Exware Solutions Inc.

ExSite 4 Developer Guide

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. The Kernel and Plug-in systems are very similar to v3. However, although the Plug-in system is similar architecturally, many plug-ins will nevertheless get major revisions going into v4, because:

- they dupe features that are now in the CMS, and should be rebuilt as special CMS classes
- their interactions with the CMS will change
- their UI will be updated and modernized
- the v4 transition is a good opportunity to "break" with old and tired methodologies since backward-compatibility is not a concern

Changes to the CMS layer are covered in a separate document, *ExSite 4 Content Model*. This developer guide covers new v4 features in the other layers.

CGI Programs

All CGI programs have been replaced with a single program, ex.cgi that provides a single entry point with common security, configuration, database, error handling, and setup tools.

The general format of a URL is

/cgi/ex.cgi/COMMAND/path_info

The commands, and their relations to old v3 CGI programs are shown in this table:

v3	v4
/cgi/login.cgi	/cgi/ex.cgi/login
/cgi/logout.cgi	/cgi/ex.cgi/logout
/cgi/home.cgi	/cgi/ex.cgi/admin
/cgi/ctrl-panel.cgi/Module	/cgi/ex.cgi/admin/Module
/cgi/page.cgi	/cgi/ex.cgi/page
/cgi/content.cgi	/cgi/ex.cgi/view
/cgi/doform.cgi	/cgi/ex.cgi/doform

v3	v4
/cgi/getdata.cgi	/cgi/ex.cgi/peek
/cgi/publish.cgi	/cgi/ex.cgi/publish (or /cgi/publish.cgi)
/cgi/dcd.cgi	/cgi/ex.cgi/dcd
/cgi/dlg.cgi	/cgi/ex.cgi/dialog

There is still a publish.cgi program (and unprivileged publish.pl counterpart) that handles the job of writing files to HTdocs. However, it can be invoked using ex.cgi/publish (or by calling a publish method directly) – they will automatically spawn a privileged publish job if necessary.

ExSite can be sped up by changing the #! line to use /usr/bin/speedy instead of /usr/bin/perl.

URLs can be simplified using URL rewrite rules that do something like:

> /**function**/path -> /cgi/ex.cgi/**function**/path

for example:

> **/cgi/ex.cgi/page/foo** could shorten to **/page/foo**

then set

> server.CGIpath =
> prog.page = page

to have the system generate its URLs in this format.

Note that when you use an URL shortening rule like this, you cannot name your base-level content objects the same as any of your functions, as that creates an ambiguity that will prevent the content from being served as you expect it to.

Database Changes

v3	v4
ExSite::SQL	ExSite::MySQL
section, page, content	content (with a type column)
content_data	revision, view
member	user
member_site_link	content_key
keyword, keyword_tag	deprecated, replaced with Index class and aliases

Notes:

- section_id columns contain the content_id of a content record of type “section”. They might be replaced with content_id columns, if they can be related to something more specific than a section.
- member_id has generally been replaced with UID.

DB Maps

DB maps now support a help column. This replaces the old help files in cgi/dbmap/help/table/column. These columns will be used to automatically add tooltips to your forms.

The **enum**: datatype works like a list: datatype, except that it returns an integer value instead of the text value. For example:

```
list:onoff          on|off
```

returns text values of either "on" or "off".

```
enum:onoff          1:on|0:off
```

returns numeric values of 1 or 0, but displays text values of *on* or *off* to the user.

Attributes & Metadata

The attribute table (and ExSite::Attribute class) provides a generic metadata functionality, as in v3. This can be overloaded to provide a more dedicated/specialized metadata feature for specific tables or types of data. V4 has a few predefined specialized metadata tables/classes:

table	class	extends	notes
attribute	ExSite::Attribute	any table	used for preferences
metadata	ExSite::Metadata	content	see v4 Content Model document
contact_info	Modules::ID::ContactInfo	contact	contact information fields

Use/inherit the ObjectMeta class on the extended object to get these metadata features. When calling the new() or setup() methods of the extended object, you can optionally pass a metadata option to preload the metadata. This can save some database reads in cases where you already have the data handy:

```
my $article = new ExSite::Article(id=>456, data=>$article, metadata=>$article_metadata);
```

```
my $contact = new Modules::ID::Contact(id=>123, data=>$contact_record, metadata=>$contact_info);
```

Object Handling

As in v3, an ExSite object is a software representation of a database record. This all works similarly to v3, with some efficiency enhancements.

