NAME

Template - Front-end module to the Template Toolkit

SYNOPSIS

```
use Template;

# some useful options (see below for full list)
my $config = {
    INCLUDE_PATH => '/search/path',  # or list ref
    INTERPOLATE  => 1,               # expand "$var" in plain text
    POST_CHOMP   => 1,               # cleanup whitespace
    PRE_PROCESS  => 'header',        # prefix each template
    EVAL_PERL    => 1,               # evaluate Perl code blocks
};

# create Template object
my $template = Template->new($config);

# define template variables for replacement
my $vars = {
    var1  => $value,
    var2  => \%hash,
    var3  => \@list,
    var4  => \&code,
    var5  => $object,
};

# specify input filename, or file handle, text reference, etc.
my $input = 'myfile.html';

# process input template, substituting variables
$template->process($input, $vars)
    || die $template->error();
```

DESCRIPTION

This documentation describes the Template module which is the direct Perl interface into the Template Toolkit. It covers the use of the module and gives a brief summary of configuration options and template directives. Please see *Template::Manual* for the complete reference manual which goes into much greater depth about the features and use of the Template Toolkit. The *Template::Tutorial* is also available as an introductory guide to using the Template Toolkit.

METHODS

new(\%config)

The `new()` constructor method (implemented by the *Template::Base* base class) instantiates a new `Template` object. A reference to a hash array of configuration items may be passed as a parameter.

```
my $tt = Template->new({
    INCLUDE_PATH => '/usr/local/templates',
    EVAL_PERL    => 1,
}) || die $Template::ERROR, "\n";
```

A reference to a new `Template` object is returned, or undef on error. In the latter case, the error message can be retrieved by calling *error()* as a class method or by examining the `$Template::ERROR` package variable directly.

```
my $tt = Template->new(\%config)
    || die Template->error(), "\n";

my $tt = Template->new(\%config)
    || die $Template::ERROR, "\n";
```

For convenience, configuration items may also be specified as a list of items instead of a hash array reference. These are automatically folded into a hash array by the constructor.

```
my $tt = Template->new(INCLUDE_PATH => '/tmp', POST_CHOMP => 1)
    || die $Template::ERROR, "\n";
```

process($template, \%vars, $output, %options)

The `process()` method is called to process a template. The first parameter indicates the input template as one of: a filename relative to `INCLUDE_PATH`, if defined; a reference to a text string containing the template text; or a file handle reference (e.g. `IO::Handle` or sub-class) or `GLOB` (e.g. `\*STDIN`), from which the template can be read. A reference to a hash array may be passed as the second parameter, containing definitions of template variables.

```
# filename
$tt->process('welcome.tt2')
    || die $tt->error(), "\n";

# text reference
$text = "[% INCLUDE header %]\nHello world!\n[% INCLUDE footer %]";
$tt->process(\$text)
    || die $tt->error(), "\n";

# file handle (GLOB)
$tt->process(\*DATA)
    || die $tt->error(), "\n";

__END__
[% INCLUDE header %]
This is a template defined in the __END__ section which is
accessible via the DATA "file handle".
[% INCLUDE footer %]
```

By default, the processed template output is printed to `STDOUT`. The `process()` method then returns `1` to indicate success. A third parameter may be passed to the `process()` method to specify a different output location. This value may be one of: a plain string indicating a filename which will be opened (relative to `OUTPUT_PATH`, if defined) and the output written to; a file GLOB opened ready for output; a reference to a scalar (e.g. a text string) to which output/error is appended; a reference to a subroutine which is called, passing the output as a parameter; or any object reference which implements a `print()` method (e.g. `IO::Handle`, `Apache::Request`, etc.) which will be called, passing the generated output as a parameter.

