Velocity Users Guide

The Jakarta Velocity Developers

Version 1.5

Table of Contents

1. Preface

1.1 About this Guide

The Velocity Users 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!

1.2 Acknowledgements

This guide has been compiled from the xdoc Version which was included with Velocity up to version 1.4. We explicitly acknowledge the efforts of all the contributors, especially Jason Van Zyl and John Castura in preparing this first version.

1.3 Feedback

If you encounter any mistakes in this manual or have other feedback related to the Velocity Users Guide, please send email to one of the Velocity mailing lists:

`<velocity-user@jakarta.apache.org>`
> This mailing list is intended for discussion about using Velocity, e.g. writing templates or integrating Velocity into your application. If you need clarification about the content of this guide or have questions about the examples, please send them to this list.

`<velocity-dev@jakarta.apache.org>`
> This mailing list is intended for discussion about developing Velocity itself. If you found a mistake in this guide, want to improve its contents or the formatting framework, please use this mailing list.

Please check also the Velocity Homepage located at `http://jakarta.apache.org/velocity/` for up-to-date information, new releases and general Velocity information.

2. What is Velocity?

Velocity is a Java-based template engine. It permits web page designers to reference methods defined in Java code. Template developers can work in parallel with Java programmers to develop web sites according to the Model-View-Controller (MVC) model, meaning that they can focus solely on creating templates, and programmers can focus solely on writing Java code. Velocity separates Java code from the template pages, making a project more maintainable over the long run.

In a web context, Velocit provides a viable alternative to Java Server Pages (JSPs) or PHP.

Velocity can be used to generate web pages, Java source code, 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.

Velocity provides template services for a number of web frameworks, e.g. the Turbine web application framework. In this contect, Velocity offers a template service that will allow web applications to be developed according to a true MVC model.

2.1 The Fruit Store Example

Suppose you are a page designer for an online store that specializes in selling fruits. Let's call it *The Online Fruit Store*. Customers place orders for various types and quantities of fruits. They login to your site using their username and password, which allows them to view their orders and buy more fruits. Right now, apples on sale, which are very popular. A minority of your customers regularly buy mangos, which are 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 fruit to the customers who are most interested in those types of fruits?*

Velocity makes it easy to customize web pages to your online visitors. As a web site designer at *The Online Fruit Store*, 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 `$customer` will hold information pertaining to the customer currently logged in, that `$fruitSpecial` will be all the types of fruit on sale at present. The `$promoTool` 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( $fruit in $fruitSpecial )
    #if ( $customer.hasPurchased($fruit) )
      <tr>
        <td>
          $promoTool.getPromo($fruit)
        </td>
```

```
      </tr>
    #end
  #end
</table>
</body></html>
```

The exact details of the `#foreach` and `#if` statement will be described in greater depth in the Directives chapter; what's important is the impact this short script may have on your web site. When a customer with a penchant for mangos logs in, and mangos are on sale, that is what this customer will see, prominently displayed. If another customer with a long history of apple purchases logs in, the notice of an apple sale will be front and center. The flexibility of Velocity is enormous and limited only by your creativity.

This manual should help you get started with Velocity and the Velocity Template Language. It has a companion manual, *The Velocity Developers Guide*, which describes how to integrate Velocity into your own web applications (this is what the software engineers at your company want to read).

2.2 Velocity Template Language (VTL): An Introduction

The *Velocity Template Language* (*VTL*) is meant to provide an easy, simple and clean 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. They can refer to something defined in the Java code, or it can get its value from a VTL *statement* in the web page itself.

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

Example 2.1 A simple VTL statement

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. String values are always enclosed in quotes, either single or double quotes. Single quotes will ensure that the quoted value will be assigned to the reference as is. Double quotes allow you to use velocity references and directives to interpolate, such as `"Hello $name"`, where the $name will be replaced by the current value before that string literal is assigned to the left hand side of the =.

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.

2.3 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( $name = "Velocity" )
Hello $name World!
</body></html>
```

The result is a web page that reads

```
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.

3. Language elements

Like other formal languages, Velocity consists of a number of elements:

- Statements and Directives

- References

- Comments

Each of these elements will be introduced in this chapter.

3.1 Statements and Directives

Velocity directives are language elements than allow the template designer to manipulate the rendering of the template. A directive is part of a Velocity statement:

```
#set($a = 10)            ## Velocity single-line statement
^\_/_____/
| |      |
| |      +---------- expression in brackets
| |
| +---------------- Velocity directive
|
+----------------- begin of Velocity statement




+----------------- begin of Velocity statement
|
|    +-------------- Velocity directive
|    |
|    |         +--- expression in brackets
| __|__  _____|_____
v/     \/           \
#foreach($i in [0..10])   ## Velocity multi-line statement
  Counter is $i           ## statement body
#end                      ## end of Velocity multi-line statement
```

Velocity knows about single-line and multi-line statements. A single-line statement starts with a single # character, followed by directive itself. Some directives also take parameters in brackets. A single-line statement then ends immediately.

