Roxen CMS 5.0

Web Developer Manual

2010-06-02
www.roxen.com

Roxen Internet Software
Box 449
SE-581 05 Linköping
Sweden
www.roxen.com

Contents

1 Introduction

This part of the documentation is intended for anyone who creates and publishes web pages using a Roxen WebServer. It describes the functionality Roxen WebServer provides that can be used to automate and ease the creation of both advanced static content as well as dynamic Internet applications. Roxen WebServer and Roxen CMS. It describes the functionality Roxen CMS and Roxen WebServer provides that can be used to automate and ease the creation of both advanced static content as well as dynamic Internet applications.

Most of Roxen CMS's and Roxen WebServer's functions are available as RXML tags, easily learned by anyone who knows HTML, XHTML or XML. A basic introduction is provided but the reader is encouraged to obtain more knowledge from other sources, should this information be new.

The manual is structured into four different parts. First is the introductory part, which you have just entered, that outlines different technologies and concepts important for creating websites with Roxen products. After the general introductions follows a closer look at Roxen specific technologies with the main focus on RXML, including the uses of RXML and the syntax and semantics of the RXML language. The third part of the manual is a reference part which lists and describes all user tags. The list is grouped by the different tags general application so that it should be easy to find a tag in a solution-centric approach. Several reference chapters start out with a closer look at some common properties of the tags in the group. The last chapter is an alphabetical list of all user tags.

2 About the document

The reference documentation found in this manual is directly derived from the documentation in the source code of Roxen WebServer and its modules. The purpose of this approach is to get the documentation as close to the actual implementation so that is should be as easy as possible to keep it up to date. The big benefit of having the documentation in the source code is possibility to online documentation within the product. All tag documentation found in this manual is available through the `<help>` tag as well as the documentation tab in the administration interface.

If you want to know more about a tag when you are developing a web page, simply add the <help> tag to a page and view it to see the latest documentation for that tag. Note that the module in which the tag is defined must be loaded into the server in order for this feature to work. The online documentation feature is however not restricted to only modules originating from Roxen Internet Software. Any third party or locally developed modules containing a compatible documentation may be viewed by the online documentation feature.

2.1 Notation example

This is how the tag documentation looks like. This tag has been flagged as a container tag, i.e. you can put content into it like this: `<example-tag>content</example-tag>`. A tag may also be tagged as a tag-only tag, i.e. you may only write it as `<example-tag/>`.

```
<example-tag></example-tag>
```

Provided by module: TagDoc Notation exemplifier

2.1.1 Attributes

age="number"

This is the documentation of the 'age' attribute to the `<example-tag>`. In this case the attribute accepts a number, e.g. `<example-tag age='42'></example-tag>`. This attribute is required. If it doesn't exist in the tag you will get an RXML parse error.

sort="{up, down}" (up)

This is the documentation of the 'sort' attribute. The sort attribute may have either the value 'up' or the value 'down'. If the attribute is omitted, the tag will assume the value 'up'.

`&_.ent;` (provided by *TagDoc Notation exemplifier*)

This entity is an internal entity of the `<example-tag>` and only available inside it, just like `<internal>`.

2.1.1.1 <internal/>

Provided by module: *TagDoc Notation exemplifier*

This is an internal tag to `<example-tag>`, which means that it is only available inside the `<example-tag>`. Below is an example of how tag usage examples looks like. These may be a single box just showing how to write the tag or it could be a double box showing both the code and the result.

```
<example-tag><internal/></example-tag>
```

3 Basic Concepts

In this chapter you'll find an overview of the concepts and standards that web servers in general and Roxen WebServer and RXML in particular are built upon. These are only provided as a short introduction and background information to what follows in this manual.

We will take a look at three main topics, URLs, HTTP and XML. URLs are used to point out and locate a resource on the Internet. HTTP is used to retrieve that resource and interact with the webserver, and XML is used as the basis for describing resources. The URL section will introduce http/https/ftp URLs and outline the differences between absolute URLs, relative URLs and absolute path URLs. The HTTP section will discuss the stateless properties of the HTTP protocol and explain how this affects HTTP authentication and the need for cookies. It then ends by quickly mention the role of MIME types. Finally the XML section introduces the XML syntax and outlines the XML application XHTML. There is no mention of HTML, since those who do not know HTML are better off learning XHTML, and those who do know HTML have no problems applying XML restrictions on HTML to get XHTML.

3.1 URL

3.1.1 Absolute URLs

One of the most important concepts of the WWW is the URL standard, Uniform Resource Location. A URL is used as a pointer to a resource, usually a web page, on the Internet. A typical URL may look like this:

```
http://www.roxen.com/index.xml
```

This URL can be split into three different parts. Everything before "://" is the protocol, in this example "http", which is the hyper text transfer protocol normally used on the WWW. Between the "://" and the first "/" is the host and the rest is the path. What is then the actual information contained in the line above? It simply means retrieve the document "/index.xml" from the host "www.roxen.com" with the HTTP protocol. Other common protocols are "https", which is secure HTTP utilizing the security layer SSL or TLS, and "ftp", which is an older way of transporting files.

The host part of the URL may optionally include a port description. If you consider the host domain as an address to the host, you can consider the port number the apartment number. The port number is implied by the selected protocol, e.g. 80 for http and 443 for https, but sometimes you want to reach a site that is not on the default port. This is done by appending ":" and the port number to the host part of the URL. The following URL points to the same resource as the above one.

```
http://www.roxen.com:80/index.xml
```

These types of URLs, containing protocol, host and path, are referred to as absolute URLs.

3.1.2 Relative URLs

Another often used type of URLs is relative URLs. They refer to another document from within a document, e.g:

```
search.xml
```

The URL above means "the file called search.xml in the same directory as the document referring to it". If a link to "search.xml" was found in "`http://www.roxen.com/index.xml`" it means "`http://www.roxen.com/search.xml`". But if the same URL was found as a link in "`http://www.roxen.com/platform/index.xml`" it would mean "`http://www.roxen.com/platform/search.xml`".

A relative URL may also point to directories above or below its position. If we want to link to search.xml in a subdirectory games it would be "`games/search.xml`". If we on the other hand want to link to search.xml in the directory above we would use the ".." to denote up, e.g. "`../search.xml`". It is possible to combine several ".." tokens, or combine them with directory names to walk down another path branch, e.g. "`../../platform/search.xml`".

3.1.3 Absolute path URL

Somewhere between absolute and relative URLs we'll find the absolute path URLs. They are not relative with respect to where on the server the URL was found, but it is relative with respect to on which server it was found.

```
/index.xml
```

The above URL means "the file index.xml in the top directory of the current server". Choosing the right URL for the right occasion will often reduce maintenance problems. A group of files that uses relative links between them can easily be moved to another directory or server. A file that links to the main search page with absolute path URLs can easily be moved to another directory or server. Luckily the shortest possible URL is often the best choice.

URLs are standardized in RFC 2396. Allowed and forbidden characters as well as characters with special meaning are of particular interest when developing Internet applications and web sites.

3.2 HTTP

The most popular protocol for transferring documents over the WWW is the hypertext transfer protocol, HTTP. Typically the browser connects to the web server sending over a request for a URL, gets a response from the server and then the connection is closed. This means that the browser has to connect to the server for every thing it downloads, e.g. if a HTML page has 40 images the browser needs to make 1+40 separate requests to the server. Since there is no persistent connection between the browser and the server there is no way to know if the user is looking at the web page just sent to him, or if she continued to look at other web pages from another server.

The original intent with HTTP was to make it a "stateless" protocol, i.e. that all requests to the server is independent, from the servers point of view. In practice that means that each server response should only rely on the information given in that very request. The benefit with this approach is that it becomes easier to make an efficient server implementation that can server web pages to any number of users, since no information about the users or the requests needs to be stored once a response has been transmitted.

3.2.1 Authentication

A stateless protocol has some interesting implications for authentication. As you perhaps know it is possible for a web server to respond with an "authentication required" response, telling the browser that it has to provide a user name and a password in order to get the contents at the given URL. This is often referred to as a login request. If the webserver accepts the user name and the password the browser remembers the user name/password pair as a valid authentication for that server, and will use them for all subsequent requests to the server. (This is an oversimplification. Read about authentication realms in RFC 2617) As a consequence there is, contrary to common belief, no logout mechanism. This is often emulated by temporary rejecting the valid user name/password causing the browser to drop the user name/password pair as valid authentication. The drawback is that the user will be presented with a new login request that she has to cancel.

3.2.2 Cookies

Another way of overcoming the drawbacks of the stateless protocol model is cookies. A cookie is a browser variable that can be set and altered by the server. Once set the browser will include the cookie in all requests to the server, thus large cookies will "waste" a lot of bandwidth. When the server sets a new value to a cookie it also gets to decide the realm of the cookie, e.g. to which URLs the browser should send the cookie, and an expiration date, at which time the browser automatically removes the cookie. There is no defined method to remove a cookie, so that operation is often simulated by setting the cookie value to an empty string and setting the expiration time to a date that has already occurred. Some browsers remove the cookie at once while others wait until the next time it is restarted. Read more about cookies in RFC 2109.