Examples:

```
# output filename
$tt->process('welcome.tt2', $vars, 'welcome.html')
    || die $tt->error(), "\n";

# reference to output subroutine
sub myout {
```

```
        my $output = shift;
        ...
    }
    $tt->process('welcome.tt2', $vars, \&myout)
        || die $tt->error(), "\n";

    # reference to output text string
    my $output = '';
    $tt->process('welcome.tt2', $vars, \$output)
        || die $tt->error(), "\n";

    print "output: $output\n";
```

In an Apache/mod_perl handler:

```
    sub handler {
        my $req = shift;

        # ...your code here...

        # direct output to Apache::Request via $req->print($output)
        $tt->process($file, $vars, $req) || do {
            $req->log_reason($tt->error());
            return SERVER_ERROR;
        };
        return OK;
    }
```

After the optional third output argument can come an optional reference to a hash or a list of `(name, value)` pairs providing further options for the output. The only option currently supported is `binmode` which, when set to any true value will ensure that files created (but not any existing file handles passed) will be set to binary mode.

```
    # either: hash reference of options
    $tt->process($infile, $vars, $outfile, { binmode => 1 })
        || die $tt->error(), "\n";

    # or: list of name, value pairs
    $tt->process($infile, $vars, $outfile, binmode => 1)
        || die $tt->error(), "\n";
```

Alternately, the `binmode` argument can specify a particular IO layer such as `:utf8`.

```
    $tt->process($infile, $vars, $outfile, binmode => ':utf8')
        || die $tt->error(), "\n";
```

The `OUTPUT` configuration item can be used to specify a default output location other than `\*STDOUT`. The `OUTPUT_PATH` specifies a directory which should be prefixed to all output locations specified as filenames.

```
    my $tt = Template->new({
        OUTPUT      => sub { ... },       # default
        OUTPUT_PATH => '/tmp',
    ...
    }) || die Template->error(), "\n";
```

```
    # use default OUTPUT (sub is called)
    $tt->process('welcome.tt2', $vars)
        || die $tt->error(), "\n";

    # write file to '/tmp/welcome.html'
    $tt->process('welcome.tt2', $vars, 'welcome.html')
        || die $tt->error(), "\n";
```

The `process()` method returns `1` on success or `undef` on error. The error message generated in the latter case can be retrieved by calling the *error()* method. See also *CONFIGURATION SUMMARY* which describes how error handling may be further customised.

error()

When called as a class method, it returns the value of the $ERROR package variable. Thus, the following are equivalent.

```
    my $tt = Template->new()
        || die Template->error(), "\n";

    my $tt = Template->new()
        || die $Template::ERROR, "\n";
```

When called as an object method, it returns the value of the internal _ERROR variable, as set by an error condition in a previous call to process().

```
    $tt->process('welcome.tt2')
        || die $tt->error(), "\n";
```

Errors are represented in the Template Toolkit by objects of the *Template::Exception* class. If the *process()* method returns a false value then the `error()` method can be called to return an object of this class. The *type()* and *info()* methods can called on the object to retrieve the error type and information string, respectively. The *as_string()* method can be called to return a string of the form `$type - $info`. This method is also overloaded onto the stringification operator allowing the object reference itself to be printed to return the formatted error string.

```
    $tt->process('somefile') || do {
        my $error = $tt->error();
        print "error type: ", $error->type(), "\n";
        print "error info: ", $error->info(), "\n";
        print $error, "\n";
    };
```

service()

The `Template` module delegates most of the effort of processing templates to an underlying *Template::Service* object. This method returns a reference to that object.

context()

The *Template::Service* module uses a core *Template::Context* object for runtime processing of templates. This method returns a reference to that object and is equivalent to `$template->service->context()`.

template($name)

This method is a simple wrapper around the *Template::Context* method of the same name. It returns a compiled template for the source provided as an argument.

CONFIGURATION SUMMARY

The following list gives a short summary of each Template Toolkit configuration option. See *Template::Manual::Config* for full details.

Template Style and Parsing Options

START_TAG, END_TAG

Define tokens that indicate start and end of directives (default: '`[%`' and '`%]`').

TAG_STYLE

Set `START_TAG` and `END_TAG` according to a pre-defined style (default: '`template`', as above).

PRE_CHOMP, POST_CHOMP

Removes whitespace before/after directives (default: 0/0).

TRIM

Remove leading and trailing whitespace from template output (default: 0).

INTERPOLATE

Interpolate variables embedded like `$this` or `${this}` (default: 0).

