

Python-Powered Templates with Cheetah

by [Andrew Glover](#)

01/13/2005

XML transformation via XSL Transformations (XSLT) is quite popular and indeed powerful. Well-constructed XSL can produce HTML, PDF, XML, and just about any other text format imaginable. XSLT, however, requires that the subject data be a well-structured XML document, which often is not the case. Consequently, developers often transform native data structures (that is, business objects) into XML so as to use XSLT. This process, unfortunately, can increase code complexity and development time for a problem that has an easier solution: a *template engine*.

Template engines facilitate the construction of various formatted documents by allowing a static template to contain placeholders for dynamic output. Hence, there is no temporary format such as XML.

If the desired output is an HTML document, a template engine will operate on an HTML template that contains placeholders for values substituted at runtime. Using a template engine's transformation process is as simple as reading the template and providing a mapping of runtime values. The output of the transformation is the native format of the template--in this case, an HTML document. Similar to XSLT, template engines can produce any format possible, such as XML, HTML, and SQL; additionally, one can even use template engines as code generators. Template engines additionally facilitate the Model View Controller architecture by embodying the view component in templates.

Introducing Cheetah

[Cheetah](#) is an extremely effective [Python](#)-powered template engine that can generate any text-based format. Cheetah's impressive yet simple template language (based on Python) can yield the most complex of documents; moreover, Cheetah's object-oriented representation of documents creates plenty of opportunities for the reuse of code. Cheetah also possesses an impressive caching mechanism that fits a variety of performance scenarios.

Cheetah is surprisingly simple to use, as it essentially has two language constructs: *placeholders* and *directives*. Placeholders are values to substitute at runtime, and directives in effect are commands to execute at runtime. Placeholders are signified by \$ signs and directives by # signs.

Is It Really That Easy?

With placeholders and directives in mind, the code below demonstrates a simple Cheetah template for generating [PyUnit](#) test cases.

```
1 import unittest
2
3 class ${classundertest}_test(unittest.TestCase):
4     def setUp(self):
5         pass
```

```

6     def tearDown(self):
7         pass
8     #for $testcase in $testcases
9     def test${testcase}(self):
10         pass
11 #end for

```

Cheetah will substitute the placeholder `$clssundertst` at runtime to create a string, such as `query_test`, that will form the test case's class name. Additionally, notice the directive, which in this case is a `for` loop, which iterates over a collection of `testcases` to create a series of class methods with names starting with `test`.

More on \$Placeholders

To make ambiguous placeholders less ambiguous, you can surround them with `{s}`, `(s)`, `[s]`, or nothing at all. Placeholders, furthermore, can be complex objects, which Cheetah can navigate quite easily through *autocalling*.

```

1 <person>
2   <firstname>${fname}</firstname>
3   <middleinitial>${mi}</middleinitial>
4   <lastname>${lname}</lastname>
5   <dateofbirth>${p.dob}</dateofbirth>
6 </person>

```

As demonstrated in the above example, Cheetah is quite flexible when it comes to syntax. Notice that `p.dob` actually calls the `dob` attribute of the `p` object. Autocalling is quite flexible; you can use dictionaries as well as lists. Here's an example of autocalling with a dictionary:

```

1 <person>
2   <firstname>${dict.fname}</firstname>
3   <middleinitial>${dict['mi']}</middleinitial>
4   <lastname>${dict.lname}</lastname>
5   <dateofbirth>${dict.dob}</dateofbirth>
6 </person>

```

In the code above, `$dict` is a dictionary object defined something like this:

```

1 mp = {"fname": "Emily", "mi": "M", \
2       "lname": "Smith", "dob": "04/21/74"}
3 inputmap = {"dict": mp}

```

The `$dict` placeholder maps to a dictionary, `mp`, which contains the keys accessed in the previous code.

Directives

Directives are constructs that control the flow of template logic. With directives, templates can contain `if/else` logic blocks as well as looping constructs. What's more, it's possible to define Python functions through directives and call them throughout a template.

