About

- Overview
- Getting Started
- Download
- Install
- Design
- Contributors
- ChangeLog
- Coding Standards
- License
- TODO
- Report Issues

Community

- Powered By Velocity
- Get Involved
- Mailing Lists
- CVS Repositories

Docs

- User's Guide (English)
- User's Guide (Finnish)
- User's Guide (French)
- User's Guide (Spanish)
- Developer's Guide
- VTL Reference Guide
- Specification
- Javadoc

Table of Contents

Tools

- Tool Subproject
- Anakia : XML->doc tool
- Texen : text generation
- DVSL : XML xformation
- Veltag : JSP taglib
- Migration to Velocity
- Editors and IDEs

Comparisons

- YMTD
- VM/WM Differences
- JSP vs. Velocity
- XMLC vs. Velocity

Site Translations

- English
- Japanese

About this Guide

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

Thanks for choosing Velocity!

What is Velocity?

Velocity is a Java-based template engine. It permits web page designers to reference methods defined in Java code. Web designers can work in parallel with Java programmers to develop web sites according to the Model-View-Controller (MVC) model, meaning that web page designers can focus solely on creating a well-designed site, and programmers can focus solely on writing top-notch code. Velocity separates Java code from the web pages, making the web site more maintainable over the long run and providing a viable alternative to Java Server Pages (JSPs) or PHP.

Velocity can be used to generate web pages, SQL, PostScript and other output from templates. It can be used either as a standalone utility for generating source code and reports, or as an integrated component of other systems. When complete, Velocity will provide template services for the Turbine web application framework. Velocity+Turbine will provide a template service that will allow web applications to be developed according to a true MVC model.

What can Velocity do for me?

The Mud Store Example

Suppose you are a page designer for an online store that specializes in selling mud. Let's call it "The Online Mud Store". Business is thriving. Customers place orders for various types and quantities of mud. They login to your site using their username and password, which allows them to view their orders and buy more mud. Right now, Terracotta Mud is on sale, which is very popular. A minority of your customers regularly buys Bright Red Mud, which is also on sale, though not as popular and usually relegated to the margin of your web page. Information about each customer is tracked in your database, so one day the question arises, Why not use Velocity to target special deals on mud to the customers who are most interested in those types of mud?

Velocity makes it easy to customize web pages to your online visitors. As a web site designer at The Mud Room, you want to make the web page that the customer will see after logging into your site.

You meet with software engineers at your company, and everyone has agreed that *$customer* will hold information pertaining to the customer currently logged in, that *$mudsOnSpecial* will be all the types mud on sale at present. The *$flogger* object contains methods that help with promotion. For the task at hand, let's concern ourselves only with these three references. Remember, you don't need to worry about how the software engineers extract the necessary information from the database, you just need to know that it works. This lets you get on with your job, and lets the software engineers get on with theirs.

You could embed the following VTL statement in the web page:

```
<HTML>
<BODY>
Hello $customer.Name!
<table>
#foreach( $mud in $mudsOnSpecial )
   #if ( $customer.hasPurchased($mud) )
      <tr>
        <td>
          $flogger.getPromo( $mud )
        </td>
      </tr>
   #end
#end
</table>
```

The exact details of the *foreach* statement will be described in greater depth shortly; what's important is the impact this short script can have on your web site. When a customer with a penchant for Bright Red Mud logs in, and Bright Red Mud is on sale, that is what this customer will see, prominently displayed. If another customer with a long history of Terracotta Mud purchases logs in, the notice of a Terracotta Mud sale will be front and center. The flexibility of Velocity is enormous and limited only by your creativity.

Documented in the VTL Reference are the many other Velocity elements, which collectively give you the power and flexibility you need to make your web site a web *presence*. As you get more familiar with these elements, you will begin to unleash the power of Velocity.

Velocity Template Language (VTL): An Introduction

The Velocity Template Language (VTL) is meant to provide the easiest, simplest, and cleanest way to incorporate dynamic content in a web page. Even a web page developer with little or no programming experience should soon be capable of using VTL to incorporate dynamic content in a web site.

