civet

Description

Civet is an XML-based templating system. It is intended primarily for generating dynamic web pages in XHTML, but may be useful for other XML applications.

Authors

[REDACTED]

Requirements

utf8, uri-common, ssax, sxpath, sxml-serializer

Introduction

Civet was created as a templating engine for the soon-to-be-released **Coq au Vin** blogging library. The name comes from CVT, which is an abbreviation for 'Coq au Vin Templates'.

Civet supports insertion of dynamic content via variables, common control structures such as **if** and **for**, and has a simple inheritance mechanism. It does not, and probably will never, support the inclusion of arbitrary code in Scheme or any other language.

All Civet templates must be well-formed XML. Each template consists of markup from the target document type (e.g., XHTML), which represents literal output, combined with markup from the Civet template vocabulary, which inserts dynamic content and controls the structure of the output. For the sake of brevity, these two types of markup may be referred to as 'literal markup' and 'template markup' respectively.

A **template set** consists of a **base template** and one or more **extension templates**. The base template determines the document type of the output and its overall structure. As of language version 0.2, the content of extension templates is restricted to metadata contained in a **head** element, and content contained in one or more **blocks**. Any content outside of these structures in an extension template will be ignored.

Extension templates are used to specialize various aspects of the base template. They may also be chained so as to specialize other extension templates. Each extension template is associated with its **parent** (i.e., the template that it extends) by means of the **`extends`** attribute on the document element.

XML namespaces are used to distinguish the template vocabulary from literal content. For best results, we suggest that the default namespace be that of the target document type, while template vocabulary markup should use a prefix. The namespace URI for the template vocabulary is

```
http://xmlns.therebetygers.net/civet/<version_number>
```

For best results, please ensure that all templates in your installation use the same prefix for this namespace. Here at Coq au Vin World Headquarters, we use the prefix **`cvt`**.

Scheme API

PROCEDURES

[procedure] `(render TEMPLATE CONTEXT #!key (PORT #f) (FILE #f))`

This is the main function used to transform a template to some form of useful output (such as an XHTML web page). The TEMPLATE argument should be the filename of a template, including the extension but not including the directory path. CONTEXT is a context object (see **`make-context`** below for a description). There are two keyword arguments to specify an output destination: PORT is an output port, and FILE is a filename. If both are given, PORT takes precedence; if neither is given, the output will be returned as a string.

```
[procedure] (process-template-set TEMPLATE CONTEXT)
```

This function is similar to **`render`**, but returns an SXML document rather than rendering to XML.

```
[procedure] (process-base-template TEMPLATE BLOCK-DATA CONTEXT)
```

This function takes an SXML template, TEMPLATE, and an alist containing SXML blocks (which generally will have been read from extension templates, but could potentially be created programmatically), and a CONTEXT object, and returns a transformed SXML document. The BLOCK-DATA alist consists of '((NAME . (LOCALE VARS MACROS BLOCK)) ...), where NAME corresponds to the name of a block in the source template, LOCALE, VARS, and MACROS are, respectively, locale options, variables, and macros read from the template, and BLOCK is an SXML fragment consisting of a **`cvt:block`** element and its content.

```
[procedure] (load-template NAME #!optional (NSMAP '())
```

Loads a template from a file. **`NAME`** should be a filename including the extension but excluding the directory path. The optional NSMAP argument is an alist in the form **`'((PREFIX . NAMESPACE-URI) ...)`**, where **`PREFIX`** is a symbol (except in the case of a default namespace, in which case it should be **`#f`**. **`NSMAP`** overrides or extends the list of namespace bindings defined in **`*default-nsmap*`**.

A value of **`#f`** for **`NSMAP`** will cause the template to be parsed with no namespace bindings.

Note that when you load a template, a primitive form of caching is peformed behind the scenes. Specifically, whenever you load a new or modified template from XML, the resulting SXML document is saved to a file in the cache path (**`<site-path>/templates/.cache`** by default). On subsequent invocations of this procedure, if the cached SXML file exists and is newer, it will be loaded in place of the XML file. At present it is not possible to override this behavior.

```
[procedure] (build-template-set NAME #!optional (NSMAP '())
```

Given the NAME of an template file and an optional **`NSMAP`**, this procedure loads that template and any ancestors it references, and returns two values, the base template and an alist (referred to elsewhere as **`BLOCK-DATA`**) of the blocks extracted from any extension templates in the set.

```
[procedure] (make-context KWARGS)
```

