Mustache templates in Pascal

Michaël Van Canneyt

June 27, 2021

Abstract

Templates are widely used in websites and web applications. One popular template language definition is called Mustache. For a long time `dmustache` was the only Pascal implementation of the mustache template language. Now there is a second implementation available.

1 Introduction

Templates are very old. You can find them in many places: from Microsoft Mail merge to many web templating languages: smarty, twig etc. One such templating system is Mustache:

```
http://mustache.github.io/
```

The nice thing about mustache is that it is not bound to a programming language: it is just a description of how the templating should work, with a reference implementation. As can be seen on the website, implementations in many programming languages are available.

Mustache is also very low-level: its motto is 'Logic-less templates'. That means there is no way to put logic in the templates. The only logic are sections, which are an encapsulation of array data.

In its most basic form, Mustache will change a template `{{Something}}` by the value of the "variable" `Something`. You can also use scopes: `{{Customer.Name}}` will be replaced by the `Name` property of the `Custom` object. It further supports sections, and some special whitespace handling. Mustache is geared towards generating HTML in that the specification mandates that substitutions must be html escaped (although this can be disabled). The complete specification is available at the above website.

Mustache assumes JSON as the source of data, so the above 2 templates would need JSON data in the following form:

```
{
  "Something" : 12,
  "Customer" : {
    "Name" : "Overbeek",
    "FirstName" : "Detlef"
  }
}
```

But the source of the template data can be anything, an implementation of Mustache can provide alternate sources of data.

The dMustache implementation exists since many years. So why a new implementation of mustache templates? There are 2 reasons. The first reason is that dMustache, available at:

```
https://github.com/synopse/dmustache
```

depends on the Synopse mORMot framework (part of it is included in the repository). This means it does not compile for all platforms that Delphi supports. The second reason is that it fails several key tests in the Mustache testsuite. If you don't care about these reasons, dmustache is a good choice, it is highly optimized and offers several extensions to the Mustache templating specification, including limited logic.

2 fpMustache and mustached

Free Pascal now contains a fpMustache unit in its distribution. It has no depencencies except the JSON support of Free Pascal. To be able to use the new engine in Delphi, it has been ported to Delphi, and can be downloaded freely at

```
https://gitlab.com/mvancanneyt/mustached
```

The unit in the Delphi implementation is called mustache, in Free Pascal it is called fpmustache. The Delphi implementation comes with 2 GUI demo programs.

The mustache unit contains many objects, most of which are auxuliary classes used to compile the template. The central object in the implementation is the TMustache class. It is a component and can be dropped on a form or datamodule.

```
TMustache = class(TComponent)
Public
  Procedure Compile;
  Procedure Render(aContext : TMustacheContext;
                   aOutput : TMustacheOutput);
  Function Render(aContext : TMustacheContext) : TMustacheString;
  Function Render(const aJSON : TJSONValue) : TMustacheString;
  Function Render(const aJSON : String) : TMustacheString;
  Class function CreateMustache(aOwner: TComponent;
                                aTemplate: TMustacheString): TMustache;
  Class Function Render(aTemplate : TMustacheString;
                        const aJSON : String) : TMustacheString;
Published
  Property Template : TMustacheString;
  Property OnGetValue : TGetTextValueEvent;
  Property StartTag : TMustacheString;
  Property StopTag : TMustacheString;
  Property Partials : TStrings;
end;
```

From this declaration, you can see that the most simple usage of this component is as follows:

```
program t1;
{$APPTYPE CONSOLE}
// For delphi, only the mustache unit needs to be used.
```

```
uses jsonparser, fpmustache;

Const
  JSON = '{ "products" : [ {"name" : "BMW" }, '+
         '{"name" : "Mercedes"}, '+
         '{ "name" : "Audi" }] }';

  // Mock markdown table
  Template =
     '| name |'+sLineBreak+
     '|------|'+sLineBreak+
     '{{#products}}| {{name}} |'+sLineBreak+
     '{{/products}}';

begin
   Writeln(TMustache.Render(Template,JSON));
end.
```

The output will look like this:

```
| name |
|------|
| BMW |
| Mercedes |
| Audi |
```

Which is a markdown table, containing the 3 cars specified in the JSON.

As you can see, for simple cases like this it is not even necessary to create an instance of the `TMustache` class.

However, if you do create an instance of `TMustache` by putting it on a form, You can set the following properties and events:

Template A String with the Mustache template.

StartTag The starting characters for a tag, by default this is `{{`.

StopTag The ending characters of a tag, by default this is `}}`.