```
#set($name = "apple")

#include("header.html")

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

```
#stop
```

Example 3.1 Velocity single-line statements

A multi-line statement starts like a single-line statement, but also has a statement body. This body ends with `#end`.[1]

```
#foreach($i in [1..10])
  The count is $i
#end

#literal
 This is a literal block of text
#end

#if ($fruit == "apple")
  This is an apple.
#else
  This is not an apple.
#end
```

Example 3.2 Velocity multi-line statements

> **Note**
>
> The `#if`/`#else`/`#end` combo is a special case because it might contain multiple statement bodies between its elements.

If you need to explicitly separate a Velocity directive from surrounding text, it is possible to wrap it in curly braces ({ and }):

```
#{set}($name = "apple")

#{foreach}($i in [1..10])
 The count is $i
#{end}
```

All available Velocity directives are listed in the Directives chapter.

> **Note**
>
> While the hash sign is technically not part of the directive, we will still speak about the `#if` or `#set` directive in the latter chapters.

---

[1]Technically speaking, the opening statement and `#end` can be on the same line. We will still call it a multi-line statement.

3.2 References

Velocity references are the glue which connect templates to the application. Template designers and application developers must agree on a common set of references used by the Java application and the Velocity templates. Every Velocity reference represents a Java object.

A Velocity reference starts with a $ character followed by an *identifier* name.

```
## Simple reference
The total amout is $total.

## Property reference
You currently have $cart.items Items in your Shopping cart.

## Method reference
We offer you a discount of $cart.calculateDiscount() for your purchase.

## Creating a new reference from Velocity
#set($linesPerPage = 20)
```

Example 3.3 Velocity References

If you need to separate a reference identifier name from the surrounding template, you can wrap the reference identifier in curly braces ({ and }):

```
The total amout is ${total}.

You currently have ${cart.items} Items in your Shopping cart.
```

Velocity references are discussed in depth in the References chapter.

3.3 Comments

Comments allows descriptive text to be included in a template 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.

Like the Java programming language, Velocity has single-line and block comments.

A single line comment begins with ## and finishes at the end of the line. This is an example of a single line comment in VTL:

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

This is visible in the output ## This is a comment.
```

If you are going to write a few lines of commentary, there is 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 will not
  see this text because the Velocity Templating Engine will
  ignore it.
*#

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

Multiline comments can start and end in arbitrary columns of the template.

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. It is similar to the Javadoc comment block in the Java programming language:

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

3.4 Escaping VTL elements

Most of the time, there is no problem, rendering a Velocity template because the rendering engine is quite smart about finding Velocity Statements and References and distinguishing them from regular text.

```
## Renders as normal text, because the dollar sign
## is not followed by a letter.
I bought a 4 lb. sack of potatoes for only $2.50!

## Renders as variable
I bought a 4 lb. sack of potatoes for only $money.
```

Velocity allows for explicit escaping of References and Directives using the \ (backslash) character. If the character following the \ would start a new directive or reference, then this character is output verbatim. This can lead to some unexpected behaviour, especially with directives.

```
\          ## Renders as a single backslash (no # or $ follows)

\\ \# \$   ## Renders as \\ \# \$ (no directive or reference follows)
```

```
\#end      ## Renders as #end, backslash escapes an existing directive
```

Example 3.4 VTL escaping examples

Unfortunately, the escaping of VTL elements contains a number of quirks that make them actually hard to use[2]. As you have seen, the behaviour of the \ differs whether a directive or just text follows:

```
$a      ## $a is a reference but an unbound on (it has no value)
        ## This renders as $a in the output

\$a     ## Escaping an unbound reference renders it as \$a

#set($a = 10)  ## When a value is assigned to the reference...

$a      ## Now renders as 10

\$a     ## Now renders as $a
```

Example 3.5 Escaping VTL references

Escaping VTL statements works in the same way.

