Written by Arnaud Bouchez

SynMustache
and the Mustache Logic-less Template Engine

7.3.2. Web clients

7.3.2.1. Mustache template engine

Mustache - see http://mustache.github.io.. - is a well-known *logic-less* template engine.
There is plenty of Open Source implementations around (including in JavaScript, which can be very convenient for AJAX applications on client side, for instance). For *mORMot*, we created the first pure Delphi implementation of it, with a perfect integration with other bricks of the framework.

Generally speaking, a Template system can be used to separate output formatting specifications, which govern the appearance and location of output text and data elements, from the executable logic which prepares the data and makes decisions about what appears in the output.

Most template systems (e.g. PHP, smarty, Razor...) feature in fact a full scripting engine within the template content. It allows powerful constructs like variable assignment or conditional statements in the middle of the HTML content. It makes it easy to modify the look of an application within the template system exclusively, without having to modify any of the underlying "application logic". They do so, however, at the cost of separation, turning the templates themselves into part of the application logic.

Mustache inherits from Google's *ctemplate* library, and is used in many famous applications, including the "main" Google web search, or the Twitter web site.
The *Mustache* template system leans strongly towards preserving the separation of logic and presentation, therefore ensures a perfect MVC - *Model-View-Controller* - design, and ready to consume SOA services.

Mustache is intentionally constrained in the features it supports and, as a result, applications tend to require quite a bit of code to instantiate a template: all the application logic will be defined within the *Controller* code, not in the *View* source. This may not be to everybody's tastes. However, while this design limits the power of the template language, it does not limit the power or flexibility of the template system. This system supports arbitrarily complex text formatting.

Finally, *Mustache* is designed with an eye towards efficiency. Template instantiation is very quick, with an eye towards minimizing both memory use and memory fragmentation. As a result, it sounds like a perfect template system for our *mORMot* framework.

7.3.2.2. Mustache principles

There are two main parts to the *Mustache* template system:

- Templates (which are plain text files);
- Data dictionaries (aka *Context*).

For instance, given the following template:

```
<h1>{{header}}</h1>

{{#items}}
  {{#first}}
    <li><strong>{{name}}</strong></li>
  {{/first}}
  {{#link}}
    <li><a href="{{url}}">{{name}}</a></li>
  {{/link}}
{{/items}}

{{#empty}}
  <p>The list is empty.</p>
{{/empty}}
```

and the following data context:

```
{
  "header": "Colors",
  "items": [
      {"name": "red", "first": true, "url": "#Red"},
      {"name": "green", "link": true, "url": "#Green"},
      {"name": "blue", "link": true, "url": "#Blue"}
  ],
  "empty": true
}
```

The *Mustache* engine will render this data as such:

```
<h1>Colors</h1>
<li><strong>red</strong></li>
<li><a href="#Green">green</a></li>
<li><a href="#Blue">blue</a></li>
<p>The list is empty.</p>
```

In fact, you did not see any "`if`" nor "*for*" loop in the template, but *Mustache* conventions make it easy to render the supplied data as the expected HTML output. It is up to the MVC *Controller* to render the data as expected by the template, e.g. for formatting dates or currency values.

7.3.2.3. Mustache templates

The *Mustache* template logic-less language has five types of tags:

- Variables;
- Sections;
- Inverted Sections;
- Comments;
- Partials.

All those tags will be identified with mustaches, i.e. `{{...}}`. Anything found in a template of this form is interpreted as a template marker. All other text is considered formatting text and is output verbatim at template expansion time.

Marker	Description
`{{variable}}`	The `variable` name will be searched recursively within the current context (possibly with dotted names), and, if found, will be written as escaped HTML. If there is no such key, nothing will be rendered.

`{{{variable}}}`
`{{& variable}}`
	The `variable` name will be searched recursively within the current context, and, if found, will be written directly, *without any HTML escape*.
	If there is no such key, nothing will be rendered.

`{{#section}}`
...
`{{/section}}`
	Defines a block of text, aka *section*, which will be rendered depending of the `section` variable value, as searched in the current context:
	- If `section` equals `false` or is an *empty list* `[]`, the whole block won't be rendered;
	- If `section` is non-`false` but not a list, it will be used as the context for a single rendering of the block;
	- If `section` is a non-empty list, the text in the block will be rendered once for each item in the list - the context of the block will be set to the current item for each iteration.