VTL uses *references* to embed dynamic content in a web site, and a variable is one type of reference. Variables are one type of reference that can refer to something defined in the Java code, or it can get its value from a VTL *statement* in the web page itself. Here is an example of a VTL statement that could be embedded in an HTML document:

```
#set( $a = "Velocity" )
```

This VTL statement, like all VTL statements, begins with the # character and contains a directive: *set*. When an online visitor requests your web page, the Velocity Templating Engine will search through your web page to find all # characters, then determine which mark the beginning of VTL statements, and which of the # characters that have nothing to do with VTL.

The # character is followed by a directive, *set*. The *set* directive uses an expression (enclosed in brackets) -- an equation that assigns a *value* to a *variable*. The variable is listed on the left hand side and its value on the right hand side; the two are separated by an = character.

In the example above, the variable is *$a* and the value is *Velocity*. This variable, like all references, begins with the *$* character. Values are always enclosed in quotes; with Velocity there is no confusion about data types, as only strings (text-based information) may be passed to variables.

The following rule of thumb may be useful to better understand how Velocity works: **References begin with *$* and are used to get something. Directives begin with # and are used to do something.**

In the example above, *#set* is used to assign a value to a variable. The variable, *$a*, can then be used in the template to output "Velocity".

Hello Velocity 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 = "Velocity" )
Hello $foo World!
</body>
<html>
```

The result is a web page that prints "Hello Velocity World!".

To make statements containing VTL directives more readable, we encourage you to start each VTL 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 VTL statements are doing, or any other purpose you find useful. Below is an example of a comment in VTL.

```
## 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 Velocity 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 VTL comment block, which may be used to store such information as the document author and versioning information:

```
#**
This is a VTL 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 VTL: variables, properties and methods. As a designer using the VTL, 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 Velocity 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 VTL *Identifier*. A VTL 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 VTL:

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

When VTL references a variable, such as *$foo*, the variable can get its value from either a *set* directive in the template, or from the Java code. For example, if the Java variable *$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 VTL references are properties, and properties have a distinctive format. The shorthand notation consists of a leading *$* character followed a VTL Identifier, followed by a dot character (".") and another VTL Identifier. These are examples of valid property references in the VTL:

```
$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, Velocity will determine which of these two possibilities makes sense, and then return the appropriate value.

Methods
A method is defined in the Java code and is capable of doing something useful, like running a calculation or arriving at a decision. Methods are references that consist of a leading "$" character followed a VTL Identifier, followed by a VTL *Method Body*. A VTL Method Body consists of a VTL Identifier followed by an left parenthesis character ("("),

followed by an optional parameter list, followed by right parenthesis character (")"). These are examples of valid method references in the VTL:

```
$customer.getAddress()
$purchase.getTotal()
$page.setTitle( "My Home Page" )
$person.setAttributes( ["Strange", "Weird", "Excited"] )
```

The first two examples -- *$customer.getAddress()* and *$purchase.getTotal()* -- may look similar to those used in the Properties section above, *$customer.Address* and *$purchase.Total*. If you guessed that these examples must be related some in some fashion, you are correct!

VTL Properties can be used as a shorthand notation for VTL Methods. The Property *$customer.Address* has the exact same effect as using the Method *$customer.getAddress()*. It is generally preferable to use a Property when available. The main difference between Properties and Methods is that you can specify a parameter list to a Method.

The shorthand notation can be used for the following Methods

```
$sun.getPlanets()
$annelid.getDirt()
$album.getPhoto()
```

We might expect these methods to return the names of planets belonging to the sun, feed our earthworm, or get a photograph from an album. Only the long notation works for the following Methods.

```
$sun.getPlanet( ["Earth", "Mars", "Neptune"] )
## Can't pass a parameter list with $sun.Planets

$sisyphus.pushRock()
## Velocity assumes I mean $sisyphus.getRock()

$book.setTitle( "Homage to Catalonia" )
## Can't pass a parameter list
```

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}
${purchase.getTotal()}
```

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: "Jack is a pyromaniac." or "Jack is a kleptomaniac.". Using the shorthand notation would be inadequate for this task. Consider the following example:

```
Jack is a $vicemaniac.
```

There is ambiguity here, and Velocity 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.

```
Jack is a ${vice}maniac.
```

Now Velocity 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 Velocity 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 VTL 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 Velocity's normal behavior; instead of using *$email* in the VTL you would use *$!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}"/>
```

Getting literal

VTL uses special characters, such as *$* and *#*, to do its work, so some added care should be taken where using these characters in your templates. This section deals with escaping the *$* character.