```
\#end     ## Renders as #end
\# end    ## Renders as \# end (note the space)

\#set ($a = 10) ## Renders as #set ($a = 10)

## If $a has been assigned a value:

#set ($a = 10)
\#set ($a = 20) ## Renders as #set (10 = 20)

## The backslash does not escape a whole multiline statement
## This is a syntax error: (#end without a starting statement)
\#if ($a == 10)
#end
```

Example 3.6 Escaping VTL statements

Using Velocity Macros, the behaviour of escaping # changes a bit:[3]

```
#apple    ## Renders as #apple, because it is not a VTL
          ## statement
```

---

[2]It is really mind-boggling how we came up with this. It is tied more to the internal workings of the Velocity engine than any user-friendlyness. This is definitely subject to change in future Velocity versions. Until then, please be kind to our mistakes.
[3]...and you probably thought, the worst part was over...

```
\#apple    ## Renders as \#apple

#macro(apple)
 Apple
#end

#apple     ## This is now an syntax error (#apple needs now
           ## brackets because it's a macro)

#apple()  ## Renders as Apple (VTL Macro)

\#apple    ## renders as #apple, because it is now a VTL
           ## statement after macro definition.
```

Example 3.7 Escaping Macros

4. References

There are three types of references in the VTL: *variables*, *properties* and *methods*. They are the glue that connects the business logic of your application written in Java to your templates. The Java code and the templates must use the same set of references, so you, as a template designer and the developers writing the application logic must agree on the references available for use on the templates.

Every reference that is output when a template is rendered, is converted to a Java string by calling the `toString()` method on the Java object which represents the reference. If you want a reference rendered in a special way, you can ask your Java developers to implement or override this method.

4.1 Identifiers

All references consist 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 and underscore: `-` and `_`

Velocity identifiers are, just as Java variable names, case-sensitive.

4.2 Variables

Variables are the simplest type of references, because each reference is also a variable. Each variable represents a Java object.

```
$foo
$bar
$foo-bar
$foo_bar
$fooBar1
```

Example 4.1 Valid variable names

Velocity keeps an internal map from Identifiers to Variables called the *Context*. This map is where Java code can place objects to be reference from the template.

A variable must be assigned a value before it can be referenced from the template. Assigning a value to a Velocity variable is the same as placing an object in the Context from the Java code of your application.

```
## Puts a String object representing "bar" in the Context
#set( $foo = "bar" )
```

Note

Assigning a new value using the #set statement to a variable does change the object present in the Context.

4.3 Properties

Velocity allows you to access properties through a short-hand notation. The objects to look up the properties must be available through a Velocity variable and the notation consists of a leading variable followed by the dot (".") character and another VTL Identifier.

```
$customer.address
$purchase.total
$cart.customerDiscount
```

Example 4.2 Valid property names

A property name can represent the following elements depending on the object used for look-up:

- If the object has a method get<property> where the property name is not modified, this method is invoked

- else if the object is a Java bean (has methods conforming to the Sun Java Bean specification for accessing bean properties), the bean getter is executed to access the value

- finally if the object used to look up the property has a get(String) method, invoke this method.

Take the first example, $customer.address. It can have multiple meanings:[1]

- when the object has a method getaddress(), invoke this method

- when the object is a Java bean with a property address, invoke its getter, getAddress()

- when the object has a method get(String), invoke this method, passing address as parameter.

- when the object has a method isAddress(), invoke this method.

Note

When a property name refers to a getter method, $obj.property and $obj.Property will both invoke the same method (either getproperty() or getProperty()). However, if the object represented by $obj has a get(String) method, $obj.property and $obj.Property will pass different values to this get(String) method. This can lead to hard-to-find problems. It's a good practice to standardize the capitalization of property names in your application.

If you wonder about setting property values, please look up the #set() directive chapter. The setting of properties is discussed there.

---

[1]The sequence of these lookups is determined by the *Velocity Uberspector*. This is merely the default lookup. A Java developer can customize this behaviour by using a custom Uberspector.

Default property lookup rules

Velocity is quite clever when figuring out which method corresponds to a requested property.[2] It tries out different alternatives based on several established naming conventions. The exact lookup sequence depends on whether or not the property name starts with an upper-case letter. For lower-case names, such as `$customer.address`, the sequence is

- `getaddress()`
- `getAddress()`
- `get("address")`
- `isAddress()`

For upper-case property names like `$customer.Address`, it is slightly different:

- `getAddress()`
- `getaddress()`
- `get("Address")`
- `isAddress()`

As regular bean properties tend to be written in lowercase notation, you should talk to your programmers that they don't add both lower-case and upper-case methods to their objects. It would be a bad idea anyway.

Caution

There are some exotic naming cases, such as a property starting with multiple upper-case letters which are treated specially according to the Java Bean specification. Velocity does not conform to this specification to the last letter, so if you have a method called `getURL()`, you cannot use `$obj.url` to invoke the method, you must use `$obj.URL` or `$obj.uRL`. This might change in future versions of Velocity.

4.4 Methods

A method is part of a context object, written in Java code, and is capable of doing something useful, like running a calculation or arriving at a decision. Method invocation is indicated by round parentheses following the method name. These parentheses can optionally contain a parameter list.