Using conditional logic is straightforward, as the syntax is Python.

```

1 #if $status == 'rejected'
2 <b>Don't call us, we'll call you</b>
3 #elif $status == 'passed'
4 We'll be calling you soon.
5 #end if

```

Looping constructs, like the `for` loop, are just as simple.

```

1 #for $person in $people
2 <TR>
3   <TD>${person.name}</TD>

```

```

5   <TD>$person.weight</TD>
6   <TD>$person.height</TD>
7 </TR>
8 #end for

```

In the above example, a list named `people` contains a collection of objects having attributes of `name`, `weight`, and `height`.

If you need repetition, use the `repeat` directive:

```

1 <p>
2 Remember, your proposal was: <br/>
3 #repeat $times
4 $status <br/>
5 #end repeat
6 </p>

```

Notice how the `#repeat` directive takes a placeholder, in this case `$times`, which represents the number of times to repeat the text in the directive's body.

Occasionally, templates may need additional logic defined in a function. Strict Model-View-Controller design tries to avoid putting too much logic in a view, but you can define functions in templates and call them throughout the template at runtime.

The following example defines a function named `caps` switches the case of the passed-in parameter. Line 8 shows how to reference the directive and pass in placeholders.

```

1 #def caps($var)
2 $var.swapcase() #slurp
3 #end def
4
5 <html>
6 <body>
7 <p>
8 Your proposal was <b> $caps($status) </b>
9 </p>

```

The `#slurp` declaration in the code above consumes the newline so the resulting output is:

```

1 <html>
2 <body>
3 <p>
4 Your proposal was <b> REJECTED </b>
5 </p>

```

Caching

As mentioned earlier, Cheetah possesses a powerful caching mechanism for use in templates. As with everything else found in Cheetah, putting it to work is easy.

In performance-intensive scenarios, caching placeholders can increase a template's rendering speed. To cache a placeholder indefinitely (or as long as the template resides in memory), use the `*` syntax.

```

1 <p>
2 Submissions will be accepted
3 for $*days calendar days so please keep
4 trying until then.
5 </p>

```

In the above example, Cheetah caches the `days` placeholder as long as the template resides in memory.

If a template has more sophisticated caching requirements, you can cache placeholders for time intervals or even cache entire template regions. To cache for a specific time limit, add the desired value in terms of

seconds, minutes, hours, days, or weeks.

```
1 <p>
2 After that, our $*4d*judges judges will
3 annouce the winners!
4 </p>
```

The above example caches the placeholder, `judges`, for four days. For seconds, use an `s`, minutes an `m`, hours an `h`, and weeks a `w`.

Caching an entire region is as simple as wrapping the region with a `#cache` directive.

```
1 #cache
2 <p>
3 Thanks again, <br/>
4 $staff
5 </p>
6 #end cache
```

Incidentally, you can also fine-tune the `#cache` directive with time intervals. See the [Cheetah documentation](#) for more details.

Using Templates

There are a few different ways to use templates. One of the easiest is to define the template in a file (commonly ending with a `.tmpl` suffix). To use it, read the template at runtime and supply it with a list of placeholder values. The list of placeholder values is a simple map, where the key should match the actual placeholder name. See the code below for an example.

```
1 tcs = ['runquery', 'executequery', 'deletequery']
2 mp = {"classundertest": "query_runner", "testcases": tcs}
3
4 from Cheetah.Template import Template
5 t = Template(file="default_pyunit.tmpl", searchList=[mp])
```

As demonstrated above, Cheetah performs placeholder mappings at template instantiation; hence, you create a template and retrieve its associated output in two lines of code (lines 4 and 5). Notice the map defined in line 2 contains keys, which presumably match placeholders found in the template, *default_pyunit.tmpl*, as shown in the code from the first example.

Cheetah treats templates as objects; moreover, these template objects are quite flexible and offer a few other usage strategies. To use templates as objects (accessing attributes, functions, and so on), you must compile them with Cheetah's `compile` command. For more information regarding compiling Cheetah templates, see the sidebar below.

