O'REILLY®

Early Release
RAW & UNEDITED

Web Content Management

SYSTEMS, FEATURES, AND BEST PRACTICES

Deane Barker

Web Content Management

Deane Barker

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'REILLY®

Web Content Management

by Deane Barker

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Allyson MacDonald
Production Editor: Nicole Shelby
Copyeditor: FIX ME!
Proofreader: FIX ME!

Indexer: FIX ME!
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Rebecca Demarest

September 2015: First Edition

Revision History for the First Edition:

2015-03-30: Early release revision 1

2015-06-05: Early release revision 2

See *http://oreilly.com/catalog/errata.csp?isbn=9781491908129* for release details.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-1-491-90812-9

[?]

Table of Contents

Preface

Back in 1995 or so, I wrote my first HTML document. I wrote it in Notepad, and I still remember adding a TITLE tag, refreshing the page in Internet Explorer, and watching with awe as my document title filled the title bar of the browser window (the idea of tabbed browsers was still years in the future, so the document TITLE became the entire title of the window).

That first web page quickly grew into an entire website. Mainstream adoption of CSS and JavaScript was still a few years off, so I didn't have scripts or stylesheets, but I had a handful of HTML files and a bunch of images (you were nobody if you didn't have a tiled, textured background on your page of links).

With this, I ran smack into the first problem of "webmasters" everywhere: *how do I keep track of all this stuff*? I don't even think the word "content" had been popularly applied yet — it was all just "stuff."

As websites inevitably grew, so did all the stuff. Since we were largely bound to the file system, we had a copy of everything on a local machines, that we would FTP to our server. Huge problems resulted if you had more than one editor, with two people trying to manage files, they would inevitably get out of sync and be unintentionally overwritten.

Additionally, the process of managing a website was enourmously tedious. Linking from one page to another assumed the two pages would always exist, yet broken links were common, and if you decided to reorganize the structure of your site or rename pages, you had to hunt through all your files to find where the previous name might have been used.

(The most valuable thing in my toolkit might have been a global search and replace utility that would let me look for — and correct — links in hundreds of files at once. Sadly, it had no backup or undo. Remind me sometime to tell you the story of how I accidentally did a irreversible find and replace on the single letter "e.")

Fast forward almost 20 years, and web technologies have since evolved to remove most of the tedium. Today's web manager largely works with abstract notions of content, without needing to understand the underlying technology, file system, or programming languages.

But even abstracting content from technology has still left us with eternal problems to solve: how do we structure and model our content? How do we allow for its organization? How do we search it? How do we work together on content without causing conflicts?

These transcendent problems exist "above" content management systems. They are the core, domain level problems which this genre of software is tasked with solving. There are hundreds of options available, most of which claim to be the correct choice for various problems. Open source advocates push a philosophical agenda, while commercial options have their eyes fixed firmly on a license fee. How can anyone know which option is right for solving their specific problem?

What You Need to Know About This Book

Who Is This Book For?

This book is an attempt to approach web content management from the outside, without pushing any particular technology or methodology. Thus, this book is designed for readers who want to understand the larger context in which a specific content management system might work, or understand the transcendent content management problems that any particular system will need to solve.

These readers might be:

- Project managers tasked with managing the implementation of a new CMS
- Experienced developers new to content management in particular
- Web managers embarking on an evaluation project to acquire a new CMS
- Content producers transitioning from offline content to web content management in a CMS
- CMS developers wanting to step outside their chosen platform and look at their discipline from a new perspective
- Anyone trying to understand and justify a new CMS purchase

What is Not in This Book?

This book is *not* a technical programming manual. There will be very few (likely zero) code samples.

Additionally, this book is intended to be language- and platform-agnostic. I will discuss many different systems, technologies, and platforms. I neither explicitly endorse nor condemn any of them. I have made an attempt to draw examples and screen captures from a wide variety of systems.

As a consultant working in this field for almost two decades, I assure you that I have many opinions. However, I also understand that even a system I might loathe still has fans somewhere who must see something in it that I do not. As such, I will do my best to keep my opinions to myself.

I also understand that — like any consultant — my experience is likely biased toward one type of project or problem in ways that I might not even notice. Just like you can't walk a mile in another man's shoes, I can't completely relate to problems (perhaps *your* problems) that I have never been tasked with solving. Systems I find enormously lacking for projects I have worked on might be entirely appropriate for the problem in front of you.

If you are looking for specific opinions or recommendations, I refer you to one of the many CMS selection consultants and analysts working in this space.

How is this book organized?

This book will be grouped into three parts:

- *Part 1: The Basics*. This lays the groundwork for the larger discussion of content management. We'll talk about what content is, paradigms with which to compare different systems, the roles that make up a CMS team, and how your organization might acquire a CMS.
- *Part 2: Feature Analysis*. This part will analyze the major functional areas of a modern CMS — how do they model content, aggregate content, coordinate workflow, manage assets, etc.
- *Part 3: Implementation Best Practices*. This final part will discuss the scope and sructure of a CMS implementation project, and the best practices and process of running one successfully (or even just surviving one).

A Note on Nomenclature

As we'll discuss in the very first chapter, "content" and "content management system" can mean many different things, from HTML files to images to Word documents.

I will use the terms "content" and "CMS" loosely for convenience. However, understand that this is a book about *web* content management specifically, so I'm talking about "web content" and "WCMS."

In dropping the “web” and “W” qualifiers, I am not staking claim to content as purely a web asset or problem. Rather, I am merely bowing to convention and brevity.

PART I
Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/oreillymedia/title_title*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Book Title* by Some Author (O'Reilly). Copyright 2012 Some Copyright Holder, 978-0-596-xxxx-x."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of plans and pricing for enterprise, government, education, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://www.oreilly.com/catalog/<catalog page>*.

To comment or ask technical questions about this book, send email to bookquestions@oreilly.com.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

CHAPTER 1

What Content Management Is (And Isn't)

We tend to look at content management as a digital concept, but it's been around for as long as content has. For as long as Mankind has been creating content, we've been searching for solutions to manage it.

The library at Alexandria was an early attempt at managing content. It preserved content and presumably controlled access to it. Librarians could be legitimiately considered to be the first content managers.

The information architecture work of S.R. Raganathan, an Indian mathemetician from the early part of last century, is another example — Ragnathan found new ways to structure and organize content to make it easier for people to find and work with it.

So, the need for content management didn't begin with the World Wide Web, but simply shifted into fast forward when the web was born in the the early 90s. At that moment, the ability to create and publish content tumbled down from its ivory tower and into the hands of the masses. Almost anyone could create a web page about virtually anything.

Content subsequently exploded. I was a college student at the time, and was slightly obssessed with James Bond. Suddenly, I could find reams and reams of information on 007. The amount of sheer trivia was staggering. Of course, I attempted to print most of it, because if it wasn't on paper, how would I manage it?

It wasn't long before I created my own James Bond website (*Mr. Kiss Kiss Bang Bang*, which for some years was at *http://ianfleming.org*), and I learned that keeping track of content was a challenge. An article only existed as an HTML file in the web server's root directory at any given time. I wasn't an IT professional back then, so the only backup outside that file was the one on my local Gateway 2000 Pentium tower. There was no versioning or access control — one fat finger mistake and the entire thing could be gone.

This struck me as as … dangerous. Even then, I knew that a website is essentially a content-based business, and with nothing more than a bunch of files lying around, I was effectively performing without a net. Content was my main business asset, and I remember thinking it was so brittle; one unforseen problem and it could simply "blow away" like a dandelion in the wind.

Additionally, each HTML file was a mass of mid-90s era markup, complete with nested TABLEs and FONT tags all over the place. There was no way to separate what was content and what was presentation, and each redesign of the site (there were many) involved manually reworking many of these files. Server-side includes helped to a certain extent, but each file was still a massive glob of mixed content and formatting code.

Sometime later, I was working for The Microsoft Network as a forum manager for The World of James Bond. We were sill writing HTML files in text editors, but Microsoft had introduced their content managers to the wonders of Visual Source Safe, which was a long since-deprecated source code management system. It provided backups, versioning, and file locking.

This clearly made us safer from a risk management perspective, but there was a mental shift too. We had a safety net now. The content we were creating had solidity to it. There was history and context. Content didn't exist only in simple files, but it lived inside a larger system which provided a set of services to protect and safeguard it. We had gone from hiding money inside our mattress to depositing it at an FDIC-insured financial institution.

Finally, at some crude level, my content was *managed*. Without knowing it, Visual Source Safe effectively became my first content management system.

A lot has changed since then, but let's start at the beginning. Along the way, we'll hopefully answer:

1. What is *content*?
2. What is content *management*?
3. What is a content management *system*?
4. What are the different types of content management systems?
5. What does a content management system do?
6. What *doesn't* a content management system do?

What is Content?

Many people have tried to draw a distinction between the fuzzy concepts of "data," "information," "content," and even "knowledge." Bob Boiko, in this seminal work *The*

Content Management Bible, dedicated the entire first part of his book to it — some five chapters and 61 pages.

We're not going to go that far, but we'll summarize by simply differentiating between content and raw data. This is likely the highest value return we can get out of the question, by framing it as: how is content management any different from managing any other type of data?

There are two key differences:

1. Content is created differently
2. Content is used differently

Created by Humans via Editorial Process

Content is created through "editorial process." This process is what humans do to prepare information for publication. It involves modeling, authoring, editing, reviewing, approving, versioning, comparing, and controlling. The creation of a news article, for example, is labor-intensive, time-intensive, and highly subjective.

The creation of content pivots largely on the opinions of human editors:

- What should the subject of the content be?
- Who is the intended audience of the content?
- From what angle should the subject be approached?
- How long should the content be?
- Does it need to be supported by media?

Despite significant advances in computing, these are not decisions a computer will make. These are messy, subjective, imperfect decisions which pour forth from the mind of a human editor sitting behind a keyboard. A small deviation in any of them (the proverbial flapping of a butterfly's wings…) can spin a piece of content in an entirely different direction.

Compare this to the creation of the record of a retail sale. There is no editorial process involved with swiping your credit card across a terminal. Furthermore, the data created is not subjected to any other process, will not be reviewed and approved, and is not subjective. The transaction happened in an instant, a historical record was created, and that's that.

The sales transaction is designed to be as repeatable and devoid of subjective opinion as possible. It will never be edited (indeed, to do so would likely violate a policy and/or a law). It is cold, sterile, and inert by design.

The news article, by contrast, might be tinkered with by multiple editors. The news article is subjective, and open for evaluation and interpretation. It has nuance. Compared to the sales transaction, it's downright artisinal.

Consequently, management of these two pieces of data is quite different.

Intended for Human Consumption via Publication

Content is stuff we create for a specific purpose: to publish it with the intention of it ultimately being consumed by other humans. Sure, it might be scooped up by another computer via an API and rearranged and published somewhere else, but eventually the information is going to make its way to a human somewhere.

Our sales transaction has no such destiny. It was created as a backwards-looking record of a historial event. It will likely not be consumed by someone in the future, except in aggregate through reporting of some kind. It may be retrieved and reviewed, but only by necessity and likely on an exception basis.

Our news article, by contrast, was created as a forward-looking item to be published in the future and consumed by humans, through whatever channel (perhaps more than one). It might be repurposed, abbreviated, rearranged, and reformatted, but the ultimate goal for it is to be consumed and evaluated by another human being.

Our content has value in the future. It might be consumed for years (even centuries or millenia), and can continue providing value to the organization far into the future. Every time our article is read, or every time a new employee reads the payroll policy, there is a benefit imparted to the content creator.

Content is an investment in the future, not a record of the past.

A Definition of Content

Bringing these two concepts together, we arrive at a concise defintion:

Content is data produced through editorial process and ultimately intended for human consumption via publication.

This definiton also points to a core dichotomy of content management: the difference between (1) management and (2) delivery. Content is created and managed, *then* it is published and delivered. The two disciplines require different skills and mindsets, and the state of current technology is creating more and more differences every day.

We'll revisit this two-sided approach to content management throughout this book.

What is a Content Management System?

A content management system (CMS) is a software package that provides some level of automation to the tasks required to effectively manage content.

A CMS is usually (though not always) server-based, multi-user software which interacts with content stored in a **repository**. This repository might be on the same server, as part of the same software package, or on a separate storage facility entirely.

A CMS allows editors to create new content, edit existing content, perform editorial processes on content, and ultimately make that content available to other people to consume it.

Logically, a CMS is comprised of many parts. The editing interface, repository, publishing mechanisms, etc. might all be separate, autonomous parts of the system behind-the-scenes. However, to a non-technical editor, all of these parts are generally viewed as a single, monolithic whole: "the CMS" (this is especially true of web content management as opposed to other types of content management, as explained below).

The Discipline vs. The Software

What's important to note is that a "content management system" is a specific manifestation of *software* designed to enable the *discipline* of content management. Just like a Ford Taurus is a specific manifestation of a device enabling personal transportation, ModX, Concrete5, and Expression Engine are a specific manifestions of software enabling content management.

The discipline of content management — the theories, best practices, and accepted patterns of the field — transcends any specific system. In this sense, it's platonic ideal: an abstract, subjective representation of how content is to be managed.

The specifics of this ideal can be very different depending on the experiences, preferences, and needs of the observer. This means there's no single, accepted defintion for content management as a discipline, just a set of debatable best practices.

Additionally, this means that skill with a particular content management system can be somewhat transferable. Even if System A differs from System B in extreme ways, they both still need to solve transendent problems of the discipline, like workflow, versioning, publishing, etc. While specific technical skill might not transfer, working with a content management *system* requires the exercise and development of skill in the content management *discipline*.

So, a CMS is a tool to assist in and enable the theoretical ideal of content management. How well any one CMS successfully brings that ideal to life is the subject of great debate and Internet flame wars.

Types of Content Management Systems

Previously we defined content as "information created through editorial process and intended for human consumption." Note that there was no mention of the web in this defintion (nor of the Internet itself, really).

However, given that this is a book about *web* content management, it's probably best that we define some different flavors of content management rather than lump them into one big bucket.

The "big four" of content management might be separated as:

- *Web Content Management (WCM):* the management of content primarily intended for mass delivery via a website
- *Enterprise Content Management (ECM):* the management of general business content, not necessarily intended for mass delivery or consumption (e.g.: employee resumes, incident reports, memos, etc.). This flavor was more traditionally known as "document management," but the label has been generalized over the years.
- *Digital Asset Management (DAM):* the management and manipulation of rich digital assets such as images, audio, and video for usage in other media
- *Records Management:* the management of transactional information created as a byproduct of business operations (e.g.: sales records, access records, etc.)

Clearly, the line blurs here quite a bit. A DAM system (an acronym which is the source of countless jokes…) is often used to provide content for a website through integration with a WCM. Furthermore, some ECM systems have system by which they can publish some of their information to the web.

So, the definitions are loose and many software systems are known only through their *intended* use and their perception in the industry. Drupal is well-known as a WCM system, but there are undoubtedly organizations using it to manage internal, enterprise content. Conversely, Documentum is a ECM system, but some organizations might use it to deliver all or part of their websites.

What Does "Enterprise" Mean?

You see that word a lot as part of the phrase "enterprise software" or "enterprise content." It has no precise definiton, but it generally means "big" or "intended for large organizations."

It's a vague term, and there is no opposite — few CMS would describes themselves as "provincial" or "boutique." And nothing is stopping the world's smallest CMS from describing itself as "enterprise" either. It's highly subjective — what's big to one person, is small to another.

Vendors use the term to indicate that their system can handle large amounts of content, or fit into a very distributed, sophisticated hosting environment (multiple load-balanced servers across multiple data centers, for instance)

"Enterprise content" is often used to refer to internal content which is not usually published outside the organization. A news article release is not "enterprise content," whereas a sexual harrassment report would be.

An "Enterprise Content Management" system is vaguely accepted to mean a system which is designed to manage this type of internal organizational content. They are consequently heavy on management tools and light on publication tools.

DAM is interesting in that it is differentiated primarily on what it *does* to content. While almost any content management system can store video and image files, DAM goes a step further by providing unique tools to render and transform digital assets. Images can be mass resized and video can be spliced and edited directly inside the system, making a DAM system's point of differentiation is one of processes that can be applied to content. Therefore, the core management features of a DAM system overlay quite closely to those of an ECM system, with the DAM system layering a level of functionality on top of that. (Indeed, many DAM systems are sold simply as add-ons to ECM systems.)

Additionally, there are other, even blurrier shades of gray. Some examples:

- *Component Content Management (CCM):* management of extremely fine-grained content (sentences or paragraphs) used to assemble documentation or highly technical content
- *Learning Management System (LMS):* management of learning resources and student interaction; most colleges and universities manage student interaction and class participation via an LMS
- *Portals:* management, aggregation, and presentation of multiple streams of information into a unifed system

Again, the lines here are very blurry.

Very little an LMS does is specific and unique to an LMS. Many different WCM systems, for instance, have add-ons and extensions that claim to turn them into an LMS, and many more are simply used as such out-of-the-box.

In the end, a given software system is mentally classified among the public based on several factors:

1. The market in which it promotes itself and in which it competes
2. The use cases and examples the user community creates and promotes

3. The specific features designed to meet the needs of a particular user or type of content

CMS software is *targeted* at particular markets. That has never stopped anyone from using it in ways outside of the one the vendor designed it for.

For the purposes of this book, we will concentrate on mainstream WCM — that software designed to manage a website intended for public delivery and consumption.

(Though, even this designation is blurry. Organizations are commonly powering their social networking platforms from their WCM system — managing content that ends up as Facebook updates, tweets, or even emails. While this is not technically “website content,” our definiton will have to suffice.)

What a CMS Does

Let’s break down the ultimate functions of a CMS. In very large, broad terms, what is the value proposition? Why are we better off with a CMS rather than without?

Content Control

A CMS allows us to get control of our content, which is something you’ll understand well if your content is out of control. A CMS keeps track of content. It “knows” where our content is, what condition it’s in, who can access it, and how it relates to other content. Furthermore, it seeks to prevent bad things from happening to our content.

Specifically, a CMS provides core management functions, such as:

- *Permissions*: Who can see this content? Who can change it? Who can delete it?
- *State Management and Workflow*: Is this content published? Is it in draft? Has it been archived and removed from the public?
- *Versioning*: How many times has this content changed? What did it look like three months ago? How does that version differ from the current version?
- *Dependency Management*: What content is being used by what other content? If I delete this content, how does that affect other content? What content is currently “orphaned” and unused?
- *Search and Organization*: How do I find a specific piece of content? How do I find all content that refers to X? How do I group and relate content so it’s easier to manage?

Each of these items increases our level of control over our content. It reduces risk — there is less chance that the shareholder report will be released early, or that the only copy of our procedures documentation manual will be deleted accidentally.

It's been said that content control can be measured as inversely proportional to the blood pressure of the average editor. Less control means higher blood pressure, and vice versa. The content maturity of an organizaton might be accurately measured via blood pressure cuff.)

Allow Content Reuse

Using content in more than one place and in more than one way increases its value. Some examples:

- A news article appears on its own page, but also as a teaser on a category page and in multiple "Related Article" sidebars.
- An author's bio appears at the bottom of all articles written by them.
- A Privacy Statement appears at the bottom of every page on our website.

In these situations, this information is not created every time in every location, but simply retrieved and displayed from a common location.

This reuse of content was one of the original problems that vexed early web developers. Remember the James Bond site I discussed earlier? One of the great frustrations was creating an article, and then adding it to all the index pages where it was supposed to appear. If we ever deleted the article or changed the title, we'd have to go find all the references and remove or change them.

This problem was mitigated somewhat by server-side includes that allowed page editors to insert a snippet of HTML by simply referring to a separate file — the files were combined on the server prior to delivery. Later platforms tried to automate this even further; Microsoft FrontPage, for an example, had a feature it explicitly called "Shared Borders."