ANYCASE

Allow directive keywords in lower case (default: 0 - UPPER only).

Template Files and Blocks

INCLUDE_PATH

One or more directories to search for templates.

DELIMITER

Delimiter for separating paths in `INCLUDE_PATH` (default: '`:`').

ABSOLUTE

Allow absolute file names, e.g. `/foo/bar.html` (default: 0).

RELATIVE

Allow relative filenames, e.g. `../foo/bar.html` (default: 0).

DEFAULT

Default template to use when another not found.

BLOCKS

Hash array pre-defining template blocks.

AUTO_RESET

Enabled by default causing `BLOCK` definitions to be reset each time a template is processed. Disable to allow `BLOCK` definitions to persist.

RECURSION

Flag to permit recursion into templates (default: 0).

Template Variables

VARIABLES

Hash array of variables and values to pre-define in the stash.

Runtime Processing Options

EVAL_PERL

Flag to indicate if PERL/RAWPERL blocks should be processed (default: 0).

PRE_PROCESS, POST_PROCESS

Name of template(s) to process before/after main template.

PROCESS

Name of template(s) to process instead of main template.

ERROR

Name of error template or reference to hash array mapping error types to templates.

OUTPUT

Default output location or handler.

OUTPUT_PATH

Directory into which output files can be written.

DEBUG

Enable debugging messages.

Caching and Compiling Options

CACHE_SIZE

Maximum number of compiled templates to cache in memory (default: undef - cache all)

COMPILE_EXT

Filename extension for compiled template files (default: undef - don't compile).

COMPILE_DIR

Root of directory in which compiled template files should be written (default: undef - don't compile).

Plugins and Filters

PLUGINS

Reference to a hash array mapping plugin names to Perl packages.

PLUGIN_BASE

One or more base classes under which plugins may be found.

LOAD_PERL

Flag to indicate regular Perl modules should be loaded if a named plugin can't be found (default: 0).

FILTERS

Hash array mapping filter names to filter subroutines or factories.

Customisation and Extension

LOAD_TEMPLATES

List of template providers.

LOAD_PLUGINS

List of plugin providers.

LOAD_FILTERS

List of filter providers.

TOLERANT

Set providers to tolerate errors as declinations (default: 0).

SERVICE

Reference to a custom service object (default: *Template::Service*).

CONTEXT

Reference to a custom context object (default: *Template::Context*).

STASH

Reference to a custom stash object (default: *Template::Stash*).

PARSER

Reference to a custom parser object (default: *Template::Parser*).

GRAMMAR

Reference to a custom grammar object (default: *Template::Grammar*).

DIRECTIVE SUMMARY

The following list gives a short summary of each Template Toolkit directive. See *Template::Manual::Directives* for full details.

GET

Evaluate and print a variable or value.

```
[%   GET variable %]    # 'GET' keyword is optional
[%       variable %]
[%       hash.key %]
[%         list.n %]
[%     code(args) %]
[% obj.meth(args) %]
[%  "value: $var" %]
```

CALL

As per *GET* but without printing result (e.g. call code)

```
[%  CALL variable %]
```

SET

Assign a values to variables.

```
[% SET variable = value %]    # 'SET' also optional
[%     variable = other_variable
       variable = 'literal text @ $100'
       variable = "interpolated text: $var"
       list     = [ val, val, val, val, ... ]
       list     = [ val..val ]
       hash     = { var => val, var => val, ... }
%]
```

DEFAULT

Like *SET*, but variables are only set if currently unset (i.e. have no true value).

```
[% DEFAULT variable = value %]
```

INSERT

Insert a file without any processing performed on the contents.

```
[% INSERT legalese.txt %]
```

PROCESS

Process another template file or block and insert the generated output. Any template *BLOCK*s or variables defined or updated in the PROCESSed template will thereafter be defined in the calling template.

```
[% PROCESS template %]
[% PROCESS template  var = val, ... %]
```

INCLUDE

Similar to PROCESS, but using a local copy of the current variables. Any template BLOCKs or variables defined in the INCLUDEd template remain local to it.

```
[% INCLUDE template %]
[% INCLUDE template  var = val, ... %]
```