You can preload your objects with partial data if you have some of the record data handy. Simply pass your incomplete datahash to the data=> parameter in new(). If getdata() requests data that is already present, no database load is required. But it needs an efficient way to check whether it needs to load the full record if the data is not present. See ExSite::Content::getdata() and ExSite::Content::loaded() for an example implementation of this feature.

The ObjectMeta class also lets you preload metadata into your objects if you happen to have it handy, which can also save on database calls. Pass the metadata as a hash of key=>value pairs.

```
my $content = new ExSite::Content(data=>$cdata,metadata=>\%meta);
```

The ObjectList class will try to track metadata as well, if you provide it, and use this feature to preload object metadata when you pull objects out of the list. To push a record and its metadata onto the list, use:

```
$list->pushmeta($record,$metadata);  # one object
$list->pushmeta(\@records,\@metadata);   # multiple objects
```

Not all list operations will be able to preserve the metadata. In those cases, the metadata might get cleared, which will force it to be reloaded from the database.

Objects also support an unload() method that tries to drop internal attributes that are no longer needed. This is useful when looping over large data sets, to keep memory leaks under control.

When working with lists of content objects, extracting an object from the list will return a content subtype class, eg. a Blog object instead of a generic Content object.

Security

There are now 10 access levels:

#	old#	name	notes
0	0	public	unauthenticated visitor
1	1	user	authenticated user
2		member	official or paid member
3		site-defined	board members, committee leads, ... ???
4		site-defined	
5		site-defined	reviewers/moderators ???
6		site-defined	
7	2	manager	can perform all section-specific admin tasks
8	3	system administrator	can perform system-wide tasks, can make keys
9	4	root	all security rules disabled

Levels 0-4 are regular website **visitors**, permitted to use the website front-end. Levels 5-9 are **executive** users, permitted to access the website back-end. Levels 3-7 are undefined for now, and can be used for customizing local site privileges.

DB maps must be upgraded to reflect this. Code should not test for absolute levels, but for user types. For example:

- is_admin (level 8+)
- is_manager (level 7+)
- is_executive (level 5+)
- is_visitor (level 4 or lower)
- is_member (level 2+)
- is_user (level 1+)

Privacy settings and access controls are now numeric. A privacy/access level of 2, for instance, means that the viewer must be at least level 2 (an active member) to view the data or access the function.

Groups

You can create user groups using the security tools, and assign or remove arbitrary users to/from these groups.

Keys

Access keys can be assigned for any content object, not just sections. (However, sections remain the default way of partitioning security.)

Access keys can also be revoked for particular content. To do this, set the key type to “revoke” instead of “grant”. Revoked keys disable a certain type of access in that part of the site. For example, say user Admin has editorial permissions on the main section. But on a particular page, you revoke those editorial permissions. The admin user’s permissions will not extend to that page, or any below it.

Keys can be granted to individual users, or to groups. Group keys can be used by everyone in that group.

Security Roles

v3	v4
editor, designer, administrator	editor, designer, administrator, reviewer, owner

The reviewer role controls moderation and approval of submitted and draft content.When new content requires approval, the reviewers will be consulted first.

The owner role is for content that originates from non-executive users. (For example, a regular member could have permission to publish content that they owned, such as postings, profiles, etc.)

If the auth.view_keys config option is enabled, any key will grant access to view a content object. That would allow you to grant specific users or groups permission to view admin-only content, for instance.

Configurations

Configuration files work similarly to v3, with some enhancements:

You can setup an array with only a single element using

```
conf.setting += foo
```

(In v3, this appends to a setting; in v4 it initializes an array if the setting is undefined.)

Module config files normally set config settings in the module namespace (eg. foo=1 in Blog.conf actually sets Blog.foo=1). You can access the global namespace in Module config files using "-", for example:

```
-.content.metadata.profile.organization.datatype = string
```

Settings

Settings are configuration parameters that can be set either globally, on a content type, or on a specific content object. They can be set in configuration files, or in preferences. For example, blogs have a configurable parameter, index_min, which defines the minimum number of articles that will be shown in a blog index.