```
$cart.calculateTotal()
$customer.updatePurchases($cart)
$shop.checkInventory()
$cart.addTaxes(8, 16)

# Property access
$page.setTitle( "My Home Page" )
$customer.getAddress()
$purchase.getTotal()
```

Example 4.3 Valid method references

---

[2] Some people would say that it is entirely too clever.

The last few examples are similar to the examples seen earlier when discussing properties. As stated there, the properties notation is a short-hand notation for the invocation of bean getters and setters.[3]

The main difference between Properties and Methods is that you can specify a parameter list to a Method.

```
$shop.getFruits()
$sun.getPlanets()
$album.get("logo")

$shop.setCustomer($customer)
$sun.setPlanetCount(9)
$album.put("logo", "newlogo")

## equal property getters and setters

$shop.fruits
$sun.Planets
$album.logo

#set ($shop.customer = $customer)
#set ($sun.planetCount = 9)
#set ($album.logo = "New Logo")
```

Example 4.4 Methods that can be used with shorthand notation

Warning

The last `#set()` example will not work in Velocity versions before 1.5 unless the `$album` object implements `java.util.Map`. This is a bug in older Velocity versions.

Not every method invocation can be replaced by short hand notation. Even if a method is called `set<property>()` or `get<property>()`, it must still adhere to bean getter and setter method signatures.

```
## Can't pass a parameter with $sun.planet
$sun.getPlanet(3)

## Velocity only allows shorthand notation for getter and setter
$shop.orderFruits()

## Can't pass a parameter list
$book.setAuthorAndTitle("George Orwell", "Homage to Catalonia")
```

Example 4.5 Methods that can not be used with shorthand notation

---

[3]By explicitly stating the method name and parameters, the Uberspector is not used. This is useful if you object has both, `get(String)` and explicit property getter methods.

4.5 Reference Miscellany

Separating Identifiers and template text

When writing templates, you might encounter situations in which it is necessary to explicitly separate a reference from the surrounding template text. In the examples above, this was done implicitly through whitespace. Additionally there is a formal notation which wraps the identifiers with curly braces:

```
${fruit}
${customer.address}
${purchase.getTotal()}
```

Example 4.6 Formal notation for Velocity references

Suppose you were building an extension to your fruit shop where juices are sold. `$fruit` contains the name of the juice which should be sold. Using the shorthand notation would be inadequate for this task. Consider the following example:

```
You have selected $fruitjuice.
```

There is ambiguity here, and Velocity assumes that `$fruitjuice`, not `$fruit`, is the identifier that you want to use. Finding no value for `$fruitjuice`, it will return `$fruitjuice`. Using formal notation can resolve this problem.

```
You have selected ${fruit}juice.
```

Now Velocity knows that `$fruit`, not `$fruitjuice`, 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 `$email` has no value, but you probably prefer a blank text field to one with a value of *$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:

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

Caution

It is very easy to confuse the quiet reference notation with the boolean *not*-Operator. Using the not-Operator, you use `!${foo}`, while the quiet reference notation is `$!{foo}`. And yes, you will end up sometimes with `!$!{foo}`...

5. Directives

References allow template designers to generate dynamic content for web sites, while *Directives* permit web designers to truly take charge of the appearance and content of the web site. A directive is a script element that can be used to manipulate the rendering of the template.

As described in Chapter 3.1, Velocity Directives are part of either single or multi-line statements and are preceded by a hash sign (#). While the hash sign is technically not part of the directive, we will still speak about the `#if` or `#set` directive.

Velocity knows about the following directives:[1]
- `#set()` (single line statement)
- `#literal()` (multi line statement)
- `#if() / #elseif() / #else` (multi line statement)
- `#foreach()` (multiline statement)
- `#include()` (single line statement)
- `#parse()` (single line statement)
- `#stop()` (single line statement)
- `#macro()` (multi line statement, see Chapter 7)

5.1 The `#set` directive

The `#set` directive is used for setting the value of a reference. It is used in a single-line statement.

A value can be assigned to either a variable reference or a property reference.