The ability to resue content is highly dependent on the structure of that content. Your ability to structure your content accurately for optimial reuse is highly dependent on the features your CMS provides for you.

Allow Content Automation and Assembly

By having all of our content in a single location, we have a system we can use to query and manipulate it for greater effect. If we want to find all news articles that were written last week and mention the word "peppermint," we can do that because there is one system that "knows" all about our content.

If our content is structured correctly, we can manipulate it to display in different formats, publish it to different locations, and rearrange it on-the-fly to serve the needs of our visitors more effectively.

- We can repackage content to be published in different formats.
- We can automatically create lists and navigation (more generally, "assemblies of content") for our website.
- We can create multiple translations of content to ensure we deliver the language most appropriate to the current user.
- We can alter the content we publish in real time based on the specific behaviors and conditions exhibited by our visitor.

A CMS enables this by structuring, storing, examining, and providing query facilities around our content. It becomes the single source of information about our content; the thing that has its arms around the entire repository; the oracle we can consult to find information about our content.

Increase Editorial Efficiency

The ability for editors to create and edit content quickly and accurately is enormously affected by the platform used. It's rare to find an editor who has unconditional love for a CMS, but the alternative, editing a website manually, is clearly much less desirable.

(Who are editors? Hang on — we'll describe them, and the other members of a content management team, a few chapters from now.)

Editor efficiency is increased by a system which controls what editors can and can't add, what formatting tools are available to them, how their content is structured in the editing interface, how the editorial workflow and collaboration is managed, and what happens to their content after they publish.

A good CMS enables editors to publish more content in a shorter timeframe (it increases "editorial throughput"), and control and manage the published content with a lower amount of friction or drag on their process.

Editorial efficiency has a huge impact on morale, which is intangible but important. Editors have a historically antagonistic relationship with their CMS, and nothing destroys editorial efficiency more quickly than a clunky editorial interface and flow.

What a CMS Doesn't Do

Now for the bad news: there are things a CMS doesn't do. More specifically, these are things that a CMS doesn't do *but that people often assume it does,* which leads to problems and unfulfilled expectations.

Create Content

A CMS simply *manages* content, it doesn't create content. It doesn't write your news articles, procedure documents, or blog posts. You must still provide the editorial horsepower to generate the content that it's supposed to be managing.

Many times, a CMS implementation has ended with a group of people looking at a each other and thinking, "So…now what?" Every web development shop in the country can tell you stories about the shiny new CMS that was never once used by the client because they never changed their site after the day it launched. (We've fielded calls from clients *years* after their sites launched wanting to know how to login to their CMS for the first time.)

Related to this, a CMS won't ensure that your content is any good, either. Although a CMS might offer several tools to minimize poor quality content from a technical standpoint (ensuring that hyperlinks are valid, or that all images have ALT tags, for instance), a CMS cannot edit your content to be sure it makes sense and meets the needs of your audience.

The best laid plans to create massive amounts of quality content often fall through when confronted with the hard reality of schedule pressure and business deadlines. You need to ensure that your content creation process exists apart from your CMS.

Create Marketing Plans

Assume your content is created consistently and managed well — that doesn't mean it actually provides your organization with any value.

A CMS doesn't "know" anything about marketing. While some systems have marketing tools built into them, they still depend on human beings to direct those systems. A CMS can make executing your marketing plans easier and more efficient, but they still need to be conceived, created, and launched.

A CMS doesn't take the place of a creative team that understands your marketplace, your customers, your competitors, and what you need to do to differentiate yourself. No software can take the place of a good digital marketing strategy or team.

Format Content

While a CMS can structure content and automatically format it during publication, there is still an unfortunate amount of room for a human editor to screw it up. At some point, most CMS have a rich text editor or some other interface element that allows editors to create format text and images. This can lead to things like:

- Too much use of **bold** and *italics*
- Inconsistent alignment of content

- Random and inconsistent hyperlinking
- Poor image placement

Editors have never seen a button on an editing interface that they didn't want to press. The only way to limit this seems to be to remove as many editing options as possible, then try to withstand the hailstorm of editor complaints which inevitably will follow.

Create Governance Plans

"Governance" describes the access and processes around your content: who has access to what, and what processes/steps they follow to make changes to it.

- If Bob adds a news article, who needs to approve this and what does that approval look like? Does someone copyedit and someone else edit for quality, voice, and tone? Can you diagram this process out on a piece of paper?
- If John wants to change how the news archives are organized, and the CMS allows him to do this…can he? What process does he have to go through to do this?
- If Jennifer wants an account on the CMS to start creating content, how does she get that? Who decides who is allowed to become an editor?

Every CMS has some method to limit the actions a user can take, but these limits have to be defined in advance. The CMS will simply carry out what your organization directs it to do. These plans have to be created through human interaction and judgement, then converted into the permissions and access limits the CMS can enforce.

Governance is primarily a human discipline. You are determining the processes and policies that humans will abide by when working with your content. The CMS is just a framework for enforcement.

The Homebuilding Analogy

The home you live in is a rough combination of three things:

1. The raw building materials (wood, nails, glass)
2. The tools and building equipment (hammers, saws)
3. The human power to make it all go (Ted, your contractor)

None of those things builds a house by itself. A pile of wood is just a pile of wood until Ted takes his hammer and makes something happen. Ted is the key here. The materials and tools are inanimate. Ted is the prime mover.

In terms of content management:

- Raw materials = your content
- Tools and equipment = your CMS
- Ted = *you*

All the content in the world doesn't do much if it's not managed. And all the management in the world doesn't do much if there's no content. Neither of them do anything without human processes and effort to make them work together, just like a pile of wood and a hammer doesn't magically build a house.

You are the thing that ties that all together. You are the one that makes it all go. A CMS is just a tool.

CHAPTER 2
Points of Comparison

In medicine, certain conditions are known as “spectrum disorders” because they’re not simple binary conditions — rather, they exist along a spectrum of severity. One can suffer from a condition slightly or severely, and the difference might manifest as entirely different symptoms and require entirely different treatments.

Web content management can be the same way.

To demontrate this, it might help to examine the aspects of WCM as a series of comparisons or dichotomies. By understanding the range of available options along a particular axis and what the extremes are on either side, we can begin to understand the full breadth of options.

There are numerous facets to systems, implementations, and practices that are simply not black and white. In fact, there are few absolutes in WCM. What’s correct in one situation is clearly wrong in another. What’s right for one organization would be a disaster at another. The key in making WCM work is making the right decisions for *your* situation, which often makes it seem like more art than science.

Furthermore, the fundamental differences we’re going to describe below make it difficult to compare CMS accurately. Instead of apples-to-apples, you end up with apples-to-pot roast. For example:

- Drupal Gardens is a hosted service built in PHP using a coupled model, offering few marketing tools, and little in the way of customization or implementation
- Ingenuix is an installed system built in ASP.NET using a decoupled model, offering marketing automation and deep customization requiring significant implementation

Technical comparisons of those two systems are difficult because they lie at opposite ends of multiple axes of comparison.

The correct solution for a particular aspect of your situation will fall somewhere between two ends of the scale. Therefore, we need to understand what each end of that scale looks like.

Systems vs. Implementations

It’s important to separate a content management system from a CMS implementation. An implementation (also called an “integration”; I’ll use both terms) is the process by which a CMS is installed, configured, templated, and extended to deliver the website you want.

Unless you build your CMS from scratch, you are not the only one using it. Other organizations are using it to solve different problems and deliver different types of websites, so it’s not pre-configured to do any one thing particularly well. This means a necessary step is the one-time effort of adapting the CMS to do exactly what your organization and circumstances require from it.

To revisit our homebuilding analogy from the first chapter, a pile of wood and set of tools is not the house you want. Ted the Contractor exerts effort to use the tools to build the house. This is a one-time effort and the final product is a completed home. Furthermore, Mike the Contractor might use the same materials to build an entirely different home.

(“One-time” is an over-simplification. The fact is, CMS projects never seem to end. Once you launch, you usually already have a list of changes. Websites are constantly in flux. The idea that you’ll launch your website and never have to do anymore development is naive.)

An implementation is a significant programming project. The skillsets required are not unlike other development efforts: you need designers, programmers, front-end specialists, project managers, and experts in the CMS itself. Organizations sometimes do their own implementations, but it’s often contracted out to a development firm that specializes in the CMS being implemented.

The expense of the implementation is usually the largest expense in the budget, far eclipsing even the license fees for a commercial CMS. The rule of thumb differs depending on who tells it, but implementation fees are usually some multiple of the licensing cost of the software.

A friend, when asked if an organization should “buy vs. build” responded, “There’s no such thing as buy *or* build. It’s always buy *and* build.

The implementation should be considered at least as important to the success of the project as the CMS software itself. There are many decisions to make during an implementation, and two different implementations of the same website using the same CMS might bear little resemblence. Any of these decision points can be implemented well or poorly, and those results will have a huge impact on the final result.

The most perfect CMS in the world can be rendered all but useless by a poor implementation. Conversely, a good implementation can vastly improve the performance and function of a poor CMS. (Though, there are CMS so bad or so inappropriate for a particular situation that no implementation in the world can save them.)

The CMS Selection Process and Unknown Unknowns

The selection of the appropriate CMS for your specific set of problems can be very complex. Getting the decision right is critical, and it's especially problematic because you often "don't know what you don't know."

I'll quote a former Secretary of Defense:

> …there are known knowns; there are things that we know that we know. We also know there are known unknowns; that is to say we know there are some things we do not know. But there are also unknown unknowns, the ones we don't know we don't know.
>
> — Donald Rumsfeld

Obtuse as this seems, Rumsfeld has a clear point: unless you understand the entire industry and feature landscape, there might be unanswered questions that you don't even know to ask. It's hard to pick the right option when you don't even know a set of options exists.

I do not know the chemical symbol for Boron offhand. Neither does my 8-year-old daughter. The key difference: *I know that this thing exists*, whereas she does not. If I need the answer, I know to go look for it, whereas she does not. To me, the chemical symbol for Boron is a *known* unknown; to her it's an *unknown unknown*.

Because of this it's generally a best practice to break off the CMS selection process into its own project, managed by someone who knows what questions to ask.

There are consultants that specialize in this exact type of decision. Large technology analyst firms like Forrester and Gartner provide general analysis and assistance in this area, and small, CMS-specific firms such as Real Story Group do CMS selection process consulting as a large segment of their business.

Platform vs. Product

If we consider a CMS a range of functional "complete-ness," the extremes might look like this:

- No CMS functionality at all. Just a raw programming platform (see the Buy vs. Build commentary above)
- A fully-functional CMS ready to go, out-of-the-box, complete with pre-built features to solve all your content problems with no changes

> "Out-of-the-box" is a phrase used often in the CMS world. It means functionality that theoretically works with no implementation necessary. It's commonly used to oversell software of all types. Be skeptical whenever you encounter it.

In between these two extremes, we can insert a third option:

- A programming framework providing flexible API access common to content management features and functions which can be used to develop your own solution, along with a default user interface and configuration to support common needs

Tony Byrne from Real Story Group has referred to this type of CMS as a "platform," and the opposite, pre-built extreme as a "product."

Platform-ish systems are designed to be rearranged and customized during implementation. Product-ish systems are designed to solve specific problems quickly and without significant effort.

Platform-ish systems are flexible but effort-intensive. Product-ish systems are rigid but supposedly easy to implement.

It's a natural tradeoff. With a product, you trade reduced implementation costs by agreeing to accept how the system works. With a platform, you trade increased implementation costs for more flexibility and control.

Many vendors market their systems as products which are ready to solve all content problems out-of-the-box with very little effort. The inverse is more rare: very few vendors want to be known as the system that requires heroic customization and programming in order to build a website. (While this certainly appeals to hardcore developers, they are not usually the people making purchasing decisions.)

The "platform-ness" of a system can be used to explain the ability and expectation of a system's extensibility and customization. Some CMS are highly customizable and this is absolutely expected in every implementation. Other CMS are designed to go in as-purchased and provide few options for customization.

This is simply due to the fact that every CMS vendor or development community has use cases (literally, "cases for use") in mind when their system is developed. Whether these use cases are explicitly recorded somewhere or not, every CMS is created to solve a set of theoretical problems. One system might be designed to manage a blog, another might be designed to manage an intranet.

Your ability to use a product-ish CMS to solve *your* problems depends highly on how closely your situation resembles these theoretical problems. If they don't, then your ability to use this CMS for a purpose outside its original intention depends highly on the extensibility features of the CMS.

Often, a product-ish system will provide pre-built functionality that gets very close to a desired solution to one of your problems. In situations where 90% isn't enough, the product will have to be customized to bridge the gap. Some systems are designed to allow this, others are not.

Actual vs. Theoretical Features

Whenever discussing the feature set of any type of software, it's important to differentiate between *actual* and *theoretical* features:

- *Actual Feature:* a feature your organization uses and derives benefit from
- *Theoretical Feature:* a feature that exists and could theoretically provide a benefit to your organization, but that you do not actually use

Software is largely sold on theoretical benefits. Those in sales know how to paint a positive picture in your head of how you would use all the functionality they offer, and a negative picture of what might happen to your organization (and you, professionally) if you were caught without this theoretical benefit.

More than one CMS has been selected based solely on features and benefits that the organization had no immediate plans to use (and usually never does). The idea of wanting a feature-rich product is not inherently wrong, but it becomes problematic when the desire for a particular feature causes the abandonment or minimization of a more relevant feature.

(We call this "Ferrari Syndrome," in honor of the car buyer who will never drive over 80 m.p.h. but nevertheless loves the idea that they could go 200 if they really wanted to. The excitement of things "possible but never realized" have been driving purchasing decisions since the begining of commerce.)

Be realistic about the features you will actually use. Identify those core features that you absolutely cannot function without (the "must haves") and make sure those are well-served before moving on to features you think might work well for you (the "nice-to-haves").

open source vs. Commercial

Like any other software, both commercial (paid license) and open source options exist. This is probably more true of content management than any other genre. Literally thousands of options exist, and extremely common open source CMS platforms like WordPress and Drupal host a significant percentage of websites worldwide.

Given the installed base and depth of use, open source CMS are generally well-tested, feature-rich, and have a large volume of contributed code and modules. The availabilty, responsiveness, and accuracy of community support varies highly.

An open source CMS project can usually trace its roots back to a developer champion who originally created it to solve a specific personal or professional problem. Very few systems are "clean sheet" designs built from scratch with the intention to be distributed. Rather, they grow as a small, internal project until reaching critical mass and being contributed to the open source community for continued development.

This results in a fair amount of developer-centrism. Most open source projects subconciously evolve to be interesting and desirable for developers first, then everyone else second. Editors and marketers are the second-class citizens in these cases.

This results in a common pattern — an "open source syndrome," if you will:

- Platform-ish systems with highly extensible APIs
- Emphasis on the database-like features of a CMS, such as content modeling
- An assumption that a developer will always be available for implementation and management
- Average to below-average user interfaces with some rough edges
- A tendency to overwhelm editors with numerous options
- An emphasis on generalization, configurability, and elegance of code
- Lack of higher-end marketing or delivery tools

Additionally, open source systems normally go through large amounts of teardown and reconstruction over time. For many developers, the code and process of solving content problems is an end goal in itself. Re-architecting the system to be more elegant, efficient, or generalizable is held to be a worthwhile project. Over a decade ago, Joel Spolsky was complaining about this exact problem.

> We're programmers. Programmers are, in their hearts, architects, and the first thing they want to do when they get to a site is to bulldoze the place flat and build something grand. We're not excited by incremental renovation: tinkering, improving, planting flower beds.

Commercial systems have a built-in limitaton — they need license fees, so *the end goal is selling the software, not architecting it*. Clearly, this can result in some bad decisions and wallpapering over of genuine technical problems, but it also results in more end-user-focused development, because features for the end user is what sells licenses.

New open source systems are quite common, though as the market is more crowded, it's harder and harder for any new system to gain any traction and achieve a significant installed base. Consequently, the most successfull open source CMS are also some of the oldest.

PHP-based systems form the lion's share of the open source CMS landscape. The three most common CMS in popular use (WordPress, Drupal, and Joomla!) are all PHP systems.

Systems lose traction and developer-support primarily due to the inability of their underlying technology stack to attract new developers — there are dozens of systems from the mid-90s written in Perl, ColdFusion, and (occasionally) C++ that are slowly dying off. New Java-based open source systems are also becoming more and more rare as Java becomes a less popular web framework.

Using an open source CMS provides numerous benefits:

- The software is free
- Community support is often plentiful
- Contributed code is often available to solve common problems
- Developers and contractors are usually highly available

Some drawbacks exist:

- The "open source syndrome" discussed above
- Ubiquitous usage results in large amounts of malware, penetration attempts, and security patches (the sheer number of WordPress installations makes it a shockingly attractive target for hackers)
- Community support for especially complicated problems will often run short
- Professional service-level support may not be available
- Usage of open source software may violate an organization's IT policies
- Open source software (not just CMS) is heavily weighted toward Linux-based technology stacks (LAMP and J2EE)

Technology Stack vs. Technology Stack

All software runs on a "stack" of supporting software, databases, and languages. A CMS is always implemented in a specific language and storage framework (which may or may not “swappable”). This language and storage framework strongly influence what hosting environment the CMS needs to run.

The stack includes the following:

- The CMS Itself
- Programming Framework
- Programming Language
- Database Server
- Web Server
- Operating System

You can envision that as a pile of technologies, with the CMS sitting on top of it all and requiring everything below it run properly. Here is an example stack comparison for two very different systems: EPiServer and eZ publish:

Table 2-1. Comparison of the technology stacks of EPiServer and eZ publish

Stack Item	EPiServer	eZ publish
Programming Framework	ASP.NET MVC	Symfony 2
Programming Language	C#	PHP
Database Server	SQL Server	Multiple (usually MySQL)
Web Server	Internet Information Server (IIS)	Multiple (usually Apache)
Operating System	Windows	Multiple (usually Linux)

The crudest categorization of CMS might be by technology stack. The most common stacks are:

- LAMP (Linux, Apache, MySQL, and PHP/Python/Perl; although almost always PHP)
- ASP.NET (Windows)
- Java / J2EE (either operating system)

Less common stacks are:

- Ruby on Rails
- Python (usually the Django framework)
- Node.js

Systems cannot be swapped into different runtime languages, but hosting environments can vary slightly. For instance, while PHP normally runs on Apache and Linux, it can run reasonably well on Windows. ASP.NET almost always runs on Windows, but can sometimes run on Linux via the Mono framework.

This matters primarily if your organization limits the technology stacks it will support. While it would be ideal to select a CMS based solely on features and its fitness for your particular situation, the CMS still has to be installed and hosted somewhere. If you organization is hosting it, they might dictate all or parts of the stack. The same limitation applies if your organization is going to implement in-house — if your development group is full of Java programmers, then there's a good chance you're going to be limited to that stack.

It's quite common for an IT department to only support specific combinations of technology. Windows servers are a common requirement in corporate IT, as are specific database frameworks. Some companies dictate Oracle as their only officially-supported database, while others might be more liberal. If these limitations exist in your organization, they will necessarily pare down the CMS you are able to implement.

The desirability of particular stack is the subject of great debate and far beyond the scope of this book. The important point is that technology stack limitations — if they exist — are usually very rigid. If your organization dictates that only Windows servers can run in its datacenter, *this is something you absolutely need to know before picking a CMS.*

(Of course, hosting your CMS outside the reach of your organization's IT policy is commonly-used tactic to make an end run around imposed limits. Many a marketing department has added hosting and monitoring services to an RFP with the goal of not having to deal with the limitations their IT department enforces.)

Management vs. Delivery

While almost everything a CMS does is lumped under the umbrella of "management" by default, the lifecycle of a piece of content can effectively be split at a hypothetical "Publish" button.

Everything that happens to content from the moment it's created until the moment it dies is "management." The subset of everything that happens to the published version of that content from the moment it's published is "delivery." The two disciplines are quite different.

