Qt Handlebars

Michel D'HOOGE

2016-01-22

Contents

1 Summary

This document introduces a Qt adaptation of the Handlebars templating language. The goal is to provide the ease-of-use of Handlebars to Qt developments. Everything that the *genuine* Handlebars does has been coded (as far as I can tell) and adapted to Qt.

2 Example

An example program can be generated with `demo.pro`.

This program loads all the available *helpers* and creates a *demo* context (`DemoContext.cpp`). This context contains predefined values of different types that can be used in any Handlebars templates. Then it opens the filename received from the command line and translates its content to `stdout` (no HTML conversion by default).

The `doc/Demo.txt` file can be provided to have a quick (and complete) overview of what this implementation of Handlebars can do.

3 Handlebars

This chapter describes all that can be done with this implementation of Handlebars. Since C++ is *strongly typed*, you may want to have a look at the implementation section to have more insight about what can be dynamically done (thanks to Qt objects).

3.1 Expressions

Handlebars *expressions* are available, of course. Everything described by the page http://handlebarsjs.com/expressions.html can be done:

- Simple *identifier*: `{{ title }}`
- Path: `{{ article.title }}`
- *Segment-literal* notation: `{{ articles.[10].[#comments] }}`
  Note: *dots* are needed; square brackets are not mandatory for *10* (but since they are present in the *official* example, I kept them here).
- Legacy syntax with `/`

Remember that `flex` is used to slice the input text into tokens; and `flex` doesn't speak Unicode, only plain, dumb, 8-bit characters. However this shouldn't be a problem because characters forbidden in *identifiers* are all ASCII (see the declaration in `Handlebars.l::ID_char`).

3.2 Path

It isn't possible to use `../` in a path because this isn't coded. As a replacement, if one of the elements of the path doesn't exist, the path is searched again within the parent context, and so on up to the outer context. This has more or less the same effect (unless an element in a given context hides an element with the same name in a more outer context).

3.3 Escaping

By default, the renderer applies no text escaping.

To set an escaping function (it doesn't have to be HTML-centric), one must call the `setEscapeFunction()` method. Any function that has the `escape_fn` signature (defined in `HandlebarsHelpers.h`) can be provided. For example, `ExtraHelpers` defines 2 such functions: `fn_noEscape` and `fn_htmlEscape`.

Note: With current code, the escape function must be defined before calling `parse()`. This could be changed but doesn't seem that useful...

3.4 Helpers & Block Helpers

Helpers are available with the full, complete syntax:

```
{{{ helper-id param1 param2 ... key1=val1 key2=val2 }}}
```

For example:

```
{{{link "See more..." href=story.url class="story"}}}
{{ hex 0x12345 fill='X' width=12 }}
```

Block helpers are also available. They all accept an `{{else}}` member (if this doesn't make sense for a given *helper* isn't the problem of the parser...)

3.5 Subexpressions

Subexpressions are available. For example:

```
{{ upper (hex 42) }}
```

3.6 Blanks & other spaces

Within *mustaches*, all blanks and line feeds are ignored. The layout of any expression is thus free.

There is a single, sensible exception to this rule: Constants of type strings (simple or double quotes) are never filtered. But they must fit on a single line.

It is even possible to use blanks before the `#` and `/` characters that define a *block helper*. On the other hand, comments and blanks removals must always stick to the brackets. Only the following layouts are valid:

```
{{! Simple comment }}
{{~! Comment with blanks removal before/after ~}}
{{!-- Comment that cannot be used with '~' --}}
```

3.7 Partials

A *partial* is a text block that can be inserted several times in another text. All *expressions* contained in the partial are evaluated according to the context of the calling block (and *not* when the partial is created). The content of a partial is thus *dynamic*.

Currently a partial can only be defined through a call to the *block helper* `registerPartial`. Thus it isn't possible to *pre-initialize* the initial, outer context from C++. If that would be needed, one would have to make the `RenderingContext` available and call the method `registerPartial`.

A *partial* is declared in the text stream by the command `}}>`.

3.8 Includes

To split the Handlebars *code* into several files, it is possible to include the content of another file with the command `{{+`. This command accepts one or more filenames, as characters strings (single ou double quotes).

This command is processed by `flex` during the *parsing* phase. This implies that it cannot accept Handlebars expressions for the filenames; because everything must be *resolved* before the beginning of the *rendering*.

Relative paths (that don't start with `/`) are anchored into the folder of the calling file. For more details, see the `openFile()` method of `IStreamManager`.

4 Helpers

Helpers are C++ functions registered with the Handlebars *parser*. They must be registered before calling the `render()` method.