```
## Assigning a variable value
#set( $fruit = "apple" )

## Assigning a property value
#set( $customer.Favourite = $fruit )
```

Example 5.1 Value assignment using the set directive

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

---

[1]If is possible to add custom directives through the Velocity configuration. These are merely the directives shipped with the default Velocity configuration. It is even possible to turn some of these off.

- List
- Map
- Expression

```
## variable reference
#set( $fruit = $selectedFruit )

## string literal
#set( $fruit.flavor = "sweet" )

## property reference
#set( $fruit.amount = $cart.total )

## method reference
#set( $fruit.color = $colorlist.selectColor($select) )

## number literal
#set( $fruit.value = 123 )

## List
#set( $fruit.sorts = ["Apple", "Pear", "Orange"] )

## Map
#set( $fruit.shapes = {"Apple" : "round", "Pear" : "conical"})
```

Example 5.2 Valid reference assignments

Note

For the List example the elements defined with the [..] operator are accessible using the methods defined in the java.util.List class. So, for example, you could access the first element above using $fruit.sorts.get(0).

Similarly, for the Map example, the elements defined within the {..} operator are accessible using the methods defined in the java.util.Map class. So, for example, you could access the first element above using $fruit.shapes.get("Apple") to return the String round, or even $fruit.shapes.Apple to return the same value.

The RHS can also be an arithmetic expression as described in the Math chapter below.

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

Example 5.3 Expression examples

Assigning null values to references

In its default configuration, Velocity treats `null` values on the RHS special. If your developers don't change the default configuration, you must understand that it is not possible to assign a null value through a `set` directive to a reference. Starting with Velocity 1.5, a runtime property (`directive.set.null.allowed`) exists that removes this restriction. If you do not want `null` values to be treated special, tell your application developers that they should look up this property in the *Velocity Developers Guide* and set it to true.

Note

Removing this restriction is one of the new features of Velocity 1.5. If you are maintaining existing template code from older Velocity versions, you probably want to keep the old behaviour.

For new developments based on Velocity 1.5 and beyond, we strongly recommend that you don't treat null values on the RHS special (set `directive.set.null.allowed` to be true). This will be the default behaviour in future Velocity versions.

The remainder of this chapter assumes that you kept the default configuration of Velocity.

If the RHS is a property or method reference that evaluates to `null`, it will *not* be assigned to the LHS and the LHS *will not be altered*. This is very different from the behaviour in e.g. the Java programming language and needs some discussion.

In the following piece of code, two queries are executed using the same set of template references:

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

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

If `$query.criteria("name")` returns the string "apple", and `$query.criteria("customer")` returns null, the above VTL will render as the follows:

```
The result of the first query is apple

The result of the second query is apple
```

In the second `set` directive, the RHS is null, because `$query.criteria("customer")` returned null. So the RHS will not be modified (and the old value retained).

Unfortunately, this can lead to hard-to-detect errors, for example with `#foreach` loops that attempt to change a reference using a `#set` directive, then immediately test that reference with an `#if` directive:

```
#set( $criteria = ["fruit", "customer"] )

#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).

One solution to this would be to reset the value of $result in every loop iteration:

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

#foreach( $criterion in $criteria )

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

   #if( $result != "" )
     Query was successful
   #end

#end
```

However, the easiest way is to use the quiet reference notation:

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

#foreach( $criterion in $criteria )

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

   #if( $result != "" )
     Query was successful
   #end

#end
```

If the query fails, $result gets the empty string assigned.

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( $fruit = "apple" )
$fruit

#set( $veggie = '$fruit' )
$veggie
```

This renders as:

```
apple

$fruit
```

By default, rendering unparsed text using single quotes is activated in Velocity. This default can be changed by changing the `runtime.interpolate.string.literals` property to be false. (See the *Velocity Developers Guide* for more information).

5.2 The #literal Directive

The `#literal` directive allows you to easily use large chunks of uninterpreted content in VTL code. This can be especially useful in place of escaping multiple directives. #literal is a multi-line statement and needs a #end directive to close the statement body.

```
#literal()
#foreach ($fruit in $basket)
  nothing will happen to $fruit
#end
#end
```

Renders as:

```
#foreach ($fruit in $basket)
  nothing will happen to $fruit
#end
```

Even if the $fruit or $basket references have a value assigned, the text will still be output uninterpreted.

> ⚠️ **Caution**
>
> While #literal() suppresses the evaluation of Velocity directives and references at run-time, the body is still parsed. #literal() is not a comment block, if you have a syntax error (e.g. a missing #end) inside the block, Velocity will still report a parse error.

5.3 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. Consider the following piece of code:

```
#if( $display )
   <strong>Velocity!</strong>
#end
```

The variable $display is evaluated to determine whether it is true, which will happen under one of two circumstances:

- $foo is a java.lang.Boolean object (True/False) which has a true value.
- The value is not null.

The Velocity context only contains Java objects, so any method that returns a boolean primitive will automatically wrapped into a java.lang.Boolean object.

The content between the #if and the #end statements becomes 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:

```
#set ($direction = 15)
#set ($wind = 6)

#if( $direction < 10 )
   <strong>Go North</strong>
#elseif( $direction == 10 )
   <strong>Go East</strong>
#elseif( $wind == 6 )
   <strong>Go South</strong>
#else
   <strong>Go West</strong>
#end
```

Example 5.4 Using #if/#elseif/#else to select multiple template pieces

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

When you wish to include text immediately following a #else directive you need to use curly brackets immediately surrounding the directive to differentiate it from the following text:

```
#if( $foo == $bar)it's true!#{else}it's not!#end</li>
```

5.4 Loops - #foreach

#foreach iterates over the elements given in a List or Array element.

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

Example 5.5 foreach loop example

Unlike other languages, the Velocity templating language allows only loops where the number of iterations is predetermined. There is no do..while or repeat..until loop in Velocity. This has been a conscious decision to make sure that the rendering process of Velocity does always terminate.[2]

A VTL foreach loop takes two arguments, the first one is the *loop variable* and the second is an object that can be iterated over. Velocity supports a number of object types in its default configuration:[3]

• Any array type

• java.lang.Collection The loop iterates over the Iterator returned by obj.iterator()

• java.lang.Map The loop will iterate over the values of the Map using the iterator returned by values().iterator()

• java.lang.Iterator Velocity uses this iterator directly

• java.lang.Enumeration Velocity wrapping the Enumeration into an Iterator

Warning

Using an Iterator or Enumeration object directly in a template has the side effect that the loop cannot be evaluated multiple times because the loop consumes the object that gets iterated. Be especially careful if you use such an object inside a Macro or an included template.

Velocity provides you with a loop counter reference which contains the number of the current iteration of the loop:

---

[2]It is stil possible to write a template that contains endless loops. It is just harder.
[3]The list of objects can be changed and extended, please see the Developers guide on how to do this.

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

The default name for the loop counter and its starting value is specified through runtime properties. By default it starts at 1 and is called $velocityCount.

```
# 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
```

The Velocity loop counter variable does support nested loops, so the following example would work as expected:

```
#foreach ($category in $allCategories)
   Category # $velocityCount is $category

   #foreach ($item in $allItems)
     Item # $velocitycount is $item
   #end

#end
```

Example 5.6 Nested Loops and the loop counter

The loop counter is local to its surrounding loop and any inner loops hides but not resets it. Please note that it is not possible to access the count variable of the outer loop directly inside the inner loop.[4]

To avoid deadlocks and denial of service attacks, it is possible to set a maximum allowed number of times that a loop may be executed. By default there is no limit, but it can be set to an arbitrary number in the Velocity configuration.

```
# The maximum allowed number of loops.
directive.foreach.maxloops = -1
```

5.5 Loading resources

---

[4]You can assign its value to another variable before entering the inner loop using #set and access it through this variable inside the inner loop.

In Velocity just as in other rendering techniques like JSP it is important to build output from multiple input files. For this, you need to be able to pull external resources into the rendering process.

Velocity supports the loading of external files by the #include and #parse directive, which both load external resources into the rendering process of a template.

Velocity does not allow you to include arbitrary files into your templates for security reasons. All included files will be loaded from ResourceLoaders configured in the Velocity configuration. Please consult the Developers Guide for information on how to do this.

File inclusion - #include

The `#include` directive allows the template designer to import a local file. Its contents replace the `#include` directive in the template. These contents are inserted as-is, they are not rendered through the template engine.

Multiple files can be loaded through a single #include directive, their names must be separated by commas.

```
## Load the file one.txt into this template
#include( "one.txt" )

## Load multiple files into this template
#include( "one.txt","two.txt","three.txt" )
```

Example 5.7 Including files into a Velocity Template

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.

```
#set ($seasonalgreetings = "christmas.txt")
#include( "greetings.txt", $seasonalgreetings )
```

Example 5.8 Including a file named by a variable reference

Template inclusion - #parse

The `#parse` directive works similar to `#include`, but the contents of the file will be part of the template rendering process and all VTL elements in the file will be evaluted. The output will replace the `#parse` directive in the importing template.

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

Example 5.9 Including an external template

Like the `#include` directive, `#parse` can take a variable rather than a template. Unlike the `#include` directive, `#parse` will only take a single argument, you cannot load multiple templates with a single `#parse` directive.

Recursion is permitted; VTL templates can have `#parse` statements referring to templates that in turn have `#parse` statements.