A compiled template's placeholders become attributes, which can be set at runtime. Directives become executable Python code. As shown below, using a compiled template is as simple as importing it and setting the object's associated attributes.

```
1 from default_pyunit import default_pyunit
2
3 tmpl = default_pyunit()
4 tmpl.classundertest = "query_runner"
5 tmpl.testcases = ['runquery', 'executequery']
6
7 print tmpl
```

Putting It All Together

Armed with a basic knowledge of how to tap Cheetah's power, you can quickly build dynamic applications and have a good time of it! Imagine a scenario where a development team, using a

bug-tracking system (such as [Bugzilla](#)), would like a weekly email report (in HTML, of course) that summarized all new bugs. The desired viewable information for a bug is the bug's ID number, the project containing the bug, the developer to whom the bug has been assigned, and a short description of the bug.

The template for this reporting application, as it turns out, is quite easy. Given a list of `bug` objects, a `for` loop directive will iterate over the list, creating a row in an HTML table with the corresponding ID, project name, owner, and description. In addition, some logic will determine whether the collection is empty so as to display an alternate message (perhaps congratulating the team for not creating any bugs for the week!).

With the view (the template) and the model (a `bug` object) defined, the controller becomes quite simple. First, query the bug database, then fill the template, and finally send the corresponding HTML email report via SMTP.

```
1 def runreport():
2     """
3     main method to run the report
4     """
5
6     import com.vanward.roach.template.template_engine as teng
7     import com.vanward.roach.email.email_engine as eeng
8
9     buglist = _getbuglist(7)
10    bmap = {"bugs":buglist, "numbugs":len(buglist), "numdays":7}
11    message = teng.getcontent(bmap)
12
13    email = _getemail(message)
14
15    eeng.sendemail(email)
```

In the code above, the `runreport()` function does a few things. Line 9 retrieves a list of bugs from the `_getbuglist()` function, which queries a database for bugs by time. Line 10 places the returned list of bugs in a map, along with the number of bugs, the length of `buglist`, and the number of days on which the query performed a search. Line 11 retrieves the content of the report from the `teng` object (see below) and then passes it to an email engine, which sends the resulting report.

```
1 def getcontent(mapvls):
2     from Cheetah.Template import Template
3     t = Template(file=_templatename(), searchList=[mapvls])
4     return t.respond()
5
6 def _templatename():
7     import com.vanward.roach.util.properties as prop
8     tmp1 = prop.templateproperties
9     return tmp1["tmpl"]
```

The above code demonstrates a simple template engine that, when given a map of values, will instantiate a template and return the resulting body. Lines 6 through 9 define a function that returns the template's fully qualified name, such as `$/var/tmp/bug_report.tmpl`.

```
1 #include "header.txt"
2 <span>Bugzilla Weekly Summary</span>
3 #if $numbugs > 0
4 <p>Over the past $numdays days, the following bugs were created.
5 Click on the bug id to view the actual bugzilla bug report page.</p>
6 <br/>
7 <table border="0" width="400" valign="top">
8 <TR>
9     <TD>Bug Id</TD>
10    <TD>Project</TD>
11    <TD>Assigned To</TD>
12    <TD>Short Description</TD>
13 </TR>
14 #for $bug in $bugs
15 <TR>
```

```

16      <TD><A HREF="http://acme.org/show_bug.cgi?id=$bug.id">$bug.id</A></TD>
17      <TD>$bug.product</TD>
18      <TD>$bug.assignedto</TD>
19      <TD>$bug.shortdescript</TD>
20 </TR>
21 #end for
22 </table>
23 #else
24 <p>No bugs were created in bugzilla this week. </p>
25 #end if
26 #include "footer.txt"

```

The code above shows the resulting HTML bug report template. Line 1 and 26 demonstrate the `#include` directive, which as you have probably guessed by now includes text from outside a template. In this case, the report includes header and footer files, each containing a bit of static HTML. Line 3 demonstrates a simple `#if` directive, which will print some summary information if there are any bugs. The trailing `#else` and `#end if` are found on lines 23 and 25, respectively. Lines 14 through 21 use a `#for` directive to iterate over a list of bugs, creating an HTML table row for each one. Figure 1 shows the results of the Cheetah bug-reporting application.