`{{^section}}`
...
`{{/section}}`
	Defines a block of text, aka *inverted section*, which will be rendered depending of the `section` variable *inverted* value, as searched in the current context:
	- If `section` equals `false` or is an *empty list*, the whole block *will* be rendered;
	- If `section` is non-`false` or a non-empty list, it won't be rendered.

`{{! comment}}`	The comment text will just be ignored.
`{{>partial}}`	The `partial` name will be searched within the registered *partials list*, then will be executed at run-time (so recursive partials are possible), with the current execution context.
`{{=...=}}`	The delimiters (i.e. by default `{{...}}`) will be replaced by the specified characters (may be convenient when double-braces may appear in the text).

In addition to those standard markers, the *mORMot* implementation of *Mustache* features:

Marker	Description
`{{.}}`	This pseudo-variable refers to the context object itself instead of one of its members. This is particularly useful when iterating over lists.
`{{-index}}`	This pseudo-variable returns the current item number when iterating over lists, starting counting at 1
`{{#-first}}` ... `{{/-first}}`	Defines a block of text (pseudo-section), which will be rendered - or *not* rendered for inverted `{{^-first}}` - for the *first* item when iterating over lists
`{{#-last}}` ... `{{/-last}}`	Defines a block of text (pseudo-section), which will be rendered - or *not* rendered for inverted `{{^-last}}` - for the *last* item when iterating over lists
`{{#-odd}}` ... `{{/-odd}}`	Defines a block of text (pseudo-section), which will be rendered - or *not* rendered for inverted `{{^-odd}}` - for the *odd* item number when iterating over lists: it can be very usefull e.g. to display a list with alternating row colors
`{{<partial}}` ... `{{/partial}}`	Defines an in-lined *partial* - to be called later via `{{>partial}}` - within the scope of the current template
`{{"some text}}`	This pseudo-variable will supply the given text to a callback, which will for instance transform its content (e.g. translate it), before writing it to the output

This set of markers will allow to easily write any kind of content, without any explicit logic nor nested code. As a major benefit, the template content could be edited and verified without the need of any *Mustache* compiler, since all those `{{...}}` markers will identify very clearly the resulting layout.

7.3.2.3.1. Variables

A typical Mustache template:

```
Hello {{name}}
You have just won {{value}} dollars!
Well, {{taxed_value}} dollars, after taxes.
```

Given the following hash:

```
{
  "name": "Chris",
  "value": 10000,
  "taxed_value": 6000
}
```

Will produce the following:

```
Hello Chris
You have just won 10000 dollars!
Well, 6000 dollars, after taxes.
```

You can note that `{{variable}}` tags are escaped for HTML by default. This is a mandatory security feature. In fact, all web applications which create HTML documents can be vulnerable to Cross-Site-Scripting (XSS) attacks unless data inserted into a template is appropriately sanitized and/or escaped. With Mustache, this is done by default. Of course, you can override it and force to *not-escape* the value, using `{{{variable}}}` or `{{& variable}}`.

For instance:

Template	Context	Output

`* {{name}}`
`* {{age}}`
`* {{company}}`
`* {{{company}}}`
	`{`
	`  "name": "Chris",`
	`  "company": "<b>GitHub</b>"`
	`}`
		`* Chris`
		`*`
		`* &lt;b&gt;GitHub&lt;/b&gt;`
		`* <b>GitHub</b>`

Variables resolve names within the current context with an optional dotted syntax, for instance:

Template	Context	Output

`* {{people.name}}`
`* {{people.age}}`
`* {{people.company}}`
`* {{{people.company}}}`
	`{`
	` "people": {`
	`  "name":"Chris",`
	`  "company":"<b>GitHub</b>"`
	` }`
	`}`
		`* Chris`
		`*`
		`* &lt;b&gt;GitHub&lt;/b&gt;`
		`* <b>GitHub</b>`

7.3.2.3.2. Sections

Sections render blocks of text one or more times, depending on the value of the key in the current context.