With these basic properties, one can get started. However, for more advanced usage, the following properties are also available:

OnGetValue This is an event that allows you to return values for template names that are not in the JSON which will be fed to the mustache engine.

It has the following signature:

```
TGetTextValueEvent = Procedure (Const aName : TMustacheString;
                                var aHandled : Boolean;
                                var aValue : TMustacheString) of Object;
```

The `aName` parameter contains the name of the value the engine needs, and you must return the value for this name in `aValue`. On return, `aHandled` must be set to `True`. If not, an empty value will be assumed.

Partials Mustache specifies that you can include other templates, called partials:

```
{{> subtemplate}}
```

When the engine encounters this, it must locate the `subtemplate` partial template, and process it as if its contents was inserted instead of the above template. The `Partials` property allows you to map partial names to actual partial templates:

```
name={{#names}}{{name}}{{/names}}
```

With all these properties set, the methods of `TMustache` can be used to actually render a template:

Compile Compiles the template: this method will parse the template and compile it into a form that will allow it to be rendered multiple times without the need to parse it again at each invocation of the `Render` method. Under normal circumstances, it should not be necessary to call this method: the `Render` method will call it when needed.

Render Will render the template with given data. This method has several forms.

The simplest form takes a string with JSON data as input, and returns the template as rendered with the JSON data:

```
Function Render(const aJSON : String) : TMustacheString;
```

The second form is actually used by the first form: here the data to use when rendering is actually specified as a JSON object:

```
Function Render(const aJSON : TJSONValue) : TMustacheString;
```

The third form uses a context object: the `TMustacheContext` class, which is responsible for providing the data to the render process:

```
Function Render(aContext : TMustacheContext) : TMustacheString;
```

The 2 `Render` objects that accept JSON actually call this method and use a descendent of the `TMustacheContext` class called `TMustacheJSONContext`, to fetch the data from the provided JSON string (or object).

The output of the above `Render` methods is always a string. But in fact, the output can be sent to whatever destination you want: a stream, a file, whatever.

This brings us to the last form of the `Render` method:

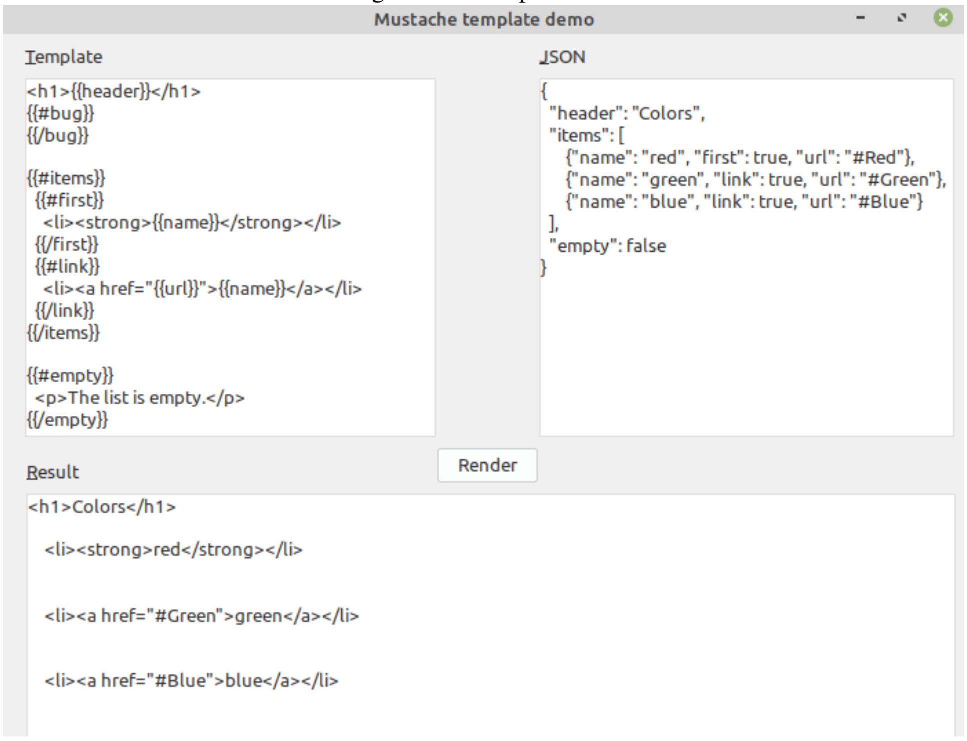
```
Procedure Render(aContext : TMustacheContext; aOutput : TMustacheOutput);
```

The `TMustacheOutput` class is very simple

```
TMustacheOutput = Class(TObject)
Public
  Procedure Output(Const aText : TMustacheString); virtual; abstract;
  Procedure Reset; virtual; abstract;
end;
```

The `output` method is called whenever rendered text needs to be appended to the output. The standard `Render` methods use a `TMustacheStringOutput` descendent of this class, which simply concatenates all text into one string, which is then returned by the `Render` function.