Management is about security, control, and efficiency. It's comprised of functionality like content modeling, permissions, versioning, and workflow. These are features that ease the creation of content, enable editorial collaboration, and keep content secure.

Delivery is about optmization and performance. The features involved during delivery depend highly on the capabilities of the CMS. These capabilities are currently evolving quickly in the marketplace. Until quite recently, delivery simply meant making content available at a public location. Today, the modern CMS is highly concerned with the performance and optimization of the content it delivers.

In the commercial space we've seen a plethora of tools which enable advanced marketing during delivery. Features like personalization, A/B testing, and analytics have proliferated as different vendors try to set their systems apart. These features used to be provided by separate "marketing automation" software packages which operate soley in the delivery environment. More and more, these tools are being built into the CMS.

The unintended result is that core management tools have changed little in the last half-decade. These tools have reached maturity in many cases and the focus is currently clearly on marketing and optimization tools during delivery. Management is generally considered "good enough."

Coupled vs. Decoupled

The "Management vs. Delivery" dichotomy manifests itself technically when considering the coupling level of a CMS. What hosting relationship do the management environment of a CMS have to do with the delivery environment?

In a coupled system, management and delivery occur on the same server (or farm of servers). Editors manage content on the same system where visitors consume it. Management and delivery are simply two sides of the same software.

This is an extremely common paradigm. Many developers and editors know of nothing else.

In a decoupled system, management and delivery are (wait for it) *decoupled* from one another. Content is managed in one environment (one server or farm) and then published to a separate environment (another server or farm). In these situations, the management functions are sometimes referred to as the "repository server," and the delivery of the content takes place on a "publishing server" or "delivery server." In these cases, published content is transported to an entirely separate environment which may or may not have any knowledge of how the content was created or managed.

Fewer and fewer systems support this paradigm, and it's seen normally in high-availability or distributed publishing environments. It has the perceived benefits of security (although this is quite debatable) and editorial advantage, as editors can make large-scale changes to content without affecting the publishing environment, only to "push" all changes as a single batch when the content is ready (though this advantage is steadily finding its way into more and more coupled systems).

Actual technical benefits of decoupling include the ability to publish to multiple servers without the need to install the CMS on each (which lowers license fees, in the case of commercial CMS), and the ability to publish to highly distributed environments (multiple datacenters on multiple continents, for example). Additionally, the delivery environment could be an entirely different programming stack than the management environment, as some systems publish “inert” assets such as simple HTML files or database records, which have few environment restrictions.

The primary drawback to decoupling is that published content is separated from the repository, which makes “live” features like personalization and user-generated content more complicated. For example, accepting user comments is more difficult when those comments have to be transported “backwards” from the delivery server to the repository server, and then the blog post on which they appear has to be republished (with the new comment displayed) “forward” into the delivery server.

To counter this, decoupled CMS are moving towards publishing content directly into companion software running on the delivery servers which has some level of knowledge of the management CMS and can enable content delivery features. The result is a CMS that’s split in half, with management features running in one environment, and delivery features running in another.

Decoupled systems tend to be clustered on the Java technology stack. Some ASP.NET systems exist, but virtually no PHP systems provide this paradigm.

Installed vs. Software-as-a-Service (SaaS)

As more and more IT infrastructure moves to “the cloud,” CMS is no different. While the norm used to be installation and configuration on your server infrastructure, vendors are now offering hosted or SaaS solutions more often. It’s not uncommon to have software rented from the vendor and hosted on their environment.

The benefit purports to be a CMS which is hosted and supported by the vendor that developed it. Whether or not this provides actual benefit is up for debate. For many, “hosted” or “SaaS” just means “someone else’s headache,” and there are multiple ways to achieve this outside of the vendor themselves.

Closely related to the Installed vs. SaaS debate is whether or not the CMS supports multiple, isolated users in the same environment. So-called “single-tenant” vs. “multi-tenant” systems are much like living in a house vs. an apartment. Users of a multi-tenant system exist in the same, shared runtime environment, isolated only by login. They occupy a crowded room, but appear to be the only one there.

The purported benefit here is one of efficiency from the vendor and a “hands off” approach to technology. These systems are promoted as giving you instant access and allowing you to concentrate on your content, not on the technology running it. The tradeoff is limits on your ability to customize, since you’re sharing the system with other clients.

We’ll discuss this dichotomy in greater detail in the next chapter.

Code vs. Content

The implementaton of a CMS will almost always involve programming code at some level. The system will either have customizations that are developed in code, or custom templating and the associated HTML, CSS, and Javascript.

(Some systems promote themselves as not requiring *any* development whatsoever, but they usually never work as well as advertised beyond the simplest of scenarios.)

This code is usually managed in a source code management system such as Git or Team Foundation Server. It's usually tested in a separate environment (a test or integration server) prior to launch. Launching new code is usually a scheduled event. Depending on your IT policy, new code might have to have approved test and change plans, as well as failure and backout plans in the event something goes wrong.

Content, on the other hand, is developed by editors. In coupled systems, it's often developed in the production CMS and just kept unpublished until it's ready to launch. It might be reviewed via a formal or informal workflow process, but often isn't otherwise "tested." If an editor has sufficient permissions, it's possible to make a content change, review it, and publish it all within the span of a few minutes with no oversight.

Content will almost always change vastly more often than code. An organization might publish and modify content several dozen times a day, but only adjust the programming code behind the website every few months. When they do, it's to fix a bug or fundamentally change how something on the website functions, not simply to change the information presented to visitors.

Code and content are sometimes confused because of the legacy of static HTML websites. For an organization that built their website with static HTML, an HTML file had to be modified for a single word to change. Thus, *a code change and a content change were the same thing*.

Decoupled content management can also blur the line between code and content. In a decoupled system, modified content is often published to a test sandbox where it's reviewed for accuracy, then published to the production environment. The existence of an entirely separate environment is similar to how code is managed.

In these situations, it's sometimes mentally hard to separate the test enviroment for content from the test environment for code. You have *two* different testing paradigms, each with its own environment, each pushing changes into the production environment.

This changes with a CMS, especially a coupled CMS. Under this paradigm, content changes without code changing at all. The verbiage of a press release might be completely rearranged, but the template which renders it is the same.

Organizations moving from static websites or decoupled systems sometimes have trouble adjusting to the idea of a "virtual" test/staging environment — unpublished content is created on the production server, and just not visible while it's awaiting publication.

Their past correlation with code tempts them to treat content the same way and intermix the two concepts.

Code vs. Editorial Configuration

Many features in a CMS can be implemented through (1) developers writing code — either core code or templating code — or (2) editors working from the interface.

Developers have complete freedom, up to the limits of the system itself. There's generally no functionality that is not available from code, as code itself is the core underpinning of the system. The only limitation on a developer is how well the API is architected to allow access and manipulation. But even with a poorly implemented API, a developer has the full capabilities of a programming language to get around shortcomings.

Editors always have access to a subset of what a developer can do from code. They are limited to the functionality that has been exposed from the interface, which varies greatly depending on the system.

Why wouldn't a system expose *all* functionality from the interface? Usually because it's too complicated to be managed by an interface. The ability to write code allows for the clear expression of extremely complex concepts. Developers are used to thinking abstractly about information and codifying it in code (and that wasn't an intentional attempt at alliteration — the root of the word "codify" is "code").

Mono- vs. Bi-Directional Publishing

Some CMS are like newspapers — they're intended for a small group of editors to create and publish content to a large group of visitors who can't directly respond. If an article in your local newspaper angers you over breakfast, there's not much you can do about it except throw your eggs, then write a letter to the editor that might get published weeks later.

Other CMS are like townhall meetings — you are allowed and expected to participate. If a political candidate says something that annoys you, you can stand up, shake your first, and yell your input directly into the conversation.

Before social media was a thing, and before user participation in websites was expected (commenting wasn't even common until the "blog revolution," circa 2002 or so), most CMS were very one-way. Editors published content that visitors consumed, blindly. Visitors didn't have "accounts" on websites. They couldn't create "profiles" or have "discussions."

Times have changed, and publishing can now go in both directions.

- Mono-directional (one-way) publishing means your organization is always pushing content "outwards" to the consumer, often blindly (meaning the user is anonymous, and not logged in or otherwise identified)
- Bi-directional (two-way) publishing means the consumer can sometimes push content "backward" to the organization (by posting a comment, for instance)

Content coming back from the user is known as "user-generated content" (UCG). Some systems are design for limited UG, while others are built around it.

This has changed over the last decade, and such tools are common and expected these days. This means that older CMS (those dating from the 90s) have had to backport this functionality, while newer CMS have it built-in.

Handling UGC requires some different tools than mono-directional publishing. If your CMS doesn't provide this, then they need to be developed, or handled by other services (Disqus or Facebook for commenting, for instance).

Additionally, UGC blurs the line between editors — if a visitor can create a blog on your site and publish blog posts, are they an editor? What differentiates them from "real" editors, from inside your organization? If your entire website is built around UCG, then do you need a CMS, or do you really need a social network or a community building tool? What about software that provides both?

UGC provides additional technical challenges for decoupled CMS, as we discussed in the prior section. In effect, content can be now created from both directions, which makes concepts like the repository hard to isolate. In some cases, UGC like comments are actually stored directly in the delivery environment, rather than the repository, as they're considered "lesser" content than that created by internal editors, and less likely to require editorial process or management.

There are several CMS that position themselves in this space and promote feature-rich community tools. Other systems that don't have these features are having to play catch-up, either tacking them on through extensions and add-on software, or integrating with other services.

When I first encountered Drupal in about 2005, I thought, “This isn’t a content management system at all. This is a *community* management system.” The presence of community management tools was a bit disorienting.

In an October 2004 article entitled “Making A Better Open Source CMS,” Jeffrey Veen was almost incredulous when he said this: “Users of a public web site should never — *never* — be presented with a way to log into the CMS. […] These systems provide a mechanism for anyone to create an account and login to the CMS directly from the site being managed.”

The migration from one-way to two-way publishing caught many of us off-guard and it took some time to realize it was a natural progression. Content can now be expected to come from any direction and systems are designed around this reality.

CHAPTER 3
Acquiring a CMS

Before we discuss features and get into the specfics of CMS architecture, we're going to take a brief detour into the software business and development models. This is necessary because in order start working with a CMS, you need to get your hands on one. There are numerous options in this regard, and each option has multiple flavors and idiosyncracies.

In general, you have four options to acquire a CMS:

1. *Open source:* you download and install
2. *Commercial:* you license (purchase) and install
3. *Software-as-a-Service (SaaS):* you "rent" and use
4. *Build Your Own:* you develop from scratch, within your organization

Any of these will provide you with a working CMS, with which to begin implementation.

However, remember, as we discussed in the last chapter, CMS operates on a scale of platform to product, requiring more and less development respectively. Some systems are as pre-fab as possible, allowing you to simply configure and go. Others are little more than frameworks with a minimal user interface, providing nothing but a starting point from which you build your own custom solution.

This being the case, remember and consider the implementation. You often don't need to simply find a CMS, but you need to find an integrator to install, configure, and template it to deliver the site you want. The greatest CMS in the world isn't going to do you much good if it never gets put to use.

Traditionally, integration costs were estimated at 3-4 times the software licensing cost. This is considerable, as a customer paying $50,000 for a commercial system was faced with another $150,000 to $200,000 in implementation costs.

However, as systems have become more cost-effective, this number is coming down to 1-2 times licensing costs. While less, this is still either equal to or significantly above the licensing costs of a commercial CMS, making those costs a minority of the total project budget.

Open Source CMS

There are likely more open source options available in CMS than any other genre of software. The most commonly-used platforms in the world — systems like WordPress, Drupal, and Joomla! — are all open source. (In fact, almost all LAMP-stack systems are open source, representing the close relationship between that technology stack and the open source ethos).

An examination of the intricacies of open source is beyond the scope of this book[1], but usually all open source CMS are free to use without paying a license fee. These systems have a website where you can download the software, install it in your own environment, and use it to the full extent of its capabilities.

That said, remember that the license cost is not the sum total of your expenses. You will still need to:

1. Host the software
2. Integrate the software

Some open source CMS are very easy to host on commodity, shared hosting accounts costing less than $20/month. Other systems require more libaries, computational power, and permissions than the average hosting account offers, and therefore require a self-hosted environment with complete control.

[1] For an examination of the core philosophy of open source software, I *highly* recommend *The Cathedral and the Bazaar* by Eric Raymond.

Open Source vs. Commercial vs. Propietary vs. Closed-Source

When discussing open source software, figuring out what to call *non* open source software can be tricky. Some call it "commercial," but others claim it's more accurate to say "proprietary" or "closed source."

For the purposes of this chapter, we'll trade hair-splitting accuracy for simplicity and simply use the descriptor of "commercial" to mean software that is not free to use, the rights over which are owned by a commercial organization that is in business to sell licenses.

Additionally, we're not going to quibble over the different varieties of open source licensing. There are several ways to license open source software, the most popular being GPL (GNU General Public License). Discussion of open source in this book will assume GPL, unless otherwise stated.

When it comes to ease of integration, open source software also varies greatly. Some projects are mature enough to have significant documentation and bootstrapping installers to get you up and running quickly. But these are often developed late in the lifecycle of open source software, so many younger systems fall quite short in these areas. Additionally, there's often a clear developer-bias in open source ("written *by* developers, *for* developers"), and a general feeling that since no one is paying for it, users will just figure it out.

Given the lack of a license fee, open source systems are used quite often in smaller projects that don't have the budgets to pay for a license. This means that applicability to much larger projects might be questionable. For every Drupal, there are a hundred other systems that have never been asked to scale much larger than a small- to mid-sized corporate site or blog.

Ubqiuitous use does present enourmous advantages in community support. Many open source systems have thriving user communities which are available to answer questions quickly and accurately. However, this can be offset by the lack of professional support (which is sometimes available at cost — see below), and the lack of a communities ability or willingness to solve more intricate questions.

And, as mentioned previously, open source CMS is tilted heavily by platform. LAMP systems are almost always open source, while there are comparatively fewer Microsoft .NET and Java systems.

Business Models of Open Source Companies

Many open source CMS products have full-fledged companies behind them, which either actively develop the software internally (example: eZ systems and their eZ Platform CMS), or guide and manage the community development (example: the Drupal Foundation). Additionally, many open source systems have commercial companies lurking around the edges of the community to provide paid services for those systems (example: Acquia, which was founded by Dries Buytaert, the creator of Drupal itself).

To pay the bills, companies behind open source software operate on one or more of the following models.

- **Consulting and Integration:** No one knows the CMS better than the company that built it, and it's quite common for vendors to implement their own software from soup to nuts, or to at least provide some higher-level consulting services with which to assist customers in integrating it themselves.
- **"Freemium":** The basic software is free, but a paid option exists that allows access to more functionality, a larger volume of managed content, or scaling options, such as the ability to load-balance. Sometimes, the free product is quite capable, and sometimes it's just a watered-down trial version meant to steer users toward paying for the full product.
- **Hosting:** Many vendors offer "managed hosting" platforms for the open source systems they develop. The purported benefit is a hosting environment designed specifically for that system, and/or system experts standing by in the event of hosting problems. In reality, the value here is a bit questionable, as there's usually little that can be done to "tailor" a hosting platform to one system over another. The value here is often just peace of mind.
- **Training and Documentation:** open source software often lacks in documentation, and developer-bias can lead to idiosyncratic, API-heavy systems. For these reasons, professional training can be helpful. Many vendors will offer paid training options, either remote or in-person. Additionally, some offer paid access to higher-quality documentation (though this is less common).
- **Commerical Licensing:** Technically, any changes to open source (usually GPL-licensed) software must be publicly released back to the community. Some vendors will offer paid, commercial licenses for their open source systems to allow organizations to ignore this requirement, close the source, and keep their changes to themselves. (In reality, however, open source enhancements are rarely released, and most organizations simply close their source in quiet violation of the license. Therefore, this option is usually only attractive for companies with strict policies governing the release of open source changes.)
- **Support:** When community support falls short, professional support can be helpful and some vendors will provide a paid support option, either on an annual subscription or per-incident basis.
- **Additional Testing and QA:** Some vendors offer a paid version of the software which is subjected to a higher, "enterprise" level of testing and QA. In these cases, the free or "community" version is presented as lightly tested and non-supported,

while the enterprise (paid) version is the only one suitable for higher-end implementations.

Commercial CMS

Like any other genre of software, numerous commercial vendors are available and eager to sell you a license to use their systems.

After our discussion of open source, why bother purchasing when so many options are available for free? Because commercial vendors generally adhere to a higher standard of quality and functionality, as (1) they have to keep paying customers happy, and (2) they have incoming revenue from license fees to fund professional development.

Like any generalization, this is not always true, as some open source systems are mature and well-used enough to compete against any commercial offering. Conversely, some commercial vendors are terrible at QA and sell products riddled with bugs. But as a general rule, it holds.

In the last five years, there has also been a distinct separation between open source and commercial CMS along the lines of marketing features. While the open source development community is obssessed with solving problems of content *management*, the commercial world has moved on to content *marketing*, which are the tools and features that help enhance your content once it's published.

It's been said that open source CMS is made for the CIO (Chief Information Offer), while commercial CMS is made for the CMO (Chief Marketing Officer). This largely holds true in how the systems are marketed, with the commercial sector concentrating their selling solely on the marketing departments of their customers, while open source is more interested in trying to capture the hearts and minds of the IT staff.

Like with open source, platform distinctions are clear. Very few LAMP stack systems are commercial, while many of the Microsoft .NET and Java systems come with price tags.

Finally, remember that the numbers under discussion in this section apply only to the license costs. Buying a commercial CMS doesn't liberate you from the costs of integrating it — you will still need to find (and pay) someone to install, configure, and template your system. In some cases, the commercial vendor provides an option for this (so-called "professional services"), and in other cases they have a "partner network" of integration firms who are experts in their system and willing to integrate for a fee.

Licensing Models

Commercial systems rarely come with a single price tag. Vendors usually have a byzantine system of formulas and tables to determine your final price tag, with the overall goal of forcing well-heeled customers to pay more, while not losing smaller sales to customers with smaller budgets.

Here are the more common ways of determining pricing.

- **By Editor/User:** The system is priced by the number of editing users, either per seat or concurrent. More rarely, the system is priced by the number of registered public users, but this usually only applies to community or intranet systems where it's expected that visitors will be registered.
- **By Server:** The system is priced by the number of servers on which it runs (less commonly on the number of CPU cores). This is quite common as larger installs will require more servers, thus extracting a higher price tag. With decoupled systems where the delivery environment is divorced from the repository environment, this can get more confusing: do you pay for repository servers or delivery servers? Or both?
- **By Site:** The system is priced by the number of distinct websites running on it. This is limited by the vagueness of what constitutes a "website." If we move our content from "microsite.domain.com" to "domain.com/microsite," are we now under the same website and license fee? What about websites with different domain names that serve the same content (branded affiinity sites, for instance)? What about alternate domains for different languages ("en.domain.com" for English and "de.domain.com" for German)?
- **By Feature:** The system is priced by add-on packages installed in addition to the core. Almost every commercial vendor has multiple features or packages that can be added onto the base system in order to increase value and price. These range from ecommerce add-ons to marketing tools. Note that each one of these features might *also* be licensed by user, server, or site, making their price variable as well.
- **By Content Volume:** The system is priced by the amount of content under management. This is somewhat rare, as the number of content objects managed is highly dependent on the requirements and implementation. For example, should you manage your blog comments? Or can you move them to a companion database (or extenal service, like Disqus) and reduce content volume by 80%?

Most systems are priced on multiple axes — for instance, by a combination of editor, server, *and* site. Final pricing can often be impossible to determine without considerable consultation with the vendor's sales department.

