Exware Solutions Inc.

ExSite 4 Content Model

Exware Solutions Inc. 102-131 Water Street, Vancouver, BC V6B 4M3 **T** 604.684.9440 **F** 604.684.5894 morgan@exware.com www.exware.com

Introduction

ExSite is built in layers:

4	Plug-in Modules (optional components with their own custom interfaces)
3	Plug-in Frameworks (optional shared data models, like Finance)
2	CMS (general-purpose tools for displaying web content)
1	Kernel (general-purpose tools for managing data & security, as well as the back-end admin system)

ExSite v4 is primarily an overhaul of layer 2, the CMS. This document covers the changes in the CMS layer. For other v4 changes in other layers, see the *ExSite 4 Developers Guide*.

Some of the disadvantages of v3 content model that we are trying to address with v4 are:

- core content management (for example, translations, revision control, etc.) was limited to normal pages, and not available to other similar content types like articles, products, or events. This often meant that features had to be reinvented when needed for other content types (eg. translations in forms).
- certain types of DB queries were slow due to the data types used in the revision table
- over-reliance on dynamic content views for anything that was not core content (=slow)

The v4 "content" object is a much more abstracted object that can serve as a section, page, or content object, plus various other types of object (such as articles, products, events, and so on). Any content that can fit this model can make use of the core content management features such as publishing/static delivery, indexing, menuing, translations, revision control, templating, metadata, security, access controls, and workflows.

Content Structure

v3	v4
section > page > content > content_data	content > revision > view
section.parent_id, page.parent_id, content.page_id	content.parent
content_data.content_id	revision.content_id
page.template_id	content.template
page.parent_id (for alternate pages)	content.master
section.template_id	n/a
page filename=index.html	section

Note that the home page of a section is the section itself, not a special page named “index.html”.

Content objects are related to each other in 3 ways:

1. Parent-child relationship (content.parent): This is the navigational structure of the content. Content objects are organized into a single tree/hierarchy, so that every content object has a single parent, 0 or more children, and a simple path that addresses the content.

2. Template (content.template): The template is a content object that is used for displaying the content and resolving relative content references, as in v3.

3. Original content reference (content.master): This is the original content that this content object refers to. This is used for translations (for example, the French version can point back to the original English version). In v3, the translation relationship was an overloading of the parent-child relationship for pages of "alternate" type. In v4, the translation relationship is a separate and distinct relationship. Original content is also used in the case of aliases, where one content object duplicates another, but they only have to be edited in one place.

Content Paths

v3	v4
foo.com/contact.html	foo.com/contact
foo.com/directions.html	foo.com/contact/directions
foo.com/cgi/page.cgi/directions.html	foo.com/cgi/ex.cgi/page/contact/directions
foo.com/map.jpg	foo.com/contact/directions/map.jpg
foo.com/cgi/content.cgi/map.jpg?id=1234	foo.com/cgi/ex.cgi/view/contact/directions/map
foo.com/_Library/images/picture.jpg	foo.com/images/picture.jpg
foo.com/_Template/base/stylessheet.css	foo.com/base/stylessheet.css

Each content object has a path, defined by its parent-child relationships. The path is used for addressing the content in dynamic queries, as well as for publishing the content to disk.

Each type publishes either as a directory or as a file. If as a directory, then the content can be viewed at its path:

http://foo.com/path/to/content

This is equivalent to:

http://foo.com/path/to/content/index.html

but the "/index.html" is optional, and usually left off to simplify the URL.

If the content is published as a file, then it can be viewed as a filename at the path of its parent:

http://foo.com/path/to/content/filename.html

Since each content type has control over its own publishing, these default rules can be bent. For example, calendars can publish their events to special dated subdirectories, for example:

http://foo.com/calendar/2012/09/event-X.html

In this example, the calendar publishes to the directory /calendar/ and the individual events publish to files, eg. event-X.html. But the calendar first creates dated subdirectories to hold these events.

If a content item is restricted access or dynamic, then the published version still exists, but consists of a redirect to the dynamic version. So the static URL always works for GET requests.

Dynamic URLs

The dynamic URL to a page at path /foo/bar is

> **http://foo.com/cgi/ex.cgi/page**/foo/bar

The dynamic URL to content at path /foo/photo is

> **http://foo.com/cgi/ex.cgi/view**/foo/photo

This will display the "bare" content directly. To display the content formatted (in the style of a page), simply change this to the page-style URL:

> **http://foo.com/cgi/ex.cgi/page**/foo/photo

This does not actually make it a page; it merely formats it as a page, with HTML wrappers, menus, etc. surrounding the content.

Note that any content object can define its own custom URL. This can be a remote URL (eg. you can create a "page" in your menus that actually links to a remote website), or it can be a local URL (which needs to either make sense to the webserver, or needs to be configured separately using some kind of server alias).

Static vs. Dynamic Content

You do not have to specify whether content should be static or dynamic. ExSite 4 content objects can intelligently make their own decisions about their presentation mode, by using their own internal logic to respond to the *publish_rule()* method (which can return static, dynamic, hourly, daily, or weekly). You can still manually set a publish rule for a particular piece of content, but that overrides the logic of the content class.

Examples: Member-only content defaults to dynamic, since it depends on the access level of the person viewing it. Calendars default to publish daily, since that is when the upcoming events listings may change. Events accepting registration default to dynamic before the event, so that fee deadlines and sellouts are displayed accurately, and static after the event is done.

Note that HTML content may include plugins; in that case, those plugins are queried using *$module->ioctl("PublishRule")* to find out if the plugin recommends a particular publish rule. For example, menu plugins can reply with "static" since menus are always the same on a particular page. Pages without a manually set publish rule will use the most dynamic publish rule requested by the plugins contained in that page.

Permalinks and Canonical URLs

The permalink to a content object is the "best" user-friendly URL to the content. This can be a relative, static URL, even if the content is ultimately delivered dynamically. Permalinks should generally be used for URLs that are targeted at people, such as internal hyperlinks and published URLs.

Permalink Example: **/sale**

The Canonical URL to a content object is the most direct absolute URL to the content. If the content is ultimately dynamic, the canonical URL must also be dynamic. Canonical URLs should be used for search engines, so they access the content directly.

Canonical URL Example: **http://foo.com/cgi/ex.cgi/page/sale**

Canonical URLs are determined automatically. However ExSite 4 also supports a manual override: allow a content type to define metadata under the name "canonical", and that will be used instead. This might be useful if referring to off-site URLs.

Revisions

v3	v4
content_data	revision

Each content object can have multiple revisions. In v3, there were two special revisions, newest and active. In v4 there is the additional concept of a draft revision:

1. the draft revision is the most recently added revision
2. the newest revision is the most recently added revision that is approved for public viewing
3. the current revision is the most recently published revision

When saving a new revision, they will automatically be saved as a “newest” revision, unless you “save as draft”.

In ExSite 4, no actual content is stored in the revision record. The revision record contains only the creation and publication timestamps, original mime-type, and comment. The content itself is stored in the views.

Views

v3	v4
content_data.data (or fdata)	view/normal
content_data.thumb	view/thumbnail

Each revision can have multiple views. A view is simply a viewable representation of the content. ExSite 3 supported only a single view of the content, with an exception for images, which could also have thumbnails. In ExSite 4 any MIME-type can have multiple views, and five views are supported:

normal: this is the default view that would normally be inserted into web pages,

large: a larger or more high-res view than the default; for example, an HD video, or high-res photo.

small: a smaller version than the default; for example, a low-res view for previews or index listings

thumbnail: a very small version, suitable for thumbnail galleries

icon: an ultra-small version, suitable for inlining into text

When content is installed, a normal view is created by default. Any of the other views can also be created. The views do not have to be the same mime-type as the original content. For example, a video could have an image as its small view (and also as its thumbnail and icon, for that matter).

When inserting views into pages, the normal view is used by default, but you can select other views optionally. If the requested view is not available, it will automatically find the closest matching view. For example, if you ask for a thumbnail

image, but there is only a small image, the system will simply resize the small image to the standard thumbnail dimensions.

Workflow

v3	v4
active	published
disabled	canceled
archived	archived
	approved, queued, submitted, draft, expired, rejected

Workflow status determines how the content is handled and displayed to visitors. The states are numbered numerically, with lower values being "more live" and higher numbers being "more dead":

0. **published** - the content is live and fully interactive
1. **archived** - the content is publicly viewable, but will be ignored in menus and indexes and is not interactive
2. **queued** - the content is approved to be taken live automatically by the publishing queue
3. **approved** - the content is approved to be published manually
4. **submitted** - the content has been submitted for the approval of a moderator
5. **draft** - the content is not ready to go live
6. **expired** - the content is no longer approved for viewing, but reinstatement is likely
7. **canceled** - the content is longer longer approved for viewing, and reinstatement is unlikely
8. **rejected** - the content is junk, and can be deleted by garbage collectors

Save as draft

New content can be set to "draft" to indicate that work is in progress, and the content should not be published. If the content already exists and is published, but a new revision should not be published, then the revision status can also be set to draft. The content will publish, but the draft revisions will be ignored. The other workflow states have no meaning in revision management.

