Quik Documentation

Release 0.2.2

Thiago Avelino

September 06, 2013

CONTENTS

A fast and lightweight Python template engine

CHAPTER

ONE

FEATURES

- Easy to use.
- High performance.
- Autoescaping.
- Template inheritance.
- Supports native python expressions.

CHAPTER

TWO

NUTSHELL

Here a small example of a Quik template

```
<ul>
    #for @user in @users:
        #if @user.age > 18:
        <li><a href="@user.url">@user.username</a></li>
        #end
    #end
</ul>
```

CHAPTER

THREE

USE IT

Render via template:

```
from quik import FileLoader

loader = FileLoader('html')
template = loader.load_template('index.html')
print template.render({'author': 'Thiago Avelino'},
                      loader=loader).encode('utf-8')
```

3.1 Contents

3.1.1 User Guide

About this Guide

The Quik User Guide is intended to help page designers and content providers get acquainted with Quik and the syntax of its simple yet powerful scripting language, the Quik Template Language (QTL). Many of the examples in this guide deal with using Quik to embed dynamic content in web sites, but all QTL examples are equally applicable to other pages and templates.

Thanks for choosing Quik!

Hello Quik World!

Once a value has been assigned to a variable, you can reference the variable anywhere in your HTML document. In the following example, a value is assigned to @foo and later referenced.

```
<html>
    <body>
        #set @foo = "Quik":
        Hello @foo World!
    </body>
<html>
```

The result is a web page that prints "Hello Quik World!". To make statements containing QTL directives more readable, we encourage you to start each QTL statement on a new line, although you are not required to do so. The set directive will be revisited in greater detail later on.

Comments

Comments allows descriptive text to be included that is not placed into the output of the template engine. Comments are a useful way of reminding yourself and explaining to others what your QTL statements are doing, or any other purpose you find useful. Below is an example of a comment in QTL.

```
## This is a single line comment.
```

A single line comment begins with ## and finishes at the end of the line. If you're going to write a few lines of commentary, there's no need to have numerous single line comments. Multi-line comments, which begin with #* and end with *#, are available to handle this scenario.

```
This is text that is outside the multi-line comment.
Online visitors can see it.

#*
Thus begins a multi-line comment. Online visitors won't
see this text because the Quik Templating Engine will
ignore it.
*#

Here is text outside the multi-line comment; it is visible.
```

Here are a few examples to clarify how single line and multi-line comments work:

```
This text is visible. ## This text is not.
This text is visible.
This text is visible. #* This text, as part of a multi-line
comment, is not visible. This text is not visible; it is also
part of the multi-line comment. This text still not
visible. *# This text is outside the comment, so it is visible.
## This text is not visible.
```

There is a third type of comment, the QTL comment block, which may be used to store such information as the document author and versioning information:

```
#**
This is a QTL comment block and
may be used to store such information
as the document author and versioning
information:
@author
@version 5
*#
```

References

There are three types of references in the QTL: variables, properties and methods. As a designer using the QTL, you and your engineers must come to an agreement on the specific names of references so you can use them correctly in your templates.

Everything coming to and from a reference is treated as a String object. If there is an object that represents *@foo* (such as an Integer object), then Quik will call its **.toString()** method to resolve the object into a String.

Variables

The shorthand notation of a variable consists of a leading "@" character followed by a QTL Identifier. A QTL Identifier must start with an alphabetic character (a .. z or A .. Z). The rest of the characters are limited to the following types of characters:

- alphabetic (a .. z, A .. Z)
- numeric (0 .. 9)
- hyphen ("-")
- underscore ("_")

Here are some examples of valid variable references in the QTL:

```
@foo
@mudSlinger
@mud-slinger
@mud_slinger
@mudSlinger1
```

When QTL references a variable, such as @foo, the variable can get its value from either a set directive in the template, or from the Python code. For example, if the Java variable @foo has the value bar at the time the template is requested, bar replaces all instances of @foo on the web page. Alternatively, if I include the statement

```
#set @foo = "bar" :
```

The output will be the same for all instances of @foo that follow this directive.