Currency
There is no problem writing "I bought a 4 lb. sack of potatoes at the farmer's market for only $2.50!" As mentioned, a VTL identifier always begins with an upper- or lowercase letter, so $2.50 would not be mistaken for a reference.

Escaping Valid VTL References
Cases may arise where there is the potential for Velocity to get confused. *Escaping* special characters is the best way to handle VTL's special characters in your templates, and this can be done using the backslash (\ ) character.

```
#set( $email = "foo" )
$email
```

If Velocity encounters a reference in your VTL template to *$email*, it will search the Context for a corresponding value. Here the output will be *foo*, because *$email* is defined. If *$email* is not defined, the output will be *$email*.

Suppose that *$email* is defined (for example, if it has the value *foo*), and that you want to output *$email*. There are a few ways of doing this, but the simplest is to use the escape character.

```
## The following line defines $email in this template:
#set( $email = "foo" )
$email
\$email
\\$email
\\\$email
```

renders as

```
foo
$email
\foo
\$email
```

Note that the \ character bind to the *$* from the left. The bind-from-left rule causes *\\\$email* to render as *\\$email*. Compare these examples to those in which *$email* is not defined.

```
$email
\$email
\\$email
\\\$email
```

renders as

```
$email
\$email
\\$email
\\\$email
```

Notice Velocity handles references that are defined differently from those that have not been defined. Here is a set directive that gives *$foo* the value *gibbous*.

```
#set( $foo = "gibbous" )
$moon = $foo
```

The output will be: *$moon = gibbous* -- where *$moon* is output as a literal because it is undefined and *gibbous* is output in place of *$foo*.

It is also possible to escape VTL directives; this is described in more detail in the Directives section.

Case Substitution

Now that you are familiar with references, you can begin to apply them effectively in your templates. Velocity references take advantage of some Java principles that template designers will find easy to use. For example:

```
$foo

$foo.getBar()
## is the same as
$foo.Bar

$data.getUser("jon")
## is the same as
$data.User("jon")

$data.getRequest().getServerName()
## is the same as
$data.Request.ServerName
## is the same as
${data.Request.ServerName}
```

These examples illustrate alternative uses for the same references. Velocity takes advantage of Java's introspection and bean features to resolve the reference names to both objects in the Context as well as the objects methods. It is possible to embed and evaluate references almost anywhere in your template.

Velocity, which is modelled on the Bean specifications defined by Sun Microsystems, is case sensitive; however, its developers have strove to catch and correct user errors wherever possible. When the method *getFoo()* is referred to in a template by `$bar.foo`, Velocity will first try `$getfoo`. If this fails, it will then try `$getFoo`. Similarly, when a template refers to `$bar.Foo`, Velocity will try *$getFoo()* first and then try *getfoo()*.

Note: *References to instance variables in a template are not resolved.* Only references to the attribute equivalents of JavaBean getter/setter methods are resolved (i.e. `$foo.Name` does resolve to the class Foo's `getName()` instance

method, but not to a public `Name` instance variable of Foo).

Directives

References allow template designers to generate dynamic content for web sites, while *directives* -- easy to use script elements that can be used to creatively manipulate the output of Java code -- permit web designers to truly take charge of the appearance and content of the web site.

#set

The *#set* directive is used for setting the value of a reference. A value can be assigned to either a variable reference or a property reference, and this occurs in brackets, as demonstrated:

```
#set( $primate = "monkey" )
#set( $customer.Behavior = $primate )
```

The left hand side (LHS) of the assignment must be a variable reference or a property reference. The right hand side (RHS) can be one of the following types:

- Variable reference
- String literal
- Property reference
- Method reference
- Number literal
- ArrayList

These examples demonstrate each of the aforementioned types:

```
#set( $monkey = $bill ) ## variable reference
#set( $monkey.Friend = "monica" ) ## string literal
#set( $monkey.Blame = $whitehouse.Leak ) ## property reference
#set( $monkey.Plan = $spindoctor.weave($web) ) ## method reference
#set( $monkey.Number = 123 ) ##number literal
#set( $monkey.Say = ["Not", $my, "fault"] ) ## ArrayList
```

NOTE: In the last example the elements defined with the [..] operator are accessible using the methods defined in the ArrayList class. So, for example, you could access the first element above using $monkey.Say.get(0).