In our "wining template" above, what happen if we do want to hide the tax details?
In most script languages, we may write an `if ...` block within the template. This is what *Mustache* avoids. So we define a section, which will be rendered on need.

The template becomes:

```
Hello {{name}}
You have just won {{value}} dollars!
{{#in_ca}}
Well, {{taxed_value}} dollars, after taxes.
{{/in_ca}}
```

Here, we created a new section, named `in_ca`.

Given the hash value of `in_ca` (and its presence), the section will be rendered, or not:

Context	Output

`{`
`  "name": "Chris",`
`  "value": 10000,`
`  "taxed_value": 6000,`
`  "in_ca": true`
`}`
	`Hello Chris`
	`You have just won 10000 dollars!`
	`Well, 6000 dollars, after taxes.`

`{`
`  "name": "Chris",`
`  "value": 10000,`
`  "taxed_value": 6000,`
`  "in_ca": false`
`}`
	`Hello Chris`
	`You have just won 10000 dollars!`

`{`
`  "name": "Chris",`
`  "value": 10000,`
`  "taxed_value": 6000`
`}`
	`Hello Chris`
	`You have just won 10000 dollars!`

Sections also change the context of its inner block. It means that the section variable content becomes the top-most context which will be used to identify any supplied variable key.

Therefore, the following context will be perfectly valid: we can define `taxed_value` as a member of `in_ca`, and it will be rendered directly, since it is part of the new context.

Context	Output

`{`
`  "name": "Chris",`
`  "value": 10000,`
`  "in_ca": {`
`     "taxed_value": 6000`
`  }`
`}`
	`Hello Chris`
	`You have just won 10000 dollars!`
	`Well, 6000 dollars, after taxes.`

`{`
`  "name": "Chris",`
`  "value": 10000,`
`  "taxed_value": 6000`
`}`
	`Hello Chris`
	`You have just won 10000 dollars!`

`{`
`  "name": "Chris",`
`  "value": 10000,`
`  "taxed_value": 3000,`
`  "in_ca": {`
`     "taxed_value": 6000`
`  }`
`}`
	`Hello Chris`
	`You have just won 10000 dollars!`
	`Well, 6000 dollars, after taxes.`

In the latest context above, there are two `taxed_value` variables. The engine will use the one defined by the context in the `in_ca` section, i.e. `in_ca.taxed_value`; the one defined at the root context level (which equals 3000) is just ignored.

If the variable pointed by the section name is a list, the text in the block will be rendered once for each item in the list. The context of the block will be set to the current item for each iteration.
In this way we can loop over collections. *Mustache* allows any depth of nested loops (e.g. any level of master/details information).

Template	Context	Output

`{{#repo}}`
` <b>{{name}}</b>`
`{{/repo}}`
	`{`
	`  "repo": [`
	`     "name": "resque" ,`
	`     "name": "hub" ,`
	`     "name": "rip"`
	`  ]`
	`}`
		`<b>resque</b>`
		`<b>hub</b>`
		`<b>rip</b>`

`{{#repo}}`
` <b>{{.}}</b>`
`{{/repo}}`
	`{`
	`  "repo":`
	`     ["resque", "hub", "rip"]`
	`}`
		`<b>resque</b>`
		`<b>hub</b>`
		`<b>rip</b>`

The latest template makes use of the `{{.}}` pseudo-variable, which allows to render the current item of the list.

7.3.2.3.3. Inverted Sections

An inverted section begins with a caret (^) and ends as a standard (non-inverted) section. They may render text once, based on the *inverse* value of the key. That is, the text block will be rendered if the key doesn't exist, is false, or is an empty list.

Inverted sections are usually defined after a standard section, to render some message in case no information will be written in the non-inverted section:

Template	Context	Output
`{{#repo}}`		
` <b>{{.}}</b>`	`{`	
`{{/repo}}`	`  "repo":`	`No repos :(`
`{{^repo}}`	`     []`	
`No repos :(`	`}`	
`{{/repo}}`		

7.3.2.3.4. Partials

Partials are some kind of external sub-templates which can be included within a main template, for instance to follow the same rendering at several places. Just like functions in code, they do ease template maintainability and spare development time.