Content Object Types

v3	v4
native section	content/section with no url
standalone section	content/section with an url defined
page	content/page
template	content/template
library	content/library
alternate	content/page with a language and master defined
content	content/content
blog	content/blog
article	content/article

v3	v4
comment	content/comment
attachment	content/content
catalog_category	content/catalog
catalog_product	content/product
event, evt	content/calendar, content/event

Every content object has a type, which provides the properties of the content. Every type includes a Perl class to manage the type; all of these classes ultimately inherit from the base Content class. However, you can override and customize any of the core behaviours.

Among the recognized types are: section, page, library, template, content, blog, article, calendar, event, catalog, product, comment. These replace the similar objects in ExSite 3, and have comparable behaviours.

Types are extensible using the following procedure:

1. add a new type to the **content_type** table, defining some of the basic rules:
    1.1. Class: the perl module that overloads Content.pm in managing content of this type (model/view)
    1.2. Plugin: the perl module that provides a specialized UI for managing content of this type (controller)
    1.3. Role: the default role of content of this type. This defines where the content comes from, and who is allowed to manage it. **Editorial** content is the regular website content; **design** is the templates and styles of the website; **user** is content that is contributed by website visitors.
    1.4. Publish As: **file** (eg. /foo.html), **directory** (eg. /foo/), or **never** (do not publish this content type). If publishing as a directory, the content may nevertheless be written to a file, index.html.
    1.5. Publish Method: **static**, **dynamic**, **hourly**, **daily**, **weekly**, or **never**. This is used as a default publish_rule if the content does not otherwise specify a rule.
    1.6. Publish Descendants: when publishing this content, select which sub-content items should also be published at the same time
    1.7. Navigation Type: **page** (treat these content objects as primary navigation destinations, like pages in the site map), **item** (treat these as secondary navigation destinations, like items in a list), **none** (this content is not a navigation destination)
    1.8. Display Type: **raw** (the content should be displayed in its bare form), **formatted** (the content should be built using a format, and inserted into a template), **template** (the content defines a complete HTML document, with CMS tags/codes for inserting other content), **none** (the content is not displayed)
2. define the allowed relations with other types in the **content_rel** table
3. install a Perl module that controls the behaviours of the new type; it can inherit from the closest existing type

Metadata

v3	v4
page.title	content.title
page.description	metadata/description
page.keywords	metadata/keywords
article.author	metadata/author

Content objects have their own attributes class called Metadata, which can track arbitrary metadata for every content object. Each content type can define the metadata properties that are relevant to that type, using configuration settings like:

```
content.metadata.article.author.datatype = string
```

Replace *article* with the content type, *author* with the property name, and *datatype* with any other dbmap parameter you want to set on the metadata property. Standard metadata properties are set up in $config{content}{metadata}, so you only need to define exceptions to those.

There are some metadata functions, which will return useful values even if the metadata has not been explicitly defined. These include: title(), author(), description(), and imeta() (see below).

The method editmeta() (and do_editmeta) automatically includes all type-specific metadata (and flags, below) when making and processing a form to edit a content object.

You can manually insert metadata values into your markup using <!--$meta_property_name-->, for example:

```
<meta name="author" content="<!--$author-->">
```

or

```
<p>Written by <!--$author--></p>
```

You can also auto-include metadata using simply, <!--$metadata-->. This will include <meta> tags for any metadata that you have set to be shown using a setting like this:

```
content.metadata.article.author.show =1
```

Auto-included meta tags will be shown if they have either an explicit or implicit value.

Generic Metadata

Generic metadata applies to all content types. We support three generic metadata schemes:

- Dublin Core
- OpenGraph (Facebook)
- Twitter Cards

Explicit and Implicit Metadata

Metadata can explicitly set on a piece of content, or it can be implicitly inferred from other content settings.

There is some overlap between the properties of the generic metadata schemes, and with ExSite metadata. For example, all of them include a variation on description (including DC.Description, og:description, and twitter:description). If one of these has been defined, but you request an undefined one, imeta() will pick a defined value and give that to you instead of undef.

Many of the generic metadata properties map roughly to properties that ExSite tracks in the CMS. These include things like authors, publication dates, images, URLs, MIME-types, dimensions, and others. If you request a generic metadata property that has not been explicitly set, but which has an implicit value, the implicit metadata function imeta() will return the implicit value.

The Modules::Content::metadata() method provides an interface to metadata management, which shows the implicit value as input placeholders if there is no explicit value defined.

If you set a metadata property to be shown automatically, the implicit value will be used if no explicit one is defined. That makes it easy to include standard meta tags such as twitter cards and have them auto-populate with no special effort.

Flags

Flags are boolean metadata. They can simply be checked off to activate them. Allowed flags are defined in $config{content}{flags}.

To test a flag setting, use:

```
if ($c->flag($flag_name)) {
```

To fetch all flag settings, use:

```
my %flag = $c->flags();
```

To set and remove flags, use:

```
$c->set_flag($name);
$c->unset_flag($name);
```

Tags and Indexes

v3	v4
keyword	content/index, content/keyword
keyword_tag	content/alias

You can setup multiple indexes under a site, each with its own dictionary of keywords. The indexes publish to directories, and the keywords to files, so you end up with URLs like:

```
http://foo.com/tags/cats.html
```

The individual tags are content objects of type **alias**, placed under the appropriate keyword.

Alias content objects simply redirect to the content ID defined in the master column. Alias content objects can be used elsewhere; they are not just for tagging.

Translations

v3	v4
version	langauge

Any content object can be setup in translated forms, simply by selecting the language and setting the master to point back to the original content.

When done with page objects, this allows for the sort of page mirroring we normally do. The sitemap in other languages will be based off the default/English sitemap, no matter how the alternate versions are structured.

The mirrored home page should be placed into a special page named for the language (or its abbreviation), so that the English page

http://foo.com/

becomes

http://foo.com/Fr

Building a Web Page

v3	v4
find “page” content object	use latest revision of this page, or its template

There is no longer a “page” content object that provides the skeleton HTML for the page. Instead, we use the current revision of the page itself. If the page has no revisions, we take the current revision of its template instead.

When we expand the HTML skeleton; we support all of the ExSite 3 CMS tags. However, the rules for finding matching content are a bit different. We search the following locations, in order (underlined rules are different from v3):

1. *search the children of the current node*
2. search the children of the current page
3. *search the children of our templates and their parent(s)*
4. search the children of the current page’s templates and their parent(s)
5. *search all of our descendant nodes*
6. *seach the children of our section’s template(s)*
7. search the libraries of this section and its parent sections

The content expansion is done by ExSite::Content::expand(), which works the same as v3’s expand(), with the following minor changes:

- The meta tags <!--$page_header--> and <!--$page_footer--> will be replaced with the contents of $share{page_header} and $share{page_footer}. The intention is that these tags should be placed in the <head> section and after the </body> in the template. Things like scripts and stylesheets can be placed there for better document structuring. Best to append to these variables, so you don’t wipe out anything already placed there by other objects/modules.

- In v3, expand() included MySite code to allow for inline editing. This is gone in v4 - MySite handles its own page generation.
- The page URL substitution {{page}} will also accept non-page objects, but will format them like a page. (see below)

Note that any content can be formatted to be displayed like a page. To do this manually:

```
http://foo.com/cgi/ex.cgi/page/path/to/content
```

This will format “content” in a full template, as if it were a page.

To do this programattically, use:

```
$out = $my_content->show_templated();
```

To write out a file that has your content formatted like a page, use:

```
my $file = new ExSite::Diskfile( base=>$base_path, path=>$web_path,
                                 filename=>$fname, contents=>$this->show_templated() );
$file->publish();
```

where $base_path is your DOCROOT, $web_path is the path in the URL, $filename is the HTML filename (eg. something.html), and contents is the complete HTML for the document.

Formatted Views

In addition to the standard views, you can define your own views using templates with merge codes. The following styles of merge code are supported:

style	description
[[foo]]	insert foo here
[[?foo]]TEXT[[/?foo]]	include TEXT if foo is defined
[[!foo]]TEXT[[/!foo]]	include TEXT if foo is not defined

The recognized merge codes are:

code	description
title, label, name, language	taken from the content record
id, about, author, caption, description, icon, index, info, navpath, summary, tools, height, width	obtained by calling the given method
type	the content type, eg. “page”, “article”
date	the posting date
curl	the canonical URL to the content
url	the URL to the content
html	the HTML to display the content
meta_X, eg. meta_author	metadata ‘X’
url_VIEW, eg. url_thumbnail	the URL to a particular VIEW (eg. thumbnail)
html_VIEW, eg. html_thumbnail	the HTML for a particular VIEW (eg. thumbnail)

code	description
index_TYPE, eg. index_contact	an index (summarized listing) of just the named subtype
contents_TYPE, eg. contents_contact	a full listing of just the named subtype
CONTENT:..., eg. logo:url_thumbnail	any of the above parameters for CONTENT
_FIRST:..., eg. _FIRST:title	any of the above parameters for the first subcontent item
_LAST:..., eg. _LAST:title	any of the above parameters for the last subcontent item
_IMG:..., eg. _IMG:html	any of the above parameters for the first image of the subcontent items

You can create your own formats and place them in exsite.conf: content.format.NAME = ... or you can simply create them on the fly. Use this call to obtain specially-formatted content:

```
$out = $my_content->show_formatted($format,%data)
```

where $format is either the name of a preconfigured format, or a complete template, and %data is any merge code values you want to override.