```
## #####################################
## Template "countdown.vm"
## #####################################
Count down.

#set( $count = 8 )
#parse( "counter.vm" )
All done with countdown.vm!
## #######################################


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


The resulting output:

Count down.
8
7
6
5
4
3
2
1
0

All done with counter.vm!
All done with countdown.vm!
```

Example 5.10 Parsing a recursive template

To avoid infinitive recursion, the number of levels is limited through a configuration property. It is not possible to configure an infinite recursion depth. The default maximum number of recursions is 20.[5]

```
# Maximum number of #parse levels
directive.parse.max.depth = 10
```

---

[5]As we said, we wanted to make it hard to write templates that contain endless loops. Allowing recustion would have been too easy.

5.6 Stop template rendering - #stop

The `#stop` directive stops the rendering of a template. The remaining part of the template after this directive is discarded. The most common usage of this directive is debugging templates.

```
This part gets rendered in the output.
#stop
This part does not get rendered.
```

Example 5.11 Using #stop to end template rendering

The `#stop` directive can also be written as `#stop()`.

Caution

While `#stop` ends the rendering process, it *does not* affect the template parsing. If you have a syntax error in the VTL code after the `#stop` directive, this error will still be reported.[6]

---

[6]It is possible to do strange things by using the `#stop` directive inside `#if ... #else ... #end` blocks. As `#stop` only ends the rendering if it is actually rendered, it is possible to skip over the directive.

6. Operators

Velocity knows a number of relational and logical operators. They can be used everywhere an expression is evaluated, most prominently in `#if` and `#set` directives.

Each operator is available in two notations, a short version which is roughly equal to the Java notation and a text representation which can be used to avoid problems e.g. with XML templates and the `<`, `>` and `&&` operators.

```
#set ($a = true)
#set ($b = false)

#if ($a || $b)
  short version 'or' was true
#end

#if ($a and $b)
  text version 'and' was true
#end
```

Example 6.1 Using operators

Table 6.1. Velocity Relational and logical operators

Type of operator	short version	text version
equal[1]	==	eq
not equal	!=	ne
greater than	>	gt
greater or equal than	>=	ge
less than	<	lt
less or equal than	<=	le
logical and	&&	and
logical or	||	or
logical not	!	not

[1]Note that the semantics of the equal operator are slightly different than Java where== can only be used to test object equality. In Velocity the equivalent operator can be used to directly compare numbers, strings, or objects. When the objects are of different classes, the string representations are obtained by calling `toString()` for each object and then compared.

⚠ Caution

Unlike other languages, Velocity does not consider the number 0 (zero) or the empty String ("") to be

equivalent to false. Only the boolean value false (primitive oder `Boolean.FALSE`) and null are evaluated to false. Everything else evaluates to true.[1]

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
```

The text versions of all logical operators can be used to avoid problems with < and > in XML templates.

6.1 The AND Operator

```
## 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.

6.2 The OR Operator

Logical OR operators work the same way as AND operators, 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

---

[1]This was a conscious decision by the Velocity designers and is not likely to change.

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

6.3 The NOT Operator

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.

7. Velocity Macros

The #macro directive allows you to name a section of a VTL template and re-insert it multiple times into the template. Velocity macros (or short *Velocimacros*) can be used in many different scenarios.

In the following example, a Velocimacro is created to save keystrokes and minimizing typographic errors:

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

This example defines a Velocimacro called d. It now can be uses as any other VTL directive:

```
#d()
```

When this template is rendered, Velocity replaces #d() with the HTML table row containing a single, empty cell defined above. Using a macro in a template is called *macro invocation*.

Velocimacros start with the #macro directive and are ended with #end. The enclosed block is the *macro body*. Whenever a Velocimacro is invoked, its body is inserted in the template. It is rendered at every invokation through the templating engine, with the invocation arguments replacing the arguments in the macro definition.

A Velocimacro can take any number of arguments including zero, but when it is invoked, it must be called with exactly the same number of arguments with which it was defined.

Here is a Velocimacro that takes two arguments, a color and a list of objects:

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

#set( $greatlakes = ["Superior","Michigan","Huron","Erie","Ontario"] )
#set( $color = "blue" )

<table>
  #tablerows( $color $greatlakes )
</table>
```

The tablerows macro takes exactly two arguments. The first argument takes the place of $color, and the second argument takes the place of $values. Anything that can be put into a VTL template can go into the body of a Velocimacro.