Partials are rendered at runtime (as opposed to compile time), so recursive partials are possible. Just avoid infinite loops. They also inherit the calling context, so can easily be re-used within a list section, or together with plain variables.

In practice, partials shall be supplied together with the data context - they could be seen as "template context".

For example, this "main" template uses a `{{> user}}` partial:

```
<h2>Names</h2>
{{#names}}
  {{> user}}
{{/names}}
```

With the following template registered as "user":

```
<strong>{{name}}</strong>
```

Can be thought of as a single, expanded template:

```
<h2>Names</h2>
{{#names}}
  <strong>{{name}}</strong>
{{/names}}
```

In *mORMot*'s implementations, you can also create some *internal* partials, defined as `{{<partial}} ... {{/partial}}` pseudo-sections. It may decrease the need of maintaining multiple template files, and refine the rendering layout.

For instance, the previous template may be defined at once:

```
<h2>Names</h2>
{{#names}}
  {{>user}}
{{/names}}


{{<user}}
<strong>{{name}}</strong>
{{/user}}
```

The same file will define both the partial and the main template. Note that we defined the internal partial after the main template, but we may have defined it anywhere in the main template logic: internal partials definitions are ignored when rendering the main template, just like comments.

7.3.2.4. SynMustache unit

Part of our *mORMot* framework, we implemented an optimized *Mustache* template engine in the `SynMustache` unit:

- It is the first Delphi implementation of *Mustache*;
- It has a separate parser and renderer (so you can compile your templates ahead of time);
- The parser features a shared cache of compiled templates;
- It passes all official *Mustache* specification tests, as defined at http://github.com/mustache/spec.. - including all weird whitespace process;
- External partials can be supplied as `TSynMustachePartials` dictionaries;
- `{{.}}`, `{{-index}}` and `{{"some text}}` pseudo-variables were added to the standard *Mustache* syntax;
- `{{#-first}}`, `{{#-last}}` and `{{#-odd}}` pseudo-sections were added to the standard *Mustache* syntax;
- Internal partials can be defined via `{{<partial}}` - also a nice addition to the standard *Mustache* syntax;
- It allows the data context to be supplied as JSON or our *TDocVariant custom variant type*;
- Almost no memory allocation is performed during the rendering;
- It is natively UTF-8, from the ground up, with optimized conversion of any string data;
- Performance has been tuned and grounded in `SynCommons`'s optimized code;
- Each parsed template is thread-safe and re-entrant;
- It follows the *Open/Close principle*, so that any aspect of the process can be customized and extended (e.g. for any kind of data context);
- It is perfectly integrated with the other bricks of our *mORMot* framework, ready to implement dynamic web sites with true *Model-View-Controller* design, and full separation of concerns in the views written in *Mustache*, the controllers being e.g. interface-based services;
- API is flexible and easy to use.

7.3.2.4.1. Variables

Now, let's see some code.
First, we define our needed variables:

```
var mustache: TSynMustache;
    doc: variant;
```

In order to parse a template, you just need to call:

```
  mustache := TSynMustache.Parse(
    'Hello {{name}}'#13#10'You have just won {{value}} dollars!');
```

It will return a compiled instance of the template.
The `Parse()` class method will use the shared cache, so you won't need to release the `mustache` instance once you are done with it: no need to write a `try ... finally mustache.Free; end` block.

You can use a `TDocVariant` to supply the context data (with late-binding):

```
  TDocVariant.New(doc);
  doc.name := 'Chris';
  doc.value := 10000;
```

As an alternative, you may have defined the context data as such:

```
  doc := _ObjFast(['name','Chris','value',1000]);
```

Now you can render the template with this context:

```
  html := mustache.Render(doc);
  // now html='Hello Chris'#13#10'You have just won 10000 dollars!'
```

If you want to supply the context data as JSON, then render it, you may write:

```
  mustache := TSynMustache.Parse(
    'Hello {{value.name}}'#13#10'You have just won {{value.value}} dollars!');
  html := mustache.RenderJSON('{value:{name:"Chris",value:10000}}');
  // now html='Hello Chris'#13#10'You have just won 10000 dollars!'
```

Note that here, the JSON is supplied with an extended syntax (i.e. field names are unquoted), and that `TSynMustache` is able to identify a dotted-named variable within the execution context.

