GoogleSummerOfCode2007

Velocity @ Google Summer of Code 2007

Macro Improvements

Proposal

Velocity is a template engine written in Java. A very important feature of the velocity template language (VTL) is the ability to define Velocimacros. Velocimacros enable template writers to easily re-use the code. However, the current macro implementation requires some improvements and fixes. A common use of Velocimacros would be to define them in separate files and include them with the #parse directive. The other way to include macros defined in separate files would be the Java API.However, both methods are not supported in the current implementation. Another problem with Velocimacros is the recursive calls. When a Velocimacro calls itself recursively without meeting stopping conditions, the current implementation does not work properly. One way to solve this problem is to introduce a maximum recursion depth for Velocimacros. If the Velocimacros can be overloaded with different arguments it will be very user friendly. This allows more freedom in writing Velocimacros. This is another improvement to be made. This project will mainly focus on restructuring and adding new features to the existing Velocity macro handling code, to solve the above issues. This process inevitably involves dependencies to the other aspects of Velocity as well. So the project requires a good in depth understanding of the overall Velocity structure.

Suggestions on where to work

(I added +++ signs after each item to denote how hard I think a given item is. + == easy, ++++ = very hard)

- The current behaviour of the parser with respect to macros is a bit 'irky'. See Escaping Velocity. Getting this to behave more predictable would be great. (++)
- For macros, there is a strong distinction between template parsing time and run time. In a nutshell, it is not possible to define a macro after parsing, because the AST contains different tokens when a piece of text has been identified to be a defined macro or just 'plain text'. It would be interesting to find out whether it is possible to make the parser emit the same tokens no matter whether a macro is defined or not and just have the AST emit at run time different outputs (macro invocation if a macro exists and characters if the macro does not exist). This would allow to define macros at run time. (++++)
- Macro arguments should behave like '#set' expressions. Especially it should be possible to use Velocity expressions for macro arguments. (+ - ++)
- The 'blank as argument delimiter' convention for macros sucks. It is counterintuitive, because $foo(xxx, yyy, zzz) but #foo(xxx yyy zzz). We should have a property that switches to using ',' as delimiter for macro arguments in both definition and invocation.
- Get a real local variable scope that is nestable. This is not as difficult as it sounds because we already have that e.g. for nested #foreach loops. (++)
- I guess, by overloading you mean that you want to be able to define multiple macros with the same name but different argument counts. I do not buy into this at all. If we want to go there, we should have optional arguments and especially, we should have the ability to assign named arguments. Think Python:

```
## Optional parameters:

#macro myOptionalMacro($arg1, $arg2=100, $arg3="bar")
  ... do something with $arg1, $arg2, $arg3
#end

#myOptionalMacro("foo") ## $arg1 = "foo", $arg2 = 100, $arg3 = "bar"
#myOptionalMacro("foo", "baz") ## $arg1 = "foo", $arg2 = "baz", $arg3 = "bar"

## named parameters

#macro myNamedMacro($arg1, $arg2, $arg3)
  ... do something with $arg1, $arg2, $arg3
#end

#set ($foo = "bar")
#myNamedMacro($arg2=100, $arg3="", $arg1=$foo)
  --> $arg1 is $foo, $arg2 is an integer value 100, $arg3 is an empty String
```

Having both features combined would be a killer. 😉 (++++)

- I (henning) did some refactoring in the Macro management routines inside the engine. Still, lots of that code can use some improvement, but for 1.5 we did not want to change too much here because we wanted to stay compatible with the older releases. This is a part that can benefit from fresh ideas. (+++ for understanding the code, ++ for improving it)