In CMS, the "mid-market" is defined as systems with a list price (the stated "official" price) between $20,000 and $80,000, and most everything is valued in relation to this — vendors are categorized as being either above or below the mid-market. The range of pricing is vast — ExpressionEngine sells for $299 at this writing, while Adobe CQ can't be had for less than $250,000 and large installations can edge into the seven figures.

When to Buy a CMS

Finally, the realities of business dictate that the end of a quarter is usually a very good time to negotiate a license sale. Vendors have to report their quarterly numbers to Wall Street or their board of directors at those times, and they're therefore highly incentivized to discount in order to pump up their revenues.

Given that the marginal cost of production is essentially static with sofware (it costs no more to "create" 100,000 copies than it does to create one), the only disincentive they have to discounting is that by doing so, they increase the expectation among future customers that the list price is up for negotation.

Software Subscription

One thing you can always count on with commercial vendors is the need to pay for software subscription, which is a continuing annual fee based on the purchase price. (Note that this is not at all unique to CMS — most all enterprise software is priced similarly.)

Subscription is usually a percentage of the purchase price — typically 18% to 22%. The first year is often built into the purchase, but the customer will be invoiced on their anniversary date every year after.

At an average of 20%, this adds up quickly. Simple math will tell you that you will effectively "re-buy" your CMS every 4-6 years.

Whether or not you *have* to pay this varies by vendor. With most, you can simply stop paying subscription at any time and continue to use the product, but you will lose all the value-added benefits that your subscription fee "buys." With most vendors, those benefits are some combination of the following.

1. On-demand support
2. Upgrades and patches as they are released
3. Free licenses for development or test servers
4. License management, in the event you need to license new servers or sites

The fear that keeps customers paying subscription is that they might be stranded in some situation with a problem they can't solve or a critical security bug they can't patch. Vendors play on this fear by forcing customers to "catch up" their subscription if they stop it and want to restart it, in order to prevent customers from only paying for subscription when they need it. For example, if a customer stops paying subscription afer Year 1, and has a problem at Year 3, the vendor will require retroactive payment for

Years 2 and 3 in order to restart (and sometimes with an added penalty). In some cases, vendors will force customers to re-purchase the entire system from scratch.

Software subscription has become so ingrained in the enterprise software indutry, that few customers question it. Vendors simply require it in the sale of their systems and customers expect to pay it.

Subscription revenue is the engine that keeps vendors in business. If their new license sales ever slow down or stop completely, they still have a large cushion of subscription-paying customers to keep the lights on. This continuing revenue is critical to their viability and valuation, and subscription is sometimes more important to them than the initial license sale itself.

SaaS CMS

Software-as-a-service used to be a quite clear and simple proposition: rather than purchase and install a CMS, you simply paid “rent” and ran your website inside a larger system managed by the vendor. You became one of many customers running their websites inside this same system.

This is known as “multi-tenant” software. Whereas the purchased-and-installed software was “single tenant" — you were the sole occupant, much like a single-family home your built yourself — multi-tenant software is like an apartment building in which many people share.

The purported benefits are quick ramp-up time, and no hosting issues. Indeed, the system is already running and just waiting for you, and since it runs on the vendor’s servers, you don’t need to worry about any of the headaches that go along with infrastructure management. SaaS was “cloud” before that term became common (and over used).

CMS companies were early entrants into the cloud paradigm, and when vendors like Clickability and CrownPeak came on the scene in about 2000, this model was new and original and provided clear benefits for customers who didn’t want to install and manage the software themselves. What happened in the intervening years is that the market changed, and the difference between true multi-tenant SaaS and everything else has gotten very blurry.

Today, “purchase and install” is just one way of getting open source or commercial software. If you want either of those options but don’t want to host yourself, there is a large ecosystem of vendors willing to install and host anything for you.

The “instant on” feature of SaaS was further marginalized with the advent of server virutalization and computing grids like Amazon’s EC2 and Microsoft Azure. You can now get an instance of almost any CMS in less than an hour. These systems aren’t multi-tenant, but offer the same benefits of minimal ramp-up time and third-party hosting.

This raises the question: just what is SaaS? Is SaaS defined as when a vendor else can give you an instance immediately and also provide the hosting environment? If so, then almost any CMS can be purchased and managed in such a way that it fulfills this criteria.

Or does SaaS refer to the true multi-tenant systems we discussed above? If so, then these systems are becoming less common at the higher edges of the market (in terms of price, capabilities, and complexity of integration), and more common at the lower edges. Vendors like WordPress.com, Drupal Gardens, and Squarespace offer "unattended" multi-tenant CMS where you can get a fully content-managed platform in minutes with nothing but a credit card, and without needing any human interaction.

At the enterprise level, SaaS CMS vendors are struggling. Some still exist, but their value proposition has dwindled precipitously. Many enterprise SaaS vendors are offering a lower monthly fee and comparing that to six- and seven-figure licensing costs of purchased systems. Traditional commercial vendors have responded by offering to "rent" licenses, whereby customers pay a monthly fee for the license and lose it when they stop paying. (This is also valulable for customers who might need to bring up extra sites or server for a limited time, in response to demand or load.)

If considering multi-tenant SaaS, several questions become important:

- **Is it appropriate for your industry?** SaaS systems tend to group by the vertical in which they serve. For instance, OmniUpdate traditionally services higher education and Clickability has numerous publishing clients. This enables these vendors to develop features common to those industries which will be used by multiple tenants of their system.
- **How much control do you have over the system?** To what extent can you integrate with other systems or inject your own business logic? Since these systems are multi-tenant, vendors are leery of allowing significant customization, lest this destabilize the larger system for other clients. Some vendors will offer a dedicated, "sandboxed" environment in which you have more control, but this raises the question of how you're now any better off than installing a CMS yourself.
- **Who can develop the website?** Some vendors might require that template development or other integration be performed by their own professional services group. This is common in SaaS systems that grew out of the in-house CMS of a development shop. In some cases, the subscription fee to use the system is simply a minimal gateway to enable the vendor generate professional services income.
- **If you part ways with the vendor, what happens to your data?** Can you export it? In what format? What about your templates which contain all of your presentation logic? Can you get that information out? Vendors in all software genres are notorious for lock-in, in order to prevent customers from leaving. Wise customers evaluate the process for leaving a vendor as a system feature like any other.

In the end, what once was a clear market for SaaS vendors has now been muddied considerably through changes in technology and business models. If the idea and benefits of SaaS are attractive to you, understand that almost any CMS vendor is willing to engage in a model that effectively emulate the SaaS model which used to be unique to a handful of vendors.

Build Your Own

Like any other software, a CMS can be built in-house. In some senses, a CMS resembles any other database-driven application, and it's not difficult to build a simple CMS fairly quickly.

There are several common justifications behind this:

- An in-house CMS doesn't require a license fee (clearly, this is rendered moot by open source options, but it's still quite common in project justifications)
- The organization will be experts in the usage of the resulting system and not have to suffer a learning curve on a pre-built system
- The organization will only build the needed functionality, avoiding software bloat and unecessary complication

Unfortunately, few of these reasons withstand scrutiny. Often, the project is justified based on a very superficial understanding of the needs of the organization or the overall disciple of content management. While it's possible to generate quick wins, the initial thrill of progress wears off too quickly, and the organization eventually finds itself rebuilding large pieces of core CMS functionality that other systems have long-solved.

From the outside, a CMS looks like a simple exercise in editing and publishing content. But get deeper into the project and editors begin asking for features like versioning, multiple languages, workflow, multi-site management, etc. These functions can be deceptively complex to develop — even more so when they have to be back-ported into a running system.

CMS development tends to "hockey stick" over time. It starts very quickly and development teams make huge strides early, often having a simple, workable proof-of-concept in a matter of weeks or even days. Simple CRUD operations (CReate, Update, Delete) are very quick to implement, especially with modern development frameworks.

Other features, however, such as content aggregation, editorial usability, and especially advanced marketing features, will cause development time to shoot skyward and forward progress to drop to a snail's pace. It wouldn't be surprising to spend as much time a small marketing feature as you did building the entire content editing interface.

> Years ago, when building a CMS from scratch with my business partner, I remarked that the work we were doing that week seemed much more tedious and slow-going than the prior week.
>
> His response was simple but telling: “Well, we solved all of the easy problems last week.”

Additionally, in-house CMS efforts are often developer-led, and developers tend to treat content as data, not as a business asset. To a developer, a page of content is simply a database record like any other, not a marketing piece designed to generate revenue. As such, marketing, optimization, and enhacement features tend to take a backseat to developer-centric features like core data management.

Eventually, developing the CMS itself begins to take more time than solving the organization’s core content problems (which the CMS was originally needed to remedy). Additionally, the organization realizes it has invested far more time than it would have ever spent becoming an expert in an existing CMS.

The resulting software can be idiosyncratic and unstable. It’s also unknown outside the organization, resulting in an inability to find outside contractors and creating a significant training curve on new hires.

Finally, the system is also “locked in,” meaning it has been developed to service the stated requirements and nothing more. While this sounds like a sound development practice, some additional features beyond what the editors immediately need is often helpful for experimentation and understanding what’s possible.

Typically, most organizations cross a “line of regret” when they’d like to rewind and choose a pre-built option. It’s not common to see a positive result from an in-house effort over the long-term.

However, there are situations when it might be the right choice:

- When the content model (more on that in later chapters) is *very* specific to the organization. If all you publish is cat videos, building a management platform might not be difficult.
- When the management needs are very simple and future development plans are known to be limited
- When the CMS is built by heavily leveraging existing frameworks to avoid as much rework as possible (e.g. — Symfony 2 for PHP, Django for Python, Entity Framework for ASP.NET)

If your organization is pushing for an in-house CMS, don’t begin until you carefully review the open source options available. Ask yourself this question: if we spent as much time developing expertise in Platform X as we would building a CMS from scratch, would we be better or worse off?

Questions to Ask

When considering the acquisition of a CMS, the combination of variables make your options almost almost limitless. The following questions might help give you some perspective.

- Where will the final CMS reside? Are we hosting it ourselves, or having someone else host it?
- If we're hosting it ourselves, does our IT department have platform limitations we must abide by?
- What is our capacity for a license fee? How much of the project budget can we carve out for this? Do we need to consider a lower fee, payable over time rather than all at once?
- Have we budgeted for continuing subscription costs in the years after launch?
- Are we going to integrate the CMS in-house, or do we need to find a partner firm to do this?
- Do we have the skill and capacity to build a CMS in-house? Can we manage maintenance and feature upgrades along with our other workload?

Finally, it's important to note that none of the questions above are perhaps the most important of all: *will the system under consideration meet the functional needs of our organization?*

Do not invest in a system just because the method of acquisition seems easy. An open source system that's free but doesn't offer marketers the tools they want isn't a good value — you will save on a license fee, but invest money in an integration that will not provide the result you want. Conversely, a commercial vendor offering a good deal on tools you will never use is simply offering to help you throw money away.

The method you acquire a CMS is but one aspect of a much larger question of matching need to platform. This is the topic we'll discuss in the next section.

CHAPTER 4

The Content Management Team

From inception to launch and ongoing usage, a CMS project might impact many people throughout your organization, all with different roles and responsibilities.

While a comprehesive look at web operations and governance is beyond the scope of this book, it's helpful to discuss these roles before digging into CMS features so we can have some clarity and perspective about exactly which people a particular feature might affect.

Primarily, members of the CMS team can be divided into:

1. Editors
2. Site Planners
3. Administrators
4. Developers
5. Stakeholders

Note that these labels are *roles*, not *people*. The lines between the roles are not absolute, and it would be rare to see a project where every single role described below is staffed by a separate person. Members of the team usually always fill multiple roles. In fact, for a very small project, the entire team might consist of a single developer and a single editor (and developer hobby projects might be an entirely one-person show).

That said, the content management team is usually comprised of some combination of the following.

Editors

Editors are responsible for creating, editing, and managing the content inside the CMS. We'll talk about editors a lot through this book, as they're the role which will interact with the CMS most intimately post-launch.

Editors tend to get lumped into a single group, but all editors are not created equally, and "editor" is a crude generalization for what might be a wide variety of capabilities.

What is a "normal" or "mainstream" editor is project-specific. Therefore, it might be helpful to discuss how editors can be limited in their capabilities to refine their subrole.

- **Section / Branch / Location:** Editors might be able to edit only a specific "area" of content on the website, whether that be a section, a branch on the content tree, or some other method of localization. They might have full control over content in that area (the press section, or the English department, for example), but no control over content in other areas.
- **Content Type:** Editors might be able to edit only specific types of content (we'll talk much more about content types in later chapters). They might manage the "faculty profiles," which appear in multiple department sites, or manage company news articles, regardless of location. In fact, some editors might be better defined by what content types can are *not* allowed to create — some editors, for instance, might not be allowed to create advanced content like aggregations or image carousels.
- **Editing Interface:** Editors might be limited by the interface they're allowed to use. In larger installations, it's not uncommon to "channel" certain editors through specialized, custom-built interfaces designed to allow them to manage only the content under their control. For instance, if the receptionist at your company is responsible for updating the lunch menu on the intranet and nothing else, he does not need an understanding of the larger CMS and all the intricacies that go with it. Instead, it might be appropriate to build him a special editing interface to manage the lunch menu and nothing else.

In contrast to these limitations is the so-called "Power Editor," who can perform all content operations across the website. This person sometimes performs multiple duties as a site administrator, trainer, subject matter expect, and all-around CMS champion inside the organization.

Several other specific editorial roles are common:

- **Approvers**: The role is responsible for reviewing submitted content, ensuring it's valid, accurate, and of acceptable quality, and then publishing that content. Approvers are usually a step in one or more workflows. Many editors are also approvers, responsible for vetting content submitted by more junior editors. These editors may also have the right to "self-approve" their own content. An approver might have the ability to edit submitted content prior to publication (an editor-in-chief, for example), while other approvers might only have the ability to approve or reject with comment (the legal or compliance department, for example). This role will only need to understand the content approval features of the CMS.

- **Marketers:** The role is responsible for reviewing content for marketing impact, and managing the marketing value of the entire website. This role will need an understanding of the marketing and analytics features of the CMS.
- **UGC / Community Managers:** The role is responsible for verifying the appropriateness of content submitted by users ("User Generated Content" or UGC), such as user profile information and blog comments. These managers are similar to approvers, but only have control over UGC, rather than core editorial content (in some cases, this might be the majority of the content on the site). Additionally, given that submission volume of UGC is often high, it's commonly managed post-publication — inappropriate content is reviewed and removed after publication (or after a complaint is received), rather than holding it from publication until review. This role will only need to understand the CMS to the extent that allows them to moderate UGC — in some cases, the CMS provides separate tools for this, while in others this is handled as normal content.
- **Translator:** This role is responsible for the translation of content from one language to another. The role only needs to understand the editorial functionality of the CMS to the extent required to add translations of specific content objects (perhaps even of only specific content properties, in the event that content objects are only partially translated).

In smaller installations, all of these roles might be the same person. Additionally, not all roles will be filled — sites without UGC or multiple languages will not require roles to manage them.

UGC / Community Managers might not be employed by the organization — in community sites, it's not uncommon to depend on the community itself to self-monitor, empowering specific members to moderate content.

Content translation is often contracted to third-party firms. In these cases, the Translator will be remote, and will likely therefore not work with the CMS at all, instead moving content in and out via a translation-specific workflow and exchange format, such as XLIFF.

Site Planners

Site planners are responsible for designing the website the CMS will serve to manage. Most of their involvement will be prior to launch, with continuing involvement as the site develops and changes over time.

They can be any one of the following sub-roles:

- **Content Strategists:** This role is responsible for designing content, both holistically and tactically. As a byproduct of the content planning process, this role will define the content types and interactions the website must support. This role will require knowledge of how the CMS models and aggregates content in order to understand any limitations on their design. Additional knowledge of the marketing features will be provided if the Content Strategist is responsible for optimizing the marketing value of the site prior to launch.
- **UX Designers and Information Architects**: These roles are responsible for wireframing the user interaction with the website. They will need to understand how the CMS organizes content, and what facilities are available to aggregate and present content to end users.
- **Visual Designers:** This role is responsible for the final, high-fidelity design of the website (as opposed to lower-fidelity wireframes and walkthroughs provided by previous roles). This role doesn't need intimate knowledge of the CMS, as CMS-related limitations will have guided the process up to their involvement.

Developers

Developers are responsible for installing, configuring, integrating, and templating the CMS to match the requirements of the project.

How much development effort this takes is specific to the complexity of the requirements and how well-suited the CMS to those requirements out-of-the-box. Deploying a simple blog powered by WordPress will take very little development (perhaps even *no* development), while an enterprise intranet built from scratch is a huge undertaking.

Like editors, not all developers are created equally. Under the umbrella of "development," there are multiple sub-roles.

- **CMS Configuration:** This role is responsible for the installation and configuration of the CMS itself, including the establishment of the content model, creation of workflows and other editorial tools, creation of user groups, roles, and permissions, etc. This work is done at a fairly high level, through facilities and interfaces provided by the CMS.
- **Back-end (Server) Development:** This role is responsible for more low-level development performed in a traditional programming language (PHP, C#, Java, etc.) to accomplish more complex content management tasks, or to integrate the CMS with other systems. This developer might not know much about the CMS, but is an expert in the required programming language.

- **Front-end (Client) Development or Templating:** This role is responsible for the creation of HTML, CSS, JavaScript, template logic, and other code artifacts required to present managed content in a browser. This role needs only to know the templating language and architecture provided by the CMS, and how it integrates with HTML, CSS, and JavaScript.

In many cases, all three of these development roles are performed by the same person. Alternately, a very common split is to have the front-end development performed by one developer, and the CMS and back-end development performed by another developer. In these cases, the front-end developer is responsible for templating content that the back-end developer has configured the CMS to manage and provide.

The split between CMS and back-end development depends on the CMS. Some systems allow an enormous amount of development to be performed from the interface, and writing programming code is considered the exception, rather than the rule (Drupal is famous for this). Also, in SaaS environments, the option to write programming code might not be available.

Other systems are designed as programming platforms, and most of the project might consist mainly of writing, compiling, and deploying programing code (EPiServer is a good example of this).

Administrators

Administrators are responsible for the continued operation of the CMS and the associated infrastructure. Within this group are several sub-roles:

- **CMS Administrator:** This role is responsible for managing the CMS itself, which includes user and permission management, workflow creation and management, licensing management, and all other tasks not related to content creation.
- **Server Administrator:** This role is responsible for the maintenance and support of the server(s) on which the CMS runs and/or deploys content. This is a traditional IT role, and often has no understanding of the CMS itself other than the basic architecture required for it to run without error. This role comes to the rescue when there is an underlying server issue that prevents the CMS from running.
- **Database / Storage Administrator:** This role is responsible for managing the database server and storage networks that hold the CMS content. This role needs very little understanding of the CMS, other than the file types, sizes, and aggregate volume which will need to be stored and backed up.

A CMS Administrator is often also a Power Editor.

It's very common to see the Server Administrator and Database / Storage Administrator roles combined in the same person. However, many larger organizations have separate groups of data administrators responsible for managing storage and nothing else.

Stakeholders

The stakeholders of a CMS project are a vague group representing the people responsible for the results that the CMS is intended to bring about. Stakeholders are normally business people (as opposed to editorial or IT staff) who look at the CMS simply as a means to an end.

In general, stakeholders are looking to a CMS to do one of two things:

1. Increase revenue
2. Reduce costs and/or risk

These goals can be achieved a number of different ways, a CMS simply being one of them. Stakeholders often have no direct contact with the CMS, and they might not care about the specific features the CMS enables — their only goal is the result the CMS can bring about.