As an alternative, you could use the following syntax to create the data context as JSON, with a set of parameters, therefore easier to work with in real code storing data in variables (for instance, any `string` variable is quoted as expected by JSON, and converted into UTF-8):

```
  mustache := TSynMustache.Parse(
    'Hello {{name}}'#13#10'You have just won {{value}} dollars!');
  html := mustache.RenderJSON('{name:?,value:?}',[],['Chris',10000]);
  html='Hello Chris'#13#10'You have just won 10000 dollars!'
```

You can find in the `mORMot.pas` unit the `ObjectToJSON()` function which is able to transform any `TPersistent` instance into valid JSON content, ready to be supplied to a `TSynMustache` compiled instance.
If the object's published properties have some getter functions, they will be called on the fly to process the data (e.g. returning 'FirstName Name' as FullName by concatenating both sub-fields).

7.3.2.4.2. Sections

Sections are handled as expected:

```
mustache := TSynMustache.Parse('Shown.{{#person}}As {{name}}!{{/person}}end{{name}}');
html := mustache.RenderJSON('{person:{age:?,name:?}}',[10,'toto']);
// now html='Shown.As toto!end'
```

Note that the sections change the data context, so that within the `#person` section, you can directly access to the data context `person` member, i.e. writing directly `{{name}}`

It supports also inverted sections:

```
mustache := TSynMustache.Parse('Shown.{{^person}}Never shown!{{/person}}end');
html := mustache.RenderJSON('{person:true}');
// now html='Shown.end'
```

To render a list of items, you can write for instance (using the `{{.}}` pseudo-variable):

```
mustache := TSynMustache.Parse('{{#things}}{{.}}{{/things}}');
html := mustache.RenderJSON('{things:["one", "two", "three"]}');
// now html='onetwothree'
```

The `{{-index}}` pseudo-variable allows to numerate the list items, when rendering:

```
mustache := TSynMustache.Parse(
  'My favorite things:'#$A'{{#things}}{{-index}}. {{.}}'#$A'{{/things}}');
html := mustache.RenderJSON('{things:["Peanut butter", "Pen spinning", "Handstands"]}');
// now html='My favorite things:'#$A'1. Peanut butter'#$A'2. Pen spinning'#$A+
//            '3. Handstands'#$A,'-index pseudo variable'
```

7.3.2.4.3. Partials

External partials (i.e. standard *Mustache* partials) can be defined using `TSynMustachePartials`. You can define and maintain a list of `TSynMustachePartials` instances, or you can use a one-time partial, for a given rendering process, as such:

```
mustache := TSynMustache.Parse('{{>partial}}'#$A'3');
html := mustache.RenderJSON('{}',TSynMustachePartials.CreateOwned(['partial','1'#$A'2']));
// now html='1'#$A'23','external partials'
```

Here `TSynMustachePartials.CreateOwned()` expects the partials to be supplied as name/value pairs.

Internal partials (one of the `SynMustache` extensions), can be defined directly in the main template:

```
mustache := TSynMustache.Parse('{{<partial}}1'#$A'2{{name}}{{/partial}}{{>partial}}4');
html := mustache.RenderJSON('{name:3}');
// now html='1'#$A'234','internal partials'
```

7.3.2.4.4. Internationalization

You can define `{{"some text}}` pseudo-variables in your templates, which text will be supplied to a callback, ready to be transformed on the fly: it may be convenient for i18n of web applications.

By default, the text will be written directly to the output buffer, but you can define a callback which may be used e.g. for text translation:

```
procedure TTestLowLevelTypes.MustacheTranslate(var English: string);
begin
  if English='Hello' then
    English := 'Bonjour' else
  if English='You have just won' then
    English := 'Vous venez de gagner';
end;
```

Of course, in a real application, you may assign one `TLanguageFile.Translate(var English: string)` method, as defined in the `mORMoti18n.pas` unit.