3.2.3 Content-Type

The URL of a resource doesn't necessarily give away the type of its contents, and URLs were never intended to be used for that either. Hence every response from the web server contains a content type header containing the contents "MIME type", e.g. text/html for HTML files or image/gif for GIF files. In Roxen WebServer these MIME types are usually derived from the file extension by the content type module, but scripts and other modules may choose another MIME type. MIME types are handled by IANA and a complete list of all official MIME types can be found at www.iana.org.

3.3 XML

3.3.1 Tags

XML is a way of describing how to describe things. You can think of XML as the alphabet which you can use to create words with. Then you take these words and actually perform the actual act of describing things. These uses of XML are often referred to as XML applications. One of these XML applications is XHTML, which is a markup language made for use on the WWW. E.g. if I want the word "blind" to be bold in the sentence "Three blind mice." I would write

```
Three <b>blind</b> mice.
```

The `<b>` and `</b>` are called tags, a start tag and an end tag. The starttag, `<b>`, turns bold on and the endtag, `</b>`, turns off the bold property. The XML rules regarding tags are fairly simple, you must turn off, or close, all tags that you have opened and you must do it in the reverse order of how you opened them. An example with both bold and italic in XHTML:

```
<b>Three <i>blind</i></b> <i>mice</i>.
```

An empty tag `<x></x>` may be compressed into `<x/>`. This is useful for tags that doesn't need any content to be meaningful, e.g. the line break tag `<br/>` in XHTML.

3.3.2 Attributes

Tags can be made more specific by the use of attributes, which consists of an attribute name and an attribute value, or argument.

```
<b lang="en-uk">Three blind mice.</b>
```

Note that either two `"` or two `'` may be used around the argument, depending on the contents. Example:

```
Did you hear that someone actually donated a pie to
<person name='William "Bill" Gates'>him</person>?
```

3.3.3 Entities

In addition to tags XML also specifies entities, which are used as a constant that represents something else. It is for example forbidden to use the characters < and > for anything else than making tags in XML. Thus there must be some other way to write these characters when you need them, &lt; (less than) and &gt; (greater than).

More information about both XML and XHTML is available at www.w3.org.

3.4 XSLT

A common application of XML is to invent XML formats of your own, where text or other data is stored in an orderly fashion, compliant to a given vocabulary and rules of what tag constellations are legal. Such a ruleset is called a Document Type Definition, or DTD for short, but we will not go into greater depths about them here, and you need not be intimately familiar with them to use the XML and XSLT features of CMS Advanced.

XSLT, short for Extensible Stylesheet Language Transformations, is a language for transforming XML documents into other XML documents. Or, in other words, XSLT is a generic standardized template system that can be used to separate your data from its layout, formatting and other aspects of presentation, so that you can change either independent of the other.

This template system transforms the source files stored in the site repository, or the site repository and a database, into files suitable for the browsers used. For web browsers this most commonly means converting to XHTML, for WAP phones WML.

The transformation consists of choosing which parts of the source files will be displayed and creating the layout necessary to display it nicely. Thus the template system ensures a consistent look and feel of the whole site.

4 Managing templates in a Basic site

The layout in the Basic site included in Roxen CMS is controlled by XSLT templates. The templates translate the high-level XML markup in the pages into HTML which the web browser can display.

You can customize the layout by changing e.g. fonts, colors, background patterns, borders, margins and so on. It is also possible to make more complex changes involving, among other things, the site navigation and the news archive view. You can choose between altering parameters in the existing layout, extend the factory-installed stylesheets, write new stylesheets from scratch, or use a combination of all methods.

4.1 Factory-installed template files

The templates are shipped as read-only files which cannot be edited directly. This is done to make future server upgrades easier since it would be complex to add new components if the files have been edited in customer installations. However, thanks to the architecture of Roxen CMS you can override a single template definition or a complete file by copying the factory-provided files into a writeable directory and then apply your changes.

Fine-tuning the font parameters and some of the graphics is even easier. This can be accomplished without writing a single line of XSLT code, instead making your selections in a user-friendly customization wizard.

To access the templates, enter the SiteBuilder editing interface by going to the URL `http://my-server/edit/`. The read-only directory with factory templates and graphics is named `/roxen-files`. Inside this directory there are a number of subdirectories with a prefix of either *i-* or *cms-*. We will only cover the *cms-* directories in this chapter; the *i-* directories are used by CMS Instant and not covered in this chapter.

In addition to the read-only files there are a handful of templates in the root directory of the Basic site. They inherit all the necessary read-only templates from the `/roxen-files/` directory in the following order:

```
 /cms-common.xsl
   |
   +-> /cms-components.xsl
   |      |
   |      +-> /roxen-files/cms-sites/4.5/strings.xsl
   |      |
   |      +-> /roxen-files/cms-sites/4.5/params.xsl
   |      |
   |      +-> /roxen-files/cms-sites/4.5/applications.xsl
   |      |      |
   |      |      +-> /roxen-files/cms-sites/4.5/components.xsl
   |      |      |      |
   |      |      |      +-> /roxen-files/cms-sites/4.5/
   |      |      |      |         fallback.xsl
   |      |      |      |
   |      |      |      +-> /roxen-files/cms-sites/4.5/
   |      |      |      |         components/header-comp.xsl
   |      |      |      |
   |      |      |      +-> /roxen-files/cms-sites/4.5/
```

```
  |     |     |     |        components/footer-comp.xsl
  |     |     |     |
  |     |     |     +-> ...plus all other factory-installed components
  |     |     |
  |     |     +-> /roxen-files/cms-sites/4.5/
  |     |     |        forum/forum.xsl
  |     |     |
  |     |     +-> /roxen-files/cms-sites/4.5/
  |     |     |        booking/booking.xsl
  |     |     |
  |     |     +-> /roxen-files/cms-sites/4.5/
  |     |              poll/poll.xsl
  |     |
  |     +-> /roxen-files/cms-sites/4.5/css/base.xsl
  |     |     |
  |     |     + > /roxen-files/cms-sites/4.5/css/common.xsl
  |     |     |
  |     |     + > /roxen-files/cms-sites/4.5/css/screen.xsl
  |     |     |
  |     |     + > /roxen-files/cms-sites/4.5/css/print.xsl
  |     |     |
  |     |     + > /roxen-files/cms-sites/4.5/css/handheld.xsl
  |     |     |
  |     |     + > /roxen-files/cms-sites/4.5/css/components.xsl
  |     |
  |     +-> your custom templates
  |
  +-> /roxen-files/cms-sites/4.5/page-layout.xsl
        |
        +-> /roxen-files/cms-sites/4.5/navigation.xsl
              |
              +-> /roxen-files/cms-sites/4.5/
                       navigation-xml-tree.xsl
/cms-print.xsl
  |
  +-> /cms-common.xsl
        |
        +-> see above
```

All XML files which are generated by the component editor will get `cms-common.xsl` as their template. Note however that when you are working inside the component editor the system will apply `cms-components.xsl` instead in order to temporarily hide the navigation interface. This means that custom components must not be imported directly into `cms-common.xsl` since they would then become unavailable to the editor.

The cms-print.xsl template is only applied temporarily to a page when a printer friendly layout is requested.

4.1.1 Navigation menus

Code for producing web page navigation is contained in two template files, `navigation.xsl` and `navigation-xml-tree.xml` in the `/roxen-files/cms-sites/4.5/` directory. The latter template constructs an internal representation of the navigation structure where menu entries are placed in a tree based on the contents of menu files, directory structure, current page and other properties. This tree is then passed to the layout code in the first navigation template for styling. By dividing the code in two parts you can easily replace the presentation logic while preserving the generation of the tree.

4.1.2 CSS styles

The files in `/roxen-files/cms-sites/4.5/css/` contain CSS styles controlling the page layout. Styles specific to a component resides in the XSL template for the component. These component styles are then called from `/roxen-files/cms-sites/4.5/css/components.xsl`.

4.1.3 External visibility

All factory template files have their external visibility metadata set to never as indicated by the eye icon in the file browser. The reason for this is to prevent site visitors from downloading the source code to the templates. The stationery file for new XSLT templates has been configured in the same way.

4.2 Customizing template parameters

Many of the XSLT templates have predefined parameters which can be changed using the command Customize Template in the Edit popup menu. Since parameter customization is inherited through `<xsl:import>` statements you should select Customize Template on the `/cms-components.xsl` file to change settings for any of its imports, including the read-only files.

You will find that there are parameters for controlling font colors, font names, page colors, margins, component spacing, header logo file, image directory path and more. If you change a parameter you also need to check the Override Imported check box next to the parameter or else the new value will not take effect over the inherited template's default value.

The file/directory path parameters all start with `&page.mountpoint;`. This is needed for the site to work correctly in case the administrator relocates the site to a mountpoint other than /. Be sure to preserve the mountpoint placeholder at the beginning of the paths if you enter a different value.