For example:

- The Chief Marketing Officer (CMO) is dissatisifed with the the rate that visitors complete the "Get a Quote" form after browsing the website. She is convinced that a personalization strategy — varying site content to market to each visitor specifically — will increase this conversion rate.
- The manager of the Support department feels that the company is taking too many support calls because of the sorry state of the online product documentation. Attempts to improve the documentation have been thwarted by technical limitations, which a new CMS might solve, and hopefully result in a lower volume of incoming support calls.
- The Editor-in-Chief is trying to increase article volume, but the current CMS forces hours of editorial overhead and re-work to get an article published. The editor is hoping to increase content throughput with a CMS that has streamlined editorial workflow.

Note that, in each case, the goal was not "to install a new CMS." The CMS is simply the means to achieving a larger business goal.

In each of these examples we have someone who (1) is not directly going to use the CMS, and (2) is not going to develop or integrate the CMS. So, why are the stakeholders important? *Because they are usually the decision makers on a CMS purchase who control the budget from which the project will be funded.*

They are included in this discussion with the CMS team because sometimes it's easy to lose sight of the forest for the trees. The closer you get to a CMS project — as an editor, administrator, site planner, or developer — the easier it is to obsess over small details.

Never lose sight of the fact that stakeholders have little regard for anything beyond the critical question: will this expense bring about the business goal we are seeking? The specifics of exactly *how* the CMS does this are simply details.

CHAPTER 5

CMS Feature Analysis

This section of the book is devoted to describing the component features of common content management systems. We'll start with a warning to set your expectations, then we'll give you an overview of what's to come.

Rather than being overly pessimistic, this chapter is intended to set your expectations for a feature-level evaluation of content management. Understand that this isn't an exact science and if the borders around particular features feel fuzzy and vague, that's likely an accurate observation.

The Difficulties of Feature Analysis

Before we embark on a detailed analysis of content management features, we need to make an important point: *feature by feature analysis and comparison is hard.* As much as we want this to be a clear science, it's messy and imperfect.

Mathematics is a very objective science. You're not going to get much argument about the answer to two plus two. In math, there is a "grand unified theory" that has been accepted and perfected over millennia about how math works. This truth is something that mathematicians can remove from debate.

Content management is not like this. There is no grand unified theory of content management. You can pose an architectural question to five different bonafide experts and get five different answers (maybe six), all of which would serve to solve the problem posed by the question and can thus be considered "correct" to some extent.

Why is this?

"Fitness to Purpose"

In evaluating anything, we tend to think in terms of relativity.

If we say that some thing is "on the left side," we're implying that some other thing is to the right of it. You can't be on the left side of nothing, so the concept of left-ness only exists because something else is on the right.

Likewise, content management systems exist only in clear relation to a problem they need to solve. Their competence can only be evaluated as their distance from what is needed for *your situation*.

The correct answer for a content management question lies in an intersection of multiple factors, including:

- The type and goals of the website
- The "shape" of the content being managed
- The publishing requirements
- The sophistication of the editors
- The business environment, decision-making process, and budget
- and on and on…

When comparing systems, it's easy to look at two features and say "this one is better…". In doing this, the unspoken end to that sentence is "…for the particular requirements in my head right now." What is right for one set of requirements could be clearly wrong for another.

Furthermore, some applications simply don't need certain features. If you are the only editor of your web content and you know for certain there won't be any other editors in the future (this is more common than you think), then concepts of workflow, collaboration, and permissions become largely meaningless. If a system is bad at those things, it might be an accurate observation, but it's not relevant.

Content management expert Tony Byrne once declared that the single criteria of CMS selection should be "fitness to purpose."" That phrase describes an intersection between between:

1. The capabilities of a system ("fitness")
2. The requirements under consideration ("purpose")

I doubt a better method of evaluation exists.

"Do Everything" Syndrome

Content management, like other software, tends to try to include as much functionality in one package as possible. No vendor (or open source community) wants to have to say, "We don't offer that feature," so they're motivated to handle every possible situation or eventuality. If an editor has to go outside the system, that would be considered a failure.

This results in two things: software bloat and poorly-implemented features with fewer options to work around them.

Content management systems are getting more and more complex every year. The industry has steadily crept "outward" from the clearly-defined core of 15 years ago. We've already discussed the drift from content into community management, and now we have systems managing social media, offering powerful marketing suites, and even systems trying to act as general application development frameworks.

The price we pay for this is complexity. The list of features you're going to ignore (and try to find ways to turn off) can easily be longer than the list of features you're actually going to use.

As systems become more complex, they tend to become more generic. Developers working on any system too long drift into larger architectural concepts and frameworks. They become what Joel Spolsky has called "architecture astronauts."

Conversations like this actually happen:

> If we can manage web pages, then we can manage general content too! In fact, let's just manage random strings of text — if the user wants to make a web page out of them, they can do that! And why even text? Everything is comprised of bytes in the end, so let's just manage sequences of bytes…

Remember: a system designed to do everything tends to do nothing well, and you should only evaluate the features you actually need.

When extended features outside core content management are added, they're often added poorly. In many cases, the vendor is trying to "check a box" that they see on customer requirements lists. Just because a feature exists doesn't mean it's done well.

Form-building tools are a classic example. Many systems have them, as they've become a de facto requirement to compete in the marketplace. However, I've never evaluated a form-building system that was developed at the same level as the core content management tools. In almost all cases, this was an add-on feature that had to survive only as long as a sales demo.

Furthermore, since the system now "has" a form builder, there's less incentive for the vendor to provide extensibility hooks to manage user generated content. Why would someone need to do that if the functionality is built in?

We have a natural desire to think that one system can solve all of our problems. However, the solution lies in multiple systems and how we use them together. Going outside your shiny new CMS is not necessarily a failure. It might be exactly the right decision rather than suffer through a poor feature for nothing but a desire for it to "do everything."

The Whole is Greater Than the Sum of its Parts

Content management systems are complex, and full of moving parts. The effect of all these parts working together forms a whole that might not be representative of the parts that make it up.

It's hard for a system of poorly-designed features to rise above them, but the inverse is sadly common: a portfolio of features that are stellar when evaluated individually, but just don't quite come together in the entire system.

This can be caused by poor usability in the intersections between features — a workflow system that offers a stunning array of functionality but simply doesn't interact well with editors when they're creating content is not much use to anyone.

Another problem can be misplacement of priorities, when one feature is over-developed compared to others. The perfect templating system in service of content that can't be modeled accurately won't help you much and the end result is like a Ferrari in a traffic jam — lots of power with no place to go.

The only way to effectively evaluate the whole is by subjecting it to as long of a use case as possible. Don't just just pick small snippets of functionality to compare ("can we select an approver when starting a workflow"), but rather complete a full cycle of interaction ("let's publish a press release from initial conception all the way through distribution"). Scenarios like this can poke unexpected holes in systems that seemed sound from the feature level.

Implementation Details Matter

The value of a CMS feature is not just in the final result. It also matters how you get there. Just because different systems can check a box on a feature list doesn't mean that the features were implemented equally well in both systems.

Usability matters. Many features were conceived, designed, and built by developers (this is especially true with open source systems), and this may have resulted in interfaces and user models which make little sense to editors. Developers tend to be more forgiving on rough edges, and more concerned with ultimate flexibility rather than usability.

Beyond just the interface, does the mental model of the feature make sense? When an editor is working with it, is it easy to describe how it works and theorize ways in which it can be used? Simple features can be made obscure by idiosyncratic philosophies and ideas that made sense to the person who designed them, but few others.

Some features can be hopelessly complex, either through poor design or just because they are, in fact, very complex. Consider the Drupal Views module — this is a system designed to let you aggregate content any way you like. That seemingly simple goal is extremely complex in practice, and no matter how well you design the interface, it's going to be somewhat overwhelming.

Other features might be built specifically for developers. A workflow system might be very powerful, but if it has to be configured in code, then compiled and deployed, this drastically limits its utility for anyone other than a developer. Similarly, templating systems that are not extractable from the core code of the system limit their utility to only people who have access to that code.

Some features might require additional software. It's very common for vendors to "partner" with other firms to provide plugin functionality. This functionality looks great in a sales demo, but requires extra expense and often a separate purchase and license from the third-party company.

The lesson here is that just because a feature exists, it doesn't mean it's any good. When you have a list of boxes you're trying to check — either an actual list, or a mental list — there's seldom room to say, "yes this feature exists, but it doesn't work very well." In most feature-matrix spreadsheets, there's a column for "yes" and a column for "no," and nothing in between.

It's a very cynical observation, but vendors often know this. They realize that someone evaluating their software is wondering about functionality and just assuming it works well. It's very hard to dive deep enough into a software demo to uncover all the warts. The vendor might be counting on this, which is why you'll often hear: "Sure, we do that. Now, let me show you this other thing we do…".

Do not evaluate features based merely on their existence or on a cursory examination of the result. Find out exactly how the feature is implemented and ensure that you'll be able to use it to acheive the same result.

An Overview of CMS Features

We'll start by covering The Big Four featureset of content management. These are the four features that are required in some form to manage content at the most basic level. They are:

- Content Modeling
- Content Aggregation
- Editorial Workflow and Usability
- Publishing and Output Management

If a system fails at one or more of these four, it's hard to manage any content effectively.

From there, we'll review extended functionality:

- Multi-site Management
- Language Handling

- File Handling
- Page Composition
- User Management
- Marketing Support
- Extensibility and Programmability
- Community
- Form Building
- Performance

CHAPTER 6
Content Modeling

At the risk of triggering bad memories, consider the form you have to complete at the local Department of Motor Vehicles when renewing your driver's license. You're envisioning a sheet of paper with tiny boxes, aren't you?

But what if it wasn't like that? Let's pretend that instead of a form, you just get a blank sheet of paper on which you're expected to write a free-form essay identifying yourself and the reasons you want a driver's license. Then someone at the DMV sits down to read your essay and extract all the particular information the DMV needs. If the information isn't there — for instance, you forgot to include your birthdate because no one told you to put it in your essay — they send you back to try again.

Clearly, this would be inefficient. Instead, the DMV has forms with separate boxes for you to input different information: your name, your birthdate, etc. These boxes even have labels describing them and prompts to ensure you enter the information in the correct format: there might be "mm/dd/yyyy" in the birthdate field, or there might be two dashes in the Social Security number field, or checkboxes for "male" and "female."

The people who designed this form considered the range of information they needed from citizens, and then structured it. They broke it into separate boxes on the form, and took steps to ensure it was entered correctly.

Put another way, someone "modeled" the information the DMV requires. They extracted an explicit structure from the amorphous chunk of information the DMV needed, broke it down into smaller pieces, and provided an interface for managing it (the form itself).

Similary, a core goal of any CMS is to accurately represent and manage your content. To do that, it has to know what your content is. Just as you can't build a box for something without knowing the dimensions of that thing, your CMS needs to know the dimensions of your content to accurately store and manage it.

In most cases, you start with a *logical idea* of the content you need to manage in order to fulfill the project requirements. For instance, you know that you need to display a press release. This is a general notion of content. But what is a press release? What does it look like? How is it structured? Ask five different people and you might get five different answers.

A CMS can't read your mind (much less the minds of five different people) and therefore has no idea what *you* think a press release is. So, this logical notion of a press release needs to be translated into a concrete form that your CMS can actually manage.

To do this, you need to explain to the CMS what a press release is — what bits of information make up a press release, and what are the rules and restrictions around that information? Only by knowing this will your CMS know how to store, manage, search, and provide editing tools around this content.

This process is called **content modeling**. The result is a description of all of the content your CMS is expected to managed. This is your **content model.**

The stakes can be bit high here, as content modeling is often done poorly either through mistakes in judgement or because of the built-in limitations of a particular CMS. These mistakes breed multiple problems for a content management project and can be tough to recover from.

Warning: Theory Ahead

This chapter (and the next) might seem a bit abstract. We're going to discuss the core characteristics of content in general, separated from concrete representations like the interface in which editors create content or the web pages that the CMS generates.

Even if some of this seems theoretical, please try to stick with it. The foundation laid in these chapters will make it easier for you to understand more specific topics later in the book.

Data Modeling 101

Modeling is not unique to content management. "Data modeling" has been around as long as databases. For decades, database designers have needed to translate logical ideas of information into database representations that are in an optimally searchable, storable, and manipulatable format.

CMS and Databases

The similarties between traditional databases and content management are obvious: both are systems to manage information at some level. In fact, CMS are usually always built *on top* of a relational database. Almost every CMS has a database underneath it where it stores much of its information.

In this sense, a CMS might be considered a "super database," by which we mean a database extended to offer functionality specific to managing content. As such, many of the same concepts, paradigms, benefits, and drawbacks of relational databases also apply to content management systems in varying degrees.

If you're thinking that perhaps a background in database design would be helpful, you're absolutely correct. To this end, I recommend *Database Design for Mere Mortals* by Michael Hernandez. Even if you never have to design a traditional database, the ability to separate a data model from the information stored within it is a key professional skill.

Another word for this process is "reificiation," which is from the Latin prefix "res" which means "thing." To "reify" something is literally "to make it a thing."

Computers don't understanding vagueness. They want hard, concrete data that's restricted so they know exactly what they're working with. Reification is a process of moving from an abstract and therefore unlimited idea of something to a concrete representation of it, complete with the limitations and restrictions this brings along with it.

The classic example of a modeling problem is a street address.

123 Main Street Suite 1 New York, NY 10001

This is quite simple to store as a big lump of text, but doing so limits your ability to ask questions of it:

- What city are you in?
- What floor of the building are you on?
- What other businesses are nearby?
- What side of the street are you on?

To answer these questions, you would have to parse — or break apart — this address to get at the smaller pieces. These smaller pieces only have meaning when they're properly labeled so a machine knows what they represent. For example, the "1" in the second line makes perfect sense when it refers to a unit number, but doesn't work as a zip code.

Consider this alternative model of the above address.

- **Street Number:** 123
- **Street Direction:** [none]
- **Street Name:** Main
- **Street Suffix:** Street
- **Unit Label:** Suite
- **Unit Number:** 1
- **City:** New York
- **State:** NY
- **Postal Code:** 10001

By storing this information in smaller chunks and giving those chunks labels, we can manipulate and query large groups of addresses at once. What we've done here is "reify" the general idea of an address into a concrete representation that our CMS can work with.

(So, could you just parse the entire address every time you wanted to work with it? Sure. But it's much more efficient to do it once when the content is created, rather than every time we want to read something from it. Common sense says that you'll read from it far more often than you'll create or change it. Additionally, when it's created, a human editor can make proactive decisions about what goes where instead of a parsing algorithm taking its best guess.)

It's worth noting that by creating a model for the address, we've made it *less* flexible. As inefficient as storing the big lump of text may be, it's certainly flexible. A big lump of text with no rules around it can store anything, even an address like this:

123 Main Street South Suite 200 Mail Stop 456 c/o Bob Johnson New York, NY 10001-0001 APO AP 12345-1234

This address wouldn't fit into the model we created earlier. To make that model work for this content, we would need to expand our model to fit. For instance, we'd need to create a space for APO/FPO/DPO numbers (this is a method for addressing miltary units overseas).

Addresses in this format might be relatively rare, but when creating a content model, your judgment is an unavoidable part of the process. Is this situation common enough to justify complicating the model to account for it? Does the exception become the rule? Only your specific requirements can answer that.

Edge Cases

In software design, our complicated address is called an "edge case," since it's a usage case at the "edges" of what you're planning.

Software design is littered with these situations and you can't possibly account for them all. Trying to handle *everything* will lead to bloated software that has become so generic and complicated that it even fails at its original purpose.

Only experience and knowledge of your users and requirements can help you decide what to do about edge cases.

Data Modeling and Content Management

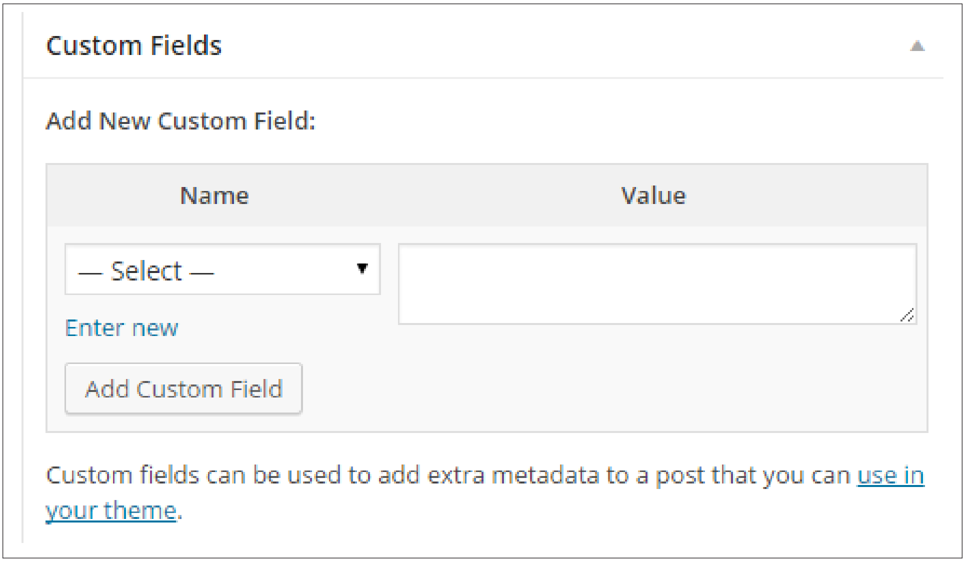
Every CMS has a content model, whether it's called that or not. Even the simplest CMS has an internal, concrete representation of how it defines content.

The simplest model might be a wiki, where you have nothing but a title and a body of text. Simplistic and rigid as this is, it's clearly a pre-defined content model.

The original blogging platforms — early WordPress, Movable Type, Blogger, etc. — were largely the same way: everything you put into the system had a title, a body, an excerpt, and a date. This is effectively a built-in content model designed specifically to fit the requirements of writing a blog.

In most cases, you need more flexibility than this. You need to store information beyond what's offered by default. When this happens, you're limited by the content modeling features of your CMS.

Some systems offer a limited number of "custom fields" in addition to the built-in model (blogging platforms like WordPress have moved in this direction), while other systems assume nothing and depend on you to create a content model from the ground up. To this end, they can offer a dizzying array of tools to assist in content model defintion.



Custom fields can be used to add extra metadata to a post that you can use in your theme.

Figure 6-1. A free-form custom fields interface in WordPress

The unspoken standard that most CMS are chasing is that of a custom relational database, which has been the traditional way to model data and information since the early 1970s. CMS have varying levels of fidelity to this ideal — some force you to simplify more than you'd like, while others are essentially thin wrappers around your own custom relational database (a few systems are *literally* of this type — they just add a few fields and tables to a database of your own creation).

Why aren't all CMS like this? Because it's often more than you need. The CMS industry has evolved around common content problems, and has created patterns for dealing with situations that are seen over and over again. Most web content management problems will fall within the range covered by these patterns — the exceptions are…wait for it…*edge cases* — so they're enough to function well under most requirements.

Where a particular CMS falls in the range of modeling capabilities has a huge impact on the success or failure of your project. Some projects have complex content models and absolutely hinge on the ability of the CMS to represent them. In these cases, content modeling limitations can be hard to work around.

Separating Content and Presentation

It's tempting to look at some form of your content — your press release page rendered in a browser, for instance — and say, "This is my content."

But is it? In a pure sene, your actual content is the structure and words that make up the press release. The information in your browser is just a web page. So, are you managing words or are you managing web pages?

I argue that it's the former. Your content is as close to pure information as possible. The web page is actually piece of media created by your content. Put another way, it's simply a *representation* of your content.

Your content has been combined with presentational information — in this case, converted to HTML tags — and published to a specific location. Ideally, you could take those same words and use them to create a PDF, which you could email to someone. Or after publishing your web page, you could use the title of your press release and the URL to create a Facebook update.

These publication locations (email, web, Facebook, etc.) are often called **channels.** The media that is published to a channel might be referred to as a **rendition** or a **presentation** of content in a particular channel.

The relationship of content to published media is one to many: one piece of content can result in many presentations. In that sense, a presentation is *applied* to content in order to get some piece of published media.