Properties

The second flavor of QTL references are properties, and properties have a distinctive format. The shorthand notation consists of a leading @ character followed a QTL Identifier, followed by a dot character (".") and another QTL Identifier. These are examples of valid property references in the QTL:

```
@customer.Address
@purchase.Total
```

Take the first example, @customer.Address. It can have two meanings. It can mean, Look in the hashtable identified as customer and return the value associated with the key Address. But @customer.Address can also be referring to a method (references that refer to methods will be discussed in the next section); @customer.Address could be an abbreviated way of writing @customer.getAddress(). When your page is requested, Quik will determine which of these two possibilities makes sense, and then return the appropriate value.

Formal Reference Notation

Shorthand notation for references was used for the examples listed above, but there is also a formal notation for references, which is demonstrated below:

```
@{mudSlinger}
@{customer.Address}
```

In almost all cases you will use the shorthand notation for references, but in some cases the formal notation is required for correct processing.

Suppose you were constructing a sentence on the fly where @vice was to be used as the base word in the noun of a sentence. The goal is to allow someone to choose the base word and produce one of the two following results:

"Avelino is a Pythonmaniac." or "Avelino is a Developermaniac.". Using the shorthand notation would be inadequate for this task. Consider the following example:

```
Avelino is a @vicemaniac.
```

There is ambiguity here, and Quik assumes that @vicemaniac, not @vice, is the Identifier that you mean to use. Finding no value for @vicemaniac, it will return @vicemaniac. Using formal notation can resolve this problem.

```
Avelino is a @{vice}maniac.
```

Now Quik knows that @vice, not @vicemaniac, is the reference. Formal notation is often useful when references are directly adjacent to text in a template.

Quiet Reference Notation

When Quik encounters an undefined reference, its normal behavior is to output the image of the reference. For example, suppose the following reference appears as part of a QTL template.

```
<input type="text" name="email" value="@email"/>
```

When the form initially loads, the variable reference @email has no value, but you prefer a blank text field to one with a value of "@email". Using the quiet reference notation circumvents Quik's normal behavior; instead of using @email in the QTL you would use !@email. So the above example would look like the following:

```
<input type="text" name="email" value="@!email"/>
```

Now when the form is initially loaded and @email still has no value, an empty string will be output instead of "@email".

Formal and quiet reference notation can be used together, as demonstrated below.

```
<input type="text" name="email" value="@!{email}"/>
```

Conditionals

If / ElseIf / Else

The #if directive in Quik allows for text to be included when the web page is generated, on the conditional that the if statement is true. For example:

```
#if @foo:
   <strong>Quik!</strong>
#end
```

The variable @foo is evaluated to determine whether it is true, which will happen under one of two circumstances: (i) @foo is a boolean (true/false) which has a true value, or (ii) the value is not null. Remember that the Quik context only contains Objects, so when we say 'boolean', it will be represented as a Boolean (the class). This is true even for methods that return *boolean* - the introspection infrastructure will return a *Boolean* of the same logical value.

The content between the #if and the #end statements become the output if the evaluation is true. In this case, if @foo is true, the output will be: "Quik!". Conversely, if @foo has a null value, or if it is a boolean false, the statement evaluates as false, and there is no output.

An #elseif or #else element can be used with an #if element. Note that the Quik Templating Engine will stop at the first expression that is found to be true. In the following example, suppose that @foo has a value of 15 and @bar has a value of 6.

```
#if @foo < 10:
    <strong>Go North</strong>
#elseif @foo == 10:
    <strong>Go East</strong>
#elseif: @bar == 6:
    <strong>Go South</strong>
#else
    <strong>Go West</strong>
#end
```

In this example, @foo is greater than 10, so the first two comparisons fail. Next @bar is compared to 6, which is true, so the output is Go South.

Relational and Logical Operators

Quik has logical AND, OR and NOT operators as well. For further information, please see the QTL Reference Guide Below are examples demonstrating the use of the logical AND, OR and NOT operators.

```
## logical AND

#if @foo && @bar:
    <strong>This AND that</strong>
#end
```

The #if() directive will only evaluate to true if both @foo and @bar are true. If @foo is false, the expression will evaluate to false; @bar will not be evaluated. If @foo is true, the Quik Templating Engine will then check the value of @bar; if @bar is true, then the entire expression is true and This AND that becomes the output. If @bar is false, then there will be no output as the entire expression is false.