Returns a context object: a closure that is used to maintain state information during the processing run. The following keywords are supported:

- **`vars`** alist, default = '()
- **`attrs`** alist, default = '()
- **`nsmap`** alist, default = (*default-nsmap*)

- **`locale`** alist, default = '()
- **`blocks`** alist, default = '()
- **`macros`** alist, default = '()
- **`state`** symbol, default = 'init

Directly manipulating the context object is not recommended. In general you should use **`context->context`** (see below). However, should you need to get or set any values, the closure responds to the following messages:

- **`(set-var! SYMBOL VALUE)`**
- **`(update-vars! ALIST)`**
- **`(set-vars! ALIST)`**
- **`(get-var SYMBOL)`**
- **`(get-vars)`**
- **`(get-field OBJ-NAME FIELD-NAME)`**
- **`(pfx->uri NAMESPACE-PREFIX)`**
- **`(uri->pfx NAMESPACE-URI)`**
- **`(set-ns! PREFIX URI)`**
- **`(update-nsmap! ALIST)`**
- **`(set-nsmap! ALIST)`**
- **`(get-nsmap)`**
- **`(set-attrs! ALIST)`**
- **`(set-attr! SYMBOL VALUE)`**
- **`(get-attrs)`**
- **`(delete-attrs!)`**
- **`(set-block! SYMBOL SXML-FRAGMENT)`**
- **`(get-block SYMBOL)`**
- **`(get-blocks)`**
- **`(set-macro! SYMBOL SXML-FRAGMENT)`**
- **`(get-macro SYMBOL)`**
- **`(get-macros)`**
- **`(set-locale! ALIST)`**
- **`(set-lang! LANG-CODE)`**
- **`(set-country! COUNTRY-CODE)`**
- **`(set-encoding! ENCODING-NAME)`**
- **`(set-date-format! DATE-FORMAT)`**
- **`(get-locale)`**
- **`(set-state! SYMBOL)`**
- **`(get-state)`**

[procedure] `(context->context CONTEXT KWARGS)`

Returns a new context object based on the existing one, with all its data copied from the original except as modified by the KWARGS. The following keyword arguments are supported.

- **`+vars`** Updates or sets one or more variables. Takes an alist.

- **+attrs** Updates or sets one or more attributes. Takes an alist.
- **+nsmap** Updates or sets one or more namespace bindings. Takes an alist.
- **+locale** Updates or sets one or more locale options. Takes an alist.
- **+blocks** Updates or sets one or more template blocks. Takes an alist.
- **+macros** Updates or sets one or more template macros. Takes an alist.
- **-vars** Unsets one or more variables. Takes a list of symbols.
- **-attrs** Unsets one or more attributes. Takes a list of symbols.
- **-nsmap** Unsets one or more namespace bindings. Takes a list of symbols.
- **-locale** Unsets one or more locale options. Takes a list of symbols.
- **-blocks** Unsets one or more template blocks. Takes a list of symbols.
- **-macros** Unsets one or more template macros. Takes a list of symbols.
- **state** Sets the state. Takes a symbol.

PARAMETERS

```
[parameter] *site-path*
```

The root directory of your web site. The template path is automatically calculated from this path unless you set ***template-path***. If the value of this parameter is **#f**, the processor assumes that the current working directory is the site path; likewise, if you set a relative path, that path is assumed to be relative to the current directory. Therefore it is a good idea to set this parameter to an absolute path.

Default: #f

```
[parameter] *template-path*
```

The directory where templates are stored. If the parameter is set to **#f**, the processor will look for templates in a **templates** subdirectory of the site path. Storing templates in subdirectories of the template path is not currently supported.

Default: #f

```
[parameter] *template-cache-path*
```

The directory where cached SXML templates are stored. If it is set to **#f**, cached templates will be searched for in a **.cache** subdirectory of the template path.

Default: #f

```
[parameter] *enable-l10n*
```

Enable localization? Currently has no effect.

Default: #f

```
[parameter] *civet-ns-prefix*
```

A symbol representing the prefix to be used for Civet vocabulary elements. Do not override this unless you are actually using a different prefix in your templates.

Default: `'cvt`

[parameter] `*civet-ns-uri*`

The namespace URI for Civet vocabulary elements. Overriding this is not recommended.

Default: `"http://xmlns.therebetygers.net/civet/0.2"`

[parameter] `*default-nsmap*`

The default namespace map to be used when loading XML templates.