Then, you will be able to define your template as such:

```
  mustache := TSynMustache.Parse(
    '{{"Hello}} {{name}}'#13#10'{{"You have just won}} {{value}} {{"dollars}}!');
  html := mustache.RenderJSON('{name:?,value:?}',[],['Chris',10000],nil,MustacheTranslate);
  // now html='Bonjour Chris'#$D#$A'Vous venez de gagner 10000 dollars!'
```

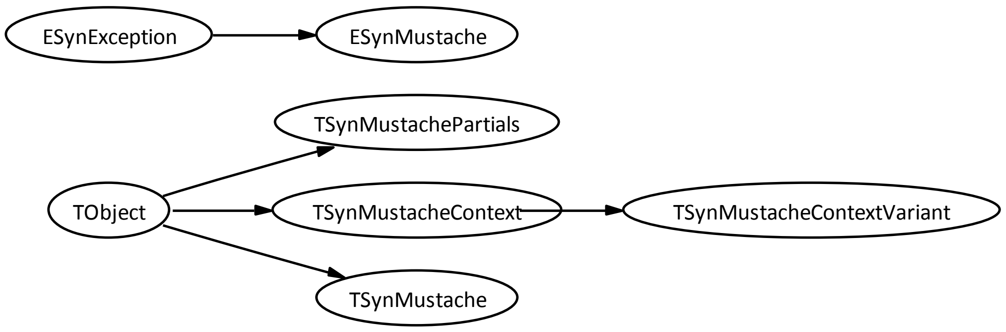
All text has indeed been translated as expected.

24.18. SynMustache.pas unit

Purpose: Logic-less mustache template rendering
- this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18

Units used in the *SynMustache* unit:

Unit Name	Description	Page
SynCommons	Common functions used by most Synopse projects - this unit is a part of the freeware Synopse mORMot framework, licensed under a MPL/GPL/LGPL tri-license; version 1.18	**Erreur ! Signet non défini.**



SynMustache class hierarchy

Objects implemented in the *SynMustache* unit:

Objects	Description	Page
`ESynMustache`	Exception raised during process of a {{mustache}} template	16
`TSynMustache`	Stores one {{mustache}} pre-rendered template	18
`TSynMustacheContext`	Handle {{mustache}} template rendering context, i.e. all values	16
`TSynMustacheContextVariant`	Handle {{mustache}} template rendering context from a custom variant	16
`TSynMustachePartials`	Maintain a list of {{mustache}} partials	17

Objects	Description	Page
TSynMustacheTag	Store a {{mustache}} tag	16

```
ESynMustache = class(ESynException)
```

Exception raised during process of a {{mustache}} template

```
TSynMustacheTag = record
```

Store a {{mustache}} tag

```
Kind: TSynMustacheTagKind;
```

The kind of the tag

```
SectionOppositeIndex: integer;
```

The index in Tags[] of the other end of this section
- either the index of mtSectionEnd for mtSection/mtInvertedSection
- or the index of mtSection/mtInvertedSection for mtSectionEnd

```
TextLen: integer;
```

Stores the mtText buffer length

```
TextStart: PUTF8Char;
```

Points to the mtText buffer start
- main template's text is not allocated as a separate string during parsing, but will rather be copied directly from the template memory

```
Value: RawUTF8;
```

The tag content, excluding trailing {{ }} and corresponding symbol
- is not set for mtText nor mtSetDelimiter

```
TSynMustacheContext = class(TObject)
```

Handle {{mustache}} template rendering context, i.e. all values
- this abstract class should not be used directly, but rather any other the overriden classes

```
constructor Create(WR: TTextWriter);
```

Initialize the rendering context for the given text writer

```
property OnStringTranslate: TOnStringTranslate read fOnStringTranslate write
fOnStringTranslate;
```

Access to the {{"English text}} translation callback

```
property Writer: TTextWriter read fWriter;
```

Read-only access to the associated text writer instance

```
TSynMustacheContextVariant = class(TSynMustacheContext)
```

Handle {{mustache}} template rendering context from a custom variant
- the context is given via a custom variant type implementing TSynInvokeableVariantType.Lookup, e.g. TDocVariant or TSMVariant

```
constructor Create(WR: TTextWriter; SectionMaxCount: integer; const aDocument:
variant);
```