To set this globally:

```
setting.blog.index_min = 3
```

To set it on a specific blog:

```
setting.blog:BLOG_ID.index_min = 1   # numeric
```

or

```
setting.blog:News.index_min = 1    # by name
```

You can also put settings into preferences. Use the first form (in this case, setting.blog.index_min), and attach to the preference to either:

- the content object (the blog) itself
- the reference object (for instance, an article setting could be put on the blog)
- the section object

Important/Useful New Configuration Settings

```
server.fileroot = /var/www/files   # where to install uploaded files
form.jquery_validation = 0  # jquery form validation enhancements (off by default; turn on if your site uses jquery)
report.compact_columns = 10  # switch to compact CSS if there are at least this many columns
report.permission_denied = (no permission to view this data)   # customize column access denied msg
search.skip.library = 1   # skip indexing of certain content types
content.viewdim.normal = 512   # optimal sizes of images
content.posting_date_format = date   # how to display posting dates
content.metadata.page.keywords.datatype = string    # set metadata and its dbmap characteristics*
content.contextual_metadata = 0    # allow content objects to inherit metadata definitions from parent objects
content.flags.form += captcha     # define a flag (boolean metadata)*
content.format.article = ...   # format string for presenting articles*
content.format.article_summary = ...   # format string for presenting article summaries in indexes*
content.format.blog_article_summary = ...   # format string for presenting article summaries in blog indexes*
content.blog.stickiness = temporary     # special config settings for content types*
spacechar = _    # what to substitute for spaces in URLs
queue.start = 10    # when to start pulling things off the queue
queue.end = 16    # when to stop pulling things off the queue
queue.perday = 2    # how many queued items to publish each day
```

* The setting shown is particular to one content type, but the general format of the setting can be used for any content type.

Forms

Forms can implement Bootstrap tooltips, by including a tooltip parameter to the input methods. DBmap help attributes will be automatically used to make form tooltips on auto-generated forms.

In FormBuilder, you can override the layout of particular questions using qtemplate(). For example, to display a checkbox before the prompt for the waiver question, you could do this:

```
$form->qtemplate("waiver","<p>[[input]] [[prompt]]</p>");
```

This lets you avoid templating the whole form when you just want to treat one question differently.

Turn on jquery form validation to get better CSS highlighting of your missing form data:

```
form.jquery_validation = 1
```

If you submit with missing data, those fields will be highlighted.

Reports

If the number of columns exceeds a configurable value (10 by default), reports automatically switch to more compact CSS.

Reports can be saved and reloaded later by a different process:

```
$report->save("unique key",$optional_number_of_days);
```

Later, you can re-load the report using:

```
$report->load("unique key");
```

and display, export, or modify it. (You need to re-save if you want your modifications preserved.)

Saved reports can be made available to the Database Query module; simply list the report keys you want to be able to view in Query.conf, like so:

```
saved_reports = Report #1
saved_reports += Report #2
... etc
```

Saved reports are a simple way of quickly viewing reports that are slow and costly to generate. For example, build the reports once on an overnight job, and then used the saved report for quickly viewing the results with no wait.

Filtered Reports

You can preload a volume of raw data into a report, and then generate numerous reports by applying filters and rules to the raw data.

Raw data is one datahash per row, ie. a set of key => value pairs, using the same keys in each row. Preload your data using:

```
$report->rawhead(%head);        # key => header name
$report->rawdata(@data);        # each array element is a row/datahash
$report->rawpush(@data);        # add more rows
```

Then generate a report using:

```
$report->select(%opt);
```

To select rows, use any of the following comparator options:

- match => match hash, matching rows will be selected for display

- notmatch => like match, but matches are excluded
- like => like match, but ignores case, charset, and whitespace
- notlike => like like, but matches are excluded
- contains => match hash, rows containing the substrings will be selected
- notcontains => like contains, but matches are excluded
- gt => match hash, rows greater than the values will be selected
- lt => { match hash, rows less than the values will be selected }
- empty => [array of keys, rows in which the key is empty will be selected]
- notempty => [array of keys, rows in which the key is not empty will be selected]

If you specify no conditions, rows will be automatically selected for display.

To apply logical operators to your conditions, use these options:

- method => and|or - use this method to join multiple conditions in one call
  - or = select rows where any condition matches
  - and = select rows where all conditions match [default]
- reselect => and|or - use this method to combine results with the results from a previous pass
  - or = add matches to the previous results
  - and = remove non-matches from the previous results

To select which columns to display in the final report:

```
columns => [ array of keys to include in the output ]
```

Then you can call make, export, or save, as usual.