ExSite will automatically look for and use formats that follow certain naming conventions:

```
TYPE_summary
CONTAINER_TYPE_summary
```

where TYPE is the content type (eg. "product") and CONTAINER is the optional parent type (eg. "catalog"). These summary formats will be used in indexes and listings.

Example Format

This format is used to display articles:

```
<div class='articlePath'>[[navpath]]</div>
<div class='article'>
	<h1 id='article[[id]]' class='articleTitle'>[[title]]</h1>
	<div class='articleAbout'>[[info]]</div>
	[[_IMG:html]]
	<div class='articleBody'>[[html]]</div>
	[[tools]]
</div>
```

Content Date Services

v3	v4
crontask, evt_date	content_date

Dates can be assigned to content objects. This can be used for a variety of purposes, including setting up scheduled tasks, placing things onto a calendar, or simply logging notes on the content. Dates can be date ranges, with a start and end. Dates have the following attributes:

- type: task, todo, calendar, log, other
- description: a text description or action
- start, end: datetime values (start is required, end is not)
- status: *active* dates will be selected by the system; *inactive* dates will be ignored; *completed* dates are tasks or todo items that have been marked as done

Tasks

Tasks are automatically scheduled actions that the system can take. To setup a task, add a content_date record with *type* "task", and the *start* date/time set to when you want the task to execute. The description should be a task action that is recognized by that content type. By default, the following actions are known:

- publish
- unpublish
- archive

Other content types can define their own actions to extend the set of permitted tasks. The cron.pl script handles the scheduling; it runs hourly, so task execution times only have a granularity of 1 hour by default.

"Active" tasks will execute once; after execution completes, the system sets the *end* date of the task to the execution time, and changes the task *status* to "completed". Tasks can also be set to "hourly", "daily", or "weekly", in which case they will run periodically after the start date and before the end date.

To-do Lists

Tasks allow the system to perform certain preset automatic jobs for you. In cases where the job is too complicated to automate, you can use to-do lists to setup reminders and checklists for yourself to do them manually. To setup a to-do, add a content_date record with *type* "todo", and the *start* date/time set to your desired reminder date. The description should be a message to yourself indicating the work that needs to be done. To-do items can be viewed using the ToDo plugin application, which also allows you to strike items off your to-do list.

You can quickly add items to a to-do list with the call:

```
$c->todo($description,$date);
```

Logbook

To log notes on a content object, set the type to "log", and the description to the log message, or use the call:

```
$c->log($message);
```

The system will automatically log workflow if you set $config{log_workflow} = 1 in your configuration settings.

Price Services

Prices can be assigned to content objects. This can be used in place of various subsystems in v3 that tracked prices for different things.

Prices are tracked in the price table, and have the following access controls:

- Access - the minimum necessary access level to get this price (use this for member pricing)
- Valid from, Valid to - the date range that this price is valid for. Can be left unbounded on either or both sides. Use this for short-term sales, early bird fees, late fees, and so on.

Any number of prices can be attached to an object. The user will get the lowest active price that they are eligible for.

The base class will manage your prices for you, but it does not handle the shopping experience itself. Your class must deal with the formatting, add-to-cart functions, and redirection to the shopping cart to check out. This is done the same way it was in v3. In the simple case, just output a Pay oyster, eg.

```
<!--&Pay(cart=view)-->
```

GETs and POSTs

GET requests are managed through regular links. To get a dynamic link to a content object with optional query string parameters set, use:

```
my $url = $c->link(%query_params);
```

The content object's show() method can check for these query string parameters when deciding what to show. For example, blogs by default show an index of current posts, but this will return a dynamic link to the archives:

```
$blog->link(archive=>1);
```

In v3 POSTs were all handled by plug-in modules. In v4, content classes can optionally accept posts directly, with no special plugin. Post methods should process normal interactions with the content from the website front-end, for example:

- comments made to articles, or replies made to comments
- submissions made to web forms
- registrations
- product orders

These posts can therefore be handled without any plug-in modules. Administrator interactions, such as content configuration, should not be run through the post() method. Deal with those in the conventional way in the module's control panel.

Post data is processed by the object's post() method. You can post direct to the post() method using:

```
http://website.ca/cgi/ex.cgi/post/path/to/content
```

but that is mostly useful for background and AJAX posts, since it is not in a page context, and will not do well at generating contextual links and content expansion. If you want to do a regular form post that returns a full page, then your show method should handle the post, using a method like

```
if ($c->allow_post()) {
        $out .= $c->post();
```

The allow_post() method in turn should perform all necessary checks that the post is permissible. It is important that it check that the URL is targeting your content object directly:

```
return 0 if (! $c->url_is_me);  # do not allow posts to us at other URLs
```

That is because content can sometimes be embedded elsewhere (such as a sidebar preview on a different page), and we only want this content responding to posts at its "home" url.

Search

Search indexes work on similar principles to v3, but with some enhancements. Indexed URLs are stored in the *searchurl* table, and relevant terms in the *searchterm* table, as before. To get a list of links, sorted by relevance, do this:

```
my $search = new ExSite::Search($section_id);
$out = $search->do_search($search_string);
```

Results are automatically filtered for the access level of the user.

Searchurls can be tagged with a "type", which is a simple way to classify URLs. By default the type is set to the content type of the content indexed at that URL (for example, "event" or "product"), or the plug-in Module if the URL originates from a plug-in. For example, to search products only, you can append the type to the parameter list:

```
$out = $search->do_search($search_string, "product");
```

Searchurls can be tagged with a content ID, indicating the object that best represents this content from a search results point of view. (For example, a content object like a body is generally not viewed directly, so searchterms found in the body should be associated with the container page.) This allows the CMS admin tools to use the search index to find content objects:

```
my @content = $search->find_content($search_string);
```

It returns an array of content objects, most relevant first, which each have a "search_score" attribute to indicate relevance. (As above, an optional $type can also be appended to the parameter list if you want to limit your searches that way.) This allows admin control panels to make use of the system search index for their own purposes. For instance, a Catalog plugin could implement an admin search tool to find products, or a Membership plugin could search for member profiles.

Not every URL in the search index is necessarily tagged with a content ID, so public searches and admin searches will not necessarily return the same results.

Content classes can support two methods to customize their search indexes:

```
$c->can_index();
```

returns true/false depending on whether the content should add terms to the index or not. False values can distinguish between 0 (meaning permission denied) and undef (meaning the content should not be indexed). True values can distinguish between -1 (meaning the content can be indexed, but the index appears to be up to date) and 1 (meaning the content should be indexed now).

```
$c->search_index($search);  # $search is an ExSite::Search object
```

this method actually creates the search index entries.

Configuration Settings

Content configurations are generally kept under $config{content}{...}. Commonly used settings are outlined below.

Some content types are optional, and only included in the system if you install certain plug-in modules. It may be more convenient to keep the content configurations in those plug-in module conf files, rather than in the system conf file. In this case, use the *back* configuration param "-" to place the conf settings in the configuration root, rather than in the module configurations. For example:

```
-.content....
```

Metadata

Metadata are setup using settings like:

```
content.metadata.TYPE.META.MAPATTR = value
```

where:
TYPE = content type
META = metadata property
MAPATTR = dbmap attribute
For example, to collect an "Author" metadata property on articles, you can set:

```
content.metadata.article.author.datatype = string
```

Flags can be defined similarly, but do not require dbmap settings, so you just need to add them to a list:

```
content.flags.article += sticky
```

Contextual Metadata
If you enable contextual metadata, using

```
content.contextual_metadata = 1
```

then we will also try to setup metadata contextually, meaning that the allowed metadata properties will depend on the parent container object. That lets you do something like:

```
content.metadata.CONTAINER_ID.TYPE.META.MAPATTR = value
```

for example:

```
content.metadata.1234.profile.degree.datatype = string
```

which defines a *degree* metadata property that applies to profile objects under content (membership type) 1234 only.