We can only do this if we separate our content and our presentation, which means modeling our core content as "purely" as possible. Ideally, we do this without regard to any particular presentation format. (We might need to add some extra information to ease the application of a particular presentation, but this information won't be used in other presentation contexts.)

This concept is not new. Gideon Burton, an English professor at Brigham Young University, has traced this all the way back to the ancient Greeks.

> Aristotle phrased this as the difference between logos (the logical content of a speech) and lexis (the style and delivery of a speech). Roman authors such as Quintilian would make the same distinction by dividing consideration of things or substance, *res*, from consideration of verbal expression, *verba*.

Our content is *logos*, the presentation method is *lexis*.

Binding your content to a specific presentation drastically limits what you can do with that content. If your requirements are simple and won't change, then perhaps this isn't a great disadvantage. But when content is structured well, its utility increases.

1. **Templating is easier.** Having content in smaller chunks allows you to use it in more specific ways when templating. Want to order authors by last name rather than first name? You can only do this if the last name is isolated in such a way that it can be evaluated for sorting.

2. **Mass presentation changes are possible.** Should all your author headshots and bios appear in the left sidebar now, rather than at the bottom of the article? If this information is separable from the main body of content, this is a simple templating change, whether you have 10 or 10,000 articles.
3. **Content can be easily presented in other contexts.** When your pages have isolated summaries, these can be posted in other contexts with shorter length requirements, for example.
4. **Editorial usability is improved.** Granular content models often allow you customize the editorial interface with specific elements to allow editors to accurately work with specific pieces of information that make up your content. Should you limit the HTML tags that editors can use when writing their article summaries? If summary is its own attribute, this is easier to do.

Going back to Burton's Aristotlean example from above, Aristole might have a theory about Man's position in the universe. This theory is his content. He can "render" this content into an essay, a play, a speech, even a drawing. Those items are the media generated from his content. They're just presentations — the core concepts of Aristotle's theory underlies them all.

Defining a Content Model

A content model is fundamentally defined by three things:

1. Types
2. Attributes
3. Relationships

These three things, used in combination, can define an infinite variety of content.

Types

A **content type** is the logical division of content by structure and purpose. Each type serves a different role in the model content and is comprised of different information.

Humans think in terms of types every day. You label specific things based on what type of thing they are — in terms of "is a."

- This metal and rubber machine *is a* bicycle.
- This food on my plate *is a* burrito.
- This house *is a* building.

In thinking this way, we're mentally identifying types. We understand that there are multiple bicycles and burritos and buildings in the world, and they are all concrete representations of some type of thing. We have mentally separated the type of thing from a specific instance of the thing.

Editors working with content think the same way. An editor wants to create a new *Page* or a new *Employee Bio*. Or, this existing content *is a* Page or *is an* Employee Bio. In your editor's head, the idea of a generic Page is separate from actual representations of those page: "About Us" or "Products."

Whether explicitly acknowledged or not, whenever you work with content you have some mental conception of the type of content you want to manage. You mentally put your content into boxes based on what that content is.

A content type is defined to your CMS in advance. Most CMS allow multiple types of content, but they differ highly on how granular and specific they allow the defintion of these types.

All content stored by a CMS will have a type. This type will usually always be selected when the content is created — indeed, most CMS won't know what editing interface to show an editor unless it knows what content type the editor istrying to work with. It is usually difficult to change the type of a content object once it's created. (More on this below.)

Types vs. Objects

It's important to draw a clear line between a Content Type and a **Content *Object***. A content type is a pattern for an object — bicycle, burrito, or building, from the example above. You might have a single content type from which thousands of content objects are created.

Consider making Christmas cookies. You have a cookie cutter, with which you cut cookie dough. You have *one* cookie cutter, which you used to create *dozens* of cookies.

The cookie cutter is your content *type*. The actual cookies are the content *objects*.

When we refer to a "content type" we're refering to a type or defintion of content object. When we refer to a "content object," we're refering to *a single piece of content created from that type.*

A content type can be considered the "pattern" for a piece of content, or the defintion of the information a particular type of content requires to be considered valid. An Employee Bio, for example, might require the following information:

- First Name
- Last Name

- Job Title
- Hire Date
- Bio Sketch
- Manager
- Image

(Why First Name and Last Name and not just Name? Because if you want to work with Last Names, you need to isolate them. You can't easily sort by Last Name or search Last Names unless they're stored separately from First Name.)

You must create this defintion in advance so that your CMS knows what information you'll be putting into what spaces.

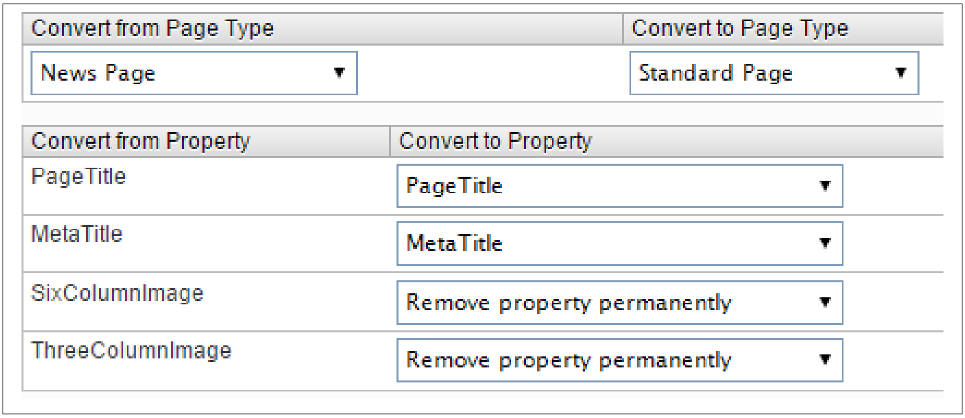
There are multiple benefits to organzing your content into types:

- **Structure:** Different content types require different information to be considered valid. A Person requires a first name. This doesn't make sense for a Page.
- **Usability:** Most CMS will automatically create an editing interface specific to the type of content you're working with. When editing the Hire Date in the above defintion, for example, the CMS might render a date selection dropdown.
- **Search:** Finding all blog posts is quite simple when they all occupy the same type.
- **Output/Templating:** Our employee profile pages will be clearly different than our marketing pages. Since the two types store different information, they obviously have different methods of outputting that information.
- **Permissions:** Perhaps only the Human Resources department can edit employee profiles. Depending on the CMS, you might be able to limit permissions based on type.

Switching the underlying content type after a content object has been created from it can be logically problematic.

Let's assume that we created a piece of content for our Employee Bio content type. Now, for whatever reason, we want to convert this to a simple Page content type. We have a problem because we have information specific to the Employee Bio type that doesn't exist in Page, which means that when we switch types, this information has nowhere to go. What happens to it?

Because of this, switching content types after content has been created is often not allowed. If it is, you have to make hard decisions about what happens to information that has no logical place in the future type.



Convert from Page Type	Convert to Page Type
News Page ▼	Standard Page ▼

Convert from Property	Convert to Property
PageTitle	PageTitle ▼
MetaTitle	MetaTitle ▼
SixColumnImage	Remove property permanently ▼
ThreeColumnImage	Remove property permanently ▼

Figure 6-2. Converting content types in EPiServer. When the new type doesn't contain matching attributes for everything defined on the old type, hard questions result.

Oftentimes, you must swallow hard and give the system permission to simply throw that information away. As such, switching types is not for the faint of heart, especially when you have hundreds or even thousands of content objects based on a specific type.

Attributes and Datatypes

Content types are essentially wrappers around smaller pieces of information. Refer back to our previous definition of an Employee Bio. An Employee Bio is simply a collection of smaller pieces of information (First Name, Last Name, etc.).

Nomenclature differs, but these smaller pieces of information are commonly referred to as **attributes**, **fields**, or **properties**. (For the sake of simplicity, we'll use "attribute.") An attribute is the smallest unit of information in your content model, and it represents a single piece of information about a content object.

Attributes will be assigned a datatype, which limits the information that can be stored within it. Common, basic datatypes are:

- Text (of varying length)
- Number
- Date
- Image or file
- Reference to other content

Depending on the CMS, there might be dozens of possible datatypes you can use to describe your content types, and you might even be able to create your own datatypes which are specific to your content model and no other.

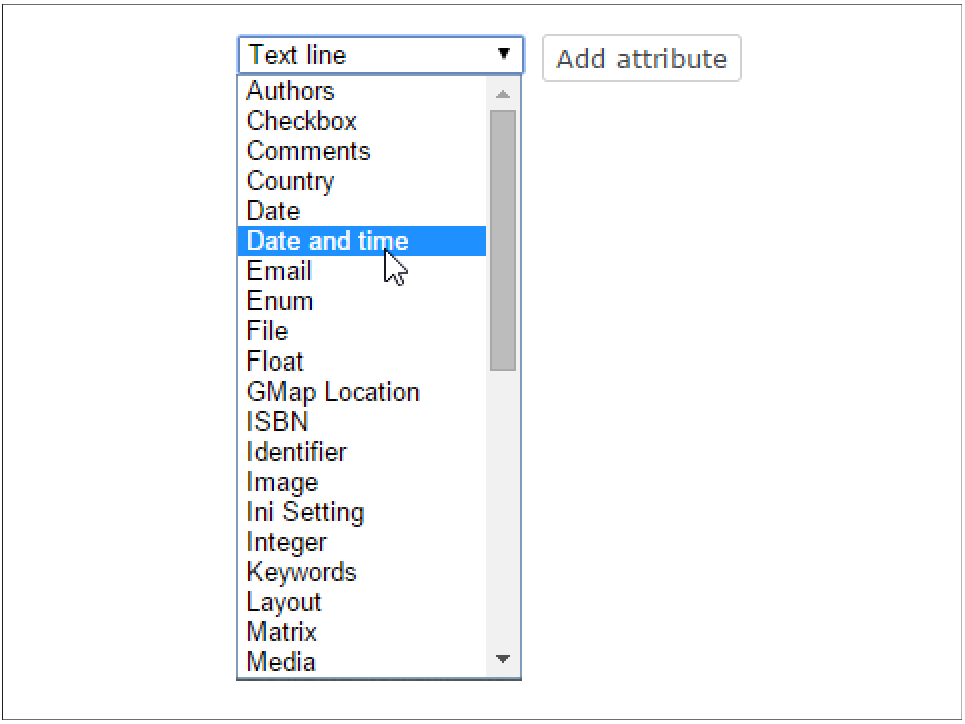


Figure 6-3. Predefined types of attributes available for eZ Platform.

Referring back to our previous model, we can apply the following datatypes:

- First Name (short text)
- Last Name (short text)
- Job Title (short text)
- Hire Date (date)
- Bio Sketch (long text)
- Manager (reference to another Employee Bio object)
- Image (reference to image file)

By specifying datatypes, you allow your CMS to do several things:

- **Validation:** The CMS can ensure that the information you enter is valid. For instance, Manager must be a valid reference to another Employee Bio object. (Additionally, perhaps the CMS prevent you from deleting that other object while there are still outstanding references to it.)

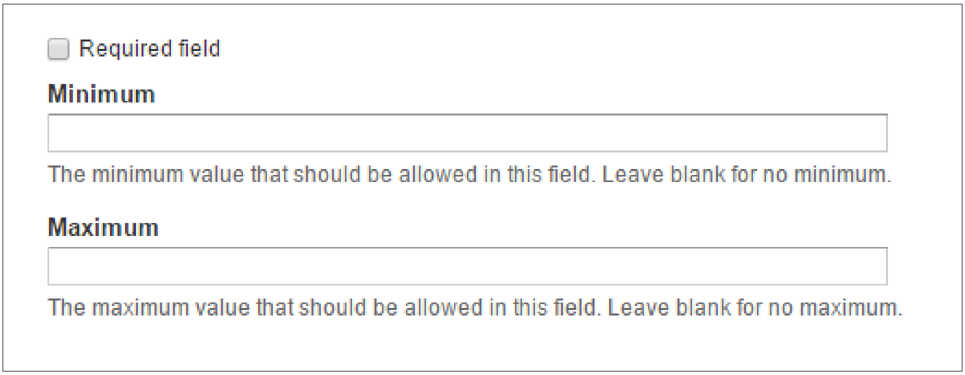
- **Editing Interface Generation:** The CMS can render different interface elements to easily work with different attributes. You might allow rich text in your Bio Sketch attribute, which means displaying a WYSIWYG editor with editing controls.
- **Sorting and Filtering:** The CMS understands how to use different datatypes to sort and filter content. By separating your Hire Date and ensuring it represents a valid date, you could locate all employees hired between two dates, and order them by seniority.

Some CMS do not allow the datatyping of attributes, but this is rare. In these cases, the CMS normally allows you to simply declare "custom fields" which it stores as simple text. In these cases, their utility is extremely limited, as you must ensure editors enter them correctly (enforcing a date format in a simple text field is tricky) and you must jump through hoops to use these values when sorting and filtering.

Attribute Validation

To ensure information is entered correctly, it needs to be validated, which means rejecting it unless it exhibits the correct format or value for its purpose.

Basic validation is enforced via the datatype. If something is meant to be a Number, then it has to be entered as a valid number. Additionally, the editing interface might enforce this by only displaying a small textbox that only allows entry of numeric characters.



□ Required field

Minimum

The minimum value that should be allowed in this field. Leave blank for no minimum.

Maximum

The maximum value that should be allowed in this field. Leave blank for no maximum.

Figure 6-4. Numeric validation options in Drupal.

However, the datatype doesn't tell the entire story. What if our number is only a number in *format*, but our intention for this number is for it to be a year. Then we need to potentially validate it in two other ways:

- It most likely needs to be a four-digit, positive integer (depending on whether or not we're allowing dates BC or not, or dates after 9999 BC)

- It most likely needs to be a specific range. For instance, we may require that it be in the past. Or that it be within a specific range (from 100 years in the past to the current year)

So, the datatype doesn't tell the entire story. We need to validate the attribute's *value*.

Pure custom validation can be valuable as well. Perhaps our product SKU from above needs to be cross-checked against our product database to ensure it exists. So, when an editor saves a content object, the CMS performs a query against the product database to ensure the product SKU exists, and rejects the content if it doesn't. Querying *your* product database is a requirement that exists in no other situation, so no system will support this out of the box. The best a system can do is provide you with the ability to program around it.

Value, pattern, and custom validation capabilities differ widely. If they don't exist, then the solution is often to create a custom datatype for the attribute which includes custom validation (even though the actual value stored is a simply numbers or text). If this isn't possible either, then the only solution available is to train editors well, and provide graceful error handling if they enter something incorrectly.

Content Type Inheritance

If content types are simply wrappers around sets of attributes, then it follows that we must create a new content type for every possible combination of attributes in our content model. This makes sense, since a Page uses a fundamentally different set of information than an Employee Bio.

But what if two content types are very similar? Many times, you'll have a content type that is exactly like another type, except for the addition of one or two extra attributes.

Consider a Page content type. To keep things as simple as possible, let's say it consists of nothing but:

1. Title
2. Body

Now we need another type for Blog Post. It needs:

1. Title
2. Body
3. Summary
4. Published Date

Do you see the similarity? A Blog Post is simply a Page with two extra attributes. You could do this for many different types of content. A Help Topic, for example, could be a Page with the addition of Software Version and Keywords. An Event could be a Page with the addition of Start Date and Location.

(The fact is, a base attribute set of Title and Body is so common to so many content types that many CMS simply build those in and you define content types by identifying only which attributes they possess *beyond* those two. Title and Body is often the base from which all other types are built.)

Sometime after your site launches, your marketing manager asks you to add a SEO Description to all the pages on the website. You're faced with the prospect of adding another attribute to all the types in your content model (and then deleting it when the marketing manager decides he doesn't want it anymore).

Wouldn't it be helpful if you could *inherit* the attribute defintion of Page and simply specify what attributes are available beyond that?

So the defintion of a Blog Post would be "everything a Page has, plus Summary and Published Date." By doing this, whenever the Page content type changes (via the addition of SEO Description, in this case) the Blog Post type — and all other types inheriting from Page — would change as well.

> To an object-oriented programmer, this is not new. Class inheritance has been a paradigm of this type of programming for decades. The same values of conciseness and manageability apply equally well to content management. By being able to extend one type into another, you gain increased control over your model as it changes in response to future requirements.

Sadly, content type inheritance is not common in CMS. Few systems offer it, though it seems to become slightly more common every year.

What's even more rare is multiple inheritance, which allows you to combine multiple types (or *partial* types) to create a new type. For instance, we could define a Content Header type as:

- Title
- Subtitle

Does this make sense as a type? Probably not — what content just has a Title and a Subtitle? That'd make for pretty useless content. However, when defined as a part of a larger type, this makes more sense. Many types might user a Title and a Subtitle as part of their defintion.

To this end, we might define an Article Body as:

- Body
- Image
- Image Caption

And we might define an Author Bio as:

- Author Name
- Author Bio
- Author Image

If we combine these three separate types, we might define an Article type as simply the combination of all three:

- Title
- Subtitle
- Body
- Image
- Image Caption
- Author Name
- Author Bio
- Author Image

How are we any better off in this situation? Because we could reuse the parts to define other types. We could use our Article Header type in an Image Gallery type, since Title and Subtitle are both common to that and an Article. Then, if we added something to Article Header, it would be added to all types which use that Type. (Note the word "use" rather than "inherit." You can "inherit" from one other type. You "use" multiple types.)

Again, this ability is rare but where available, it vastly increases the managability of a complicated content model.

Relationships

Modeling content is of two basic varieties:

1. **Discrete:** describing a type of content internal to itself
2. **Relational:** describing a type of content in how it relates to other content

In our Employee Bio example, we have both varieties. Attributes like First Name and Last Name are specific to a single content object only. The fact that one person's name is "Joe Smith" has no bearing on any other content.

However, the Manager attribute is a reference to another content object, which means it is "relational," or it defines how a content object "relates" to another content object.

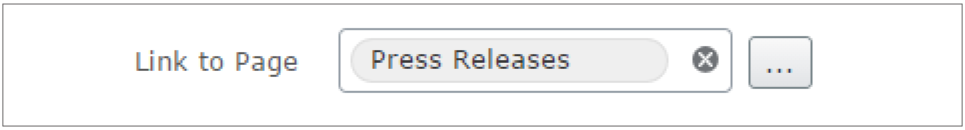


Figure 6-5. An example of relational modeling in EPiServer. A property entitled "Link to Page" allows editors to select another page in the CMS as its target.

Relational content modeling opens up a number of new challenges. Considering our Manager attribute again:

1. You must ensure that the content object to which this attribute refers is another Employee Bio, and not, for example, the "Contact Us" page.
2. Can an employee have more than one manager? This is an edge case, certainly, but if it happens, we either have to ignore it or modify the model to accomodate it. This means our Manager attribute must store multiple values.
3. How do we ensure the reference is valid? What if someone deletes the manager object? Are we prepared to handle an employee with no manager?
4. How do we allow editors to work with this attribute? Since this is a reference, our editors will likely need to go search for another employee to specify, which can make for a complicated interface

Highly relational content models can get very complicated and are enormously dependent on the capabilties of the CMS. The range of capabilities in this regard is wide. A small subset of CMS handles relational modeling well, and the "relationality" of your planned content model can and should have a significant influence on your CMS selection.

We will discuss relational modeling more extensively in the next chapter on Content Aggregation.

Content Composition

Some content isn't simple, and is best modeled as a group of content objects working together in a structure. Thus, a logical piece of content is *composed* of multiple content objects.

A classic example might be a magazine. We have a content type for Issue, but this is comprised of the following:

- One featured Article
- Multiple other Articles

We might do this by giving our Issue content type these attributes:

- Featured Article (reference to an Article)
- Articles (reference to multiple Articles)

We can create our Articles as their own content objects, and an Issue is essentially just a collection (an "aggregation") of Articles in a specific structure. Our Issue has very little information of its own (it might have a single attribute for Published Date, for example, and maybe one more for Cover Image), and exists solely to organize multiple Articles into a larger whole.