4.3 Creating customized layouts

If you want to make more changes than the template parameters support you can upload custom graphics and write your own XSLT templates.

The graphical elements are stored in two locations:

- `/header-logo.gif`
- `/roxen-files/cms-images/`

The header logo is a writable file so you can upload new content to it. Alternatively you can set the template parameter named *header-logo* to point to a different image file. To modify the other graphical elements you need to make a copy of the `/roxen-files/cms-images/` directory. Next, modify the template parameter image-path to reflect the new location so the factory templates will find your images.

Another kind of customization involves translating the language-dependent parts of the factory templates. All message strings, date format patterns and so on have been collected in the `/roxen-files/cms-templates/strings.xsl` template. To change it you start by making a copy of the original `strings.xsl`, edit the copy and finally insert an `<xsl:import>` statement pointing to your file at the end of `/cms-components.xsl`. Be sure that the import statement is placed after any of the `/roxen-files/` imports since you need to give your file a higher import precedence.

Using the same procedure you can override or extend any of the factory templates handling page layout and editor component rendering. The XSLT architecture is very flexible since it allows you to reuse template code and only provide customized code for those template rules which you want to replace. This means that you can remove duplicate code from the copy of the read-only template file and only keep those parts which should override the default layout. This in turn increases the compatibility with future updates to the factory-installed templates.

4.4 Controlling the component editor

4.4.1 Directives in template files

Roxen CMS makes use of two special XSL files in your site to control the behavior of the page editor:

`cms-components.xsl`

As seen earlier in *Factory-installed template files*, cms-components.xsl is the starting point for importing component-specific templates. This is also the XSL template that the page editor loads directly while presenting the editing interface.

By placing additional directives in this stylesheet, or any of the stylesheets it imports, you can disable specific components from being used in a given section of the web site.

Components are by default enabled until you add a <xsl:param> declaration of a variable named component tag name-enable. For instance, the Link component uses the tag <href-component>, and thus the controlling parameter is named href-component-enable. You should give the parameter necessary rxml: attributes to become customizable through the web-based user interface.

All standard components already have their visibility parameters declared in the /roxen-files/ files. For reference, here is the full declaration of href-component-enable:

```
<xsl:param name="href-component-enable" rxml:type="checkbox"
           rxml:group="Component enable"
           rxml:doc="Enable the Link component"
           select="1"/>
```

In a similar way you can also declare which variants you want to have for a component. This is described in the next section, *Defining component variants*.

`cms-fckconfig.js`

The rich-text editor used in Roxen CMS is based on a third-party software project called FCK Editor. This project also maintains a Wiki site with technical documentation that can be of interest when customizing the editor. Examples of uses include adding or removing toolbar icons and defining custom CSS styles.

Roxen CMS allows for FCK Editor features to be customized by creating a cms-fckconfig.js JavaScript definition file. The default instance of this file is /roxen-files/cms-templates/cms-fckconfig.js. However, when overriding the default copy the preferred approach is to use the following technique: Create an RXML page with the reserved file name, which, when evaluated, inserts the original file and then adds or redefines the standard FCK Editor configuration parameters.

The benefit of this model is that future Roxen CMS versions can add more standard parameters to the default file, and your customized versions will inherit these parameters with no extra effort.

The step-by-step process creating the RXML file is described in a comment inside /roxen-files/cms-templates/cms-fckconfig.js, but is repeated here as well:

- **Create/copy `cms-fckconfig.js`.** Place it where it can be found using the smart search strategy.
- **Assign a content type of HTML.** This ensures it will be parsed server-side by the RXML parser.
- **Add code to inherit the default file.** Insert the following code fragment into the file:

```
<header name="Content-Type" value="application/x-javascript"/>
<insert file="/roxen-files/cms-templates/cms-fckconfig.js"/>
<?noparse
  ...your custom JavaScript code here...
?>
```

`cms-toolbar.xsl`

This file gives you control over some of the main toolbar buttons in the Insite Editor. For detailed information, see the section *Controlling the Insite Editor toolbar*.

All of the aforementioned files are located using the smart search strategy employed throughout Roxen CMS (explained in the section *Where to Place Stylesheets*). This lets you customize the site's behavior in a flexible way, for example to implement sub-sites.

4.4.2 Directives in content files

The XML file format used in the component editor allows advanced site designers to interleave traditional XML and RXML statements together with component elements. This can e.g. be used to produce non-standard layouts such as the two-column start page in the Basic site. However, such layouts are often tuned to a specific set of components and may not render correctly if unrestricted editing takes place in the component editor.

To solve these cases the designer may want to restrict editing in one or more ways for a particular page. This can be done using some attributes in the top-level <page-components> element or, in some cases, a specific component element or its fields. All actions are unlocked by default.

rxml:insert-lock="yes"

Only allowed on the top-level <page-components> element. When set to yes the component editor will disable the insert buttons on the page.

rxml:delete-lock="yes"

Allowed on the top-level <page-components> element as well as individual component elements. When set to yes the component editor will disable the delete button for the corresponding component elements. Note that when set on the top-level element the locking state cannot be overridden in any of the page components.

rxml:move-lock="yes"

Only allowed on the top-level <page-components> element. When set to yes the component editor will disable the arrow buttons which are used for rearranging components.

rxml:edit-lock="yes"

Allowed on the top-level <page-components> element as well as individual component elements and their fields. When set to yes for the whole file or a component the component editor will disable the edit button for the corresponding component elements, and when used on a selected component field that field will be locked for editing.

4.4.3 Examples on field/component locking

The first example below shows a header component where the variant field is locked to the value 1 (but the title remains unlocked), and a Text & Picture component which cannot be deleted.

```
<?xml version="1.0"?>
<page-components>

  <!-- Lock <variant> field for this header -->
  <header-component>
    <variant rxml:edit-lock="yes">1</variant>
    <title>Fixed variant</title>
  </header-component>

  <!-- Can't delete this component -->
  <picture-component rxml:delete-lock="yes">
    <variant>0</variant>
    <title>Locked for deletion</title>
    <text>Lorem ipsum.</text>
  </picture-component>

</page-components>
```

The second example presents a page where the existing components can be edited but not moved or deleted. Moreover, new components cannot be inserted.

```
<?xml version="1.0"?>
<page-components rxml:insert-lock="yes" rxml:move-lock="yes"
   rxml:delete-lock="yes">

  <header-component/>

  <picture-component>
    <variant>0</variant>
    <title>Edit me</title>
    <text>Lorem ipsum.</text>
  </picture-component>

  <footer-component/>

</page-components>
```

4.5 Defining component variants

4.5.1 Purpose of variants

The majority of the components in the component editor present a list of variants which the page author can choose from. Variants are normally small layout variations, e.g. left/right horizontal alignment, but can be more complex if the component developer wants to. It's possible to add, rename, remove and reorder variants for standard components as well as your own.

Internally variants are numbered starting at zero. The user will not see this number but the stylesheet creator uses them in the template match pattern or in the template rule itself:

```
<xsl:template match="sample-component-a[variant = '0']">
  <!-- this code is only used for variant #0 -->
  ...
</xsl:template>
```

```
<xsl:template match="sample-component-b">
  <!-- code used for all variants -->
  <xsl:if test="variant = 1">
    <!-- only variant #1 will run this code -->
    ...
  </xsl:if>
  ...
</xsl:template>
```

4.5.2 Configuring variants – current Roxen CMS versions

Today's Roxen CMS version lets you define variants directly in the site repository through the use of XSL parameters. Compared to the deprecated method described below this approach supports sub-sites in a single repository.

XSL templates should use <xsl:param> to declare variants with a name that is constructed as component tag name-variants. Taking the Link component as an example, the parameter is named href-component-variants.

Here is the declaration used in the factory templates for the Link component:

```
<xsl:param name="href-component-variants" rxml:type="text"
           rxml:group="Component variants"
           rxml:doc="Variants for the Link component"
           select="'0:Left aligned&#10;1:Center aligned&#10;2:Right
aligned'"/>
```

As seen here, the actual choices are encoded as a string which follows a special pattern where each variant is listed as number:title, and multiple patterns are separated with newline characters (entered as a numeric XML entity using &#10;).

Note!

> In order to activate variants declared this way, the administrator must remove all old-style variant definitions in the Administration interface. See below for details on how to change these settings.

4.5.3 Configuration variants – Roxen CMS version 4.0 or earlier

The following describes the means used to define component variants in older versions of Roxen CMS. This is no longer recommended for newly created sites since it is far less flexible than the new method.

Note!

> Actions described in this section are in part performed in the Roxen administration interface and may therefore not be accessible to all SiteBuilder users.

The relationship between variant names and numbers is defined in the Roxen administration interface. Go to Sites -> (site name) -> (component module) -> Settings and you will find a module variable called Component variants. This variable defines a list of names where each entry is prefixed by the corresponding variant number. Edit the list to change what the page author sees in the variant pop up menu in the component editor. Don't forget to add matching rules for new variants to the XSLT templates using the methods shown earlier in this section.