The RHS can also be a simple arithmetic expression:

```
#set( $value = $foo + 1 )
#set( $value = $bar - 1 )
#set( $value = $foo * $bar )
#set( $value = $foo / $bar )
```

If the RHS is a property or method reference that evaluates to *null*, it will **not** be assigned to the LHS. It is not possible to remove an existing reference from the context via this mechanism. This can be confusing for newcomers to Velocity. For example:

```
#set( $result = $query.criteria("name") )
The result of the first query is $result

#set( $result = $query.criteria("address") )
The result of the second query is $result
```

If *$query.criteria("name")* returns the string "bill", and *$query.criteria("address")* returns *null*, the above VTL will render as the following:

```
The result of the first query is bill

The result of the second query is bill
```

This tends to confuse newcomers who construct *#foreach* loops that attempt to *#set* a reference via a property or method reference, then immediately test that reference with an *#if* directive. For example:

```
#set( $criteria = ["name", "address"] )

#foreach( $criterion in $criteria )

    #set( $result = $query.criteria($criterion) )

    #if( $result )
        Query was successful
    #end

#end
```

In the above example, it would not be wise to rely on the evaluation of *$result* to determine if a query was successful. After *$result* has been *#set* (added to the context), it cannot be set back to *null* (removed from the context). The details of the *#if* and *#foreach* directives are covered later in this document.

One solution to this would be to pre-set *$result* to *false*. Then if the *$query.criteria()* call fails, you can check.

```
#set( $criteria = ["name", "address"] )

#foreach( $criterion in $criteria )

    #set( $result = false )
    #set( $result = $query.criteria($criterion) )

    #if( $result )
        Query was successful
    #end

#end
```

Unlike some of the other Velocity directives, the *#set* directive does not have an *#end* statement.

String Literals

When using the *#set* directive, string literals that are enclosed in double quote characters will be parsed and rendered, as shown:

```
#set( $directoryRoot = "www" )
#set( $templateName = "index.vm" )
#set( $template = "$directoryRoot/$templateName" )
$template
```

The output will be

```
www/index.vm
```

However, when the string literal is enclosed in single quote characters, it will not be parsed:

```
#set( $foo = "bar" )
$foo
#set( $blargh = '$foo' )
$blargh
```

```
  bar
  $foo
```

By default, this feature of using single quotes to render unparsed text is available in Velocity. This default can be changed by editing `velocity.properties` such that `stringliterals.interpolate=false`.

Conditionals

If / ElseIf / Else

The *#if* directive in Velocity 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>Velocity!</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 Velocity 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 *$foo* is true, the output will be: "Velocity!". Conversely, if *$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 Velocity 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**.

Please note that currently, Velocity's numeric comparisons are constrained to Integers - anything else will evaluate to *false*. The only exception to this is equality '==', where Velocity requires that the objects on each side of the '==' is of the *same* class.

Relational and Logical Operators

Velocity uses the equivalent operator to determine the relationships between variables. Here is a simple example to illustrate how the equivalent operator is used.

```
#set ($foo = "deoxyribonucleic acid")
#set ($bar = "ribonucleic acid")

#if ($foo == $bar)
  In this case it's clear they aren't equivalent. So...
#else
  They are not equivalent and this will be the output.
#end
```

Velocity has logical AND, OR and NOT operators as well. For further information, please see the VTL 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 *$foo* is true, the Velocity Templating Engine will then check the value of *$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 *$foo* is true, the Velocity Templating Engine has no need to look at *$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

Foreach Loop

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

```
<ul>
#foreach( $product in $allProducts )
    <li>$product</li>
#end
</ul>
```

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

The contents of the *$allProducts* variable is a Vector, a Hashtable or an Array. The value assigned to the *$product* variable is a Java Object and can be referenced from a variable as such. For example, if *$product* was really a Product class in Java, its name could be retrieved by referencing the *$product.Name* method (ie: *$Product.getName()*).