Formats

Content formats are taken from the following configuration settings:

```
content.format.TYPE = html_format...
```

where "TYPE" is a content type like article or product. This is used to display the main view of the content.

```
content.format.TYPE_summary = ...
```

for example, *article_summary*. This format is used to display the content in indexes, for example when viewing the blog.

```
content.format.CONTAINER_TYPE_summary = ...
```

for example, *blog_article_summary*. This format is used to display blog articles in indexes, in case you want a different format than articles that are used elsewhere.

Note that if no special format is defined, the system will use content.format.content as the generic format.

Settings

Settings are configuration parameters that have a flexible scope. They can be set:

- on every content object
- on every object of a certain type (every article, for example)
- on a specific content object (eg. one particular article)
- on all content objects of a certain type within one section (eg. every article within one sub-site)
- on all content objects of a certain type within a certain container (eg. every article within a certain blog)

Settings can be set in configuration files, or in preferences. The general format is:

```
setting.TYPE.NAME = value
```

where TYPE is a content type (such as "blog") and "name" is the actual conf setting name (such as "index_min").

To make a setting applicable to ALL content types, just use

```
setting.NAME = value
```

To make a setting applicable to a specific content object, change TYPE to TYPE:ID or TYPE:NAME, where ID is the content ID or NAME is the content name. For example, to set the index_min setting on a particular blog, use:

```
setting.blog:100.index_min = 1
```

or

```
setting.blog:News.index_min = 1    # might be ambiguous
```

To make a setting applicable to all content objects within a certain container, place the setting into a preference that is attached to the container object. Use the generic setting name, eg. setting.TYPE.NAME, and attach it to the container object. For example, to moderate all comments in a particular forum, use the setting

```
setting.comment.moderate = 1
```

and place this as a preference on the forum in question.

To make the setting applicable to all content objects of a type within a certain section, place the setting into a preference that is attached to the section.

To fetch the setting that is most applicable to a content object, use:

```
my $value = $c->setting($name);
```

or, for a more specific example:

```
my $index_min = $blog->setting("index_min");
```

It will select the most specific setting that has been defined using the above methods.

Behaviour Inheritance

In v4 there are multiple mechanisms for inheriting behaviours from related objects and classes.

Class Inheritance

Model (content behaviour)

Content classes should inherit from the base class, ExSite::Content, to get all standard v4 content model behaviours. You can alternatively inherit from higher-level classes; for instance If the new class is page-like, it can optionally inherit from the Page class instead.

The base behaviour is fairly flexible, and is controlled by a few parameters in the content_type table:

- Publish As (publish_as) - how the content publishes itself
    - file (eg. foo.html)
    - directory (eg. foo/)
    - never if the content does not publish
- Publish Method (publish) - the default publish rule for this type of content
    - static - content publishes to a static file
    - dynamic - content is always rendered at the time of page view
    - hourly, daily, weekly - static, but republished on intervals
- Navigation Type (navtype) - how the content is linked in site navigation
    - page (object shows in menus)
    - item (object shows in indexes/listings)
    - none (object cannot be navigated to)
- Display Type (displaytype) - what the object looks like when viewed directly
    - raw (object is displayed directly)
    - formatted (object is built using a format with merge codes)
    - template (object is used as a complete HTML document template)
    - none (object is not displayed)

These behaviours can be configured without even needing a special model class, in principle.

The model is extensible using metadata and flags, which can be defined in your configuration files, using settings like:

```
content.metadata.TYPE.METANAME.datatype = string
content.metadata.TYPE.METANAME.help = some tooltip text
content.flags.TYPE += flag_name
```

View (content appearance)

The public view of the content is generated by the class' show() method. Overload this if you need specialized views.

If necessary, you can define alternative admin views using a preview() method.

Most views can be auto-generated using formats, which are template-like snippets of HTML with merge codes. Set the display type to formatted, and define a format with the content type name in the configurations under content.format... For example, content.format.article will automatically be used to format article views, if defined. Summary views (eg. for indexes) will automatically be formatted in a similar way if a format like *article_summary* is found (replace article with the appropriate content type).

Formats can include other formats to share/inherit common formats or layouts. The following merge codes will pull in other formats:

> [[.foo]] - find and insert $config{content}{format}{foo}
> [[>foo]] - find and insert a format named "foo" using ExSite::Content::find (will find foo in a template or library)
> [[>>foo]] - find and insert a format named "foo" using ExSite::Content::find_in_path (will inherit foo from a parent)

Controller (content management)

The Modules::Content plugin provides a useful base class for content management controllers and plug-ins. Use it instead of Modules::BaseDCD for plugins that manage content classes.

Modules::Content classes have a few standard query parameters that are assumed.

- id - the ID of a content object
- cmd - the control panel function being invoked. The following commands/functions are predefined: conf, edit, rollback, todo, img, price, contact, translate, publish, unpublish, copy, del.

Modules::Content provides the following methods for superclasses that inherit it:

- write($id) - outputs content object $id. This provides a simple method for embedding or aliasing content in other parts of the site. Note that you do not normally need a special write() method for content, since the content's show() method should automatically generate its own view. Plugin write() methods are used for irregular or aggregated views that are not normally generated by the object itself. For example, a blog automatically generates its own index and other views, but if you have several blogs on a system, you might need an aggregated view such as "recent articles" across all blogs. Similarly, you don't normally need service pages for content, since every content object has its own natural URL.
- pathbar($c,%opt) - output a PathBar widget to show where you are in admin control panels. $c is the object you are working on. $opt{section_id} suppresses the inclusion of ancestral objects above the specified section number. $opt{linktypes} provides a regular expression of the content subtypes that should be linked from the pathbar.
- configure($c) - creates and processes a form to reconfigure the content object $c, including content parameters, metadata, and flags.
- update($c) - creates and processes a form to revise the content object $c using the HTML editor.
- rollback($c) - rolls the latest revision back, and republishes the content object $c.

- delete($c) - unpublishes and deletes the content object $c.
- copy($c) - copies the content object $c, and redirects the admin to the configuration screen for the new object.
- pricing($c) - displays tools to manage the pricing of content object $c.
- order($c) - displays tool to manage the sort order of the contents of $c.
- translate($c) - displays tools to manage the translations of $c.
- publish($c), unpublish($c), archive($c), approve($c), queue($c), unqueue($c), expire($c), reject($c) - change the workflow status of content $c

Ancestor Inheritance

In addition to class inheritance, objects can inherit definitions from their ancestors.

As noted above, formats can include sub-formats from their ancestors using merge codes like [[>>foo]].

Metadata definitions are normally common to all objects of a given type, but you can make them context specific (ie. inherited from ancestors) if desired. Allowed metadata are normally defined as follows:

```
content.metadata.product.option_color.datatype = string
content.metadata.product.option_size.datatype = string
```

This defines option_color and option_size as possible options for any product. Say that one particular category also has an additional option (option_hand, ie. left or right) but you don't want that option to appear under every product in the system, just in this one category. If the category ID is XXX, then you can add the context-specific option as follows:

```
content.metadata.XXX.product.option_hand.datatype = string
```

Then any product under content ID XXX will get that option as well. Because this can involve extra lookups, which impacts performance, enable contextual metadata with the config setting

```
content.contextual_metadata = 1
```

You can also use settings (see above) in a similar way, making them apply only within certain sections or container objects.

Converting v3 Websites

The script bin/convert3.pl assists with conversion of v3 websites. You must modify the parameters to connect to your old v3 database. Use server.db to connect to your v4 database, and server.db1 to connect to your original v3 database. Run the script from your v4 cgi directory

It converts:

- sections, pages, templates, libraries, and content objects
- alternate language pages
- aliased pages
- blogs, e-zines, articles, comments

- calendars, events, registration fees, and registrations
- catalogs and products
- web forms
- membership databases
- accounts, contacts, and sales information
- jgalleries (if you have a jgallery table)

Pages will conform to the original sitemap as much as possible. Templates and libraries will be given a high sortkey so that they are positioned at the end of the sitemap. It will attempt to embed forms in the pages that use them; otherwise they will be placed under a Forms container page. It will attempt to embed blogs and e-zines under pages that use them, otherwise they will be placed under the section. It will try to re-write oysters for QA, Zine, Blog, VMenu, and ImageRotate to point to the converted objects.

There are some configuration settings at the start of this file that control how some content gets converted:

- $latin1_to_utf8 - enable this if the old site still uses latin1 encoding
- $profile_contact_type - if membership profiles should have a visible address card (eg. a place of business), indicate which contact type should be used for that purpose
- $profile_title_template - how to title membership profiles; the default is "[[first_name]] [[last_name]]" but you might want to include titles, organizations, or other info here.
- $copy_invalid_finances - enable this if you want to include invalid invoices, cancelled payments and other ignored financial data.