4.1 Default Helpers

All *helpers* introduced in http://handlebarsjs.com/builtin_helpers.html are coded in `HandlebarsParser.cpp`. To register them, call method `Parser::registerDefaultHelpers()`.

4.1.1 each

`{{# each <container> }}`
Content to process for each element of the container.
`{{ else }}`
Content to be displayed if the container is empty, or if the variable is *not* a container.
`{{/ each }}`

Within the block, the following variables are defined:

- `first` is *true* for `i == 0`,
- `@index` vary from 1 to *n*,
- `@index0` vary from 0 to *n-1*,
- `last` is *true* for `i == n-1`,
- `this` holds current value.

4.1.2 if

`{{# if <truthy> }} TRUE {{ else }} FALSE {{/ if }}`

Look at the source for the definition of a *truthy* value.

4.1.3 registerPartial

`{{# registerPartial <partial-name> }}`
Text recorded within the *partial*.

The text may contain expressions and blocks.
`{{/ registerPartial }}`

It is then possible to re-call the block with *{{> partial-name }}*.

4.1.4 unless

```
{{# unless <false> }} FALSE {{else}} TRUE {{/ unless }}
```

4.1.5 with

```
{{# with <property> }} ... {{/ with }}
```

4.2 Extra Helpers

Additional *helpers* are provided in `ExtraHelpers.cpp`. They are sorted by *kind* and can be registered with the parser by calling functions like `registerFileHelpers()`.

4.2.1 Bitwise Helpers

- `bit-mask <size>`
  A hex value with <*size*> bits set to 1 (counting from the LSB).
- `int_fast ["bit"|"byte"] <value>`
  The fastest integer type for this size (bits or bytes), this processor, etc.
  These C++ types are defined in `<stdint.h>`.
- `int_least ["bit"|"byte"] <value>`
  Same as above, but smallest.
- `uint[_fast|_least] ["bit"|"byte"] <value>`
  Same as above, but unsigned.

4.2.2 Boolean Helpers

All these *helpers* return a boolean value that can be tested by another *helper*.

- `[==] [!=] [<] [<=] [>] [>=]`: Comparison operators.
- `AND <bool>+`: Returns the logic AND between all parameters.
- `OR  <bool>+`: Returns the logic OR between all parameters.
- `NOT <bool>`: Returns the negation of the operand.

For example:
`{{# if ( [==] 1 2 ) }} ONE equals TWO!!! {{/ if }}`

The *brackets* around the functions names are an example of when to use the *segment-literal* notation. This can be used for any kind of identifiers.

4.2.3 File Helpers

The following *helpers* accept a list of arguments that are concatenated together. Concatenating elements together is handy to quickly address pathname split into several variables.

- `temp_path`: Returns the system temporary folder (result of a call to `QDir::tempPath()`).
- `mkdir <filepathname-element>+`: Creates a folder.
- `cd <filepathname-element>+`: Changes the current working directory (returning to the previous folder isn't handled).
- `copy_into-current-folder_from <element>+ [contentOnly=0]`
  Copies into current folder the pointed-to content (file or folder, recursively). If `contentOnly` is 1 (0 by default), only the content of the folder is copied; otherwise a folder with the same name is created first.
- `{{# create_file <filepathname-element>+ [HASH] }}`
  ...
  `{{/ create_file }}`
  Creates a file in which all the content of the block is written. If a *hash* is provided, its content is added to the context. The following variables are also available: `@basename`, `@filename`, `@filepath`, `@filepath_absolute` and `@filepath_relative`. This last variable is only created if the caller context contains a variable called `output_folder` (this isn't the case by default).

It is important to remember that all these functions are executed during the *rendering* phase and as such, they won't have any effect on the *source* content.

It is possible (and even maybe recommended) to change the current working directory between the call to `parse()` and the call to `render()`. But in this case, it is the responsibility of the software and/or the templates to return into the initial folder at the end.

4.2.4 Integer Helpers

- `range <first> <last> [<increment> = +1/-1 by default]`
  Creates a `QVariantList` with integers between `first` et `last`.

4.2.5 Property Helpers

- `objects-with-property <container> <property-name>+`
  Returns a container only filled with objects that have the `QProperty` set to *true*. If the property doesn't exist, it is considered as *false*. If several properties are listed, a logic OR is applied: an object is selected as soon as it has one of the property. To create a logic AND, one must embed sub-expressions.
  The returned container is of the same type as the one provided (`QHash`, `QList`, `QMap`).
- `container-value <container> <key>`
  Returns a value from a container. The same function is used as to retrieve a property from a context; so everything that works for an expression should also work here.