You can include any number of conditions in the options, for example:

```
# select rows where a<5 and b>10 and c==15

$report->select(lt=>{a=>5},gt=>{b=>10},match=>{c=>15});
```

By default, the conditions are combined with a logical AND, ie. all conditions must match for a row to be selected. Set method=>"or" to combine with a logical OR instead, ie. ANY condition may match for a row to be selected.

Use reselect=>and|or to perform multiple passes on the data in a similar way. This lets you create more complicated logical tests, for example:

```
# select rows where (a<5 || b<10) && (c==3 || d==6)
# first, select rows where a<5 OR b<10
```

```
$report->select(lt=>{a=>5,b=>10},method=>"or");
# of the selected rows, select rows where c==3 or d==6
$report->select(match=>{c=>3,d=>6},method=>"or",reselect=>"and");

# select rows where (a<5 && b<10) || (c==3 && d==6)
# first, select rows where a<5 and b<10
$report->select(lt=>{a=>5,b=>10}); # method=>"and" assumed by default
# also include rows where c==3 and d==6
$report->select(match=>{c=>3,d=>6},reselect=>"or");
```

User Interface

The administrator back-end is at:

/cgi/ex.cgi/admin
/cgi/ex.cgi/admin/*<Module>*

The base panel of the admin interface is a launcher menu that lets you launch other panels and interfaces, as needed. It includes all of your plugins, organized into categories. The Overview category includes general views of your whole system or site, including:

- alternative launcher panels, such as the v3 webtop, special Dashboards
- links to your website front-ends
- help

Control Panels

If your plug-in partitions its material by section, you can start with a section selector, as in v3. For CMS tools, it may be friendlier (fewer clicks) to just show all of the available child objects with their section identified. For example, a calendar tool could just show all of the available calendars in all sections the admin has access to, rather than asking them to select a section first.

2-pane control panels (as used in the old Document plug-in, for instance) are no longer supported.

UI Class

The ExSite::UI class replaces v3's ExSite::HTML class. It has a lot of the same sorts of widgets, but they are called in an OOP manner, eg.

```
$out .= &ExSiite::HTML::BasicBox(%options);   # v3

$out .= $ui->BasicBox(%options);   #v4
```

Although similar, many of the widgets have been replaced with Bootstrap equivalents. There are a few new widgets in v4:

widget	description
navitem	generic clickable element
dispitem	generic display element
ErrorMsg	displays an error message

widget	description
SuccessMsg	displays a confirmation message
HelpMsg	displays help information
SuccessBox	same as SuccessMsg, but with a title and frame
HelpBox	same as HelpMsg, but with a title and frame
AlertBox	a floating dialog, that can be dismissed
Figure	a graphic with a caption
TreeView	replaces old DynList
OverlayFrame	an iframe that floats over the main screen
TitleBar	an application titlebar with optional links
Menu	a vertically-stacked menu of links
Spinner	a please-wait spinner
FATool	a small icon tool button, using Font Awesome

The calling conventions of these widgets have been normalized and made consistent across all calls. There are some standard parameters that can be passed to many of these widgets:

parameter	description
tone	good, bad, warning, important, neutral - sets the general styling of the widget
size	l, m, s - sets the size of the widget
ucicon	name of a unicode icon to use
faicon	name of a Font Awesome icon to use
icon	url of an image to use as an icon
confirm	pops up a confirmation dialog with this message when clicked

Note that buttons and tools, which accept an url parameter to link to something, can also accept a links parameter, which is a list of things to link to. This will generate a pop-up menu when clicked.

The OverlayFrame widget is particularly useful when you switch contexts, such as:

- jumping to a different plug-in module
- showing a preview of a front-end page or item
- when your main screen is slow to generate, and you don't want to lose it as the user works through sub-views
- anywhere you might consider using a popup window

To make use of the OverlayFrame, simply insert it into your page:

```
$out .= $ui->OverlayFrame();
```

To make a link open in the OverlayFrame, use this method:

```
my $link = $ui->OLFrameLink($url);
$out .= $ml->a("open in OverlayFrame",{href=>$link});
```

ML

The markup class has been updated to support HTML5. Set the HTML level it should presume using

```
markup.html_level = 5  # 4 or 5; 5 is the default
```