Figure 1: A simple demo



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

{{#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}}
```

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

```
<h1>Colors</h1>

  <li><strong>red</strong></li>


  <li><a href="#Green">green</a></li>


  <li><a href="#Blue">blue</a></li>
```

3 a simple demo

To demonstrate this, we will recreate the demo on the mustache website as a Lazarus program. (the same program exists for Delphi in the gitlab repository)

To this end, we put 3 memos on a form, and a button to render the content. The `MTemplate` memo component contains the sample template from the website, the `MJSON` memo contains the JSON to feed as data to the rendering engine. The result will be rendered in the `mResult` memo.

The Mustache component is created in the Form's `OnCreate` event.

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  FMustache:=TMustache.Create(Self);
end;
```

Doing so we don't need to put the mustache component on the component palette. We do need to provide the path to the source of the `fpMustache` unit in the project's compiler settings. (The same is done in the Delphi demo)

When the user presses the `bRender` button, the following code is executed:

```
procedure TMainForm.BRenderClick(Sender: TObject);
begin
  MResult.Text:=fMustache.Render(mTemplate.Text,mJSON.Text);
end;
```

The result of all this can be seen in figure 1 on page 5.

4 Feeding dataset data to the renderer

The above example uses plain JSON to feed the data to the template renderer. But what if your data is in a dataset? One way would be to transform the dataset data to a JSON array of records, and then feed that JSON data to the renderer. However, this would involve a lot of copying of data, slowing things down and increasing memory requirements.

Instead, the `TMustacheContext` can be expanded: the unit `dbmustache` (or `fpdbmustache`) contains a descendant of `TMustacheContext` called `TMustacheDBContext`.

This context object will fetch the data from one or more datasets that can be attached to it. Each dataset is given a section name, which can be used in the template to refer to a dataset in the collection of datasets.

Given a dataset with name `dsFamily`, the following template

```
{{#dsFamily}}
<tr>
  <td>{{name}}</td><td>{{age}}</td>
</tr>
{{/dsFamily}}
```

would produce a set of HTML table rows with the contents of the fields `name` and `age`. Likewise, a template

```
{{dsFamily.name}}
```

would render the field `name` of the `dsFamily` dataset.

The `TMustacheDBContext` class (a class descendent from `TObject`, so not a `TComponent`) has the following methods and properties:

```
TMustacheDBContext = Class(TMustacheContext)
Public
  Procedure Clear;
  Procedure AddDataset(aDataset : TDataset; aSectionName : String = '');
  Procedure RemoveDataset(aDataset : TDataset);
  Property StaticValues : TStrings;
  Property Datasets[aIndex : Integer] : TDatasetCollectionItem;
  Property DatasetCount : Integer;
end;
```

The meaning of these methods is pretty straightforward:

Clear Clear the dataset list and the `StaticValues`.

AddDataset Add a dataset to the list of datasets. The section name can be specified. When the name is omitted, the name of the dataset component is used.

RemoveDataset Remove a dataset from the list of datasets.

StaticValues A stringlist containing `name=value` pairs. This can be used to provide the template renderer with values that are not available as a field in one of the provided datasets.

Datasets Indexed access to the Dataset/Section name items.

DatasetCount Number of registered dataset items.

We can change the GUI example program to use this dataset context: we drop a CSV dataset on the form (csvFamily), connect it to a grid (GData) and a DBNavigator (NavData), which we put instead of the MJSON memo.

The data of the CSV dataset is loaded in the form OnCreate handler:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  FMustache:=TMustache.Create(Self);
  csvFamily.FileName:=ExtractFilePath(ParamStr(0))+'family.csv';
  csvFamily.Open;
end;
```

The OnClick method of the BRender button now must use the TMustacheDBContext class to feed the data to the render method:

```
procedure TMainForm.BRenderClick(Sender: TObject);