Default:

```
`((#f . "http://www.w3.org/1999/xhtml")
  (,(*civet-ns-prefix*) . ,(*civet-ns-uri*)))))
```

[parameter] `*sxpath-nsmap*`

The default namespace map to be used for SXPath expressions and serialization.

Default:

```
`((*default* . "http://www.w3.org/1999/xhtml")
  (,(*civet-ns-prefix*) . ,(*civet-ns-uri*)))
```

[parameter] `*sort-functions*`

A mapping from data type symbols to sorting functions to be used in **`for`** loops. For each "built-in" data type there are two functions specified: the first is used for ascending sorts, the second for descending.

Default:

```
`((string . (,string<? ,string>?))
  (char . (,char<? ,char>?))
  (number . (,< ,>))
  (boolean . (,(lambda (a b) (or (not a) b)) ,(lambda (a b) (or a (not b))))))
```

Template Vocabulary, version 0.2

The following describes the complete vocabulary of elements and attributes represented by the **`civet`** namespace. At present there is no formal specification or schema for the language, and this document may be regarded as a normative reference. Please report any language in this document that you find ambiguous or insufficiently clear.

Also, each element description includes a **Contents** subsection, describing what child nodes are required or allowed. However, for all elements, unless otherwise noted, markup from the target vocabulary is permitted within any element of the template vocabulary, and comments and processing instructions are unrestricted.

NOTES:

- This document uses the **`cvt:`** namespace prefix in all element descriptions. Bear in mind that, as with all XML namespace prefixes, this is merely a convention (and is the default prefix supported by the Scheme library), and you may use a different prefix in your code and templates.
- While all elements described are handled by the processor and should not cause errors, some (e.g. locale) do not actually do anything.

cvt:template

This is the document element for an extension template. As of Version 0.2, a **`cvt:template`** element may only contain **`cvt:head`**, **`cvt:block`** elements. Any other content will be discarded by the processor.

A base template does **not** use **`cvt:template`**; rather, its document element should be the document element required by the target document type, e.g. **`html`**.

Context:

Occurs only as the document element of an extension template.

Content:

May contain, in the following order:

- **`cvt:head`** [optional]
- **`cvt:block`** [zero or more]

May not contain text nodes or any markup from the target vocabulary.

Attributes:

- **`extends`** [required] Contains a reference to the parent template, expressed as a system file path.

---

cvt:head

Contains elements that set variables and/or configure processing behavior.

Context:

First child of the document element of a template.

Contents:

May contain, in any order:

- **`cvt:locale`** [optional]
- **`cvt:defvar`** [zero or more]
- **`cvt:defmacro`** [zero or more]

---

cvt:locale

May be used to determine locale options within a template. Currently has no effect. I'm not sure if this element should be supported or not. Certainly, locale options are useful at the application level, but not necessarily within a template, so this element will probably be discontinued if it does not prove useful in the near future.

Context:

Within **`head`**.

Content:

Empty.

Attributes:

- **`lang`**
- **`country`**
- **`encoding`**
- **`date-format`**

---

cvt:defvar

Sets a variable's value within its local scope (i.e. for the template if contained in **`cvt:head`**, otherwise within the lexical scope of its parent element). The value may be specified either by a **`value`** attribute whose value is a literal string, or by the element content; if both are present, the attribute takes precedence.

Context:

Within a **`cvt:head`**, **`cvt:block`**, or **`cvt:with`**.

Content:

Any element other than **`cvt:template`**, **`cvt:head`**, or **`cvt:block`**

Attributes:

- **`name`** [required] The variable name to set.
- **`type`** [optional, default=auto] The value may be any known datatype, where *known* includes the built-in types and any types defined in the processing application. A value of **`auto`** means that the type will be **`node-set`** if this element contains any markup from the target vocabulary, otherwise **`string`**.
- **`value`** [optional] This attribute may be used instead of child nodes to specify the value, if the value consists of a single text node.

---

cvt:defmacro

Defines a macro. A macro has a name and contains XML nodes which are evaluated in the lexical scope where a macro is referenced using the **`cvt:macro`** element. Thus the macro may contain arbitrary template markup including references to variables that are unbound at macro definition time. Please note that as of library version 0.3.6, there are certain template markup elements that will not work as you may expect in a macro. These include **`cvt:attr`**, **`cvt:else`**, and **`cvt:interpolate`**. These elements are handled outside the normal processing flow; therefore, if you use one of them as the *child* of **`cvt:macro`**, there will be no result when the macro is evaluated. On the other hand, these elements *should* work as descendants of other elements within a macro definition, but this has not been systematically tested.