It now AUTOLOADs implicit methods for all tags, so you can call

```
$ml->foo(...)   # or _foo or __foo
```

and it will generate markup like

```
<foo>...</foo>
```

ML objects also support smart headings, which allows you to generate well-structured and context-appropriate headings, even in cases where you don't know the current heading level.

```
$ml->hlevel(n);  # set the heading level without generating a heading
$ml->h1($heading);  # output a heading at level 1, and set that heading level (ditto for h2 ... h6)
$ml->h($heading);  # generating a heading at the current heading level
$ml->_h($heading);  # append a heading at the current heading level
$ml->hup($heading);  # go up/back one heading level (eg. h3 to h2) and generate a heading
$ml->hdown($heading);  # go down one heading level (eg. h2 to h3) and generate a heading
```

Page and Output Handling

General CMS features are covered in the *ExSite 4 Content Model* document, but there are a few points of interest to developers, that are worth covering here.

Publishing

V4 prefers static content views, where possible. The system supports hourly, daily, and weekly publish rules, in addition to static and dynamic. These can help to keep your static content refreshed automatically even when it changes unpredictably (for example, upcoming events, new posts, member directories).

Personalized pages are harder to render as static views. If the personalizations are small, you can sometimes move the dynamic elements into Javascript (the Login plugin does this, for instance).

V4 allows for intelligent determination of whether a content object should be static or dynamic. If the publish field is not set, the system will call the object's **publish_rule_heuristic**() method to determine the most appropriate publish rule to use. For example, the default heuristic is:

- dynamic if the content is access-controlled (eg. members-only)
- the most dynamic rule of the embedded plugins. Plugins can declare their own publish rules using the PublishRule ioctl. For example, SimpleMenu has a static PublishRule, because its output is always the same on any given page.
- default is static

For example, event views can change day-to-day, as registrations open and close. In v3, that meant that they had to be dynamic, but they would continue to be dynamic even after events were over. In v4, events will automatically assume their publish rule is "daily" before the event if there is registration, and "static" after the event. That means you get best

results by not setting any publish rule at all, which allows it to be flexible enough to change the rule depending on the date.

If only a specific content object on the page needs to be dynamic or access-controlled, you can set the access or publish rule on that content only. The main page will be static, and the dynamic content will be fetched with javascript. If the object is an image, and access is denied, it will be replaced with an access denied image.

Note that if your page is completely dynamic, it will still generate a static file containing a meta-redirect to the dynamic view. That means static URLs should always work for simple GET requests.

Publishing will be invoked automatically by certain workflow operations (like "publish", obviously). In some cases, those workflow operations can be initiated by unprivileged users. For example, if comments are accepted without moderation, then threads and/or forum indexes will republish automatically when they are submitted. You must pass an approve=>1 flag to the publish() call to approve such operations.

CMS tags

You do not necessarily have to do your own URL generation when linking to content.

You can request a specific view when you link to content using [[content]] tags to get the URL. For instance, say you have a content object called *image*, then you can link to the normal default view using [[image]]. But if you have a high-res version on file, you can link to that instead, using [[image:large]].

You can also force dynamic views and even select specific revisions, using tags like [[image?revision=REV]] where REV is a revision name (newest, draft, etc) or number.

Similarly, {{page?query_string}} will substitute a dynamic link to the page, with the appropriate query string settings. If you just use {{?query_string}}, the page will link to itself. {{}} will link to itself with no query string setting, but will use a static URL, while {{!}} will always link to a dynamic URL. {{&Module}} will link to the service page for a module, so you can link to the shopping cart using {{&Pay}}, for instance.

Metadata

The metadata insertion tags <!--$special--> have a few new options:

<!--$page_header--> and <!--$page_footer--> are used for appending to the <head> section of the document, as well as to the end of the document. If your plug-in has some additions that belong properly in the <head> section (for example, <style> blocks, or metadata), then do this:

```
$share{page_header} .= $head_out;
```

Similarly, if your plug-in has some additional that are better put at the end of the document (for example, javascript code that should load after other content has loaded), then do this:

```
$share{page_footer} .= $foot_out;
```

Those %share variables will automatically be substituted in where the <!--$page_header--> and <!--$page_footer--> tags are inserted. If you didn't include your own *page_footer* tag, it will append to the end of the HTML document.

Note that ExSite automatically puts stuff into these variables. For example, optional javascript will get added to *page_footer* so that it runs after the page is loaded. For this reason, always append to these %share variables so you don't wipe out any existing settings.