4.6 Controlling the Insite Editor toolbar

4.6.1 Disabling toolbar commands

The main toolbar in the Insite Editor contains buttons that, among other things, create, edit and delete pages. In a complex web site the web developer may determine that certain parts of the web site should follow a special set of rules when content is added or modified; in such cases it may be preferred to disable one or more of the standard action buttons in order to guide editors to use other means of interacting with the site.

In the situation described above, it's not uncommon for the web developer to provide custom buttons within the web site itself to e.g. create pages. These buttons can be linked to RXML scripts which indirectly activate the same fundamental page creation functionality, but gives some additional control over location, stationery choice or similar.

For example, in a news archive it can be an advantage to give new pages a path that is generated from today's date. Using SiteBuilder tags you get a chance to compute the path, create the directories and finally the index.xml page in the proper place. If editors at the same time have the familiar "New" button accessible in the toolbar, chances are that they will not use the custom button and thereby create pages that don't conform to the intended directory structure.

To forcefully disable buttons, create a file called cms-toolbar.xsl and include one or more <xsl:param> declarations with the following names:

- toolbar-new-button-enable
- toolbar-edit-button-enable
- toolbar-delete-button-enable
- toolbar-publish-button-enable
- toolbar-update-button-enable
- toolbar-revert-button-enable

The `<xsl:param>` declaration requires a rxml:type attribute in order for Roxen CMS to detect it properly.

Here is one example:

```
<xsl:param name="toolbar-new-button-enable"
           rxml:type="checkbox" select="0"/>
```

Note!

The methods described in this section should not be seen as a secure way of preventing commands to be executed. The same result can in many cases be achieved by switching to the Content Editor environment, running RXML tags or accessing the editor through other interfaces such as FTP or WebDAV. The goal is rather to assist the users to choose the right tool and reduce the risk of making errors.

When more advanced restrictions are necessary you need to make use of the Access Control features in the product.

4.6.2 Spellchecker

It is possible to control the default dictionary with the parameter default-dictionary in the cms-toolbar.xsl file. If the dictionary for the files language is set to "*" the parameter will be used. This is useful in a subsite environment where each subsite has a non-English language but all files are language less. Example:

```
  <xsl:param name="default-dictionary"
             rxml:type="string" select="'nl'"/>
```

4.6.3 Author

It is possible to control the author wizard with the parameter author-zone in the cms-toolbar.xsl file. If the parameter is specified the author select box will only contain users from the specified zone. Example:

```
  <xsl:param name="author-zone"
             rxml:type="string" select="'Common'"/>
```

4.6.4 Local timezones

Roxen CMS includes support for letting sub-sites specify their own timezone. This only affects date/time strings shown in the user interface when working with files in the corresponding directories and not the output from RXML tags. Site sections that don't have a defined timezone will use the server's timezone.

To use this feature, declare the parameter toolbar-timezone in the cms-toolbar.xsl file. Example:

```
  <xsl:param name="toolbar-timezone"
             rxml:type="string" select="'GMT'"/>
```

4.6.5 Custom FCK Editor filters

As seen in Controlling the component editor, various aspects of FCK Editor can be controlled programmatically. Another customizable feature of the editor is the HTML cleaning that takes place when a page is saved; the default filter is used to remove unwanted tags, harmonize tag names (e.g. rename <em> to <i>) and more. If a custom behavior is needed you can add your own filter and activate it with fck-filter and/or fck-table-filter parameters in the cms-toolbar.xsl file. Here is one example:

```
  <xsl:param name="fck-filter"
             rxml:type="string" select="'MyFCKFilter'"/>
  <xsl:param name="fck-table-filter"
             rxml:type="string" select="'MyFCKTableFilter'"/>
```

Note!

> The MyFCKFilter and MyFCKTableFilter identifiers are case-sensitive. The corresponding code needs to be added to the server installation. More details on how to develop and install a filter is presented in FCK Editor Filters.

4.6.6 Detection of simultaneous editing

Roxen CMS has full support for dealing with several users editing the same page or file simultaneously, but in some setups this may be discouraged for various reasons. For instance, the knowledge of the conflict resolution procedure may be limited among the people using the CMS.

One alternative is to apply a setting in the cms-toolbar.xsl file that will present a warning if a user attempts to edit a page that is currently being edited by someone else. Note that this setting will only generate a notification and won’t block editing, but it will at least make the user aware of the situation.

Here is an example of how to define the parameter in cms-toolbar.xsl:

```
  <xsl:param name="toolbar-clash-prevention-enable"
             rxml:type="checkbox" select="1"/>
```

5 Roxen Concepts

Before looking into the actual features provided for static and dynamic page generation, let's have a look at what we want to accomplish with these features. The following list outlines four major properties in the WWW that must be implemented on the server side to be reliable and secure.

- Connection to content generation, such as image rendering or database retrieval.
- Interaction, such as posting information on chat boards, committing polls or performing searches in search engines.
- Monitoring and control, such as bandwidth throttling, access control and page access counters.
- Traffic control, such as HTTP cache settings and HTTP redirects.

Furthermore we want to be able to do content/layout separation as well as using macro functions to shorten development time and ease maintenance. Though these two properties do not have to be done at the server side, it is a good compromise between letting a content generation program creating static content and relying on the client to interpret the information correctly.

5.1.1 Scripts

Traditionally scripts have been the answer to demand for the above properties. A script is a small program that is run when a client has requested a certain URL, and the output from the program is then sent back to the client. The benefits are complete control over the transaction enabling implementation of all the above properties, but there are several drawbacks. Scripts are more expensive to handle, since you need programmers to create programs, and it is difficult to reuse the script code in an efficient manner.

Several types of scripts are supported by Roxen WebServer. Pike scripts are the most efficient, since they run inside the server process itself. The second most efficient way of using scripts is using the Extscript feature in Roxen, currently supporting Perl, which keeps the overhead of starting a Perl interpreter low by letting several requests use the same interpreter before it is discarded. Finally Roxen WebServer supports the FCGI and CGI interface making it possible to run virtually any script.

5.1.2 Modules

Modules in Roxen WebServer can be considered as scripts through which all requests passes, and it is up to the module to decide when to react on a request. This solves both the problems with scripts, that they are not general and that they are not as manageable. The script code can now be reused for any page that we like, e.g. by inserting a marker into the page or by explicitly adding the page URL to that modules configuration. The use of markers to trigger some kind of action in the module, e.g. inclusion of certain data, also makes the web site more manageable, since backend programmers and page designers now work on different levels.

This description of modules is an oversimplification, since there are many different kinds of modules in Roxen WebServer, with different tasks and ways to be activated. Read the Administrator and Programmer manual for more in-depth information.

5.1.3 RXML

RXML, RoXen Macro Language (it was invented before XML), is a functional, serverside, XML compliant scripting language that easily integrates itself with web content. Since its syntax is already familiar for many it is often very easy to learn. Its tight integration with the XML in the pages enables the module programmer to keep as little HTML as possible in the program code, thus leaving more creative freedom to the web designer, i.e. a good separation between layout and functional code.

```
<table>
<emit source="sql" query="select * from users">
  <tr><td>&_.name;</td><td><a href="mailto:&_.email;">&_.email;</a></td></tr>
</emit>
</table>
```

5.1.4 In-page scripting

To "complete the circle" it should also be mentioned that it is possible to do in-page scripting, i.e. putting functional code inside the actual web pages. This is currently possible with Pike code and Perl code.

5.2 RXML Evaluation

RXML is an XML compliant programming language which can be used to produce dynamic as well as static content. The core of RXML is tags and entities, but unlike XML both tags and entity references may be expanded into dynamic values or have serverside side-effects.

To make it easier to use RXML variables, they are grouped into scopes. You can think of a scope as a bucket with variables. When you reference a variable, you first type the name of the scope, then a period, and then the name of the variable. In the example below the variable *thing* in the scope *var* is set to "Book" and the variable *thing* in the scope *form* is set to "Chair".

```
<set variable='var.thing'>Book</set>
<set variable='form.thing'>Chair</set>
My &var.thing; is on the &form.thing;.
My Book is on the Chair.
```

In the last row of the example the two variables are inserted into the document by writing the variable reference as an XML entity reference, i.e. inside `&` and `;`. The RXML parser distinguishes RXML variable entity references from ordinary XML entity references by looking for the period.