Context:

Within **`cvt:head`**.

Content:

Any element other than **`cvt:template`**, **`cvt:head`**, or **`cvt:block`**.

Attributes:

- **`name`** [required] The name of the macro. Must be unique within the template.

---

cvt:block

This element represents the basic unit of document structure, and may contain any type of content, or be empty. The order of blocks within the document (and of any interspersed content outside of blocks) is determined by the base template, and may not be altered by extension templates.

Each block has a required **`name`** attribute. Following the usual practice, its value must be unique within any given document. Furthermore, any block ID used in an extension template must match one defined in the base template of the set.

Nested blocks are not currently allowed, and will raise an error. Nesting may be supported in future versions if it is deemed a useful feature and can be implemented in a reasonable manner.

The following rules govern the relationships of corresponding blocks (i.e., those whose IDs match) among different templates in a set.

- *Defined with content in ancestor template, omitted in descendant:*

Output includes the content defined in the ancestor template.

- *Defined as empty in ancestor template, omitted in descendant template:*

No output.

- *Defined with content in ancestor template, defined as empty in descendant:*

No output

- *Defined with content in ancestor template, and with different content in descendant:*

Content defined in the descendant template replaces that defined in the ancestor template.

Attributes:

- **`name`** [required] An arbitrary identifier that must be unique at document level.

---

cvt:var

A placeholder for inserting dynamic content. Variables are generally passed by the processing application, but may also be defined within the template (see **`cvt:defvar`**). There is no mechanism for assigning a new value to an existing variable, but names may be reused in local bindings (e.g. a block- level definition).

Attributes:

- **`name`** [required] The identifier for the variable; in order to insert content, this value must be matched in the **`vars`** alist defined in the processing application, or by a variable defined within the template. If the name is a **qualified name**, indicated with dotted notation, the processor will retrieve the named field from the named object (i.e.: <object>.<field>)
- **`required`** [optional, default: true] Whether the variable is required to be defined. Any required variable that is undefined is an error.
- **`type`** [optional, default: string] A builtin or user-defined datatype. The builtin types are:
  - **`string`**
  - **`boolean`**
  - **`char`**
  - **`number`**
  - **`integer`**
  - **`float`**
  - **`list:<type>`**
  - **`object`**
  - **`node-list`**
- **`format`** [optional] The name of a formatting function defined in the civet library or in your application. This function is usually associated with a specific data type. As of version 0.3 of the library, **`format="uri"`** will cause the output to be uri-encoded. Future versions of the library will include additional options and provide a mechanism for user-defined formatters.

---

cvt:macro

References a macro defined with **`cvt:defmacro`**. All contents of the named macro will be evaluated in the lexical scope where this element is placed.

Attributes:

- **`name`** [required] The name of the macro.

---

cvt:if

The basic conditional structure. If the test specified by the **`test`** attribute returns true, all content of the **`cvt:if`** element is written to the output; otherwise it is ommitted. May contain an optional **`cvt:else`** element.

Context:

May be contained within any element.

Contents:

Any element except **`cvt:template`** and **`cvt:block`**.

Attributes:

- **`test`** [required] A boolean expression. See **`Expression Language`** below.

Expression Language:

The **`test`** attribute uses a simple expression language, including the following expressions:

<var-name> Returns #t if the variable is defined in the current context, and is neither false nor null, false otherwise.

<var-name> = <expr> Returns #t if the variable value is equal (using equal?) to the right-side expression. The right-side expression may be a (quoted) string or numeric literal, or another variable name.

<var-name> != <expr> Returns #t if the variable value is unequal to the right-side expression.

<function>(<var-name>, <expr>) Performs a numeric comparison between the named variable and the right-side expression. Four functions are supported:

- **`lt`** Less than
- **`gt`** Greater than
- **`le`** Less than or equal to
- **`ge`** Greater than or equal to

Whitespace is allowed but not required at the beginning and end of an expression, and between any two tokens.

---

cvt:else

The content of this element is output if the **`cvt:if`** test fails.

---

cvt:for

Iterate over a list variable.

Attributes:

- **`each`** [required] Defines the local variable name for each iteration.
- **`in`** [required] Equivalent to the name attribute for **`var`** and **`block`**.
- **`sort`** [optional, default=auto] The sorting method to use. May be 'alpha', 'numeric', 'auto', or 'none'. 'Auto' means that civet will select a sorting method based on the datatype of the variable. This may degrade performance and produce unexpected results, so it is best to specify the sorting method whenever possible. Future versions of the library may allow user-defined sorting functions.
- **`sort-field`** [optional] If the variable refers to a list of objects, this attribute specifies the field that should be used to sort the list.
- **`order`** [optional, default=asc] Possible values are 'asc' (ascending) and 'desc' (descending).