The <!--$metadata--> tag will be replaced with a full set of <meta> tags, depending on which ones have been preconfigured to display automatically. To include meta tags automatically, use a setting like

```
content.metadata.page.description.show = 1   # ExSite content-specific metadata

content.metadata._.twitter:card.show =1   # generic metadata
```

All such meta tags with a show setting will be included, if they have valid explicit or implicit content.

Javascript

To include standard Javascript in your output (on pages or control panels), use:

```
$out .= &insert_js($script1, $script2, $script3, ... );
```

so that we can avoid inserting javascript that has already been included somewhere else. $script can be:

- a standard library name, like *jquery*, *jqueryui*, or *bootstrap*
- a standard Bootstrap initialization snippet, such as *tooltip*, *popover*, or *dropdown*
- a standard ExSite js file in /_ExSite/js, such as *misc* or *httprequest*
- the path to any other javascript file
- raw javascript code

If you are are including any standard javascript in your front-end website templates, you can mark them as included using config settings like

```
content.js.bootstrap = 1
```

Then any attempt by a plugin to include bootstrap will do nothing, since it knows it was already loaded by the template.

CSS

You can track what CSS files have been inserted in a similar way:

```
$out .= &insert_css($css);
```

or, to put the style links/blocks in the document head where they belong:

```
$share{page_header} .= &insert_css($css);
```

$css can be:

- a standard ExSite CSS file in /_ExSite/css, such as Report.css (the suffix is optional)
- any full path to a CSS file
- raw CSS code

AJAX

The dispatcher is a new v4 feature that tries to gang together multiuple AJAX calls into a single AJAX call, for more efficient AJAX communications. You do not need to do anything to enable this feature, except set

```
content.ajax_method = dispatch
```

which should be the default.

The dispatcher replaces all module AJAX calls, such as those invoked using single and double-AJAX notation like

```
<!--&&Module()-->
```

with a single call to dispatch(). This performs all of the AJAX calls in a single http request, and returns the results in a JSON structure. This structure is parsed, and the various outputs are dispatched back to their appropriate page elements. That means that a page should only ever generate one AJAX hit on the server, no matter how many AJAX modules are invoked on the page. (However, AJAX posts will be dispatched separately.)

Frameworks

E-Commerce Services

E-commerce and Financial reports work similarly to v3. Some significant new features have been added, however.

Shopping Cart

There is a new Cart class in the Finance framework that handles the shopping cart functionality. It inherits from the Receivable class. If you use the Pay module to generate your shopping cart views, you will not need to interact with the Cart directly. But if you need more control over the shopping cart experience, you can use the Cart class as a shopping cart API as follows:

Obtain a shopping cart object:

```
my $cart = new Modules::Finance::Cart();  # automatically connects to the current shopping cart
```

Display the cart:

```
$out .= $cart->show(%options);  # interactive, eg. delete from cart, change quantity, checkout
$out .= $cart->show_readonly(%options);  # only shows the cart, no interactivity
```

Add an item to the cart:

```
$cart->add(%item);  # %item describes a receivable_item
```

Receivable Items can now be related to each other through a parent column; this allows for multiple purchases in the cart to be related to each other as co-purchases. Removing the parent purchase will, for example, also remove the co-purchase.

Note that deleting items from the cart does not actually delete the receivable_item records any more; it merely changes the record status to *deleted*. Remember this when looping over the items in an invoice; use $item->is_active to determine whether the item is active (not deleted). Similarly, items can also have a status of “hidden” (do not display in end-user views) and “readonly” (do not permit the end-user to alter or delete the item).

When calling $invoice->show(%opt), use the options deleted=>1 or hidden=>1 to included deleted or hidden items in your views of the contents, as they will not be shown by default.

Callbacks to Purchased Objects

Individual line items can be related to database objects using the objtype and objid fields, as in v3. To obtain a purchased object, simply use:

```
my $obj = $receivable_item->purchased_object();
```

There are 6 standard callbacks that will be used automatically, if they exist in the purchased object:

```
$obj->sale_select($item); # this is called when the object is added to the cart
$obj->sale_quantity($item,$old_q,$new_q); # this is called when the quantity of the object is changed
$obj->sale_delete($item); # this is called when the object is removed from the cart
$obj->sale_activate($item); # this is called when the sale is closed and the invoice is activated
$obj->sale_complete($tem); # this is called when the sale is completed and paid in full
$obj->sale_note($item); # this is used to append purchase notes/messages to invoices and receipts
```

In each case, the receivable item object is passed to the callback.