Logical OR operators work the same way, except only one of the references need evaluate to true in order for the entire expression to be considered true. Consider the following example.

```
## logical OR

#if @foo || @bar:
    <strong>This OR That</strong>
#end
```

If @foo is true, the Quik Templating Engine has no need to look at @bar; whether @bar is true or false, the expression will be true, and This OR That will be output. If @foo is false, however, @bar must be checked. In this case, if @bar is also false, the expression evaluates to false and there is no output. On the other hand, if @bar is true, then the entire expression is true, and the output is This OR That

With logical NOT operators, there is only one argument :

```
##logical NOT

#if !@foo :
    <strong>NOT that</strong>
#end
```

Here, the if @foo is true, then !@foo evaluates to false, and there is no output. If @foo is false, then !@foo evaluates to true and **NOT that** will be output. Be careful not to confuse this with the quiet reference @!foo which is something altogether different.

Loops

For Loop

The *#for* element allows for looping. For example:

```
<ul>
    #for @product in @products:
    <li>@product</li>
    #end
</ul>
```

This #for loop causes the @products list (the object) to be looped over for all of the products (targets) in the list. Each time through the loop, the value from @products is placed into the @product variable.

The contents of the @products variable is a List, or an QuerySet. The value assigned to the @product variable is a Python Object and can be referenced from a variable as such. For example, if @product was really a Product class in Python, its name could be retrieved by referencing the @product.name method.

Lets say that @products is a Hashtable. If you wanted to retrieve the key values for the Hashtable as well as the objects within the Hashtable, you can use code like this:

```
<ul>
    #for @product in @products:
    <li>Key: @product.id -> Value: @products.get(@product.id)</li>
    #end
</ul>
```

Include

The *#include* script element allows the template designer to import a local file, which is then inserted into the location where the #include directive is defined. The contents of the file are not rendered through the template engine. For security reasons, the file to be included may only be under TEMPLATE_ROOT.

```
#include( "content.html" )
```

The file to which the *#include* directive refers is enclosed in quotes. If more than one file will be included, they should be separated by commas.

```
#include( "content.html", "tree.xml", "quik.gif" )
```

Stop

The *#stop* script element allows the template designer to stop the execution of the template engine and return. This is useful for debugging purposes.

```
#stop
```

Macros

The *#macro* script element allows template designers to define a repeated segment of a QTL template. Macros are very useful in a wide range of scenarios both simple and complex. This macro, created for the sole purpose of saving keystrokes and minimizing typographic errors, provides an introduction to the concept of macros.

```
#macro tr:
<tr><td></tr></tr>
#end
```

The macro being defined in this example is *tr*, and it can be called in a manner analogous to any other QTL directive:

```
#tr :
```

When this template is called, Quik would replace *#tr()* with a row containing a single, empty data cell.

A macro could take any number of arguments – even zero arguments, as demonstrated in the first example, is an option – but when the macro is invoked, it must be called with the same number of arguments with which it was defined. Many macros are more involved than the one defined above. Here is a macro that takes two arguments, a color and an list.

```
#macro tablerows @color @somelist:
    #for @something in @somelist:
    <tr><td bgcolor=@color>@something</td></tr>
    #end
#end
```

The macro being defined in this example, tablerows, takes two arguments. The first argument takes the place of @color, and the second argument takes the place of @somelist.

Anything that can be put into a QTL template can go into the body of a macro. The tablerows macro is a for statement. There are two #end statements in the definition of the #tablerows macro; the first belongs to the #for, the second ends the macro definition.

```
#set @greatlakes = ["Superior","Michigan","Huron","Erie","Ontario"]:
#set @color = "blue":
<table>
    #tablerows @color @greatlakes:
</table>
```

Notice that @greatlakes takes the place of @somelist. When the #tablerows macro is called in this situation, the following output is generated:

```
<table>
    <tr><td bgcolor="blue">Superior</td></tr>
    <tr><td bgcolor="blue">Michigan</td></tr>
    <tr><td bgcolor="blue">Huron</td></tr>
    <tr><td bgcolor="blue">Erie</td></tr>
    <tr><td bgcolor="blue">Ontario</td></tr>
</table>
```

Return macro

3.1.2 Team

Quik is written and maintained by the Quik Team and various contributors:

Lead Developer

- Thiago Avelino <thiago@avelino.xxx>

Contributors

3.2 Useful Links

- Quik @ PyPI
- Quik @ Github

3.3 Indices and tables

- *genindex*
- *modindex*
- *search*