You will need to do the following manual steps after the automatic conversion has completed:

- Eliminate unnecessary service pages; service pages for things like blogs, e-zines, calendars, etc. are redundant, since the blog itself serves as this page.
- Eliminate redundant container pages; for example, a News page (which typically contains a Blog plug-in) is no longer necessary, because the Blog itself serves this function. You can move the new blog in place of the old News page, and discard the old News page.
- Fix dupe names; name collisions will be resolved by adding some random text to the conflicts. You should remove the unncessary dupes, and then rename the survivors to simplify.
- Eliminate, convert, or update various plug-in oysters.
- Update stylesheets to support new mark-up, and eliminate old rules.

Plug-in Module Notes

Content

This is a rudimentary content management plug-in application, which can be used as a base class for other CMS tools. It also works as a simple barebones CMS.

Content Management

This is the full-featured CMS application that exposes the most CMS features to the end user. It replaces Website Manager, which no longer exists.

E-Zines

Use *Blogs* instead for posting article feeds. Use *Forums* for forum management, and *Comments* for comment moderation.

To auto-embed images into articles, you can insert content objects under the article.

There is a *Zine* plug-in, but it only provides some basic aggregation functions such as recent articles across all blogs.

My Website

Note that the page preview and edit screens are actually control panel URLs now, not page URLs as they used to be. Copying/pasting these URLs for the public will not work.

Photo Albums

Note that there is now a distinction between albums (photo collections that are displayed together, like a gallery) and libraries (photo collections where the images are used individually here and there on the site). Albums are a simple replacement for gallery and slideshow plugins. Galleries are manually sortable.

Keyword Tags

Note that you can maintain any number of distinct indexes/dictionaries.

Forms

Replaces the old QA (Web Forms) module.

When creating questions you have the option of defining a new question from scratch or copying one from an existing form.

Note that many HTML5 input types (such as date, time, email, etc) are supported. These use native HTML5 browser support, which varies across browsers.

Product Catalogs

To add product options, define product metadata with the name option_*Name*. For example:

```
content.metadata.product.option_Color.datatype = text
```

The "Option Color" field will then show up under product configuration. Enter your |-separated list of options into this field. If you leave this field blank for the product, the option will be ignored.

Registration

Registrations now follow a ticketing model, which means:

- events have an inventory of tickets to sell
- ad-hoc ticketing means that new tickets are created as needed on the fly; otherwise the ticket inventory determines the event capacity
- tickets can in principle be tagged with specific seats or other unique identifiers
- tickets can be in a reserved state, meaning they are designated but unpaid; they can also be in a held state, meaning selected for purchase, but not checked out. This prevents overselling events, because another registrant cannot grab a held ticket until it gets released back into the sales pool

Fees have a much simpler structure. To mask fees from registrants in other categories, provide a comma-separated list of Fee ids to hide from in the "Hide from fees" configuration field.

Pricing options will be appended to registration forms on control panel registrations. For admin-only comp fees, add a $0.00 price that is available to admins only, and it will be included in those pricing options.

Address Book

There are significant differences in the structure of address cards in v4. The contact record itself contains no contact information; the content information is in the attribute-like contact_info records. That makes it more extensible, and more flexible for security (access can be granted field-by-field rather than for whole cards. However, it will complicate contact queries, especially on multiple fields.

Memberships

Note that memberships are a product that is purchased and associated with a user. That means that one user can own multiple memberships. The content object is the membership itself - the profile, to be more specific. It sets up its expiries and other term limits using date/task management.

Security Manager

Also allows you to review all system administrators.

To-Do

This tool shows any pending/incomplete to-do tasks that have been entered through the CMS date tools.

Users

This replaces the old Members app, and also provides group management features.

Financial Reports

Includes GL code reporting options.

ExSite::Content API - CMS Base Class

The Content class provides all of the basic CMS services required by various different subsystems, including:

- revision control, archiving
- alternate views (eg. thumbnails)
- level-based access controls (public, member-only, administrator-only)
- group- and role-based access controls
- publishing
- scheduling, queuing
- workflows, moderation, approvals
- hiding content from navigation, search, robots
- metadata management
- URL management
- tagging
- translation, multilingual services
- pricing
- search

Different classes of content will inherit from this class to get the basic CMS behaviours, which they can then overload to define their specific class behaviours.

Content Identification

Every content object has a unique numeric ID. Additionally, it has a unique path, an alphanumeric string that is used to identify the content in URLs, eg. http://foo.com**/foo/bar**. The path is comprised of the names of the content object and its ancestor objects, so it is also a list of names that have to be traversed to get to the object in the overall heirarchy.

The name does not have to be unique across the whole website, but it should be unique for all content objects that share the same parent ID, in order to ensure that the path is unique.

There can be one "anonymous" node with no name, but it must be at the root of the tree. So the above path could refer to either one of

> foo -> bar
>
> [anonymous] -> foo -> bar

Content objects typically publish to directories that match their path, and can be viewed at an URL like:

> http://foo.com/foo/bar

They can also be viewed dynamically as a formatted web page:

> http://foo.com/cgi/ex.cgi/page/foo/bar

or as bare content:

> http://foo.com/cgi/ex.cgi/view/foo/bar

Different content classes can alter their publication rules to vary the published location. For example, events can publish to dated directories like /calendar/2012/11/event.html.

Setup

Create your content object in one of the following ways:

By its content ID:

```
my $c = new ExSite::Content(id=>99);
```

By its whole or partial data record:

```
my $c = new ExSite::Content(data=>\%content);
```

By its name (you must provide a parent ID):

```
my $c = new ExSite::Content(name=>"foo",parent=>99);
```

By its path:

```
my $c = new ExSite::Content(path=>"/foo/bar");
```

By the path requested in $ENV{PATH_INFO}:

```
my $c = new ExSite::Content(path=>1);
```

Converting Content Type

The above calls will instantiate an object of class ExSite::Content. This might not be the correct class, in which case you will get incorrect object behaviours. If you know the object is actually of another class, you could instantiate that class directly, eg.

```
my $p = new ExSite::Page(id=>99);
```

or, you can ask the object to convert itself to the most appropriate class:

```
$c = $c->get_content_obj();
```

If you already have another content object handy, you can often do this in one step. For example, to make an object from a content ID:

```
my $newc = $c->get_content_obj($id);
```

Or, to make an object from a content record:

```
my $newc = $c->get_content_obj(\%content);
```

When used this way, neither of these calls affects object $c in any way.

Basic Object Queries

The Content class inherits from the ObjectMeta class, and has all of the behaviours of a standard ExSite object with metadata. For example:

```
my $val = $c->getdata($column);      # fetch a raw data value
```

```
my $val = $c->showdata($column);   # safely display a data value
my @err = $c->validate();          # validate the record contents
```

...and so on. See the documentation for Object.pm and ObjectMeta.pm for more information.

In addition to the basic Object methods, there are also the following Content-specific methods for obtaining information about the object.

Content names:

```
$c->name()                         # used in URLs
$c->title()                        # used in page titles, headings
$c->label()                        # used in hyperlinks to the content
```

Relationships to other content objects:

```
$c->parent()                       # returns the parent object
$c->template()                     # returns this object's template
$c->master()                       # returns this object's original reference content object
$c->alias()                        # whether or not we were redirected to this content from an alias
$c->is_in($c2)                     # whether $c is contained within $c2
$c->my_ancestor($type)             # nearest ancestor of a certain type
$c->my_page()                      # nearest page-like object that contains us
$c->my_section()                   # the section that contains us
$c->my_root()                      # the highest-level object that contains us
```

Status of this content object:

```
$c->has_content();                 # true if this object contains its own revisions
$c->is_static();                   # true if this object publishes
$c->is_viewable();                 # true if this object is okay for regular viewing
$c->is_active();                   # true if this object is useable
                                   # (is_active is same as is_viewable for non-interactive content)
$c->is_public();                   # true if this object can be shown to the public
```

The type/behaviour of this content:

```
$c->subtype();                     # for example content, library, article, page, etc.
$c->isubtype();                    # the numeric type that corresponds to the above
$c->is_page();                     # does this content behave in a page-like way?
$c->is_subdir();                   # dies this content publish to its own directory?
$c->my_subdir();                   # the actual subdirectory name we publish to
$c->filename();                    # the filename we publish to
```

is_page() determines whether the content is page-like. A "page" is a primary navigation destination on your site, and will determine certain default behaviours, such as whether to include links in menus. For the base Content class, is_page() returns FALSE. You should overload this if your content object is page-like. (You may also want to inherit from the Page class to get other page behaviours.)

is_subdir() determines whether this content is published as directory or as a file. If as a directory, it will have a static URL like /foo/bar, whereas if as a file, it will have a static URL like /foo/bar.html (and will publish into the parent's directory). The base Content class publishes as files, but higher-level classes can override this.

my_subdir() is the actual subdirectory name we publish to (which may be blank for the anonymous root node). my_filename() is the filename we publish to, which will typically be NAME.EXT where NAME is $c->name() and EXT is the appropriate MIME-type suffix. For page-like objects that publish into their own directory and are HTML, the filename defaults to index.html.