Purchase Locations

Purchased objects can define their own location for the purpose of calculating taxes and other surcharges. When the object is added to the cart, you can pass country and provstate values that will be used for this purpose. This information is saved in the receivable_item record, so no handlers or special callbacks are needed.

If no location information is included in the purchased object, the purchaser location will be used instead.

GL Codes

Itemized financial reports can include GL codes for each item. GL codes have 2 forms, internal and external. Internal GL codes have the format “AXX.YYY.ZZZ”:

A = 1 for sales, 2 for payments received, 3 for refunds, and 4 for payments issued
XX = the type (sales type/accounting code for sales, payment method for payments)
YYY = a purchase category identifier (eg. an event, or catalog ID)
ZZZ = a purchase subcategory identifier (eg. a fee, or product ID)

To set these GL code values, set the following fields in the receivable_item table, or in the parameters to $cart->add(...):

XX: acctcode_id
YYY: acctcode2
ZZZ: acctcode3

For example, the internal GL code 102.556.562 is a sale (1xx) of Account Code 2 (event registrations, say), relating to event 556 and fee 562.

This GL code structure allows for some generalization, grouping, and categorization, for instance 102.* is all event-registration sales, whereas 102.556.* is all sales for one particular event, and 102.556.562 is all sales for one fee. Sorting by GL code will automatically group related sales together.

GL Code Mapping

Further grouping and categorization is possible with GL code mapping. Internal GL codes can be mapped to external GL codes using the glcode table. This table defines simple rules for converting internal GL codes to external GL codes for

some other accounting system. A mapping rule consists of an internal GL code, which is compared to the GL code of an actual item, and an external GL code that is used if the two match. Mapping rules can define one, two, or all three fields of the internal GL code, for example:

102 - matches any internal GL code starting with 102

102.556 - matches any internal GL code starting with 102.556

102.556.563 - matches that GL code exactly.

Example mapping rule: If event 556 is a conference with multiple different fees, but you want all conference fees to be grouped by the same external GL code, the following map rule will use “CONF2016” as the GL code for all registrations in all fees for that event.

102.556 --> CONF2016

If there are multiple mapping rules that all match, it will choose the most specific one that matches.

You can also use GL code maps to equivalence different items. For example, if you run two sessions of a course, they will end up with different internal GL codes because they are different events. If one is event 600 and the other is event 601, you can equivalence them with a simple mapping rule like:

102.601 --> 102.600

or, alternatively

102.600 --> COURSE101
102.601 --> COURSE101

Payment GL codes

Payments also use GL codes in the format "AXX.YYY.ZZZ". A is 2 for revenues, and 4 for refunds. XX is set based on the payment method. YYY is the account ID that the payment was made against, and ZZZ is the invoice that was being paid, if that information is available.

Note that payments also have a new *ident* field which contains a payment identifier. This is meant to be used for including an identification value such as last 4 digits of credit card, or cheque number. Keeping this in a separate field makes it more easily reported and searchable than putting it into the notes field.

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.

The individual contact_info fields each have their own privacy settings (numeric, equivalent to access level), which supersede the privacy setting of the contact record itself.

However, it will complicate contact queries, especially on multiple fields. To get multiple contact fields as separate rows, do something like this:

select c.contact_id,c.name,i.name,i.value from contact c, contact_info i where i.contact_id=c.contact id and ...

To get multiple contact fields on a single row, do something like this:

```
select c.contact_id,c.name,phone.value,email.value
from contact c
left join contact_info email on email.contact_id=c.contact_id
left join contact_info phone on phone.contact_id=c.contact id
where email.name="email" and phone.name="phone" and ...
```

To quickly get the contact fields for a particular contact record, use:

```
my %contact_info = $contact->get_info();
```

The v3 phone1 and phone2 fields have been replaced with phone and cell fields.

Address cards auto-validate their provstate settings, based on which country was selected.

Contact records can be attached to accounts or to contact records. The latter is used for things like:

- location addresses
- profile/directory addresses

Forms

The v3 QA framework is replaced by v4's Forms framework. Forms and questions are now content objects, which means:

- they use standard access controls and translation tools
- the revisions are treated as templates. Revisions of a form are the layout/template for the form, while revisions of a question are used as a layout/template for that one question.