Lets say that *$allProducts* 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>
#foreach( $key in $allProducts.keySet() )
    <li>Key: $key -> Value: $allProducts.get($key)</li>
#end
</ul>
```

Velocity provides an easy way to get the loop counter so that you can do something like the following:

```
<table>
#foreach( $customer in $customerList )
    <tr><td>$velocityCount</td><td>$customer.Name</td></tr>
#end
</table>
```

The default name for the loop counter variable reference, which is specified in the velocity.properties file, is $velocityCount. By default the counter starts at 1, but this can be set to either 0 or 1 in the `velocity.properties` file. Here's what the loop counter properties section of the `velocity.properties` file appears:

```
# Default name of the loop counter
# variable reference.
directive.foreach.counter.name = velocityCount

# Default starting value of the loop
# counter variable reference.
directive.foreach.counter.initial.value = 1
```

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( "one.txt" )
```

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( "one.gif","two.txt","three.htm" )
```

The file being included need not be referenced by name; in fact, it is often preferable to use a variable instead of a filename. This could be useful for targeting output according to criteria determined when the page request is submitted. Here is an example showing both a filename and a variable.

```
#include( "greetings.txt", $seasonalstock )
```

Parse

The *#parse* script element allows the template designer to import a local file that contains VTL. Velocity will parse the VTL and render the template specified.

```
#parse( "me.vm" )
```

Like the *#include* directive, *#parse* can take a variable rather than a template. Any templates to which *#parse* refers must be included under TEMPLATE_ROOT. Unlike the *#include* directive, *#parse* will only take a single argument.

VTL templates can have *#parse* statements referring to templates that in turn have *#parse* statements. By default set to 10, the *parse_directive.maxdepth* line of the `velocity.properties` allows users to customize maximum number of *#parse* referrals that can occur from a single template. (Note: If the *parse_directive.maxdepth* property is absent from the `velocity.properties` file, Velocity will set this default to 10.) Recursion is permitted, for example, if the template `dofoo.vm` contains the following lines:

```
Count down.
#set( $count = 8 )
#parse( "parsefoo.vm" )
All done with dofoo.vm!
```

It would reference the template `parsefoo.vm`, which might contain the following VTL:

```
$count
#set( $count = $count - 1 )
#if( $count > 0 )
    #parse( "parsefoo.vm" )
#else
    All done with parsefoo.vm!
#end
```

After "Count down." is displayed, Velocity passes through `parsefoo.vm`, counting down from 8. When the count reaches 0, it will display the "All done with parsefoo.vm!" message. At this point, Velocity will return to `dofoo.vm` and output the "All done with dofoo.vm!" message.

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
```

Velocimacros

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

```
#macro( d )
<tr><td></td></tr>
#end
```

The Velocimacro being defined in this example is *d*, and it can be called in a manner analogous to any other VTL directive:

```
#d()
```

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

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

```
#macro( tablerows $color $somelist )
#foreach( $something in $somelist )
    <tr><td bgcolor=$color>$something</td></tr>
#end
#end
```

The Velocimacro 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 VTL template can go into the body of a Velocimacro. The *tablerows* Velocimacro is a *foreach* statement. There are two *#end* statements in the definition of the *#tablerows* Velocimacro; the first belongs to the *#foreach*, the second ends the Velocimacro 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* Velocimacro 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>
```

Velocimacros can be defined *inline* in a Velocity template, meaning that it is unavailable to other Velocity templates on the same web site. Defining a Velocimacro such that it can be shared by all templates has obvious advantages: it reduces the need to redefine the Velocimacro on numerous templates, saving work and reducing the chance of error, and ensures that a single change to a macro available to more than one template.

Were the *#tablerows($color $list)* Velocimacro defined in a Velocimacros template library, this macro could be used on any of the regular templates. It could be used many times and for many different purposes. In the template `mushroom.vm` devoted to all things fungi, the *#tablerows* Velocimacro could be invoked to list the parts of a typical mushroom:

```
#set( $parts = ["volva","stipe","annulus","gills","pileus"] )
#set( $cellbgcol = "#CC00FF" )
<table>
#tablerows( $cellbgcol $parts )
</table>
```

When fulfilling a request for `mushroom.vm`, Velocity would find the *#tablerows* Velocimacro in the template library (defined in the `velocity.properties` file) and generate the following output:

```
<table>
    <tr><td bgcolor="#CC00FF">volva</td></tr>
    <tr><td bgcolor="#CC00FF">stipe</td></tr>
    <tr><td bgcolor="#CC00FF">annulus</td></tr>
    <tr><td bgcolor="#CC00FF">gills</td></tr>
    <tr><td bgcolor="#CC00FF">pileus</td></tr>
</table>
```

Velocimacro Arguments

Velocimacros can take as arguments any of the following VTL elements :

- Reference : anything that starts with '$'
- String literal : something like "$foo" or 'hello'
- Number literal : 1, 2 etc
- IntegerRange : [1..2] or [$foo .. $bar]
- ObjectArray : ["a", "b", "c"]
- boolean value true
- boolean value false

When passing references as arguments to Velocimacros, please note that references are passed 'by name'. This means that their value is 'generated' at each use inside the Velocimacro. This feature allows you to pass references with method calls and have the method called at each use. For example, when calling the following Velocimacro as shown

```
    #macro( callme $a )
        $a $a $a
    #end

    #callme( $foo.bar() )
```

results in the method bar() of the reference $foo being called 3 times.

At first glance, this feature appears surprising, but when you take into consideration the original motivation behind Velocimacros -- to eliminate cut'n'paste duplication of commonly used VTL -- it makes sense. It allows you to do things like pass stateful objects, such as an object that generates colors in a repeating sequence for coloring table rows, into the Velocimacro.

If you need to circumvent this feature, you can always just get the value from the method as a new reference and pass that :

```
    #set( $myval = $foo.bar() )
    #callme( $myval )
```

Velocimacro Properties

Several lines in the `velocity.properties` file allow for flexible implementation of Velocimacros. Note that these are also documented in the Developer Guide.

`velocimacro.library` - A comma-separated list of all Velocimacro template libraries. By default, Velocity looks for a single library: *VM_global_library.vm*. The configured template path is used to find the Velocimacro libraries.

`velocimacro.permissions.allow.inline` - This property, which has possible values of true or false, determines whether Velocimacros can be defined in regular templates. The default, true, allows template designers to define Velocimacros in the templates themselves.

`velocimacro.permissions.allow.inline.to.replace.global` - With possible values of true or false, this property allows the user to specify if a Velocimacro defined inline in a template can replace a globally defined template, one that was loaded on startup via the `velocimacro.library` property. The default, `false`, prevents Velocimacros defined inline in a template from replacing those defined in the template libraries loaded at startup.

`velocimacro.permissions.allow.inline.local.scope` - This property, with possible values of true or false, defaulting to false, controls if Velocimacros defined inline are 'visible' only to the defining template. In other words, with this property set to true, a template can define inline VMs that are usable only by the defining template. You can use this for fancy VM tricks - if a global VM calls another global VM, with inline scope, a template can define a private implementation of the second VM that will be called by the first VM when invoked by that template. All other templates are unaffected.

`velocimacro.context.localscope` - This property has the possible values true or false, and the default is false. When true, any modifications to the context via #set() within a Velocimacro are considered 'local' to the Velocimacro, and will not permanently affect the context.

`velocimacro.library.autoreload` - This property controls Velocimacro library autoloading. The default value is `false`. When set to `true` the source Velocimacro library for an invoked Velocimacro will be checked for changes, and reloaded if necessary. This allows you to change and test Velocimacro libraries without having to restart your application or servlet container, just like you can with regular templates. This mode only works when caching is *off* in the resource loaders (e.g. `file.resource.loader.cache = false`). This feature is intended for development, not for production.

Velocimacro Trivia

Currently, Velocimacros must be defined before they are first used in a template. This means that your #macro() declarations should come before using the Velocimacros.

This is important to remember if you try to #parse() a template containing inline #macro() directives. Because the #parse() happens at runtime, and the parser decides if a VM-looking element in the template is a VM at parsetime, #parse()-ing a set of VM declarations won't work as expected. To get around this, simply use the `velocimacro.library` facility to have Velocity load your VMs at startup.

Escaping VTL Directives

VTL directives can be escaped with the backslash character ("\") in a manner similar to valid VTL references.