As a rule of thumb, the content and attributes of an RXML tag are evaluated first and the tag itself afterwards, as shown in the next example. First var.thing is set to "Book" and then the value of var.thing is expanded in the content of the second <set> tag so that form.thing is set to the value of var.thing, i.e. "Book".

```
<set variable='var.thing'>Book</set>
<set variable='form.thing'>&var.thing;</set>
My &var.thing; is on the &form.thing;
My Book is on the Book.
```

You can also use RXML variable entity references to insert values into the attributes of tags, as shown by these two examples:

```
<set variable='var.variable'>var.title</set>
<set variable='&var.variable;'>Autour de la Lune</set>
Title: &var.title;
Title: Autour de la Lune
```

```
<set variable='var.variable'>title</set>
<set variable='var.&var.variable;'>Autour de la Lune</set>
Title: &var.title;
Title: Autour de la Lune
```

5.2.1 Evaluation order

The general evaluation order of RXML is from top to bottom of the page, and inside and out in terms of tag levels. The "top to bottom" means that if two non-nested tags appear on a page, the first one on the page is evaluated before the second one.

```
<set variable='var.value'>One</set>
&var.value;<br />
<append variable='var.value'> Two</append>
&var.value;<br />
One
One Two
```

The "inside and out" on the other hand describes how tags behave if they are nested. Then the innermost first gets to produce its result. That result is then handed over to the surrounding tag, which in turn produces its output with the first tags output as input.

```
<case case='upper'><date part='wday' type='string' /></case>
MONDAY
```

5.3 URL Extensions

Since everything after the host part of the URL is sent to the server as is, after transport encoding, the server in practice decides the meaning of that part of the URL. Normally that part of the URL consists of a path part, containing the path to a file the client wants, and a query part, containing variables that may affect the way the server accomplishes the request.

```
http://roxen.com/products/platform/tech-features.xml?page=9
```

In the URL above we request the file /products/platform/tech-features.xml from the host roxen.com with the variable page set to 9. Note that there is in principle nothing that prevents us from making a server that returns the same result with the following URL.

```
http://roxen.com/SELECT_page_FROM_features_WHERE_product=platform_AND_page=9
```

This is however not a good idea for several reasons, usability being the number one. Users are used to alter the URL of a page to get to index-pages higher up in the web site structure.

5.3.1 Index pages

The most common sidestep from the rule that the path part of the URL explicitly denotes a file is directory URLs.

```
http://roxen.com/products/platform/
```

The above URL does not denote a specific file in the /products/platform/ directory, but does instead point at the directory itself. The common approach is to find an index file in the directory and send that file instead. This is handled by the directory module, which by default looks for the files "index.html", "index.xml", "index.htm", "index.pike" and "index.cgi" in that order.

5.3.2 Path info

Sometimes it can be practical to fake a directory structure, but let all requests to the files in that directory lead to the same file. The example with the tech-features.xml URL above could look like this:

```
http://roxen.com/products/platform/tech-features.xml/9/
```

The part of the URL after the actual file will then be provided to the file/script in a special variable during its parsing.

5.3.3 Prestates

When developing and debugging is a great help to be able to turn on and off specific parts of the code that generates the current page. This is an ideal application for prestates, a mechanism invented by Roxen to enable the user to turn certain switches on and off. The name and function of the prestates is decided by the page developer. One example of how prestates are used is the Table/Image Border Unveiler module, which is used on the community.roxen.com web site.

```
http://community.roxen.com/(tables)/developers/
```

This URL signifies that we want to fetch file at the path /developers/ from the host community.roxen.com with the protocol HTTP and with the prestate tables set. In the WebServer the Table/Image Border Unveiler module recognizes the table prestate and knows that the user wants all tables highlighted in the page. Compare the result with how the page looks without the prestate. It is also possible to add several prestates to the same page in a comma separated list, e.g.

```
http://community.roxen.com/(tables,images)/developers/
```

Prestates can of course be used for many other things than switching debugging flags, e.g. moving states between pages like a browser window local cookie. See *<apre>* and *<if prestate>* for more information about how to control and detect prestates in your RXML applications.

5.3.4 Config states

A variation of prestates is the config state. Looks very similar to the prestates, but stores its value in a cookie. Looking at the following URL will store the value "bacon" in the cookie RoxenConfig, which will be valid for two years since its latest change. After the cookie has been set, the server will redirect to the page you came from, or it it was unable to determine that page, to the same URL but without the config state.

```
http://community.roxen.com/<bacon>/
```

Removing a config state value is a little trickier than with prestates, since you can not edit them by hand, as with the URL. Prepending a minus sign before a config state flag indicates that it should be removed from the RoxenConfig cookie. As with prestates it is possible to combine several states at the same time, both with and without minus signs.

```
http://community.roxen.com/<egg,-bacon>/
```

See *<aconf>* and *<if config>* for more information about how to control and detect config stats in your RXML applications.

6 RXML Variables and Entities

An RXML *variable* is a binding of a value to a variable name in a scope. Values are usually strings, but can also be numbers or more complex data types – see the next chapter. Most variables may change values during the RXML evaluation.

A *variable reference* is usually written as `scope.variable`, i.e. the name of the scope, followed by a period, followed by the name of the variable. Depending on the value in a variable, further periods can be used after the variable name to index specific parts of the value. More on this in section 8.4 Subindexing.

A *variable entity reference*, or often just *entity* for short, is when the value of a variable is inserted into attributes or contents of XML elements using the XML entity reference syntax. For example, to insert the value of the variable var.name, write `&var.name;`. This works for all XML elements, regardless whether they are RXML tags that get evaluated or ordinary XML/HTML elements that are otherwise sent as-is to the client.

Here are a couple small examples showing the use of RXML variables:

RXML	Result

`<set variable="var.foo" value="bar"/>`
`&var.foo;`
	"`bar`"

`<set variable="var" scope="form"`
`value="bar"/>`
`&form.bar;`
	"`bar`"

`<if variable="var.foo = bar">gazonk</if>`	"`gazonk`" if var.foo is equal to "bar".
`<if match="&var.foo; = *test">gazonk</if>`	"`gazonk`" if the string in var.foo ends with "test".

Variable are grouped into scopes, and each scope typically covers a specific source. There is e.g. one scope *client* for information about the browser client, and another scope *page* with info about the current page on the server. You can get a list of variables belonging to a scope by using this RXML snippet:

```
<pre>
  <insert variables="full" scope="page"/>
</pre>
```

6.1 Quoting variable references

Since periods have special meaning in variable references (be it variable entity references or not), a quoting rule is necessary to access variables whose names contain periods. This is done by writing each period in a name as two periods after each other.

For example, suppose you have a graphical submit button like this:

```
<input type="image" name="button" src="button.gif"/>
```

Here the browser will submit two form variables with the names `button.x` and `button.y`. You can access those variables as follows:

```
<if variable="form.button..x">
  You pressed on the coordinate &form.button..x;,&form.button..y;.
</if>
```

6.2 Scopes

The most common scopes that handle variables are the *var* and *form* scopes. The *var* scope is always empty when the page parsing begins and is intended for internal variables that you need in your RXML code. The *form* scope contains all returned query variables from forms etc.

The other standard scopes are:

client	Information about the browser, e.g. the User-Agent string.
cookie	All cookies sent by the client.
page	Information about the page being RXML evaluated, e.g. its path.
request-header	All request headers sent by the client.
roxen	Information about the Roxen server.
user	Information about the authenticated user. This scope is only available in Roxen CMS.

See the web manual for details about these scopes.

Some RXML tags also define scopes that only exist inside them. Most important is the <emit> tag, which is used to iterate over information from some source.

Usually these so-called *tag scopes* get the same name as the emit source. E.g. the *sql* source, which lets you query an SQL database and process results from it, defines by default a scope `sql` which contains the values retrieved from the database. If several surrounding tags define scopes with the same name, then only the innermost scope with that name is accessible.

Usually there is a way to change the name of a tag scope, to allow you to access scopes that would otherwise be hidden. The <emit> tag takes a scope="..." attribute for that purpose. You can also always refer to the innermost, i.e. *current*, tag scope with _ (underscore). Here is an example that shows both scope renaming and the use of _ to access the current scope:

```
<emit source="sql" query=" ... " scope="outer">
 &_.name; is the same as &outer.name;
  <emit source="sql" query=" ... " scope="inner">
   &_.name; is the same as &inner.name; but different from &outer.name;
  </emit>
</emit>
```

6.3 Attribute splicing

If you want to set arbitrary attributes on XML elements, you can use the special *splice attribute* `::`. The value of that attribute is inserted directly into the attribute list, and it should therefore contain a sequence of `*attribute*="*value*"` pairs. An example:

```
<set variable="var.form-attrs">
  method="POST" enctype="multipart/form-data"
</set>
<form action="query.xml" ::="&var.form-attrs;">...</form>
```
Page source:
```
<form action="query.xml" method="POST" enctype="multipart/form-
data">...</form>
```

The reason why the splice attribute is necessary here is that XML entity references cannot be used directly in an attribute context. I.e. this is invalid XML:

```
<form action="query.xml" &var.form-attrs;>...</form>
```

The splice attribute works both for RXML evaluated tags and other tags that are sent directly to the client. Note that it does introduce a certain amount of overhead in compiled RXML code, so use it only when necessary.