Var
  C : TMustacheDBContext;

begin
  C:=TMustacheDBContext.Create(Nil);
  try
    C.AddDataset(csvFamily,'data');
    C.StaticValues.Values['title']:='Our family';
    fMustache.Template:=mTemplate.Text;
    MResult.Text:=fMustache.Render(C);
  finally
    C.Free;
  end;
end;
```

In the code, a static value called title is added to the context, which will contain the title of the rendered HTML page.

After setting the Template property, and calling the Render method using the created TMustacheDBContext instance, the result will look something like figure 2 on page 8.

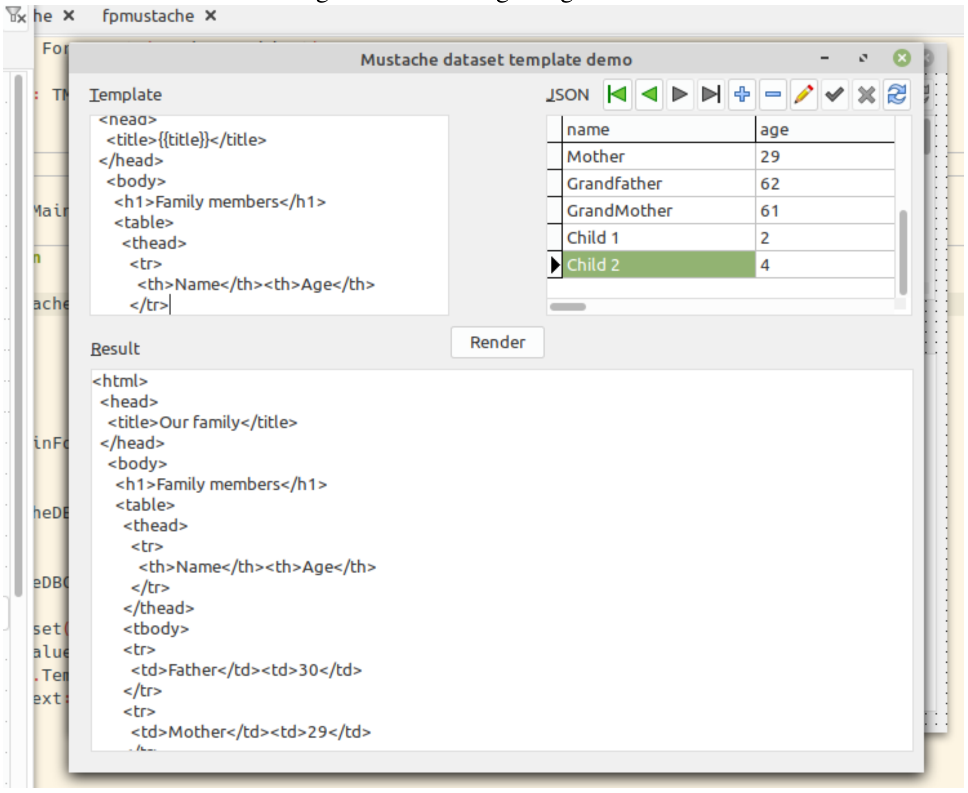
5 Adding simple logic: using expressions

The TMustache component is set up in such a manner that descendents can be created which initialize their own template parser, which must be an instance of TMustacheParser. This allows you to extend the template parser with support for new kinds of templates. The template is parsed into a DOM-like structure, based on TMustacheElement classes.

When the template is compiled, the parser uses a factory function to create the necessary mustache elements for the various template elements. You can override this function (called CreateDefault) in a descendent of TMustacheParser, to check for special characters and create a special element with customized handling for the template.

The examples from the previous sections and the above explanation will work both in Delphi and Lazarus. What follows next is only available in Free Pascal: the fpexmustache unit contains a descendent of the TMustache component, called TMustacheExpr.

Figure 2: Rendering using a dataset



```
<head>
  <title>{{title}}</title>
</head>
 <body>
  <h1>Family members</h1>
  <table>
   <thead>
    <tr>
     <th>Name</th><th>Age</th>
    </tr>
```

name	age
Mother	29
Grandfather	62
GrandMother	61
Child 1	2
Child 2	4

```
<html>
 <head>
  <title>Our family</title>
 </head>
  <body>
   <h1>Family members</h1>
   <table>
    <thead>
     <tr>
      <th>Name</th><th>Age</th>
     </tr>
    </thead>
    <tbody>
    <tr>
     <td>Father</td><td>30</td>
    </tr>
    <tr>
     <td>Mother</td><td>29</td>
```

This component uses an expression calculation engine only available in Free Pascal, hence the component is only available for Free Pascal/Lazarus, and not for Delphi.

The `TMustacheExpr` class creates a `TMustacheExprParser` parser class. It overrides the `CreateDefault` method to allow you to enter an expression:

```
{{[IF(age>=18,'adult','minor')]}}
```

The square brackets inside the template indicate that the content of the template is an expression. All expressions supported by the `TFPExpressionParser` component of Free Pascal are allowed. The above is quite simple: if the variable age is larger than 18, then `adult` will be rendered, else `minor`.

The expression engine supports various datatypes: whatever the result type, it will be converted to a string using standard functions in the `SysUtils` unit, and rendered. The expression engine of Free Pascal itself of course offers a lot of conversion functions (most of them mappings from various `SysUtils` functions), so you can format the result yourself in your expression.

When using the expression engine, it needs to know what variables are available and what their types are. The available variable names are the same ones as in the JSON data given to the renderer. However, they must be explictly registered with the expression engine.

This is done using the `RegisterVariables` method of the `TMustacheExpr` class, which comes in several overloaded versions:

```
Procedure RegisterVariables (aContext : TMustacheJSONContext;
                             aPath : TJSONStringType = '';
                             UseEvent : Boolean = True);
```

```
Procedure RegisterVariables (aJSON : String;
                             aPath : TJSONStringType = '';
                             UseEvent : Boolean = True);
Procedure RegisterVariables (aJSON : TJSONObject;
                             aPath : TJSONStringType = '';
                             UseEvent : Boolean = True);
```

As you can see, the variable values can be registered through multiple mechanisms: as a JSON string, a JSON object, or a JSON context. The `aPath` parameter is a path in the JSON object: only keys below this path will be registered. For example, given the following JSON:

```
{
  "data" : {
    "customer" : {
      "firstname" : "John",
      "lastname" : "Doe"
    }
  }
}
```

using the path `data.custmer` will only register the `firstname` and `lastname` values.

The use of `UseEvent` needs some explanation. If `UseEvent` is false, the variables will be registered as static values: their names, types and values will be taken from the JSON given to the `RegisterVariables` call. That means if you call `Render` several times, each time using a different JSON, `firstname` and `lastname` will have the value `John` and `Doe` every time when you render the template.

This may be what you want, but it is more likely that this is not what you want: in `RegisterVariables` you just want to register the names using a sample JSON, but the actual value must be fetched when rendering from the JSON supplied to the `Render` method.

If `UseEvent` is `True` (the default), then only the names and types of the variables are registered, but the actual values of these variables will be retrieved while rendering with an event. The values will be retrieved using the same mechanism as used when getting template values.

We'll demonstrate this with a small example, which demonstrates the expression given in the example above.

```
program demo2;

uses Classes, fpjson, jsonparser,fpmustache, fpexmustache;

Const
  // Mock markdown table
  Template =
     '| name | age | '+sLineBreak+
     '|------|------|'+sLineBreak+
     '{{#data}}| {{name}} | '+
     '{{[IF(age>=18,''adult'',''minor'')]}} |'+sLineBreak+
     '{{/data}}';

Var
```

```
  M : TMustacheExpr;
  C : TJSONObject;
  F : TFileStream;

begin
  M:=TMustacheExpr.Create(Nil);
  try
    F:=TFileStream.Create('family.json',fmOpenRead);
    C:=GetJSON(F) as TJSONObject;
    M.RegisterVariables(C,'data[0]');
    M.Template:=Template;
    Writeln(M.Render(C));
  finally
    M.Free;
    F.Free;
    C.Free;
  end;
end.
```

As you can see, the program is again simplicity itself.

The JSON is read from a file `family.json` that looks like this:

```
{
  "data" : [
   { "name" : "Father", "age": 30 },
   { "name" : "Mother", "age": 29 },
   { "name" : "Grandfather", "age": 62 },
   { "name" : "GrandMother", "age": 61 },
   { "name" : "Child 1", "age": 2 },
   { "name" : "Child 2", "age": 4 }
  ]
}
```

The path `data[0]` used in the `RegisterVariables` call indicates that the first record in the `data` array imust be used used to register the variables: this means `name` and `age` are registered as variables using an event to get the actual values.

The result of this program is the following markdown table:

```
| name | age |
|------|------|
| Father | adult |
| Mother | adult |
| Grandfather | adult |
| GrandMother | adult |
| Child 1 | minor |
| Child 2 | minor |
```

Which is the expected outcome.

6 conclusion

Having a standard template engine available can be useful: it can be used to generate markdown, HTML, plain text, or any other text format, starting from a template and data.

This could be used for a website, but also for reporting, and it is perfect for generating and sending mails, allowing the user to specify a template for the mail (html or plain text) and send the mail based on data in a database.