Notice that $greatlakes takes the place of $values. When this template is rendered, 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>
```

In the example above, the `#tablerows` macro was defined *inline*. Velocimacros can be defined *inline* or *global*. An inline macro is only visible in the template where it is defined, a global macro is accessible from all templates. Global definitions must be done in a a Velocimacro template library, which is a template file that contains the macros and they must be referenced explicitly through the Velocity configuration. Please see the Velocity reference guide for more information on how to define and load a Macro library.

If the `#tablerows($color $values)` Velocimacro is defined a Velocimacro template library, it could be in any template. In the fruit store it could be used to list fruits available:

```
#set( $fruits = ["apple", "pear", "orange", "strawberry"] )
#set($cellbgcol = "white" )

<table>
  #tablerows( $cellbgcol $fruits )
</table>
```

When rendering this template, Velocitywould find the `#tablerows` Velocimacro in a template library (defined in the Velocity configuration) and generate the following output:

```
<table>
  <tr><td bgcolor="white">apple</td></tr>
  <tr><td bgcolor="white">pear</td></tr>
  <tr><td bgcolor="white">orange</td></tr>
  <tr><td bgcolor="white">strawberry</td></tr>
</table>
```

7.1 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"]
• The words true and false: boolean values for true and false

When passing references as arguments to Velocimacros, please note that references are passed *by name*. This means that their value is evaluated each time it is used inside the Velocimacro. This feature allows you to pass references with method calls and have the method called at each use:

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

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

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

Caution

If you are used to the behaviour of method or procedure calls in programming languages like Java or C, please be sure to fully understand the implications of the passing by name of references. Unlike these programming languages, the reference is not evaluated at macro invocation time but at every occurence inside the macro. Don't treat macros as procedures or methods!

The easiest way to avoid any gotchas is not passing any stateful object into a Velocimacro. This can be done by assigning the result of a method invocation to a new reference:

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

7.2 Velocimacro Properties

There are a number of properties in the Velocity configuration that influence the behaviour of macros. A complete list of all configuration options is in the Velocity Reference 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 turned offin the resource loaders (see the Velocity Reference guide on how to do this). This feature is intended for development, not for production.

7.3 Velocimacro Trivia

Velocimacros must be defined before they are first used in a template. This means that your inline `#macro()` declarations must be found by the Engine before usage.

This is important to remember if you try to `#parse()` a template containing inline `#macro()` directives. Because the `#parse()` invocation happens *at runtime*, and the parser looks at the macro definitions earlier, *at parsetime*. So loading a set of macro definitions using `#parse()` does not work. If you need to use a set of macros on more than one template, declare it as a Velomacro Library in the Velocity configuration.

8. Formatting Issues

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

```
#set( $fruits = ["apple", "pear", "orange", "strawberry"] )

#foreach( $fruit in $fruits )
  $fruit
#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=["$10 and ","a pie"])#foreach($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 pie"] )
#foreach( $a in $foo )
$a
#end
please.
```

or as

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

In each case the output will be the same.

9. Other Features and Miscellany

9.1 Math

Velocity has a number 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 between two integers, the result will be an integer. Any remainder can be obtained by using the modulus (`%`) operator.

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

9.2 Range Operator

The range operator can be used in conjunction with `#set` and `#foreach` statements. It creates an array of `java.lang.Integer` objects:

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

Both `n` and `m` must either be or produce integers. If `m` is greater than `n`, the range will count down:

```
## Range: 1 2 3 4 5
First example:
#foreach( $cnt in [1..5] )
  $cnt
#end

## Range: 2 1 0 -1 -2
Second example:
#foreach( $cnt in [2..-2] )
  $cnt
#end

## Range: 0 1
Third example:
#set( $range = [0..1] )
#foreach( $cnt in $range )
  $cnt
#end
```

```
## Not a range (not in a #set or #foreach directive)
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

As any other operator, the range operator is only evaluated in Velocity directives!

9.3 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
```

9.4 Velocimacro Miscellany

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

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

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

However, there are things you can do. One easy solution is to take advantage of the fact that 'doublequote' (") renders its 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 Velocimacro, not at the calling level. In other words, the argument to the Velocimacro is passed in in its entirety and evaluated within the Velocimacro 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 Velocimacros, so you can hand Velocimacros 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 Velocimacro, and pass the value into the Velocimacro.

```
#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 Velocimacro-looking element in the template is a Velocimacro at parsetime, #parse()-ing a set of Velocimacro declarations won't work as expected. To get around this, simply use the `velocimacro.library` facility to have Velocity load your Velocimacros 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.

9.5 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.