- `{{# set_property <property-name> [<value>] [only-if="new"] }}`
  Content to put into the property, only if <value> *is* not *defined.*
  `{{/ set_property }}`
  Creates a new property that will be available *until the end* of current context; except if `only-if` is set to `new` and the property already exists, in that case, the new value is ignored.
  There are 2 ways to provide the value: either within the *moustaches* or in the contained block. Note: Because of implementation details, a *block helper* is always used even when the value is provided in the *moustaches* (because currently the source code only allows *block helper* to change the context; and I didn't think it was appropriate to also authorise the *helpers* just for this function).

4.2.6 String Helpers

- `camelize <string>`
  Removes blanks and changes all words so they start with an uppercase letter.
- `date [<format> = "yyyy-MM-ddTHH:mmt"]`
  Returns date & hour in the given format.
- `hex <num> width=0 fill=' '`
  Returns a hex string with the value of the provided integer.
- `link "Text" href=url [other attrs]`
  Returns an HTML link.
- `print <qvariant> [other vars]`
  Prints content and type of the parameter (mostly for *debug* purpose).
- `upper "string"`
  Returns an all uppercased string.

4.3 ICU Helpers

These *helpers* are separated from the previous ones because they need the `ICU` libraries.

- `camelize <string>`
  A more *complete* version of the String helper. The returned value is only composed of ASCII characters; it should be used without risk as an *identifier* by a compiler.
- `icu_transform <Compound_ID> <string>`
  Applies the provided *ICU transform* to the string (for ICU experts and copy/pasters).
- `{{# icu_transform_b <Compound_ID> }} TEXT {{/icu_transform_b}}`
  The *block* version of the previous function.

5 Implementation

The source code is not that big. But it may be a bit cryptic because of the use of C++ templates in some places (to only "handle" once or twice the Qt containers: `QList`, `QHash` & `QMap`).

5.1 Parsing & Rendering

`Flex` and `Bison` are used to parse the Handlebars templates. The parsing must be done once, and then the rendering can be done several times, with different contexts.

Two context types are defined. The `HandlebarsParsingContext` is *really* for internal use by the parser. The `HandlebarsRenderingContext` is indirectly available to the templates through *helpers* (it may be of interest if you want to add a new function).

5.2 Context & QVariant objects

To have something a bit "dynamically typed" (*à la* javascript), `QVariant` objects are used.

In the `HandlebarsRenderingContext`, the `PropertiesContext` type is defined as `QVariant`, because this is the top-level class. But one is really expected to provide instead a `QVariant` container (which happens to be a sub-class of `QVariant`).

The rendering context knows how to handle `QHash`, `QList`, `QMap` and plain `QObject`. And any user-defined `QVariant` objects are *tried* like `QObject`.

For Qt containers, the behaviour is the *expected*,*normal* one, with an addition to handle size. A `QList` returns its size when asked for properties "length" or "size". A map or hash returns its `size()` as a result of property "length", but only if it has no content with that name.

For `QObjects`, there are some subtleties. A property is first search with the `QObject::property()` method. If it isn't found, the key is compared to some pre-defined values. Both `name` and `objectName` returns the content of a call to `objectName()`. And `parent` returns `parent()`. And `className` returns the class name of the meta-object as a string.

6 Useful Tips

6.1 Text Layout

It is not always easy to have both a *nice* layout of the source file and the produced content. When the result is only to be processed by another software, that's not a big deal. On the other hand, when the resulting output must be reprocessed by a human being, this gets more tricky.

To *swallow* all blanks (including line-feeds) before or after an expression, one must use the `~` character. For example, the expression `{{~!~}}` will remove all blanks before & after itself.

It is also possible to do the opposite: to keep some blank characters (for example to force a line-feed). In that case, use an empty comment: `{{!}}`.

To correctly use the `~` character, it is important to understand when the blanks removal occurs. In our case this removal is done by `flex`, when the input text is just read. This means that the *tokens* handled by the `bison` grammar have already been *stripped*. This is

especially of interest for the *block helpers*, in case of repetition or when choosing between the *main* and the *else* blocks.

6.2 Comments

Like many other languages, it is possible to use comments to deactivate some parts of an input source file. When experimenting, if a block must be regularly deactivated, one can use the following method:

```
{{! --}}
Block is active.
{{! --}}
```

As long as there is a space between "`!`" and "`--`", the block is *active*. On the other hand, if the space of the 1st comment is removed, `flex` will remove everything between `{{!--` et `--}}`, which is the whole block.

```
{{!--}}
Hidden block.
{{! --}}
```

In addition, it is possible to also use the `~` character to remove blanks.

```
{{! --}}{{!~}}
My on/off block.
{{! --}}{{!~}}
```