---

cvt:interpolate

Inserts text or markup between the iterations of a **`cvt:for`** loop. Any **`cvt:for`** element may have up to three **`cvt:interpolate`** children (with different **`mode`** attributes). The content of **`cvt:interpolate`**, when it appears (see the description of the **`mode`** attribute below, follows all content of the **`cvt:for`** loop.

Context:

Within a **`cvt:for`** element.

Content:

Text, or any element other than **`cvt:template`**, **`cvt:head`**, or **`cvt:block`**.

Attributes:

- **`mode`** [optional, default=default] Determines whether to insert the content, depending on the state of the loop. The following values are recognized:
  - **default:** Insert this content on every iteration of the loop except the last, unless overridden by another **`cvt:interpolate`** element with a different mode.
  - **last:** Insert this content after the next-to-last iteration. Overrides 'default'.
  - **pair:** When the loop variable has exactly two elements, insert this content between them. Overrides 'last' and 'default'.

---

cvt:with

A container for variable definitions.

Context:

Anywhere in a base template, or anywhere within a block in an extension template.

Content:

Should contain one or more <defvar> elements [otherwise the **`cvt:with`** element serves no purpose], followed by any other elements (except **`cvt:block`** and **`cvt:template`**).

Attributes:

None.

---

cvt:attr

Sets an attribute on its parent element. If the literal attribute is already defined on the parent, the value specified by this element overrides it. Otherwise, it adds a new attribute to the parent. The value may be specified by a variable reference specified by a **`var`** attribute, or by the element content. If both are present, the attribute takes precedence.

NOTE: As of library version 0.3.6, the **`type`** attribute does not work. This will be fixed in an upcoming release.

Context:

Any element from the target vocabulary.

Content:

Any combination of strings and template markup that evaluates to a single string.

Attributes:

- **`name`** [required]
- **`type`** [optional, default: string]
- **`var`** [optional] References a variable to supply the value.

Variable substitution in attributes

In addition to the attributes defined above for template vocabulary elements, any unprefixed attribute from the target vocabulary may have the **`civet`** namespace prefix applied to it. This indicates to the processor that the attribute's value is a variable reference; when encountering such an attribute, the processor will substitute the variable's value for the reference and remove the prefix.

For example, if your application defines a variable **`page-classes`** with the value "single-article blog-post", the following code in a template:

```
<body cvt:class="page-classes">
```

will produce the output:

```
<body class="single-article blog-post">
```

Note that this method permits only the substitution of a primitive data type with an obvious string representation. If you require any more complex manipulations, such as converting lists or objects to strings, or conditional processing, you must use the **`attr`** element.

If an element has an attribute prefixed in this manner, and a **`cvt:attr`** element child, the **`cvt:attr`** element overrides the prefixed attribute.

Examples

There is a small but growing collection of example templates at:

[REDACTED]

Template Processors

A civet processor must fulfill the following requirements:

- It must support the entire template vocabulary as described in this document.
- If any templates in the input set are ill-formed or fail to conform to the requirements of this document, processing must end with an error.
- If template processing successfully completes, the output will be well-formed XML, with all markup from the template vocabulary removed.

Any failure with respect to meeting these requirements is a bug, and may be reported as such. Please note, however, that since the template language deliberately allows templates to include arbitrary fragments composed of markup from non-civet vocabularies, it is impossible to guarantee the validity of the output document with respect to any XML schema.

In case of bugs

If you have a GitHub account, please use the GitHub issue tracker -- likewise for any technical questions or suggestions you may have (other than how-to type questions). If you are unable to do this, the chicken-users mailing list will also work.

License

Repo

[REDACTED]

Version History

0.3.6
Implemented **`var`** attribute on **`cvt:attr`**.

0.3.4
Added version to utf8 dependency.

0.3.3
Modified expression language to treat null values as false.

0.3.1
Fixed meta file.

0.3.0
Added interpolation in **`for`** loops, macros, and **`format="uri"`** for variable references.

0.2.1
Edited docs & converted to svnwiki format.

0.2
Fixed inconsistent nesting bug.

0.1.1
Improved documentation.

0.1
Initial release.