7 RXML Variable Entity Encoding

When an RXML variable is accessed as an entity (e.g. `&var.foo;`) in an XML/HTML context, it is by default HTML encoded, i.e. < is inserted as `&lt;`, > as `&gt;` and `&` as `&amp;`. However, there are situations when that is not what you want, e.g. when inserting entities into SQL queries. Therefore, the encoding can be controlled by applying a different encoding scheme on the entity, `&scope.entity:`*`scheme`*`;`.

```
<sqlquery query="SELECT * FROM db WHERE name='&form.name:mysql;'">
```

It is also possible to combine several encoding schemes by separating them with `.` (a period). To e.g. UTF-8 encode and then URL encode with `%`*`XX`* escapes, write `&var.foo:utf8.url;`.

Here is a list of all available encoding schemes:

- `none`

  No quoting. This can potentially be dangerous if the value of the variable comes from an outside source in one way or the other. It should not be used unless you have total control of the content of the variable.

- `html`

  This is the default quoting, for inserting into regular HTML or XML, e.g. `&` is encoded to `&amp;`.

  Encoded characters: NUL `< > & " '`

- `http`

  HTTP encoding (i.e. using `%`*`XX`* style escapes) of characters that can never occur verbatim in URLs. Other URL-special chars, including `%`, `&` and `?`, are not encoded.

  Encoded characters are all control chars and additionally these:
  SPACE `: / ? # [ ] @ ! $ & ' ( ) * + , ; = " % < > \ ^ ` { | }`

  8-bit and wider chars are first UTF-8 encoded followed by `%`*`XX`* escaping, according to the IRI standard (see RFC 3987).

- `url`

  An extended variant of the `http` encoding scheme that encodes all URI reserved and excluded chars which otherwise could have special meaning; see RFC 3986. This includes characters such as `% / : " '`.

- `cookie`

  Nonstandard `http`-style encoding for cookie values. The Roxen HTTP protocol module automatically decodes incoming cookies using this encoding, so by using this for *Set-Cookie* headers etc you will get back the original value in the *cookie* scope. Note that the RXML <set-cookie> tag already does this encoding for you.

  Encoded characters are all control chars and additionally these:
  `= , ; %`

- `pike`

  Pike string quoting, for use in e.g. the <pike> tag. This means backslash escapes for chars that cannot occur verbatim in Pike string literals.

  Encoded characters: LF `\` `"`

- `js` or `javascript`

  Javascript string quoting using backslash escapes, for use in Javascript code.

  Encoded characters: BS HT FF CR LF `\` `"` `'`

- `mysql`

  MySQL string quoting using backslash escapes, for use in MySQL SQL queries.

  In Roxen 5.0 release 4 and later, the single quote character `'` is quoted as `''` according to standard SQL quoting rules. That is compatible with MySQL, and it avoids potential security problems with SQL injection if this encoding is incorrectly used with other databases.

  Encoded characters: `"` `'` `\`

- `sql` or `oracle`

  For inserting into SQL queries: The single quote character `'` is doubled to `''` (i.e. two single quote characters).

  WARNING: Do **not** use this encoding with MySQL as it might open up for security issues through SQL injection.

- `hex`

  Hexadecimal encoding. Requires octet (i.e. non-wide) strings.

```
<set variable="var.test" value="Hello World!"/>
&var.test:hex;
48656c6c6f20576f726c6421
```

- `utf8` or `utf-8`

  UTF-8 encoding.

- `utf16` or `utf16be`

  Big endian UTF-16 encoding.

```
<set variable="var.test" value="&#x1234;"/>
&var.test:utf16.hex;
1234
```

- `utf16le`

  Little endian UTF-16 encoding.

```
<set variable="var.test" value="&#x1234;"/>
&var.test:utf16le.hex;
3412
```

- `base64` or `base-64` or `b64`

  Base-64 MIME encoding. Requires octet (i.e. non-wide) strings.

- `quotedprintable` or `quoted-printable` or `qp`

  Quoted-Printable MIME encoding. Requires octet (i.e. non-wide) strings.

There are also some obsolete and deprecated encoding schemes, listed only for completeness:

- The empty string

  This means you write nothing after the colon, e.g. `&var.foo:;`. It works as an alias for `none`.

- `wml`

  Extended version of the `html` encoding that additionally encodes all `$` to `$$`.

- `wml-url`

  Alias for `url`.

- `dtag`
- `stag`

  The `dtag` scheme encodes a double quote `"` into the sequence `"'"'"`, and `stag` encodes a single quote `'` into `'"'"'`. (These were useful in attributes in RXML before Roxen 2.0.)

- `mysql-pike`

  Equivalent to the combination `mysql.pike`.

- `mysql-dtag`
- `sql-dtag`
- `oracle-dtag`

  These are equivalent to the combinations `mysql.dtag`, `sql.dtag` and `oracle.dtag`, respectively.

8 RXML Type System

RXML is usually used in XML (or HTML) documents to do things with simple strings or numeric values. In that case the underlying type system is hardly exposed, and you can usually cope quite well without a thorough understanding of it. There is however more power under the hood, and when you start doing more advanced things it is useful to get some concepts sorted out. That is what this chapter is about.

8.1 Context sensitivity

All RXML tags and variable values are *context sensitive* – where ever they occur there is a context which has a *type*, and the tags or variable values can potentially produce different results based on that type.

For example, the normal top level type in a page is `text/html`. When a variable with a string value is inserted through a variable entity reference, RXML converts the string to the `text/html` context by encoding the XML markup characters `< > & " '`. That is why a form like the following behaves well even if a user tries to play tricks by submitting something like "`'/><blink>I make you blink!`":

```
<form action='&page.self;'>
  <p>You sent: &form.text;</p>
  <input type='string' name='text'
         value='&form.text;'/>
  <input type='submit' name='send'/>
</form>
```
Page source:
```
<form action='myform.html'>
  <p>You sent: &#39;/&gt;&lt;blink&gt;I make you blink!</p>
  <input type='string' name='text'
         value='&#39;/&gt;&lt;blink&gt;I make you blink!'/>
  <input type='submit' name='send'/>
</form>
```

If the context was not `text/html` or `text/xml` then that conversion would not happen. E.g. the <set> tag uses the most generic type `any` for its content. That type accepts any value without conversion, which means the value gets assigned to the variable verbatim, regardless of what it is. In the following snippet we set var.y to a five char string with the tricky characters `< > & " '`, without any encoding occurring:

```
<set variable='var.x'><![CDATA[<>&"']]></set>
<set variable='var.y'>&var.x;</set>
<debug showvar='var.y'/>
<if sizeof='var.y = 5'>It is 5 chars.</if>
"<>&\"'"
It is 5 chars.
```

Here we use <debug showvar> to take a peek on what the variable "really is". It prints the variable using Pike notation, which means the string is printed in double quotes with Pike/C/C++/Java/Javascript style backslash escapes. Then the result is XML encoded, so `"<&>\"'"` is what you actually see in your browser. This way of showing the value usually makes it very clear what it actually contains. Still, just for good measure we also use an <if> test to check that the string really is five chars long.

8.1.1 Type propagation

Many RXML tags propagate the results of their contents unmodified to their surrounding contexts, e.g. <if>, <else> and <emit>. All such tags also first propagate the types of their surrounding contexts inwards to their own contents. That way you can use them in any context without hassle. In the following example, the <if> inside the <set> takes the `any` type from the <set> tag and propagates it to its contents, so `&var.x;` is expanded in an `any` context, just like in the last example in the previous section:

```
<set variable='var.x'><![CDATA[<>&"']]></set>
<set variable='var.y'>
  <if sizeof='var.x = 5'>
    &var.x;
  </if>
</set>
<debug showvar='var.y'/>
"<>&\"'"
```

Note in the example above that there is plenty of whitespace in the <set> and <if> tags to make the code nicely indented, yet it has no effect on the value assigned to var.y. That is another aspect of the typed contexts: Certain types, like `any`, ignore most literal whitespace and comments, while others, e.g. `text/xml` and `text/html`, does not. See section 8.3.1 for further details.

8.1.2 Tags without results

There are also some RXML tags that produce no results at all, e.g. the <set> and <nooutput> tags. Such tags have the result type `nil`, which allows them to occur in contexts of any type.

8.2 Sequential and non-sequential types

A fundamental difference between types is whether they are *sequential* or not. Basically, a sequential type is one where it makes sense to concatenate several values together. Any sort of string type is sequential – that allows RXML to paste together the many RXML tag results and the literal pieces of the page to send the completed page as a single string to the client. The two container types `array` and `mapping` are also sequential; see section 8.7.

Other types are not sequential. For e.g. integers it does not make much sense to concatenate the two numbers 1 and 2 together to 12 – although possible it is simply not useful[1].