WRAPPER

The content between the WRAPPER and correspondng END directives is first evaluated, with the output generated being stored in the `content` variable. The named template is then process as per INCLUDE.

```
[% WRAPPER layout %]
   Some template markup [% blah %]...
[% END %]
```

A simple *layout* template might look something like this:

```
Your header here...
[% content %]
Your footer here...
```

BLOCK

Define a named template block for *INCLUDE*, *PROCESS* and *WRAPPER* to use.

```
[% BLOCK hello %]
   Hello World
[% END %]

[% INCLUDE hello %]
```

FOREACH

Repeat the enclosed FOREACH ... END block for each value in the list.

```
[% FOREACH variable IN [ val, val, val ] %]    # either
[% FOREACH variable IN list %]                 # or
   The variable is set to [% variable %]
[% END %]
```

WHILE

The block enclosed between `WHILE` and `END` block is processed while the specified condition is true.

```
[% WHILE condition %]
   content
[% END %]
```

IF / UNLESS / ELSIF / ELSE

The enclosed block is processed if the condition is true / false.

```
[% IF condition %]
   content
[% ELSIF condition %]
 content
[% ELSE %]
 content
[% END %]

[% UNLESS condition %]
   content
[% # ELSIF/ELSE as per IF, above %]
   content
[% END %]
```

SWITCH / CASE

Multi-way switch/case statement.

```
[% SWITCH variable %]
[%   CASE val1 %]
       content
[%   CASE [ val2, val3 ] %]
       content
[%   CASE %]         # or [% CASE DEFAULT %]
       content
[% END %]
```

MACRO

Define a named macro.

```
[% MACRO name <directive> %]
[% MACRO name(arg1, arg2) <directive> %]
...
[% name %]
[% name(val1, val2) %]
```

FILTER

Process enclosed `FILTER` ... `END` block then pipe through a filter.

```
[% FILTER name %]                       # either
[% FILTER name( params ) %]             # or
[% FILTER alias = name( params ) %]     # or
   content
[% END %]
```

USE

Load a plugin module (see Template::<Manual::Plugins), or any regular Perl module when the LOAD_PERL option is set.

```
[% USE name %]                      # either
[% USE name( params ) %]            # or
[% USE var = name( params ) %]      # or
...
[% name.method %]
[% var.method %]
```

PERL / RAWPERL

Evaluate enclosed blocks as Perl code (requires the EVAL_PERL option to be set).

```
[% PERL %]
 # perl code goes here
 $stash->set('foo', 10);
 print "set 'foo' to ", $stash->get('foo'), "\n";
 print $context->include('footer', { var => $val });
[% END %]


[% RAWPERL %]
   # raw perl code goes here, no magic but fast.
   $output .= 'some output';
[% END %]
```

TRY / THROW / CATCH / FINAL

Exception handling.

```
[% TRY %]
 content
   [% THROW type info %]
[% CATCH type %]
 catch content
   [% error.type %] [% error.info %]
[% CATCH %] # or [% CATCH DEFAULT %]
 content
[% FINAL %]
   this block is always processed
[% END %]
```

NEXT

Jump straight to the next item in a FOREACH or WHILE loop.

```
[% NEXT %]
```

LAST

Break out of FOREACH or WHILE loop.

```
[% LAST %]
```

RETURN

Stop processing current template and return to including templates.

```
[% RETURN %]
```

STOP

Stop processing all templates and return to caller.

```
[% STOP %]
```

TAGS

Define new tag style or characters (default: `[% %]`).

```
[% TAGS html %]
[% TAGS <!-- --> %]
```

COMMENTS

Ignored and deleted.

```
[% # this is a comment to the end of line
   foo = 'bar'
%]

[%# placing the '#' immediately inside the directive
    tag comments out the entire directive
%]
```

SOURCE CODE REPOSITORY

The source code for the Template Toolkit is held in a public git repository on Github: *https://github.com/abw/Template2*

AUTHOR

Andy Wardley <abw@wardley.org> *http://wardley.org/*

VERSION

Template Toolkit version 2.23, released January 2012.

COPYRIGHT