(This is a limited example. A much more common example requires aggregating content in a "geographic" structure, which we'll talk about in the next chapter.)

Content Model Manageability

Any content model is a balancing act between complexity, flexibility, and completeness. You might be tempted to account for every possible content situation, but this can have the side effect of limiting your flexibility or increase your complexity.

For example, when working with content types, editors need to be able to understand the different types available, how they differ, and when to use one over the other. In some situations, it might make sense to combine two types for the sake of simplicity and just account for the differences in presentation.

Could a Page content type double as a Blog Post? If a Page also has fields for Summary, Author, and Publication Date, is it easier to simply display those in the template only when they're filled in? If a Page is created within a Blog content type (more on this in the "Content Aggregation" chapter), then can we treat that Page as a Blog Post and give editors one less type to understand?

Whether this is easier or harder depends on your situation and your editors. If they work with the CMS often to create very demanding content, then they might be well-served with two separate types. If they create content so rarely that they almost have to be re-trained each time, then a single type might be the way to go.

If you don't have the ability to inherit content types, then it's to your benefit to limit content types as much as is reasonable. Having a content model with 50 different types becomes very hard to manage when someone wants to make model-wide changes.

(It's hard to place general rules around manageability, but limiting content types to the bare minimum needed is usually always a good idea. More types means more training for editors, more templating, and an increased likelihood of having to switch types after creation.)

The best you can do is to keep manageability in mind when creating or adjusting your model. Examine every request and requirement from the perspective of how this will affect the model over time. Almost every change will increase the complexity of the model, and is the benefit worth it?

A Summary of Content Modeling Features

Since this chapter has been largely about the theory of content modeling, it can be hard to draw out specific features or methods of system evaluation. Here's a list of what features are implied by content modeling and what questions to ask of a system:

- What is the a built-in or default content model? How closely does this match your requirements?
- To what extent can this model be customized?
- Does the system allow multiple types?
- Does the system allow content type inheritance? Does it allow multiple inheritance?
- Does the system allow for the datatyping of attributes?
- What datatypes are available to add attributes to types?
- Can you add custom datatypes to the system based on your specific requirements?
- Does the system allow multiple values for attributes?
- What editorial interfaces are available for each datatype?
- Does the system allow an attribute to be a reference to another content object? Can the reference be to multiple objects? Can this reference be limited to only those objects of a certain type?
- Does the system allow for permissions based on types?
- Does the system allow for templating based on types?
- How close can this system get to the (admittedly unreachable) ideal of a custom relational database?

A Note About Feature Lists

We discussed this in the last chapter, but it bears repeating:

When evaluating this list (and any other feature list in this book), please remember that the objective is not simply to check every box on the list, for three reasons:

1. It's doubtful that any system could satisfy every feature
2. It's not only important that the feature exists, but also *how well it works*
3. Some features aren't binary. Rather they exist on a range of functionality. Note several items in the list above with the phrasing "to what extent" or "what is available"? These are not yes/no answers. The feature exists along a scale of competence.

Finally, always remember that features only have value in comparison to your own requirements. As such, evaluate them in that context only.

CHAPTER 7
Content Aggregation

Imagine a restaurant with no menu — no menu board, no menus at the table, and no server to recite the food options to you. When you go to this restaurant, you're expected to simply know what food is available, or else just keep guessing until you find an entree they actually serve.

Clearly, this would be awful. Entrees in this restaurants would be like little hidden islands in the vast ocean of what's possible. If you don't stumble onto them, then you don't even know they exist.

A menu at a restaurant is an example of information which requires ***aggregation***: "a group, body, or mass composed of many distinct parts or individuals." For the purposes of choosing what you want to eat, individual menu items are fairly useless in isolation.

Aggregation is when you bring things together in a group of some kind for the benefit of the information consumer. In this case, a menu would be an aggregation of entree information for the purpose of selecting what you want to eat.

Content aggregation is the ability for a CMS to group content together for display to a site visitor. Note that we're not being too specific here — there are many types of aggregations and many ways a CMS might accomplish this. Furthermore, this ability is so obvious as to be taken for granted — we tend to simply assume every CMS does this to whatever degree we need.

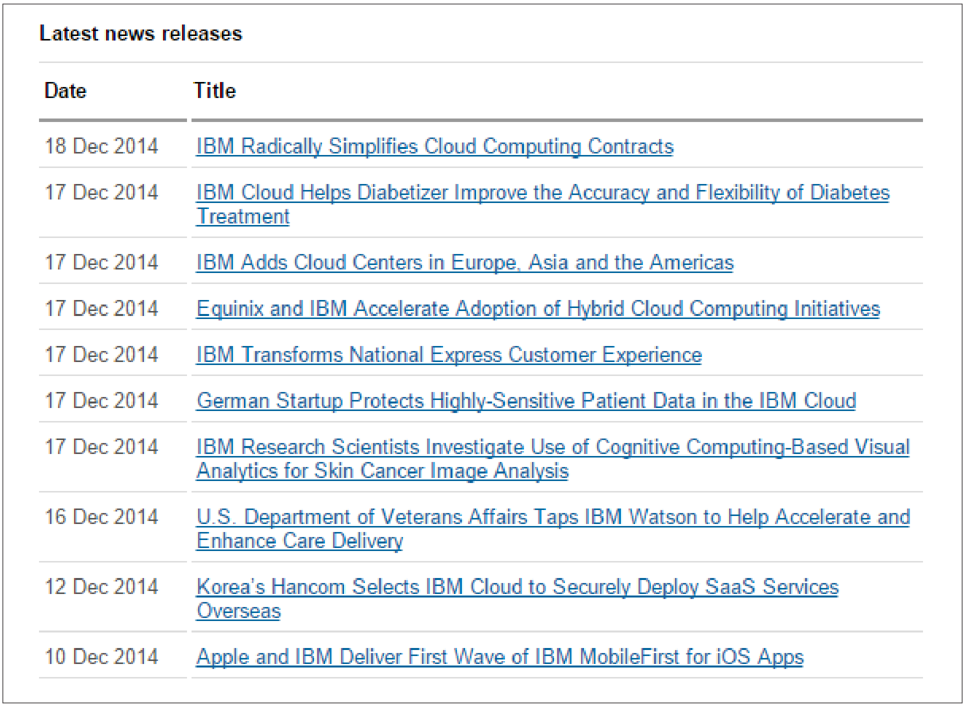
For example:

- Displaying navigation links is aggregation. At some point, you need to tell your CMS to display a series of pages in a specific order to form the top menu of every page (a *static* aggregation which is *manually* ordered).
- Index pages are aggregation. The page that lists your latest press releases is often simply a canned search of a specific content type, limited to the top 10 or so, and

displayed in descending order chronologically (a *dynamic* aggregation with *derived* ordering).

- Search is aggregation. When a user enters a search term and gets results back, this is a grouping of specific content (a *dynamic*, *variable* aggregation).

Aggregation is such a core part of most systems that it's assumed and often isn't even identified as a separate subsystem or discipline. But the range of functionality in this regard is wide, and breakdowns in this area are enourmously frustrating.



Latest news releases

Date	Title
18 Dec 2014	IBM Radically Simplifies Cloud Computing Contracts
17 Dec 2014	IBM Cloud Helps Diabetizer Improve the Accuracy and Flexibility of Diabetes Treatment
17 Dec 2014	IBM Adds Cloud Centers in Europe, Asia and the Americas
17 Dec 2014	Equinix and IBM Accelerate Adoption of Hybrid Cloud Computing Initiatives
17 Dec 2014	IBM Transforms National Express Customer Experience
17 Dec 2014	German Startup Protects Highly-Sensitive Patient Data in the IBM Cloud
17 Dec 2014	IBM Research Scientists Investigate Use of Cognitive Computing-Based Visual Analytics for Skin Cancer Image Analysis
16 Dec 2014	U.S. Department of Veterans Affairs Taps IBM Watson to Help Accelerate and Enhance Care Delivery
12 Dec 2014	Korea's Hancom Selects IBM Cloud to Securely Deploy SaaS Services Overseas
10 Dec 2014	Apple and IBM Deliver First Wave of IBM MobileFirst for iOS Apps

Figure 7-1. An aggregation of news releases on IBM's website as of December 2014

Few things are more annoying than having the content you want, but being unable to retrieve it in the format you need. I've been in numerous situations working with multiple systems where some editor threw up their hands in frustration and said, "All I want is to make *this* content appear in *that* place! Why is this so hard!?"

The answer to that question lies in a complex intersection of content shape, aggregation, and usability.

The Shape of Content

As we discussed in the last chapter, every set of requirements is different. For a content management project, these requirements revolve around the entire domain content being managed. Different domains of content have different “shapes.”

The “shape” of content refers to the general characteristics of a content model when taken in aggregate and when considered against the usage patterns of your content consumers. Different usage and models result in clear differences between content and their aggregation requirements.

- **Serial:** This type content is supplied in a serial “line,” ordered by some parameter, usually date. An obvious example is a blog, which is a reverse-chronological aggregation of posts. Very similar to that are social media updates — a tweet stream, for instance — or a news feed. This content does not need to be organized in any larger construct beyond where it falls in chronological order relative to other content.
- **Hierarchical:** This type of content is organized into a tree. There is a root point in the tree that has multiple children, each of which may have one or more children, and so on. Sibling content items (those items under the same parent), can have an arbitrary order. Trees can be broad (lots of children under each parent) or narrow (fewer children), and shallow (fewer levels) or deep (more levels). An example of this are the core pages of many simple, informational websites. Websites are generally organized into trees — there is primary navigation (Products, About Us, Contact Us), which leads to secondary navigation (Product A, Product B, etc.). Navigational aggregations for these sites can often be derived from the position of content in the tree.
- **Tabular:** This type of content has a clearly defined structure, with large amounts of content of one type, and is usually optimized for searching, not browsing. Imagine a large Excel spreadsheet with labeled header columns and thousands of rows. An example would be a company locations database. There might be 1,000 locations, all clearly organized into columns (address, city, state, phone number, hours, etc.). Users are not going to browse this information. Rather, they search it based on known parameters.
- **Networked:** This type of content has no larger structure beyond the links between individual content objects. All content is “equal” and unordered in relation to other content, with nothing but links between the content to tie it together. The obvious example of this is a wiki. Wikis have no structure (some allow heirarchical organization of pages, but most do not), and the entire body of content is held together only by the links between pages. A social network — if managed as content — would be another example. People are content in the network, who are randomly connected (“friends”) with other content.

- **Relational:** This type of content has a tightly defined structural relationship between multiple, highly structured content types, much like a typical relational database. The Internet Movie Database, for example, has Movies, which have one or more Actors, zero or more Sequels, zero or more Trivia Items, etc. These relationships are enforced — for instance, you cannot add a Trivia Item for a Movie which doesn't exist. A Trivia Item is required to be linked to a Movie, and cannot be added until the Movie exists.

Different CMS have different levels of ability to handle the different shapes of content. For example:

- WordPress is well-suited to manage **serial** content (blog posts), but you couldn't easily run a highly herarchical help topic database with it.
- MediaWiki is designed to handle **networked** content, but it would be extremely inefficient to try to run a blog from it.
- WebNodes is perfect for defining and managing **tabular** and **relational** content. Interestingly, this also gives it the ability to manage **serial** content well (a blog is essentially a database table ordered by a date field), but it would make sense to highly structure a **networked** wiki with it.

In our examples, we simplified by pigeonholing websites to one shape, but the truth is that different sections of the same website will model content differently. The average corporate website might have many marketing pages organized into a tree (hierarchical), a dealer locator (tabular), and a news feed (serial). The content managed in each section has a different shape.

Additionally, when we say a particular system is not suited to a paticular shape of content, what we're saying is that this system is not *best suited* to work with that type of content. It's important to note that almost any system can be contorted to work with any type of content, though this either requires heroic development efforts or results in a very complex and confusing editor experience (often both).

Content Geography

Most every system has some core method of organizing content. Very rarely do editors just throw content into a big bucket — normally, content is created "somewhere," which means it exists in a location relative to other content.

Much like geography refers to the spatial relationship of countries, **content geography** refers to the spatial nature of content — where it exists "in space," or how it is organized relative to the content "around" it.

The quotes in that last paragraph underscore the idea that we're trying to place some physical characteristic on an abstract concept. Geographies attempt to treat content as a physical thing which exists in some place in a theoretical space representing the domain of all of your content.

The most common example of a geography is the **content tree** where content is organized as a parent-child structure. All content objects can be the parent of one or more other content objects. Additionally, each object has its own parent and siblings (unless it's the root object, which means it's at the very top of the tree).

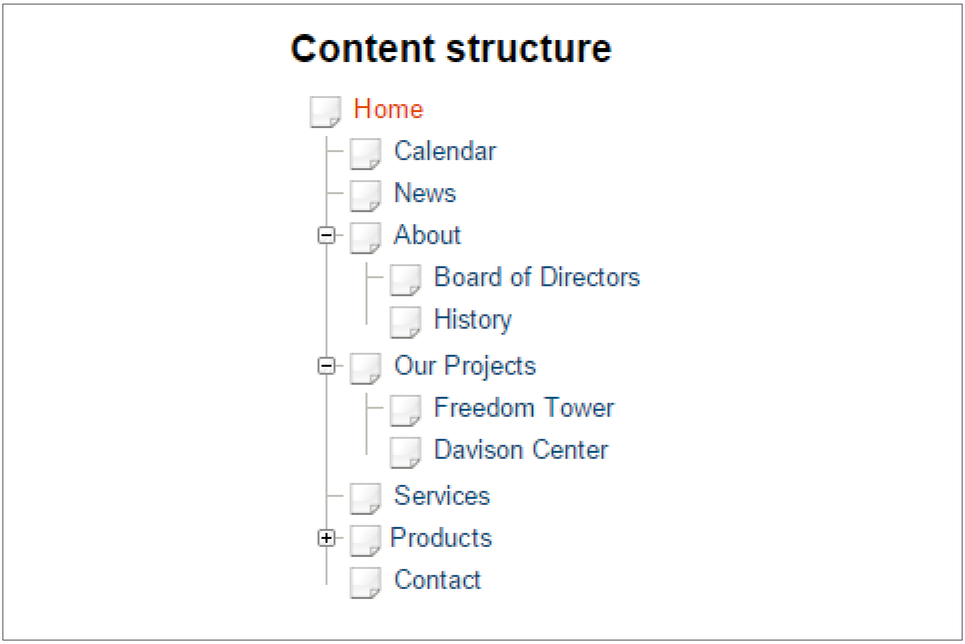


Figure 7-2. A typical content tree in an eZ Platform-powered website

In this sense, all content is created in some location. You add content as a child of some other content object, and because of this, it instantly has relationships with multiple other content objects. Depending on where it's placed, it might come into the world with siblings and (eventually) might have children, grandchildren, etc.

This content is hierarchical, and this geography allows us to traverse the tree and make decisions based on what we find. For instance, we may form our top navigation automatically from the children of the root object. Or we may list the children of a certain page as navigation options when displaying that page to a user.

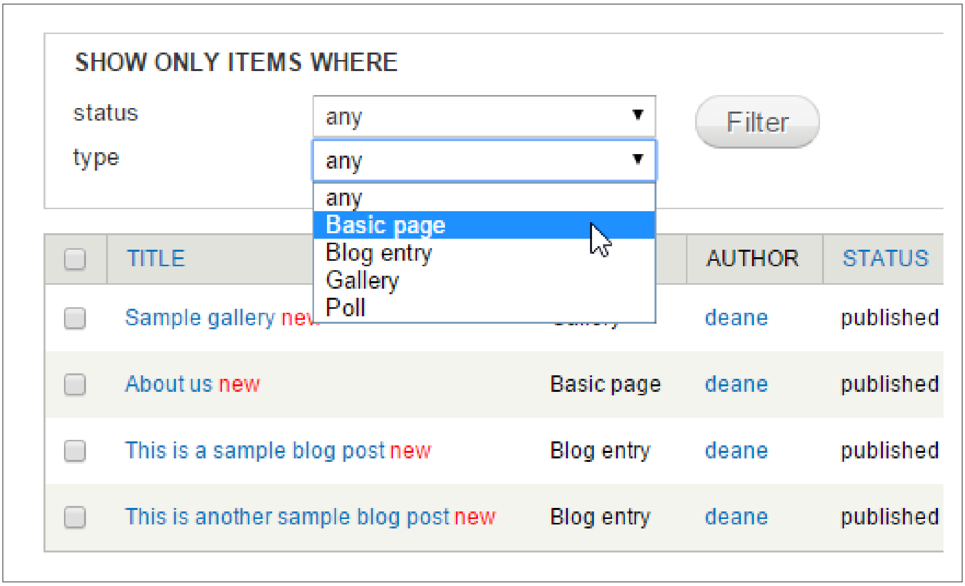
In a content tree geography, content is often discussed in terms of "levels." The most important pages in your site are "Top Level Pages" or "First Level Pages." Under that we have "Second Level Pages," then "Third Level Pages."

Less common is the **folder structure** of organizing content. Systems using this model have “folders” into which content can be placed. This might seem very similar to the Content Tree in that it’s hierarchical, but there’s an important difference: *the folder is not itself a content object*. It’s simply an administrative structure for editors to organize their content. Content objects are not children of other content objects, nor do they have children. Instead, we have folders and subfolders in which content objects are placed.

While this structure is generally clear for editors — we’ve been using files and folders for years in every operating system — it can be limiting in practice. The inability to directly relate content by parentage removes the ability to use this to aggregate content.

If we can’t have children of a page, then we have to find another way to tie these pages together. The only spatial relationship content in these system exhibits is a sibling relationship with content in the same folder. (What makes this more complicated is that folders are often treated as simple buckets which don’t allow manually ordering of the content within them — more on that below.)

Other systems might depend on a simple **type segregation** model, where the content type (from our last chapter) of a particular piece of content is the major geography. You can easily subdivide the entire domain of content by content type and perhaps some other paramaters, but sometimes not much else. We can easily see all the Press Releases, for example, but you don’t create content in any “location,” and it doesn’t exist with a spatial relationship to any other content.



SHOW ONLY ITEMS WHERE

status: any ▼

type: any ▼

□	TITLE		AUTHOR	STATUS
□	Sample gallery ne		deane	published
□	About us new	Basic page	deane	published
□	This is a sample blog post new	Blog entry	deane	published
□	This is another sample blog post new	Blog entry	deane	published

Figure 7-3. Listing content by type in Drupal

Like the folder structure, this can limit our options considerably, but it can be appropriate in many cases. For instance, if your website is a high-volume blog, then your editors might not need to place their content anywhere. They might just need to create a Blog Post or a Movie Review and let the system itself sort out how to display it. In these situations, editors are creating serial, isolated content that doesn't need any spatial relationship (or, rather, its spatial relationship to other content is derived, usually by date).

It's important to understand that a content geography is simply the main organization method a system exhibits. Systems usually offer multiple ways to aggregate content so when the core geography falls short, there are usually multiple paths to accomplish the desired aggregation.

Content geographies are administrative concepts only. Visitors to a website will not usually be aware of the underlying geography. They may be vaguely aware that pages are ordered hierarchically, or that news items are ordered by date. However, at no time will the average website call attention to the content tree (with the possible exception of a site map) or explicitly name pages as "parent" or "child."

Aggregation Models: Implicit and Explicit

There are two major models for building aggregations:

1. Implicit / Internal
2. Explicit / External

Content can implicitly be created as part of a larger structure. This is common in the content tree and type segregation models. When you create the page for "Product X" under the "Products" page, you have implicitly created a relationship. Those two pages are inextricably bound together as parent and child.