8.2.1 Non-sequential type any

The generic type `any` is not sequential since it must allow values of any type, both sequential and not. That is why the <set> tag gives the following error which most RXML coders come across sooner or later:

```
<set variable='var.user'>Ann</set>
<set variable='var.greeting'>Hello, &var.user;</set>
&var.greeting;
RXML parse error: Cannot append another value "Ann" to non-sequential type
any.
 | <set variable="var.greeting">
```

The first <set> tag works because it only contains a single literal string "Ann". In the second <set> tag however, the result first gets the value "Hello,", which is converted to the type `any`. Then it

[1] Implementing concatenation as some other operation, such as addition, is not useful either. Intuitively it would just be obscure, but in more strict mathematical terms it would make the parsing process non-homomorphic – see the doc comment for RXML.Type.sequential in the WebServer sources for further details.

complains because the **any** type cannot append the second value "Ann". The solution is to tell the <set> tag to use a sequential type which specifies how the concatenation should be done:

```
<set variable='var.user'>Ann</set>
<set variable='var.greeting' type='text/*'>Hello, &var.user;</set>
&var.greeting;
Hello, Ann
```

8.2.2 Sequential type array

The **array** type is sequential, and its values can be anything. It can therefore be thought of as a sequential counterpart to **any**, which collects the values in a vector/array:

```
<set variable='var.user'>Ann</set>
<set variable='var.greeting' type='array'>Hello, &var.user;</set>
<debug showvar='var.greeting'/>
({ /* 2 elements */
    "Hello,",
    "Ann"
})
```

In the example above, we can see that the two values "Hello," and "Ann" were collected as two separate elements in an array. The <debug> tag then printed it out using Pike notation, i.e. inside `({` and `})`, and with the elements separated by commas (the comment "`/* 2 elements */`", the line breaks, and the indentation are just for readability and are not significant).

8.3 Parsing rules

Types differ in their handling of whitespace (including XML comments), plain text literals, and unparsed (i.e. non-RXML) tags and processing instructions.

8.3.1 Ignoring comments and whitespace

For so-called *free text* types, i.e. **text/html**, **text/plain** and all the others starting with "text/", all whitespace, comments, and literal text is significant:

```
<set variable='var.greeting' type='text/*'>
  <!-- Show the greeting -->
  <if variable='var.user'>
    Hello there, &var.user;!
  </if>
</set>
<debug showvar='var.greeting'/>
"\n"
"  <!-- Show the greeting -->\n"
"  \n"
"    Hello there, Ann!\n"
"  \n"
```

Notice the leftovers from the <if> tag indentation etc. Since **text/html** is the normal top level type, this is the behavior you will usually get.

When you are programming it is however often convenient to use whitespace and comments in your code without affecting the output, so you want the parser to ignore such things. All other types do that. E.g. if we switch to **string** in the example above, the result gets much shorter:

```
<set variable='var.greeting' type='string'>
  <!-- Show the greeting -->
  <if variable='var.user'>
```

```
    Hello there, &var.user;!
  </if>
</set>
<debug showvar='var.greeting'/>
"Hello there,Ann!"
```

The comment and all separating whitespace is gone. For literal text, all the leading and trailing whitespace is trimmed away, but internal whitespace is kept (e.g. the space in "Hello there,").

Notice how the whitespace before `&var.user;` is trimmed away too. That is because it is a variable entity which separate two text nodes, "`\n    Hello there, `" and "!`\n  `", each of which is trimmed separately. In cases like that it is probably more convenient to use a free text type to keep such whitespace separation intact, and you can use the <value> tag to switch to **text/*** only for that bit:

```
<set variable='var.greeting'>
  <!-- Show the greeting -->
  <if variable='var.user'>
    <value type='text/*'>Hello there, &var.user;!</value>
  </if>
</set>
<debug showvar='var.greeting'/>
"Hello there, Ann!"
```

Due to the <value> tag, there is no longer any need to specify a type for the <set> tag; the default **any** type will do just fine (unless you later find that you want to concatenate several values – see section 8.2.1).

8.3.2 Literal text

Apart from the free text types, there are several types that accept plain text literals, in particular the **string** type which is shown in several examples in the previous section.

The **any** type accepts arbitrary text literals just like **string** does. Other types, notably **int** and **float**, try to parse them as numbers or according to some other syntax – see the sections on the respective type for details.

The remaining types do not handle literals at all. For them you have to produce the values using other tags, e.g. from <emit> sources or by using <substring>, <set ... split="...">, or in particular the <value> tag:

```
<set variable='var.arr' type='array'>
  <value>abc</value>
  <value type='int'>17</value>
  <value type='float'>3.14</value>
</set>
<debug showvar='var.arr'/>
({ /* 3 elements */ "abc", 17, 3.14})
```

The example above also shows how **int** and **float** parse literals as numbers. They will throw parse errors if the literals do not conform to the numeric syntax.

8.3.3 Handling of unparsed tags and PI's

RXML usually works by detecting certain tags, specially formatted entity references, and a few processing instructions. The rest, i.e. from the RXML point of view *unparsed* tags and processing instructions, is let through untouched. This is also controlled by the types, and it is only the free text types (i.e. those beginning with "text/") that accept unparsed tags and PI's, although they do it in different ways – see section 8.5.

All other types require RXML parsed tags and PI's, and they must also produce results of the right types. This gives better error detection in case tags are misspelled or the right tag modules aren't loaded.

```
<set variable='var.greeting'>
  <if variable='var.user'>
    <p>Hello there, &var.user;!</p>
  </if>
</set>
<debug showvar='var.greeting'/>
RXML parse error: Unknown tag "p" is not allowed in context of type any.
 | <if variable="var.user">
 | <set variable="var.greeting">
```

If you want to write tags and PI's as ordinary text in a `string` or `any` context, you must quote them using `&lt;` or `<![CDATA[...]]>` blocks, but it is probably better to use a <value> tag to switch to a `text/*` context, like in the last example in section 8.3.1:

```
<set variable='var.greeting'>
  <if variable='var.user'>
    <value type='text/*'><p>Hello there, &var.user;!</p></value>
  </if>
</set>
<debug showvar='var.greeting'/>
"<p>Hello there, Ann!</p>"
```

8.4 Subindexing

For values of some types, it is possible to pick out smaller parts using subindexing. This is done by following the variable name with a period and the subindex, similar to how the scope is followed by a period and the variable name.

8.4.1 Indexing arrays

Subindexing is mainly used with the container types. For example, assume var.arr contains the `array` value ({"1", ({"2a", "2b"})}):

```
<set variable='var.arr' type='array'>
  <value>1</value>
  <value type='array'>
    <value>2a</value>
    <value>2b</value>
  </value>
</set>
First element: <debug showvar='var.arr.1'/>
Second element: <debug showvar='var.arr.2'/>
First element in the second element: <debug showvar='var.arr.2.1'/>
First element: "1"
Second element: ({ /* 2 elements */ "2a", "2b"})
First element in the second element: "2a"
```

Notice how subindexing can continue in more than one step. The *timerange* source to the <emit> tag uses this extensively with special "intelligent" values to do date and time arithmetic in a convenient way.

Use negative values to index from the end in an array. Given the same value in var.arr as above:

```
Last element: <debug showvar='var.arr.-1'/>
Second last element: <debug showvar='var.arr.-2'/>
Last element: ({ /* 2 elements */ "2a", "2b"})
Second last element: "1"
```

It is an error to index beyond the end of an array, or beyond the beginning when using negative indices.

8.4.2 Indexing scalar values

Scalar values (i.e. strings, text, integers and floating point numbers) cannot be indexed[2]. Even so, as a convenience they produce themselves when indexed with either 1 or -1. This is useful in cases where you do not know and do not really care if a variable is a sequence of values or just a single value, typically when handling form variables that might have been sent multiple times.

For example, if your page contains a variable entity as follows and the request is `http://myserver.com/sizes/?size=12`, then it works fine:

```
Size: &form.size;
Size: 12
```

But if the request becomes `http://myserver.com/sizes/?size=12&size=0` for some reason, then the value will be an array ({"12", "0"}) and you get an odd value with a NUL inside it (due to special magic in the *form* scope – see its reference in the web-based manual for details):

```
Size: &form.size;
Size: 12�0
```

In all scopes except the form scope, you will likely get an error instead, since an array cannot be inserted directly into the page.

If you instead use a subindex 1, you will in both cases get a single value from the form variable:

```
Size: &form.size.1;
Size: 12
```

8.4.3 Indexing scopes and mappings

Mappings and scopes are indexed in much the same way. Indices can be either numbers or strings. If the index does not match the name of any member then you will get an undefined value (`RXML.nil` – c.f. section 8.5.2), which in most contexts is transformed to an empty value that does not affect the result at all. If you continue to subindex the undefined value, you will get an error (unless you index it with 1 or -1; see the previous section).

8.5 Special types and values

8.5.1 any

The `any` type is a completely unspecified non-sequential type. Every type is a subtype of this one.

This type is also special in that any value can be converted to and from this type without the value getting changed in any way (provided it's representable in the target type), which means that the meaning of a value might change when this type is used as a middle step.

[2] To pick out characters or pieces of strings, use a string manipulation tag, such as the <substring> tag.

E.g. if "`<foo>`" of type **`text/*`** is converted directly to **`text/xml`**, it is quoted to "`&lt;foo&gt;`" since **`text/*`** always is literal text. However, if it first is converted to **`any`** and then to **`text/xml`**, it still remains "`<foo>`", which then carries a totally different meaning.