New features that are made possible by this approach:

- layouts are contained in the form, not in libraries
- no question libraries; just re-use questions by copying them from old forms
- smarter question layouts (eg. checkboxes first)

Other new features that have also been added:

- HTML input types
- drag-and-drop question ordering
- context-sensitive question setup and configuration

Note that, as content objects, forms do not utilize service pages any more. However, forms effectively serve as their own service page. To embed a form into a page, place the form content object under that page, and embed it into the page content using <!--content(form_name)-->. Submitting that form effectively directs you to /cgi/ex.cgi/page_name/form_name, which allows you to customize the layout of the form response.

As a developer, the primary change to consider is that forms are not remote objects that are referenced through a qa_form_id, but are content objects that can be embedded right in the object you are working on. For example a registration fee can contain its own form as sub-content. Or a membership type can contain a sign-up or renewal form (or both) as sub-content. To check for the existence of such a form, do this:

```
$registration_form = $fee->my_content("registration_form");
```

$registration_form will now be the form object itself, if one exists.

Plugin Development

V4 plugins are structured the same as v3, with read() write(), and ioctl() methods. The same ioctl commands are recognized, and you can follow the same general methodology as in v3.

If your plugin is built on the v4 CMS framework (eg. it manages a new content type), then it can inherit from Modules::Content to get a full suite of CMS admin functions, including:

- **pathbar**() to display your current position in the site
- **configure**() to set the content record fields and specific metadata attributes
- **update**() to add a new revision, and **rollback**() to discard the latest revision
- **delete**() to remove the content and its descendants
- **copy**() to make a copy of the content
- **search**() to find content within an object (eg. find an event in a calendar)
- **pricing**() to manage the prices of an item
- **order**() to change the sorting of the items under the content
- **todo**() to add a to-do item on this content
- **translations**() to manage the translations of this content
- **images**() to manage the images under this content
- **contacts**() to manage the contact records for this content
- **about**() to display all of the relevant information about this content
- **tags**() to manage all of the keyword tags on the content
- **metadata**() to manage the specific and generic metadata for the content
- **schedule**() to manage the scheduled tasks for the content
- **wf_tools**() to display appropriate workflow actions in a toolbar or menu, and **schedule_tool**() to include a schedule button
- **publish**(), **unpublish**(), **archive**(), **approve**(), **submit**(), **queue**(), **unqueue**(), **draft**(), **expire**(), **cancel**(), and **reject**() methods to perform those workflow operations

You simply need to decide which ones to expose to the admin in your toolbar menu, and your command switch statement. These methods all accept a content object as a parameter, so they can be used to manage any content object, including ones from other content classes that are not explicitly handled by your plugin.

In cases where your content makes use of other content objects (like images, email templates, or forms), those secondary content objects should simply be embedded under their parent objects, and managed using the same tools as above. Do not place them in libraries unless it is your intent to share them between multiple content objects.

Plugin Best Practices

- inherit from the Modules::Content class to get standardized UI and methods for managing content objects
- use the global ML object to keep heading and other context

```
$ml = &get_obj("ML");
```

- do not duplicate underlying CMS features if possible
- use static or interval publishing, where possible
- use wf_tools() to suggest reasonable workflow operations
- avoid using a section selector if the next step just makes you select from a small number of content objects; select those objects on the first screen, and show all sections at once
- use $ui->OverlayFrame() when cross-linking to other plugins, live website views, or other operations that would normally use a popup to avoid losing your control panel
- control panels should display, in order, and where appropriate: pathbar, toolbar, status messages, work area/preview
- when linking to another plugin or inclined to use a popup, try an OverlayFrame
- use the **bad** tone for destructive actions, such as delete buttons.
- if your toolbars are too busy, group and hide the less-used functions in drop-down menus.
- wrap previews in a BasicBox or something similar

Should you rebuild your plug-in using the V4 CMS framework?

Your plug-in can probably be converted to v4 with some simple changes:

- if you are calling into any CMS objects (like Section, Page), those calls may need to be updated
- direct references to a section_id in the data may need to be replaced with content_id
- update the control panel UI calls

Otherwise, it should just work. But you should consider a rewrite using the V4 CMS framework if your module has any of the following features:

- hierarchical data structures
- translations

- HTML editing (TinyMCE)
- workflow, moderation, approvals
- uploaded images or files
- scheduled tasks
- search
- prices, e-commerce
- access controls
- metadata or other settings
- forms
- templates
- contact records

Sharing those common frameworks means less coding, less debugging, more UI consistency, and it's easier to extend your tools to include additional features on this list.