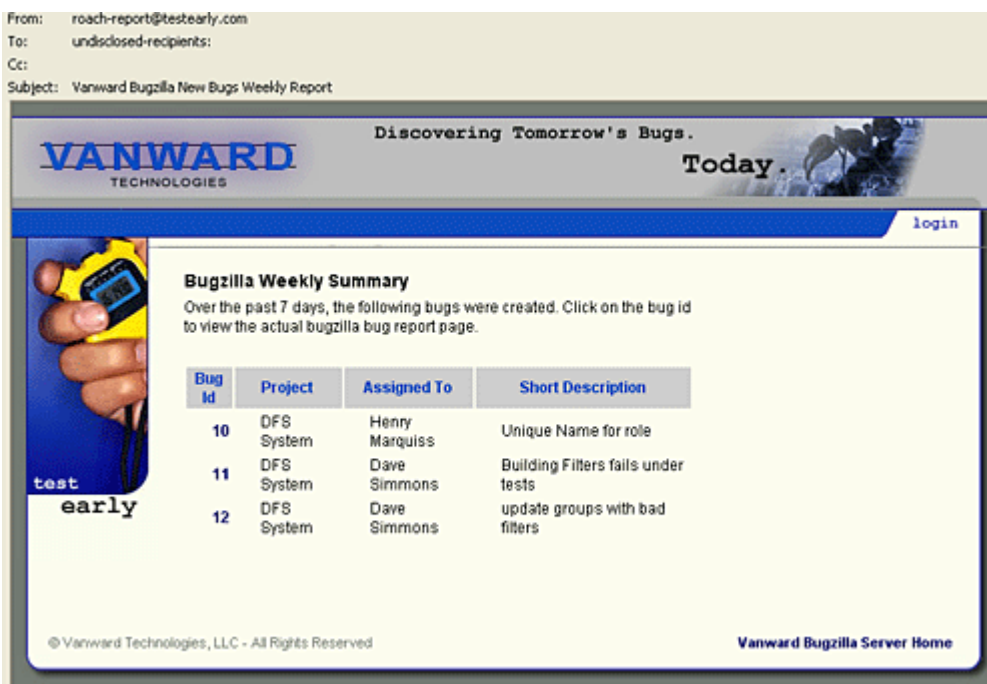


Figure 1. Output of the bug-reporting application.

Conclusion

The Cheetah template engine's simple language constructs and ease of use make data transformations a snap in terms of development time and complexity. What's more, Cheetah does not prevent architectures from using an MVC pattern, and Cheetah's built-in caching mechanism and object-oriented representation of templates yield an impressive alternative to XML transformations.

The next time you have the task of building an application with a "view," consider putting Cheetah to the test.

Resources

Cheetah offers the ability to compile template files into Python modules containing a class representing the template. These classes are easy to extend or argument. To compile a template, run the `cheetah` command with the `compile` option on the desired template:

```
$ cheetah compile default_pyunit.tpl
```

This yields a Python module with the same name as the template (in this case called `default_pyunit.py`); additionally, the module will contain a class also having the name of the template:

- [Cheetah's home](#)
- [Cheetah User Guide](#)
- [Velocity--a Java alternative](#)
- [FreeMarker--another Java alternative](#)
- [Bugzilla's home](#)

```
class default_pyunit(Template)
```

The class extends Cheetah's Template class, which is the default base class for all templates.



Related Reading

[Python & XML](#)
By [Christopher A. Jones](#), [Fred L. Drake, Jr.](#)

[Table of Contents](#)
[Index](#)
[Sample Chapter](#)

[Read Online--Safari](#)

Search this book on
Safari:

☐ Only This Book

☐ Code Fragments
only

[Andrew Glover](#) is the founder and CTO of [Vanward Technologies](#), a company specializing in building automated testing frameworks.

Return to the [Python DevCenter](#).

Copyright © 2006 O'Reilly Media, Inc.