Initialize the context from a custom variant document
- note that the aDocument instance shall be available during all lifetime of this `TSynMustacheContextVariant` instance
- you should not use this constructor directly, but the corresponding TSynMustache.Render*() methods

```
TSynMustachePartials = class(TObject)
```

Maintain a list of {{mustache}} partials
- this list of partials template could be supplied to TSynMustache.Render() method, to render {{>partials}} as expected
- using a dedicated class allows to share the partials between execution context, without recurring to non SOLID global variables
- you may also define "internal" partials, e.g. {{<foo}}This is foo{{/foo}}

```
constructor Create; overload;
```

Initialize the template partials storage
- after creation, the partials should be registered via the Add() method
- you shall manage this instance life time with a try..finally Free block

```
constructor CreateOwned(const NameTemplatePairs: array of RawUTF8); overload;
```

Initialize a template partials storage with the supplied templates
- partials list is expected to be supplied in Name / Template pairs
- this instance can be supplied as parameter to the TSynMustache.Render() method, which will free the instances as soon as it finishes

```
destructor Destroy; override;
```

Delete the partials

```
class function CreateOwned(const Partials: variant): TSynMustachePartials;
overload;
```

Initialize a template partials storage with the supplied templates
- partials list is expected to be supplied as a dvObject TDocVariant, each member being the name/template string pairs
- if the supplied variant is not a matching TDocVariant, will return nil
- this instance can be supplied as parameter to the TSynMustache.Render() method, which will free the instances as soon as it finishes

```
procedure Add(const aName: RawUTF8; aTemplateStart,aTemplateEnd: PUTF8Char);
overload;
```

Register a {{>partialName}} template

```
procedure Add(const aName,aTemplate: RawUTF8); overload;
```

Register a {{>partialName}} template

```
TSynMustache = class(TObject)
```

Stores one {{mustache}} pre-rendered template
- once parsed, a template will be stored in this class instance, to be rendered lated via the Render() method
- you can use the Parse() class function to maintain a shared cache of parsed templates
- implements all official mustache specifications, and some extensions
- handles {{.}} pseudo-variable for the current context object (very handy when looping through a simple list, for instance)
- handles {{-index}} pseudo-variable for the current context array index (1-based value) so that e.g. "My favorite things:\n{{#things}}{{-index}}. {{.}}\n{{/things}}" over {things:["Peanut butter", "Pen spinning", "Handstands"]} renders as "My favorite things:\n1. Peanut butter\n2. Pen spinning\n3. Handstands\n"
- handles -first -last and -odd pseudo-section keys, e.g. "{{#things}}{{^-first}}, {{/-first}}{{.}}{{/things}}" over {things:["one", "two", "three"]} renders as 'one, two, three'
- allows inlined partial templates , to be defined e.g. as {{<foo}}This is the foo partial {{myValue}} template{{/foo}}
- features {{"English text}} translation, via a custom callback
- this implementation is thread-safe and re-entrant (i.e. the same TSynMustache instance can be used by several threads at once)

```
constructor Create(const aTemplate: RawUTF8); overload;
```

Initialize and parse a pre-rendered {{mustache}} template
- you should better use the Parse() class function instead, which features an internal thread-safe cache

```
constructor Create(aTemplate: PUTF8Char; aTemplateLen: integer); overload;
virtual;
```

Initialize and parse a pre-rendered {{mustache}} template
- you should better use the Parse() class function instead, which features an internal thread-safe cache

```
destructor Destroy; override;
```

Finalize internal memory

```
class function Parse(const aTemplate: RawUTF8): TSynMustache;
```

Parse a {{mustache}} template, and returns the corresponding TSynMustache instance
- an internal cache is maintained by this class function
- this implementation is thread-safe and re-entrant: i.e. the same TSynMustache returned instance can be used by several threads at once

```
function Render(const Context: variant; Partials: TSynMustachePartials=nil;
OnTranslate: TOnStringTranslate=nil): RawUTF8; overload;
```