8.5.2 nil

There is both a type **`nil`** and a nil value, which most often shows up as `RXML.nil` in error messages and similar.

The value `RXML.nil` is a bit of a contradiction since it means no value at all, i.e. it is used as a marker to indicate the lack of a value. E.g. assigning `RXML.nil` to a variable (only possible from Pike) is the same as removing the variable binding.

Correspondingly, the type **`nil`** is a type that accepts no value at all (not even the empty value of some type). This type is a subtype of every other type since all the RXML evaluation functions may return no value (i.e. `RXML.nil`) regardless of the expected type.

8.5.3 The empty value

RXML has a special object that represent the empty value of any type which has such a thing, i.e. "" for strings and text, 0 for integers, an array with zero elements for arrays, etc. It is normally not visible from RXML, but it might show up in error messages etc as `RXML.empty`.

8.6 String types

All string types are sequential. They are used both for text and binary data.

When strings contain text, they will ideally contain Unicode strings. That is an abstract representation, without any charset or transmission encoding. I.e. you can view strings as a sequence of Unicode code points, and each character can be up to 32 bits wide. File system modules, RXML tags and the http protocol module in WebServer try to handle the charset issues for you so that the RXML evaluation takes place in the Unicode domain. But you can still use other charsets and control charset aspects in a variety of ways, e.g. with the <charset> and <recode> tags.

8.6.1 String values

In RXML, all string values are using the same string type internally. In other words, when you have a string in a variable then the type is not carried with it.

E.g. in the following case, you might think that you declare var.greeting to contain a **`text/html`** string, and hence it should not be quoted when inserted into the **`text/html`** context of the page.

```
<set variable='var.greeting' type='text/html'>
  <h1>Hello, my friend!</h1>
</set>
&var.greeting;
```
Page source:
```
&lt;h1&gt;Hello, my friend!&lt;/h1&gt;
```

Alas, that is not what happens. What the `type='text/html'` attribute actually did was only to tell the <set> tag that the content is **`text/html`**. After parsing it according to that type, the resulting string value is assigned to var.greeting. Then it is just a string like any other, and RXML will by default encode it safely before inserting it into the **`text/html`** context. You can use the <value> tag to type the value when you use it:

```
<set variable='var.greeting' type='text/html'>
  <h1>Hello, my friend!</h1>
```

```
</set>
<value type='text/html' from="var.greeting"/>
```
Page source:
```
<h1>Hello, my friend!</h1>
```

Now RXML is aware that a **text/html** value is being inserted, and skips the encoding. Of course, you can just as well override the default encoding with `&var.greeting:none;`.

8.6.2 string

This is the generic type for strings. As opposed to the text types in the two following sections, **string** does not allow free text, only literals; see section 8.3.

String conversions to and from this type works just like for **text/*** – see the next section for further details.

8.6.3 text/plain and text/*

These types are for plain or unspecified text. They are free text types, which means that all characters are significant, including whitespace and comments; see section 8.3.1 for details.

XML markup for quoting in the literal content, i.e. known character entity references such as `&lt;`, `&amp;` and `&ouml;` and `<![CDATA[...]]>` blocks, are converted to their unquoted values for these types. Other XML markup such as comments, tags, processing instructions, and entity references are not significant and hence parsed as any other text. That means you lose information in the parsing step. E.g. in the following example, there is no way to tell the difference between the <br/> tag and the plain string after it when the value has been parsed as text:

```
<set variable='var.text' type='text/plain'><br/>&lt;br/&gt;</set>
<debug showvar='var.text'/>
"<br/><br/>"
```

text/plain is intended to be used for text that is actually known to be plain text, while **text/*** is intended for all other text when the internal format is unknown.

In more concrete terms, the difference between the lies in how conversions to/from the other text types are done:

From	To	Result
text/plain	**text/xml** or **text/html**	XML markup chars like `<` `>` `&` get quoted.
text/xml or **text/html**	**text/plain**	XML quoting like `&amp;`, `&lt;` and `<![CDATA[...]]>` blocks get decoded.
text/* or **string**	any type	No change of the value.
any type	**text/*** or **string**	No change of the value.

What this means is that you get encoding/decoding as can be expected for the **text/plain** type, whereas **text/*** is similar to the **any** type in that the value doesn't change, which means that its meaning might change (see 8.5.1 for more details and an example).

8.6.4 text/xml and text/html

These two types are for XML data. On the RXML level there is no difference between them. `text/html` is the default top level type in the RXML parser.

8.7 Numeric types

The two numeric types `int` and `float` are non-sequential. There is also a `number` type which is a supertype for both. It is mostly useful when writing RXML tags, for attributes that accept both kinds of numbers.

8.7.1 int

The type for integers. RXML, like Pike, handles arbitrarily large integers.

When integers are parsed from string literals, they must be a string of decimal digits, optionally prepended by a minus sign.

8.7.2 float

The type for floating point numbers. The precision of these depend on the architecture (and – for advanced users – how Pike has been compiled); they are 32 bits long on 32 bit architectures, and 64 bits on 64 bit computers.

Floating point numbers are parsed and printed on this format:

[+/-]*nnnn*.*fffff*[e/E][+/-]*eee*

where *nnnn* is the integer part, *fffff* the fraction, and *eee* the exponent, all on decimal form.

8.8 Container types

Container types can hold any number of values, where each value can be of any type. As opposed to the string and numeric types, these cannot be directly inserted into text contexts – you must use subindexing to pick out specific parts in that case (see section 8.4).

The <value> tag is useful to create values of these types; see the sections below for examples. Some tags return arrays or mappings when they are used in a non-string context. For example, the <substring> tag can return an array of substrings:

```
<set variable='var.list' type='array'>
  <substring separator=',' trimwhites=''>
    a, , b:c, d::e: f
  </substring>
</set>
<debug showvar='var.list'/>
({ /* 4 elements */ "a", "", "b:c", "d::e: f"})
```

Another example is <insert source="variables"> which returns a scope as a mapping if it is used in a non-string context. That can be useful to collect results from an <emit> into an array of mappings:

```
<set variable='var.res' type='array'>
  <emit source="dir" directory=".">
    <insert source="variables" scope="_"/>
  </emit>
</set>
<debug showvar='var.res'/>
({ /* 5 elements */
    ([/* 24 elements */
      "atime": "2008-10-26",
      "atime-iso": "2008-10-26",
      "atime-unix": 1225013411,
      "counter": 1,
      "dirname": "/",
      "filename": "test.html",
      …
    ]),
    ([ /* 24 elements */
      "atime": "2008-04-26",
      …
    ]),
    …
})
```

8.8.1 array

An array is an ordered sequence of zero or more elements. They naturally occur for variables in the *form* scope since a form variable might occur more than one time and hence might get more than one value.

When arrays are printed for debugging or in error messages, they are shown in Pike syntax. That means a starting `({` followed by the elements in ascending order separated by `,` and then `})` at the end.

Arrays are usually built using the <value> tag, e.g.:

```
<set variable='var.arr' type='array'>
  <value>apple</value>
  <value>orange</value>
  <value>banana</value>
</set>
<debug showvar='var.arr'/>
({ /* 3 elements */ "apple", "orange", "banana"})
```

(The C-style comment `/* 3 elements */` is something that the <debug> tag adds for convenience. It is not significant.)

The <value> tag takes a *type* attribute to set the type of its content, just like e.g. <set>. That can be useful to build arrays inside arrays:

```
<set variable='var.arr' type='array'>
  <value>1</value>
  <value type='array'>
    <value>1.1</value>
    <value>1.2</value>
  </value>
  <value>2</value>
</set>
<debug showvar='var.arr'/>
({ /* 3 elements */ "1", ({ /* 2 elements */ "1.1", "1.2"}), "2"})
```

Note that a variable with an array value is normally spliced into an array context. Here too an extra <value> tag is useful to make it a nested array:

```
<set variable='var.x' split=','>a,b</set> <!-- ({"a", "b"}) -->
<set variable='var.arr' type="array">
  <!-- Insert all the elements in var.x -->
  &var.x;
  <!-- Compare to the following that adds the var.x array as a
       single element. -->
  <value>&var.x;</value>
</set>
<debug showvar='var.arr'/>
({ /* 3 elements */ "a", "b", ({ /* 2 elements */ "a", "b"}) })
```

8.8.2 mapping

A mapping is a hash table that contains a set of entries where each maps an index to a value. Both indices and values may be of any type, but indices are usually strings.

Like arrays, mappings are shown in Pike syntax in error messages etc. The format is:

`([`*index*$_1$`:` *value*$_1$`,` *index*$_2$`:` *value*$_2$`,` …`])`

where each *index*$_n$ is mapped to the following *value*$_n$.

Mappings can be created with the <value> tag by using the *index* attribute:

```
<set variable='var.map' type='mapping'>
  <value index='1'>first</value>
  <value index='2'>second</value>
  <value index='3'>third</value>
</set>
<debug showvar='var.map'/>
([ /* 3 elements */ "1": "first", "2": "second", "3": "third"])
```