Use the info() method to obtain a bunch of technical info about the current revision of the content, such as filenames, mime-types, sizes, and image dimensions.

```
my $info_string = $c->info();          # returns descriptive string
my %info = $c->info();                 # returns all info in a hash
```

Content Dates

The following calls will return date information about the content:

```
$c->getdata("ctime");                  # creation time of the content record, as a timestamp
$c->getdata("mtime");                  # last modification time of the content record, as a timestamp
$c->getdata("ptime");                  # original publication of this content, as a timestamp
$c->posting_date();                    # ctime, but as an ExSite::Time object
$c->age();                             # time since creation, in days
$c->revision->timestamp();             # creation time of the current revision
$c->revision->getdata("ptime");        # publication time of the current revision
```

In addition to these timestamps, there is also a general-purpose date service that content objects can make use of. This lets you assign arbitrary dates to any content object. There are 5 general types of dates you can add to a content object:

- task - this is used for scheduling purposes, ie. the date is a trigger time for some task you want to be performed automatically
- calendar - this is used for calendaring purposes, ie. the date is used for organizing and displaying content by date
- log - this is used for recording a timestamped note on the content
- todo - adds items to the To-Do list; these may generate reminder emails on the specified date, and will also appear in the ToDo plugin app
- other - any dates that do not fit into one of the above categories

To use the date service, you can use the following calls:

```
$c->get_dates($type);      # returns an ObjectList of dates; pass $type to fetch only dates of that type
$c->log($message);         # add a date of the “log” type to the content
$c->todo($message);        # add a to-do task associated with this content
```

Metadata

Object metadata is stored separately from the core data, and can be accessed using the following methods:

```
$c->meta();                # returns metadata object, for metadata manipulation
$c->get_metadata();        # returns all metadata, as a hash
```

To find the "best" metadata under a given name, use something like

```
my $val = $c->dcmeta("description");
```

This will look for an appropriate metadata item under DC.Description, Description, and description, in that order. The first one (DC.Description) is the industry-standard "Dublin Core" metadata format, which defines the following metadata names: DC.Title, DC.Creator, DC.Subject, DC.Description, DC.Publisher, DC.Contributor, DC.Date, DC.Type, DC.Format, DC.Identifier, DC.Source, DC.Language, DC.Relation, DC.Coverage, DC.Rights. Because of their industry-standard nature, these will be preferred over other, similar metadata, if they are defined.

Additionally, you can use the following methods to obtain metadata-like values, even if no metadata is actually defined. The object will find something appropriate to use regardless:

```
$c->title();
$c->author();          # checks both author and creator
$c->description();     # checks description, abstract, and summarizes content, otherwise
$c->caption();         # checks caption, description, subject, and title
```

Different content types can predeclare the metadata that they support in the system configuration. The configuration setting content.metadata.SUBTYPE.METANAME should be set to a dbmap-like hash of settings (such as datatype, size) that describe the acceptable values. For example:

```
content.metadata.location.longitude = { datatype=>"decimal" }
```

If you are creating a plug-in application, you can define metadata (among other content settings) using this configuration file notation:

```
-.content.metadata.location.longitude.datatype = decimal
```

Note the leading "-" which places the configuration setting into the global configurations, not the module configurations.

Flags

Flags are metadata-like booleans. Rather than store name/value pairs, they store only names. If the name/flag exists, the boolean value is true, otherwise it is false. Allowed flags are defined in the system configuration, using settings of the form content.flags.SUBTYPE, which holds a list of recognized flags. For example:

```
content.flags.question = required
content.flags.question += checked
```

Work with flags using the following calls:

```
$c->flag($flag);       # returns the flag setting
$c->flags();           # returns a hash of all flags
$c->set_flag($flag);   # turns on $flag
$c->unset_flag($flag); # turns off $flag
```

Content Paths

There are numerous ways to describe the path to the content. Use the following methods:

```
$c->content_path();     # returns an object list of the nodes traversed to get to this content object
$c->path();             # a simple text representation of this path, eg. "/foo/bar"
$c->navpath();          # a cookie-crumb style formatted list of links representing this path
```

When publishing to disk, we use the logical path to the content to build an actual diskpath.

```
$c->my_subdir();        # the subdirectory name that this content publishes to, if it publishes as a directory
$c->filename();         # the filename that this content publishes to
$c->basedir();          # our HTdocs directory; the root of our published files
$c->subdir();           # our complete set of subdirectories relative to basedir()
$c->diskpath();         # basedir + subdir
$c->httpbase();         # our base path element in the URL, $config{server}{HTMLpath}
$c->httpdir();          # our URL path relative to httpbase();
$c->httppath();         # our URL path relative to the docroot, httpbase + httpdir
```

Displaying the Content

To fetch and display your content, you have 3 general approaches:

- get HTML to render the content
- get an URL to visit the content
- get the raw content data (for example, the JPG data)

The following methods can be used:

Getting HTML

- show(%opt)

This returns HTML to render the content in a web browser. It should return an HTML snippet that can be inlined into another document. If the content is a page, then it should return the complete HTML document.

Use $c->show(view=>$viewname) to explicitly display a different view than the default.

- show_templated(%opt)

This call always returns a complete web document; the content will be wrapped up in HTML to give it the same appearance as a full web page on this site.

It will use the template setting for the content object to determine which template HTML to use to wrap up the object. If template is not defined, it will find something appropriate based on the pages or sections we are nesting under.

For pages, show() and show_templated() should give the same results.

- show_formatted($format,%data)

This allows for on-the-fly custom formats to display the content. $format is either the name of a predefined format (kept in $config{content}{format}{...}, or a fully-specified template with merge fields. ExSite::Misc::substitute is used to merge the data, so you can use codes like:

```
[[foo]]              - insert foo here

[[?foo]]TEXT[[/?foo]]  - insert TEXT if foo is defined

[[!foo]]TEXT[[/!foo]]  - insert TEXT if foo is not defined
```

The allowed substitution parameters include:

```
about          # posting info, such as author, date
attachments    # a listing of file-like subcontent for downloading
author         # author name
caption        # caption
curl           # canonical URL
date           # posting date
description    # description, or summarized content
gallery        # a thumbnail gallery of image subcontent
height         # image height
html           # content HTML
icon           # an icon to represent the content
id             # the content ID
image          # best image associated with the content
index          # an HTML listing of sub-contents
info           # description of the file, such as mime type, size
label          # hyperlink anchor for the content
language       # the langauge, if not the default
link           # dynamic url to the content
mime_type      # MIME type of the content
mime_descr     # natural-language description of the MIME type
mime_category  # the general type (eg. image, text, application)
name           # the content name, as used in URLs
navpath        # HTML to show a cookie crumb to this content
price          # price
purl           # permalink URL (best user-friendly URL)
summary        # description, or file information
tools          # links to operate on the content
title          # title
type           # content type, such as library, page, article, etc
url            # current URL to the content
width          # image width
```

You can also substitute any metadata, using:

```
meta_FOO       # metadata "FOO"
```

You can also substitute views other than the one you are working with. Replace VIEW in the following with the view you want, such as “small” or “thumbnail”:

```
url_VIEW        # URL to this view

html_VIEW       # HTML to render this view
```

You can also subsitute other content objects into the display of this one. (For example, inlining images into an article.) In the following parameters, replace ... with any of the above to get the appropriate parameter for the specified content:

```
NAME:...        # any of the above parameters for content object NAME

_FIRST:...      # any of the above parameters for the first subcontent item

_LAST:...       # any of the above parameters for the last subcontent item

_IMG:...        # any of the above parameters for the "best" image of the subcontent items

index_TYPE      # an index of subcontent of TYPE

contents_TYPE   # a full listing of subcontent of TYPE
```

If you pass any unknown parameter such as FOO, the system will ask the content class if it supports that parameter (via a method called param_FOO). If such a method exists, it will be called to fetch the appropriate substitution data.

- summary()

The summary is generally used when an abbreviated view is required, such as in listings or indexes.

This is just shorthand for

```
$c->show_formatted("summary");
```

However, because the summary wants to display a description, and there often is not a good description available, this call will attempt to compose a useful description first.

- preview()

preview() is a special display of the content, for administrators. By default, it shows all existing views of the content, in both normal and source form. This should be overloaded by other content classes that have different content managment requirements.

- report()

This displays technical information about the content.

- get_html()

This retrieves the raw HTML for the current revision. For simple content objects, this should give the same results as show().

Getting URLs

There are numerous URLs you can use to obtain/view content:

```
$c->get_url()                 # current best URL to the content
$c->get_url_dynamic()         # the dynamic URL to the content
$c->get_view_url_dynamic() # the dynamic URL to view the bare content
$c->get_page_url_dynamic() # the dynamic URL to view the content formatted as a page
$c->get_url_static()          # the static URL to the content
$c->get_url_canonical()       # the canonical URL to the content
$c->permalink()               # the best human-friendly URL to the content
$c->get_uri()                 # a URI object representing get_url_dynamic()
$c->get_page_uri()            # a URI object representing get_page_url_dynamic()
$c->link(%args)               # modify the dynamic URL with other parameters
$c->admin_url()               # URL to an admin control panel that is suitable for managing the content
```

With link(), the content object can generate a dynamic link in two styles:

- by path, eg. /cgi/ex.cgi/page/path/to/page - this is best for public content views
- by id, eg. /cgi/ex.cgi/admin/Foo?id=555 - this is best for back-end admin views

It will choose the latter method if the content object has its internal attribute *content_link_type* set to “id”.

