets from a database.

*User Interface (UI):* The portion of the program that the user can see and interact with.