Put another way, their relationship is *internal* (or "implicit") — each content object knows where it lives in the structure. The fact that "Product X" is a child of "Products," or that your Press Release is part of the larger aggregation of all press releases, is a characteristic that is inextricable from the content.

(This also gives implicit structures the reputation of inflexibility, which is sometimes deserved.)

The opposite of this is to explicitly create an aggregation. Many CMS (Drupal, for instance) have a menuing system where you create a structure and bind content to it. You might have a Main Menu, which is a hierarchically-organized structure of content.

In this case, the fact that Product X is a child of Products is only true in relationship to the Main Menu we created. The structure in these cases is *external* (or “explicit”). The structure doesn’t exist within the content itself, but rather in a separate object — the menu.

One of the benefits of explicitly creating aggregation structures is that content can take part in more than one. You might have a dozen different menus in use around your site, and the Products page could appear in all of them.

One potential issue is that the aggregation structure is often not a content object itself. If our menu is not a content object, we might lose considerable content-related functionality, like workflow, permissions, templating, URL mapping, etc.

Aggregation Functionality

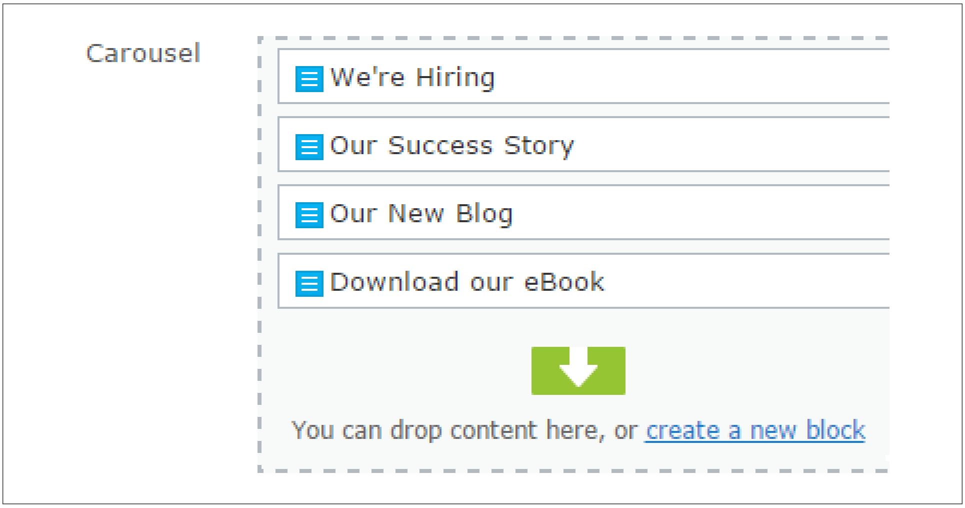
Content aggregations can have multiple features, the presence or absence of each will drastically affect an editor’s ability to get the result they want.

Static vs. Dynamic

A particular aggregation might be static, which means an editor has arbitrarily selected specific content objects to include in the aggregation, or dynamic, meaning the content of the aggregation is determined at any particular moment by specified criteria.

- A static aggregation might be an index of pages from your employee handbook that new employees should review. In this case, you have specifically found these pages, and included them in this aggregation via editorial process. For a new page to be on this list, you need to manually include it.
- A dynamic aggregation might simply be a list of all of the pages of the employee handbook. For a new page to be on this list, it simply needs to be added to the employee handbook. Its appearance on this list is simply a byproduct of that.

A dynamic aggregation is essentially a “canned search" — a set of search criteria which are executed at any moment in time to return matching content.



Carousel

- We're Hiring
- Our Success Story
- Our New Blog
- Download our eBook

You can drop content here, or create a new block

Figure 7-4. A statically-created list of content for an image carousel in EPiServer

Dynamic aggregations can often be a byproduct of the content gegraphy. In the case of a Content Tree, the children of a parent page is a dynamic aggregation. With all such systems, it's possible to obtain an aggregation of child pages, and a new item will appear in this aggregation simply by virtue of being created under the parent. This is no different than a canned search with the criteria that the returned content must be children of a particular parent.

Likewise, a dynamic aggregation might be "show me all the press releases." In a system relying on Type Segmentation as its core geography, simply adding a new Press Release content object will cause a new item to appear in this aggregation.

Variable vs. Fixed

A subset of dynamic aggregations are those that can vary based on environmental variables. Even if the domain of content doesn't change, what appears in a dynamic aggregation might be different from day to day, or user to user.

Search is a classic example of a dynamic, variable aggregation. What is displayed in a search results page depends primarily on user input — what the user searches for. You may specify some other criteria, such as what content is searched and how the content is weighted or ordered, but a search is still created in real time based on user input.

Other aggregations might be based on user behavior. A sidebar widget displaying "Recommended Content" might examine the content consumed by that visitor during the current session to determine similar content of interest.

Aggregations might be based on the date — events under the "This Day in History" sidebar, for example, clearly relies on the current date to aggregate its contents. Likewise, "Locations Near You" relies on a geolocation of the visitor's latitude and longitude.

Manual Ordering vs. Derived Ordering

Once we have content in our list, how is it ordered? What appears first, second, third, etc? In some cases, we need to manually set this order. In other cases, we need or want the order to be derived from some other criteria possessed by the content itself.

- In the case of our employee handbook, if we were creating a curated guide of pages that new employees should read, then in addition to manually selecting those pages, we'd likely want to arbitrarily decide the order in which they appear. We might want more foundational topics to appear first, with more specific topics appearing in the last further down.
- If we have a list of "Recently Updated Handbook Topics," then in addition to this list being dynamic (essentially a search based on the date the content was changed), we would want this content ordered reverse-chronologically, so the most recently updated topics appeared first.

It's obvious in our example that the characteristic of static vs. dynamic and manual ordering vs. derived ordering often go hand-in-hand. It's relatively rare (though not impossible) to find an arbitrary aggregation that should have derived ordering. However, in most cases, if editors are manually aggregating content they also want the ability to manually order it.

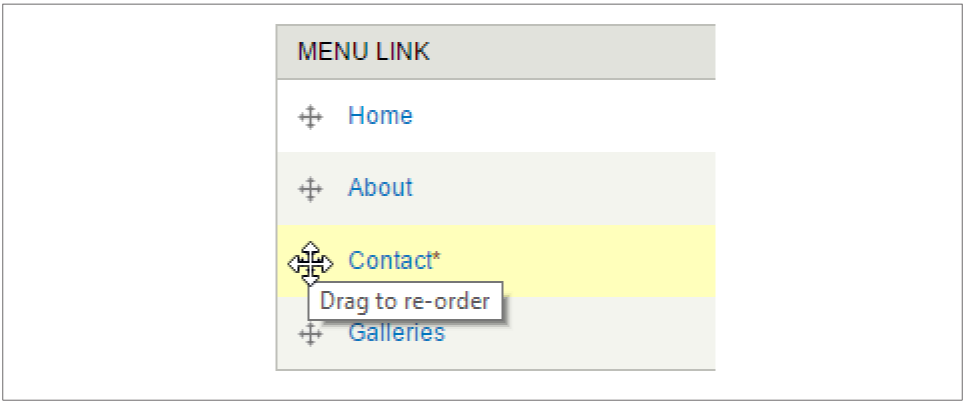


Figure 7-5. Manually re-ordering a menu in Drupal

The other option — a dynamic aggregation which is manually ordered — is logically impossible. If an aggregation is dynamic, then its contents are not known at creation time (indeed, you're not building an aggregation as much as you're simply configuring search parameters), so there's no way this aggregation can be manually ordered. You can't manually order a group of things for which the contents are unknown.

Manual ordering of dynamic aggregations can be approximated by "weighting" or "sort indexing," whereby the content has a property specifically designed to be used in sorting.

This works in most cases, but it can be quite loose. If one page has a sort index of 2 and another has 4, then there's nothing stopping an editor from inserting something at 3. Indeed, in many cases this is what the editor wants to do, but in other cases, the editor might do this ignorant of the content consequences (remember, they're editing the content itself to do this, not the content's inclusion in the larger aggregation — they may not even be aware of the larger aggregation).

Furthermore, to allow this type of ordering, you need to have *a different sort index for every single dynamic aggregation in which the content might appear*. Obviously, this is nigh impossible. Dynamic aggregations, by defintion, can be created to return arbitrary searches, so there's no way to speculate the sum total of all aggregation in which a content object might appear, nor is it possible to speculate on the *other* content in this aggregation, so as to manually tune a sort index.

Suffice to it say that in very few cases is it possible to manually order a dynamic aggregation of content.

Type Limitations

It's not uncommon to only allow certain content types in specific aggregations. If the aggregation is dynamic and we specify the type in our search criteria ("show me all the press releases"), then this is natural. However, in other situations, we might want to limit types because the content has to conform to a specific format in order to be used in the output.

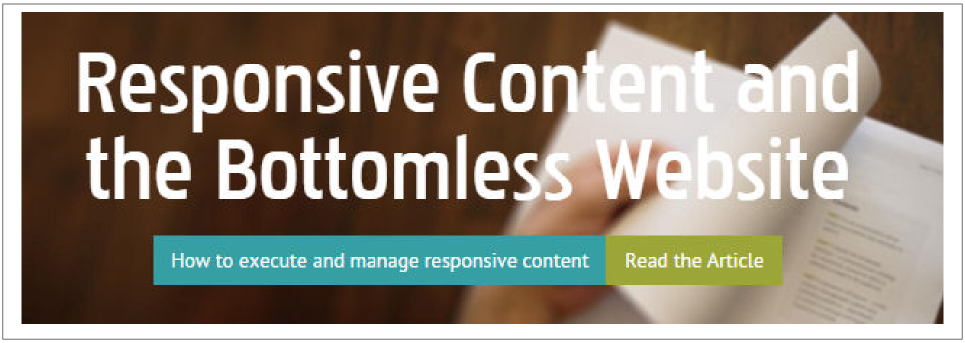

Responsive Content and the Bottomless Website
How to execute and manage responsive content
Read the Article

Figure 7-6. A image carousel frame

For instance, consider the image carousel frame depicted in Figure 7-6. This is one frame of a multi-frame gallery, powered by an aggregation (a flat list) of multiple content objects. This list is manually created and ordered.

To render correctly, every item in this aggregation must have:

1. A title
2. A short summary
3. An image
4. Some link text (the "Read the Article" text)
5. A permanent URL (to route the visitor when they click on the link text)

Only content confirming to this pattern can be included in this aggregation. This means, for example, that an Employee Bio is out, because it doesn't have a summary.

Since this aggegation is most likely static (image carousels are always a curated list of content), then we have to have a way to limit editors in the interface to only select content which is of the type we need. If we can't limit this by type, then we run the risk of our image carousel breaking if it encounters content not of the type it needs. (A smart template developer will account for this and simply ignore and skip over content that doesn't work. This prevents errors, but will likely confuse an editor who doesn't understand the type limitations.)

This limitations are not uncommon in Content Tree geographies. It's quite common to be able to specifiy the type of content that can be created as children of other content. For example, we might be able to specify that a Comment content type can only be created as a child of a Blog Post content type, and the inverse — Blog Posts can only accept children that are Comments.

Quantity Limitations

This is less common, but some aggregations can store more content than others, and some systems allow you to require a certain number of content objects, or prevent you from going over a maximum.

Consider our image carousel — it might need at least two items (or else it's not a carousel), and is limited to a maximum of five (or else the formatting will break). It would be helpful if the interface could limit editors to adding items only within those parameters.

Permissions and Publication Status Filters

In one sense, an aggregation — be it static or dynamic — should be a “potential” list of content. Every aggregation on a site should be dynamically filtered for both the current visitor, and the publication status of the content.

If you manually create an aggregation with 10 items, but the current visitor only has permissions to view three of them, what happens? Ideally that list should be dynamically filtered to remove the seven items that shouldn’t be viewed. The same is true with publication status, specifically start and end publish dates. Content prior to its start publish dates shouldn’t be included, and neither should content after its end publish date.

What this means is that an aggregation — even a static one — might show different content for different visitors, and under certain conditions *some visitors might not see any content at all*.

Flat vs. Hierarchical

Many aggregations are simply flat — our list of employee handbook pages, for example, or our image carousel. But other aggregations are often hierarchial structures of content.

In these cases, we have multiple flat aggregations with relationships to each other. A hierarchical list is basicially multiple flat lists nested in one another. The top level is one flat list, and each level can either be a single content object, or another flat list, and so on all the way down. The only difference (and it’s an important one), is that these flat aggregations are aware of each other — any one of them knows that it has chidren, or a parent.

The main menu for your website is a clear example. Websites often have an overhead menu bar that either contains dropdown submenus for second-level pages, or secondary navigation that appears in the sidebar menu.

(And I hope it’s obvious by this point that a content tree geography is one big, hierarchical content aggregation.)

“Decorated” Aggregations

In some situations, the inclusion of content in an aggregation requires additional information to make sense. In these cases, the inclusion of content becomes a content object in itself.

For example, let’s say we’re aggregating a group of Employee Bio content objects to represent the team planning the company Christmas party. To do this, we will create a static aggregation of content.

However, in addition to the simple inclusion in this aggregation, we want to indicate the role this employee plays in the group represented by the aggregation. So, in the case of Mary Jones, we want to indicate that she is the "Committee Chair." Mary is actually the Receptionist at the company, and this is the title modeled into her Employee Bio object.

The title of Committee Chair only makes sense relative to her inclusion in this aggregation, and nowhere else. Therefore, this attribute is not on the aggregation or on the Employee Bio. This attribute rightfully belongs on the attachment point between the two.

In this sense, her inclusion in this aggregation *is a content object in itself*, and our committee is really an aggregation of Committee Assignment content objects, which are modeled to have a Title and a reference to an Employee Bio. In this sense, the Employee Bio object is included in the aggregation "through" a Committee Assignment object.

Now, clearly, this gets complicated quickly, and this isn't something you would do for a one-off situation. But if situations like this are part of your requirements, then modeling an aggregation inclusion as a content type by itself can allow you to model the relationship.

By Configuration or By Code

As we briefly discussed in Chapter 2, certain things in a CMS environment can be accomplished by non-technical editors working from the interface, and other tasks need to be handled by developers working with the templating or the core code of the system.

Aggregations are no different. As a general rule, editors can aggregate content any which way the system allows — they have complete freedom. A subset of these capabilities are provided to editors to create and display aggregations as part of content creation. How big this overlap is depends highly on the system, and partially on the sophistication of your editors.

The simple fact is that aggregations can get complicated. A list of blog posts seems quite simple, but the number of variables it involves can spiral out of control more quickly than you think.

- What content should be included?
- Where should this content be retrieved from?
- How should they be ordered? Should there be controls for visitors to order by a different criteria?
- Should the posts come from one category? From one tag? From more than one tag?
- Should they be filtered for permissions?

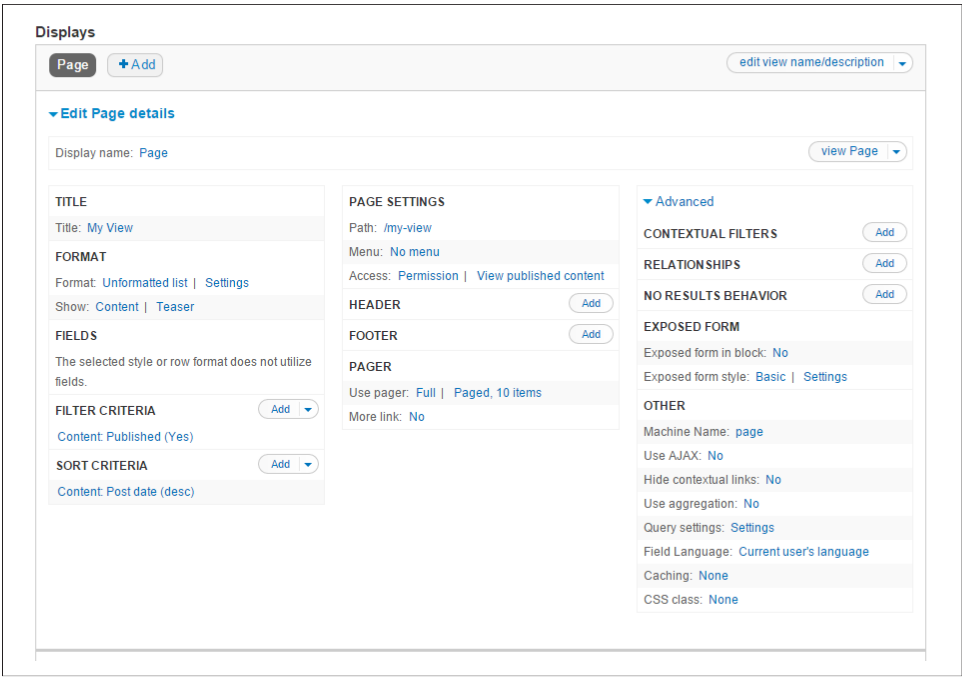
- Should they be filtered by date?
- How should they be formatted? Should we display a summary? Should we truncate the main body? Should we include the date? The author? In what font-size?
- Should they link to the post itself?

These variables are usually quite simple for a developer to code, but they get very complicated for an editor to configure via an interface.

The Drupal Views modules provides a wonderful example of this basic complexity. Views is a module that allows editors to create dynamic aggregations of content by configuring search parameters and formatting information. It provides an enormous number of options in order to provide editors extreme flexibility.

Views has been developed over many years, and the interface has been through several rewrites to be as simple and usable as possible. However, complexity remains. There's simply a basic, unresolvable level of complexity that goes with the flexibility that Views offers.

Consider the interface presented in Figure 7-7. You could very easily spend an entire day training editors on just the functionality that Views offers.



Displays

Edit Page details

Display name: Page

TITLE

Title: My View

FORMAT

Format: Unformatted list | Settings

Show: Content | Teaser

FIELDS

The selected style or row format does not utilize fields.

FILTER CRITERIA

Content: Published (Yes)

SORT CRITERIA

Content: Post date (desc)

PAGE SETTINGS

Path: /my-view

Menu: No menu

Access: Permission | View published content

HEADER

FOOTER

PAGER

Use pager: Full | Paged, 10 items

More link: No

Advanced

CONTEXTUAL FILTERS

RELATIONSHIPS

NO RESULTS BEHAVIOR

EXPOSED FORM

Exposed form in block: No

Exposed form style: Basic | Settings

OTHER

Machine Name: page

Use AJAX: No

Hide contextual links: No

Use aggregation: No

Query settings: Settings

Field Language: Current user's language

Caching: None

CSS class: None

Figure 7-7. Drupal Views module configuration

Developers have it easier, since code is more succinct and exact, and they''re are more accustomed to thinking abstractly about information concepts and codifying those abstractions. That said, the quality of the API provided varies greatly. Some are elegant, succinct, and comprehensive, while others are awkward, verbose, and have frustrating gaps that prevent even a developer from aggregating the desired content in the desired format.

Training and Repitition

They say that practice makes perfect, and the same is true of content editing. Editors will remember things they do often, and forget things they do seldom. Editing content is something they do often. Creating content aggregations is generally done much less.

This usually means that no matter how well they were trained originally, editors will forget seldom-used features which require intricate functional knowledge, and content aggreagtion configuration clearly fits this description.

This means that a system designed to give editors control over aggregated content is often more effective at generating a support call every time an editor tries to use it. It’s not uncommon for developers to have to configure aggregations for editors, using the interface that ironically was created to allow them to do it *without* developers.

A Summary of Content Aggregation Features

- What is the core content geography in use by the system? Does it have a master aggregation model, into which all content is structured?
- Can parent-child relationships be easily represented?
- What abilities do developers have to aggregate content by code?
- What abilities do editors have to aggregate content by configuration in the interface?
- Can editors create static aggregations?
- Are these aggregations flat or hierarchical?
- Can static aggregations be manually ordered?
- Can static aggregations be limited by type?
- Can editors configure dynamic aggregations? Based on what what avalable criteria?