Sub-content

Some content objects can contain other content. In the base content class, this is not possible, but higher classes (for example, libraries, blogs) may not actually have their own revisions, just sub-content.

Use the following calls to obtain sub-content. You can optionally pass a type if you want to restrict the listing to only that class of item.

```
my @clist = $c->get_contents($type);          # returns an array of content objects
my $list = $c->get_contents_as_list($type);   # returns an ObjectList
my %c = $c->get_contents_as_hash($type);      # return a hash indexed by content name
my $n = $c->count_contents($type);            # returns the number of sub-content items
my $c = $c->my_content($name);                # returns just the named content object
```

For menus and submenus, you want the subcontents that are page-like. (Note that this list includes hidden pages that should not be displayed in menus.)

```
my @submenu = $c->my_submenu_items();   # children of this node
my @menu = $c->my_menu_items();         # also includes this node as the top item
```

If you want to get all descendent nodes, not just the immediate children, use:

```
my @all = $c->get_descenants(@type);          # you can pass multiple types
```

Indexes

Indexes are HTML listings of sub-content items. Typically these are hyperlinked so that the viewer can use the index to find and visit the other content items.

```
$out .= $this->index();                       # display an index to the contents
```

Other content classes can overload this to provide their own specialized indexes.

When automatically generating links to sub-content items, you can also make use of:

```
$out .= $c->attachments();          # display sub-contents as files to be downloaded
$out .= $c->gallery();              # display sub-contents an an image gallery
$out .= $c->insert_image();         # find and display a suitable image
```

Note that an RSS feed is just a specialized index, formatted as an RSS XML file. You can generate this XML using this call:

```
$xml = $this->rss();
```

tools()

tools() generates the links that are needed to interact with the content. For regular content objects, there are none. Other content classes can overload this to provide additional functions.

Examples of common types of tools displayed by different content types are:

- RSS feeds
- post functions
- archives
- add to cart

Finding Content

Starting from any given content object, we can find other content given only a name. Because names are not unique system-wide, there is a particular order we search in:

1. search the children of the current node
2. search the children of the current page
3. search the children of our template (and the template's parents)
4. search the children of the current page's template (and parents)
5. search our descendant nodes
6. search the children of our section's template (and parents)
7. search the libraries of this section (and parent sections)

To perform such a search directly, use:

```
my $best = $c->find($name);
```

$best will be the best-match content object named $name.

Content Expansion

HTML content can contain substitution tags like the following:

```
<!--content(NAME)-->        # insert HTML for content NAME here
[[NAME]]                    # insert URL to raw content for NAME here
{{NAME}}                    # insert URL to templated (page-like) content NAME here
<!--$META-->                # insert metadata META here
<!--&MODULE(options)-->  # insert plug-in MODULE content here
```

If the content object contains HTML, you can automatically substitute all of the tags by calling:

```
my $html = $c->expand(%option);
```

%option can contain the following options:

```
html => $starting_html
```

Use this if you want to start from something different than the current content object's HTML.

```
method => "data,url,content,page,module"
```

Use this switch to control which substitutions will be performed. You can pass a list of any of the above 5 substitution types.

```
expand => "template"
```

Assume the content being expanded is a template, not actual content. This will leave certain items unexpanded.

$c->unexpand() undoes the content expansion so that it can be restarted.

get_dynamic_content()

```
my $html = $this->get_dynamic_content($module,$args,$option);
```

Return replacement text for a <!--&Module(...)--> CMS tag

Loads the $module plug-in, and calls its write() method, passing the $args to it. Returns the output of the module. If the module fails to load/compile, a Perl error string will be returned.

$option is the same as is passed to expand(). If the "expand" option or "dummy-module" methods are selected, a placeholder image will be substutited instead of the module content.

If the "static-module" method is selected, each module will be queried to see if its output is static for this page; only if static will the content will be substituted. (This allows for precompiling certain modules' output, eg. menus, while rendering others at page view time.)

get_dynamic_content_indirect()

```
my $html = $this->get_dynamic_content_indirect($module,$args,$option);
```

This is an AJAX version of get_dynamic_content().

This method returns replacement text for CMS tags of the form:

```
<!--&&Module(...)-->    # indirect substitution, direct re-links
```

```
<!--&&&Module(...)-->   # indirect substitution, indirect re-links
```

This fetches the DCD content using a separate server request, instead of inlining it directly.

The main advantage is that you can publish the main page to a static HTML file, yet keep some page elements dynamic. This is especially useful for index.html pages, which must be static, but may contain dynamic elements (eg. recent news, upcoming events, current specials, etc.). The solution is either continuous republishing of the index page (which may still be a better solution for heavily loaded sites) or using an indirect dynamic content fetch.

The disadvantage is that the full page is slower (although the base page may be much faster), and that JavaScript must be enabled to perform the secondary content fetches.

The '&&' variant does direct re-links, ie. links from the dynamic content point to full URLs that generate a new page. The '&&&' variant does indirect re-links, ie. links back to the same module in the same page only fetch the DCD content and inline it dynamically into the current page without generating a whole new page.

Revision Control

The content itself gets updated in the form of revisions. Each update creates a new revision record tied to its content record. The most recent revision is the newest revision, while the most recently publish revision is the active revision.

By default we load the active revision, since that is the one approved for public viewing. To use the newest or draft revision instead (for admins, for instance), add a cms flag when creating your content object, to tell the system that you are doing CMS work:

```
my $c = new ExSite::Content(id=>99,cms=>1);
```

To obtain the revision that the content object is using, use;

```
my $r = $c->revision();
```

To set the revision of the content object to a different revision, use:

```
$c->load_revision($rev);
```

where $rev is "draft", "newest", "active", or a numeric revision ID for some other revision.

To fetch a different revision, without changing the current content object, use:

```
$c->find_revision($rev);
```

To make calls to the current revision, use calls like:

```
$c->revision->mime_type();
```

The full list of calls is documented in the ExSite::Revision class, below.

revise(%opt)
Adds a new revision to this content object. Options are:

- data - The raw data for the view(s). For file uploads, should be an encoded file.
- format - The format of the data, ie. text, file, path, url.

- mime_type - The mime_type of the data. Optional - will be guessed if not provided.
- attributes - A hash of attributes and values.
- status - defaults to 0/published, but can be set to 5/draft
- note - A revision control comment.
- view - A hashref with instructions for generating views. Each hash item is of the form

  viewname => 1

or

  viewname => \%cview_datahash

where viewname is one of the recognized view names, ie. large, normal, small, thumbnail, or icon. If the view is defined as a datahash, it will be installed as defined. Otherwise, if the view is defined as TRUE, the system will attempt to generate the view automatically. This is normally only possible for images, which can be rescaled for different views.

Publishing

Publishing is initiated by the publish() method, but this is just a switcher that determines whether the process has permissions to publish to the HTDOCS or not. If not, it forks a privileged publisher process to continue the job; this process ends up in the publish method as well, but since it has the privileges, it calls publish_content() instead to do the actual publishing. In v3, you had to make this determination yourself, and call run_publisher() to get the necessary permissions, but v4 takes care of this automatically; you only have to call publish().

publish_content(%opt)
This performs the actual publishing of files. The generic version in ExSite::Content is probably suitable for all content types, and delegates to different publishing methods:

publish_self(%opt)
This publishes the files for the selected content object.

publish_parent(%opt)
This publishes the object that contains the selected content object. This will, for instance, republish indexes (such as a blog or calendar) when the contents (such as an article or event) is added or changed.

publish_children(%opt)
This publishes all items that are contained by the selected content object. For instance, if you publish a blog, it will publish all of its articles as well. This function is recursive, so that the children of each of the children will also be published, and so on. If you publish a section object, this will effectively republish the entire site.

Workflows

Every content object is in one of the following states:

0. **published**: visible to visitors, and all interactive features are available.
1. **archived**: visible to visitors, but content is no longer current or maintained. Will not be linked from regular menus or indexes. Interactive content types (for example, forums, comments) may cease being interactive when archived. (In other words, comments will stop accepting replies.)

2. **queued**: similar as approved, but will be published automatically by the queuing system
3. **approved**: ok to publish, but not published yet; will automatically be published next time publisher is manually run.
4. **submitted**: pending approval/moderation
5. **draft**: content is being worked on, will not publish automatically.
6. **expired**: content is no longer approved for viewing, but may be reinstated
7. **canceled**: content is not suitable for viewing by visitors, but will be retained for archival purposes.
8. **rejected**: content is considered junk, and can be removed by garbage collectors.