```
## #include( "a.txt" ) renders as <contents of a.txt>
#include( "a.txt" )

## \#include( "a.txt" ) renders as \#include( "a.txt" )
\#include( "a.txt" )

## \\#include ( "a.txt" ) renders as \<contents of a.txt>
\\#include ( "a.txt" )
```

Extra care should be taken when escaping VTL directives that contain multiple script elements in a single directive (such as in an if-else-end statements). Here is a typical VTL if-statement:

```
#if( $jazz )
    Vyacheslav Ganelin
#end
```

If *$jazz* is true, the output is

```
Vyacheslav Ganelin
```

If *$jazz* is false, there is no output. Escaping script elements alters the output. Consider the following case:

```
\#if( $jazz )
    Vyacheslav Ganelin
\#end
```

Whether *$jazz* is true or false, the output will be

```
 #if($ jazz )
     Vyacheslav Ganelin
 #end
```

In fact, because all script elements are escaped, *$jazz* is never evaluated for it's boolean value. Suppose backslashes precede script elements that are legitimately escaped:

```
\\#if( $jazz )
   Vyacheslav Ganelin
\\#end
```

In this case, if *$jazz* is true, the output is

```
\ Vyacheslav Ganelin
\
```

To understand this, note that the `#if( arg )` when ended by a newline (return) will omit the newline from the output. Therefore, the body of the `#if()` block follows the first '\', rendered from the '\\' preceding the `#if()`. The last \ is on a different line than the text because there is a newline after 'Ganelin', so the final \\, preceding the `#end` is part of the body of the block.

If *$jazz* is false, there is no output. Note that things start to break if script elements are not properly escaped.

```
\\\#if( $jazz )
    Vyacheslave Ganelin
\\#end
```

Here the *#if* is escaped, but there is an *#end* remaining; having too many endings will cause a parsing error.

VTL: Formatting Issues

Although VTL in this user guide is often displayed with newlines and whitespaces, the VTL shown below

```
#set( $imperial = ["Munetaka","Koreyasu","Hisakira","Morikune"] )
#foreach( $shogun in $imperial )
    $shogun
#end
```

is equally valid as the following snippet that Geir Magnusson Jr. posted to the Velocity user mailing list to illustrate a completely unrelated point:

```
Send me #set($foo = ["$10 and ","a cake"])#foreach($a in $foo)$a #end please.
```

Velocity's behaviour is to gobble up excess whitespace. The preceding directive can be written as:

```
Send me
#set( $foo = ["$10 and ","a cake"] )
#foreach( $a in $foo )
$a
#end
please.
```

or as

```
Send me
#set($foo       = ["$10 and ","a cake"])
                  #foreach           ($a in $foo )$a
         #end please.
```

In each case the output will be the same.

Other Features and Miscellany

Math

Velocity has a handful of built-in mathematical functions that can be used in templates with the *set* directive. The following equations are examples of addition, subtraction, multiplication and division, respectively:

```
#set( $foo = $bar + 3 )
#set( $foo = $bar - 4 )
#set( $foo = $bar * 6 )
#set( $foo = $bar / 2 )
```

When a division operation is performed, the result will be an integer. Any remainder can be obtained by using the modulus (%) operator.

```
#set( $foo = $bar % 5 )
```

Only integers (...-2, -1, 0, 1, 2...) are permissible when performing mathematical equations in Velocity; when a non-integer is used, it is logged and a null will be returned as the output.

Range Operator

The range operator can be used in conjunction with *#set* and *#foreach* statements. Useful for its ability to produce an object array containing integers, the range operator has the following construction:

```
[n..m]
```

Both *n* and *m* must either be or produce integers. Whether *m* is greater than or less than *n* will not matter; in this case the range will simply count down. Examples showing the use of the range operator as provided below:

```
First example:
#foreach( $foo in [1..5] )
$foo
#end

Second example:
#foreach( $bar in [2..-2] )
$bar
#end

Third example:
#set( $arr = [0..1] )
#foreach( $i in $arr )
$i
#end

Fourth example:
[1..3]
```

Produces the following output:

```
First example:
1 2 3 4 5

Second example:
2 1 0 -1 -2

Third example:
0 1

Fourth example:
[1..3]
```

Note that the range operator only produces the array when used in conjunction with *#set* and *#foreach* directives, as demonstrated in the fourth example.

Web page designers concerned with making tables a standard size, but where some will not have enough data to fill the table, will find the range operator particularly useful.

Advanced Issues: Escaping and !

When a reference is silenced with the *!* character and the *!* character preceded by an \ escape character, the reference is handled in a special way. Note the differences between regular escaping, and the special case where \ precedes *!* follows it:

```
#set( $foo = "bar" )
$\!foo
$\!{foo}
$\\!foo
$\\\!foo
```

This renders as:

```
$!foo
$!{foo}
$\!foo
$\\!foo
```

Contrast this with regular escaping, where \ precedes *$*:

```
\$foo
\$!foo
\$!{foo}
\\$!{foo}
```

This renders as:

```
\$foo
\$!foo
\$!{foo}
\bar
```

Velocimacro Miscellany

This section is a mini-FAQ on topics relating to Velocimacros. This section will change over time, so it's worth checking for new information from time to time.

Note : Throughout this section, 'Velocimacro' will commonly be abbreviated as 'VM'.

Can I use a directive or another VM as an argument to a VM?

Example : `#center( #bold("hello") )`

No. A directive isn't a valid argument to a directive, and for most practical purposes, a VM is a directive.

However..., there are things you can do. One easy solution is to take advantage of the fact that 'doublequote' (") renders it's contents. So you could do something like

```
#set($stuff = "#bold('hello')" )
#center( $stuff )
```

You can save a step...

```
#center( "#bold( 'hello' )" )
```

Please note that in the latter example the arg is evaluated *inside* the VM, not at the calling level. In other words, the argument to the VM is passed in in its entirety and evaluated within the VM it was passed into. This allows you to do things like :

```
#macro( inner $foo )
  inner : $foo
#end

#macro( outer $foo )
   #set($bar = "outerlala")
   outer : $foo
#end

#set($bar = 'calltimelala')
#outer( "#inner($bar)" )
```

```
```

Where the output is

```
Outer : inner : outerlala
```

because the evaluation of the "#inner($bar)" happens inside #outer(), so the $bar value set inside #outer() is the one that's used.

This is an intentional and jealously guarded feature - args are passed 'by name' into VMs, so you can hand VMs things like stateful references such as

```
#macro( foo $color )
  <tr bgcolor=$color><td>Hi</td></tr>
  <tr bgcolor=$color><td>There</td></tr>
#end

#foo( $bar.rowColor() )
```

And have rowColor() called repeatedly, rather than just once. To avoid that, invoke the method outside of the VM, and pass the value into the VM.

```
#set($color = $bar.rowColor())
#foo( $color )
```

Can I register Velocimacros via #parse() ?

Currently, Velocimacros must be defined before they are first used in a template. This means that your #macro() declarations should come before using the Velocimacros.

This is important to remember if you try to #parse() a template containing inline #macro() directives. Because the #parse() happens at runtime, and the parser decides if a VM-looking element in the template is a VM at parsetime, #parse()-ing a set of VM declarations won't work as expected. To get around this, simply use the `velocimacro.library` facility to have Velocity load your VMs at startup.

What is Velocimacro Autoreloading?

There is a property, meant to be used in *development*, not production :

```
velocimacro.library.autoreload
```

which defaults to false. When set to true *along with*

```
<type>.resource.loader.cache = false
```

(where <type> is the name of the resource loader that you are using, such as 'file') then the Velocity engine will automatically reload changes to your Velocimacro library files when you make them, so you do not have to dump the servlet engine (or application) or do other tricks to have your Velocimacros reloaded.

Here is what a simple set of configuration properties would look like.

```
file.resource.loader.path = templates
file.resource.loader.cache = false
velocimacro.library.autoreload = true
```

Don't keep this on in production.

String Concatenation

A common question that developers ask is *How do I do String concatenation? Is there any analogue to the '+' operator in Java?*.

To do concatenation of references in VTL, you just have to 'put them together'. The context of where you want to put them together does matter, so we will illustrate with some examples.

In the regular 'schmoo' of a template (when you are mixing it in with regular content) :

```
#set( $size = "Big" )
#set( $name = "Ben" )

The clock is $size$name.
```

and the output will render as 'The clock is BigBen'. For more interesting cases, such as when you want to concatenate strings to pass to a method, or to set a new reference, just do

```
#set( $size = "Big" )
#set( $name = "Ben" )

#set($clock = "$size$name" )

The clock is $clock.
```

Which will result in the same output. As a final example, when you want to mix in 'static' strings with your references, you may need to use 'formal references' :

```
#set( $size = "Big" )
#set( $name = "Ben" )

#set($clock = "${size}Tall$name" )

The clock is $clock.
```

Now the output is 'The clock is BigTallBen'. The formal notation is needed so the parser knows you mean to use the reference '$size' versus '$sizeTall' which it would if the '{}' weren't there.

Feedback

If you encounter any mistakes in this manual or have other feedback related to the Velocity User Guide, please email the Velocity user list. Thanks!