Renders the {{mustache}} template from a variant defined context
- the context is given via a custom variant type implementing TSynInvokeableVariantType.Lookup, e.g. TDocVariant or TSMVariant
- you can specify a list of partials via TSynMustachePartials.CreateOwned or a custom {{"English text}} callback
- can be used e.g. via a TDocVariant:

```
var mustache := TSynMustache;
    doc: variant;
    html: RawUTF8;
begin
  mustache := TSynMustache.Parse(
    'Hello {{name}}'#13#10'You have just won {{value}} dollars!');
  TDocVariant.New(doc);
  doc.name := 'Chris';
  doc.value := 10000;
  html := mustache.Render(doc);
  // here html='Hello Chris'#13#10'You have just won 10000 dollars!'
```

```
function RenderJSON(const JSON: RawUTF8; Partials: TSynMustachePartials=nil;
OnTranslate: TOnStringTranslate=nil): RawUTF8; overload;
```

Renders the {{mustache}} template from JSON defined context
- the context is given via a JSON object, defined from UTF-8 buffer
- you can specify a list of partials via TSynMustachePartials.CreateOwned or a custom {{"English text}} callback
- is just a wrapper around Render(_JsonFast())
- you can write e.g. with the extended JSON syntax:

```
 html := mustache.RenderJSON('{things:["one", "two", "three"]}');
```

```
function RenderJSON(JSON: PUTF8Char; const Args,Params: array of const;
Partials: TSynMustachePartials=nil; OnTranslate: TOnStringTranslate=nil):
RawUTF8; overload;
```

Renders the {{mustache}} template from JSON defined context
- the context is given via a JSON object, defined with parameters
- you can specify a list of partials via TSynMustachePartials.CreateOwned or a custom {{"English text}} callback
- is just a wrapper around Render(_JsonFastFmt())
- you can write e.g. with the extended JSON syntax:

```
   html := mustache.RenderJSON('{name:?,value:?}',[],['Chris',10000]);
```

```
class function UnParse(const aTemplate: RawUTF8): boolean;
```

Remove the specified {{mustache}} template from the internal cache
- returns TRUE on success, and FALSE if the template was not cached by a previous call to Parse() class function

```
procedure Render(Context: TSynMustacheContext; TagStart,TagEnd: integer;
Partials: TSynMustachePartials; NeverFreePartials: boolean); overload;
```

Renders the {{mustache}} template into a destination text buffer
- the context is given via our abstract TSynMustacheContext wrapper
- the rendering extended in fTags[] is supplied as parameters
- you can specify a list of partials via TSynMustachePartials.CreateOwned

```
property SectionMaxCount: Integer read fSectionMaxCount;
```

The maximum possible number of nested contexts

```
property Template: RawUTF8 read fTemplate;
```

Read-only access to the raw {{mustache}} `template` content

Types implemented in the *SynMustache* unit:

```
TSynMustacheSectionType = ( msNothing, msSingle, msSinglePseudo, msList );
```

States the section content according to a given value
- msNothing for false values or empty lists
- msSingle for non-false values but not a list
- msList for non-empty lists

```
TSynMustacheTagDynArray = array of TSynMustacheTag;
```

Store all {{mustache}} tags of a given template

```
TSynMustacheTagKind =
( mtVariable, mtVariableUnescape, mtSection, mtInvertedSection, mtSectionEnd,
mtComment, mtPartial, mtSetPartial, mtSetDelimiter, mtTranslate, mtText );
```

Identify the {{mustache}} tag kind
- mtVariable if the tag is a variable - e.g. {{myValue}}
- mtVariableUnescaped to unescape the variable HTML - e.g. {{{myRawValue}}} or {{& name}}
- mtSection and mtInvertedSection for sections beginning - e.g. {{#person}} or {{^person}}
- mtSectionEnd for sections ending - e.g. {{/person}}
- mtComment for comments - e.g. {{! ignore me}}
- mtPartial for partials - e.g. {{> next_more}}
- mtSetPartial for setting an internal partial - e.g. {{<foo}}This is the foo partial {{myValue}} template{{/foo}}
- mtSetDelimiter for setting custom delimeter symbols - e.g. {{=<% %>=}} - Warning: current implementation only supports two character delimiters
- mtTranslate for content i18n via a callback - e.g. {{"English text}}
- mtText for all text that appears outside a symbol