Workflow actions move the object to a new state, eg. **publish, archive, queue, approve, submit, draft** (save as draft)**, expire, cancel, reject**. Content moves toward the publish end of this spectrum when moving onto the site, and away from the publish end of the spectrum when being moved off the site.

The publish and archive workflow actions will also try to write out files if the content is already unpublished. The remaining actions will attempt to remove files if the content is already published.

Content moderation consists of reviewing and approving/publishing items that are in a draft or submitted state. For example, comments can be reviewed, and then either approved or rejected. If you are approving many comments, it is faster to approve them individually, and then republish the whole forum or blog post once. If approving only one, you can skip the approve step and go straight to publish; this takes longer, so it will be tedious to repeat this if approving multiple comments.

ExSite::Revision

Every time a content object is updated, we create a new revision to track the changes. Old revisions are kept on file until explicitly deleted. We can track an unlimited number of old revisions.

Certain revisions have special importance:

- **active** - The active revision is the most recently published one. It is the one that is viewable to the public.
- **newest** - The newest revision is the most recently added one. It has not necessarily been approved for public viewing yet, in which case it is only visible to administrators. When the content is published, the newest revision will become the active revision.
- **draft** - The newest revision can optionally be added as a “draft”. Unlike regular "newest" revisions, draft revisions will not be published. It must be resaved or approved as a non-draft revision before it can be published.

By default, when entering new revisions, they will save as the “newest” revision, unless you use a special “save as draft” submission button.

Setup

There are different methods for creating a revision object:

To get the current revision used by a content object:

```
my $r = $content->revision();
```

To select a specific revision for a particular piece of content:

```
my $r = $content->find_revision($rev)
```

$rev can be "active", "newest", or a particular revision ID. It defaults to active.

To create a revision directly:

```
my $r = new ExSite::Revision(id=>$revision_id);
```

```
my $r = new ExSite::Revision(data=>\%revision_data);
```

In both of these cases, you can also optionally pass content=$content_object if you know it. This will speed up some operations that need to know the context.

Revision Management

```
is_published()
```

Returns true if the revision has ever been published.

```
publish()
```

Write the revision's files to disk, if possible. Regardless of whether any files are written, we mark the revision as published, which means it is viewable by the public.

```
unpublish()
```

Remove the revision's files from disk.

```
copy(%data)
```

Copy this revision, and all its views. Updates the note to reflect that it is a copy. A revision can be copied to another content object by passing the the content_id in the parameter data.

```
restore()
```

Uses copy() to creates a new revision that is identical to this revision. This is used to "restore" an old revision to use, without rolling back through all the intermediate revisions.

Views

The actual content is stored in views. The revision itself only contains information about the update, such as timestamps and changelog notes. There are 5 supported views: normal (default), large, small, thumbnail, and icon. Most revisions will consist of a normal view, plus some optional alternate views.

The normal view is selected by default, but you can select a different view using:

```
$r->select_view($preferred_view);
```

To return the current selected view, use:

```
$r->view();
```

where $preferred_view is one of normal, large, small, thumbnail, or icon. If your preferred view is not available, it will select the closest matching view. Use the following calls to get information about the available views:

```
$r->has_views();  # returns true if ANY views are available
$r->has_view($view_type); # returns true if the given view is available
```

When you ask to display a revision or fetch its URL, the result will reflect the selected view. In fact, most calls to fetch information about the actual content will simply be passed through to the selected view, such as:

```
$r->is_file();
$r->is_image();
$r->is_text();
$r->get_fileinfo();
$r->filename();
$r->get_raw();
$r->get_html();
$r->get_html_dynamic();
$r->get_url();
$r->get_url_static();
$r->get_url_dynamic();
$r->show();
$r->httppath();
$r->diskpath();
```

There are also some shortcut calls to access commonly-used views:

```
$r->get_thumb();
$r->get_thumb_html();
$r->get_thumb_url();
$r->get_thumb_raw();
$r->get_icon();
$r->get_icon_html();
$r->get_icon_url();
$r->get_icon_raw();
$r->get_generic_icon_url();  # finds an icon, even if there is no such view
```

If you request the MIME-type of a revision, however, you will get the original MIME-type of the revision. In principle, views could have different MIME-types than the original content. For instance, if revision is a video, but you include a “small” view, which is just an image still, that view will have an image MIME-type instead.

ExSite::View

Every revision of a content object can be broken down into one or more views. The supported views are:

- **normal** - This is the default view of the content, intended for insertion into a regular web page. If the content is an image, it is scaled to a typical web page body width (by default 512 pixels).
- **large** - If the original data is larger than the normal view, we can optionally retain it in its original form as a large view. (The CMS may nevertheless scale it down if it too large to save in the CMS.) Large views are also suitable for HD video that is larger than what would be served to viewers by default, large or uncompressed documents, and so on.

- **small** - A small view is intended to be used with content summaries, such as are used in listings and indexes. Small views of text content include abstracts, teasers, and summaries. Small views of images are typically scalled down (by default to 256 pixels) so that they can be floated left or right of the surrounding text. Small views of documents might consist of simply the cover or introduction.
- **thumbnail** - Thumbnails are normally used for images. A thumbnail view is scaled to a good size for aggregating into thumbnail grids or "contact sheets". If a thumbnail view is requested, we scale it down to the system thumbnail size (typically 100 pixels). For other types of content, we can install image thumbnails so that they can also be presented this way.
- **icon** - An icon view is an extra-small view that is typically used for listing files or downloads. They are small enough to inline into text. Icon views are not automatically generated. The system can be asked to provide a generic icon (eg. suitable for the content's mime-type) if no specific icon is installed.

Setup/create your view object in one of these ways:

```
my $view = $revision->select_view($viewtype);
my $view = new ExSite::View(id=>$view_id);
my $view = new ExSite::View(data=>\%viewdata);
```

Internally, the view is stored in one of 4 formats:

- as a **file** that was uploaded
- as **text** that was entered directly
- as a **path** to some other file on the server
- as an **URL** to a remote resource elsewhere on the web

You can obtain the particular format used in a view using:

```
$view->format();
$view->is_file();  # true if file format
```

Storing files in **file** format is simple, but can be costly from a database perspective, because it results in a large amount of data transfer from the database server, plus an encoding/decoding overhead. If you set the configuration parameter

```
content.install_files = 1
```

then the underlying Diskfile object will attempt to write the files straight to disk, and then use the **path** format to reference the file. Doing this saves performance on database transfers, file decodes, and publishing overhead.

Disk Files

In cases where the view can publish as a file (such as images, stylesheets, documents, and so on), you can obtain a Diskfile object that represents this file:

```
my $file = $view->diskfile();
```

This object is used for all publishing operations. See the documentation for ExSite::Diskfile for more information.

The filename that will be used for the current view is given by:

```
my $filename = $view->filename();
```

The original filename that was used to upload the file can be retrieved using:

```
my $orig_filename = $view->original_filename();
```

The publication location of the file can be obtained using:

```
$view->httppath();   # path in URL
$view->diskpath();   # full path on server
```

MIME-type

Each view records its own mime-type, which can be retrieved using:

```
my $mime = $view->getdata("mime_type");
```

If the mime-type is not recorded, we can make a guess based on the data or file extensions:

```
my $mime = $view->guess_mime_type();
```

You can also use this to guess the mime type of other content, by passing in the data directly in one of the 4 accepted formats:

```
my $mime = $view->guess_mime_type($data,$format);
```

The simplified call:

```
my $mime = $view->mime_type();
```

does all of the above, as needed, to come up with a suitable mime-type for the content.

The following calls will also give you some more general MIME information:

```
$view->is_image;

$view->is_text;
$view->is_publishable;  # true for all non-HTML MIME-types
```

Images

If the data is an image, you can fetch an ExSite::Image object for it:

```
my $img = $view->get_image();
```

You can also do the reverse, and pass an ExSite::Image object in as the data:

```
$view->set_image($img,$filename);
```

The following calls will work for images:

```
my $w = $view->width;
my $h = $view->height;
my ($w,$h) = $view->dim;
```

Retrieving/Displaying Data

Views can be output/displayed in 3 ways:

- **HTML** - The system will generate the necessary HTML to display the content.
- **URL** - The system will return an appropriate URL to retrieve the content, suitable for use in hyperlinks or as SRC= attributes in img tags.
- **raw** - The system will return the raw content directly. This is suitable for delivering the content direct to the user's browser, or writing it to a disk file.

Use the following methods:

```
$view->get_html();
$view->get_html_dynamic();
$view->get_url();
$view->get_url_dynamic();
$view->get_url_static();
$view->get_raw();
$view->show(mode=>"source"); # outputs displayable HTML source code
$view->show(mode=>"preview"); # displays from the database, not published files
```