

## Velocity

General  
Overview  
Getting Started  
Web Applications  
Download  
FAQ (Wiki)

## Docs

User Guide  
Developer Guide  
VTL Reference  
Glossary

## Developers

License  
Javadoc  
Changes  
Resolved Issues  
Upgrading  
Dependencies  
Source Code Repository  
Building from Source  
Release Process

## Community

Wiki  
Recent News  
Powered By Velocity  
IDE/Editor Plugins  
Articles and Books  
Get Involved  
Mailing Lists

## Velocity

### Development

Road Map  
Coding Standards  
Documentation Guidelines  
Issues  
Who we are

### Translations

Site (Japanese)  
User's Guide (Finnish)  
User's Guide (French)  
User's Guide (Spanish)

## Modules

Apache Velocity - Engine  
Apache Velocity - Support  
for logging in Log4j  
Apache Velocity - Support  
for logging in Commons  
Logging  
Apache Velocity Engine -  
Servlet support  
Apache Velocity Engine -  
SLF4J support  
Apache Velocity Engine -  
Examples  
Apache Velocity Engine -  
Uberjar package  
Apache Velocity Engine -  
Assemblies

## Project

### Documentation

- Project Information
- Project Reports

## Developer Guide - Contents

1. Introduction and Getting Started
2. Resources
3. How Velocity Works
  - The Fundamental Pattern
4. To Singleton Or Not To Singleton...
  - Singleton Model
  - Separate Instance
5. The Context
  - The Basics
  - Support for Iterative Objects for #foreach()
  - Support for "Static Classes"
  - Context Chaining
  - Objects Created by the Template
  - Other Context Issues
6. Using Velocity in General Applications
  - The Velocity Helper Class
  - Exceptions
  - Miscellaneous Details
7. Application Attributes
8. Configuring Event Handlers
9. Velocity Configuration Keys and Values
10. Configuring Logging
  - Using Log4j With Existing Logger
  - Simple Example of a Custom Logger
11. Configuring the Resource Loaders (template loaders)
  - Resource Loaders
  - Configuration Examples
  - Pluggable Resource Manager and Resource Cache
12. Template Encoding for Internationalization
13. Velocity and XML
14. Summary

## Introduction

Velocity is a Java-based template engine, a simple and powerful development tool that allows you to easily create and render documents that format and present your data. In this guide, we hope to give an overview of the basics of development using Velocity.

### Building Web Applications with Velocity

Velocity is often used for building web applications. In order to use Velocity in a web app you'll need a servlet or servlet-based framework. The easiest way to get started is with [VelocityViewServlet](#) in the [Velocity Tools](#) subproject. You can also use any of a number of third party frameworks or build your own servlet using the techniques described in this document.

We suggest you read this article on [getting started with web applications](#) for more detail on the various options.



## Downloading Velocity

You can download the latest release version of [Velocity](#) or [Velocity Tools](#) from the main Apache Velocity download site. For Velocity itself, source is included with the binary download.

If you want to download the latest source, you can do so via the Subversion (svn) source control system, or download a complete [nightly snapshot](#).

Instructions for building Velocity from source can be found in the [Build](#) document.

## Dependencies

Velocity uses elements of the Java 2 API such as collections, and therefore building requires the Java 2 Standard Edition SDK (Software Development Kit). To run Velocity, the Java 2 Standard Edition RTE (Run Time Environment) is required (or you can use the SDK, of course).

Velocity also is dependent upon a few packages for general functionality. They are included in the `build/lib` directory for convenience, but the default build target (see above) does not include them. If you use the default build target, you must add the dependencies to your classpath.

- **Jakarta Commons Collections** - required.
- **Jakarta Commons Lang** - required.
- **Excalibur (ex-Avalon) Logkit** - optional, but very common. Needed if using the default file-based logging in Velocity.
- **Jakarta ORO** - optional. Needed when using the `org.apache.velocity.convert.WebMacro` template conversion utility or `org.apache.velocity.app.event.implement.EscapeReferenceReferenceInsertionEventHandler`.

## Resources

There are quite a few resources and examples available to the programmer, and we recommend that you look at our examples, documentation and even the source code. Some great sources are:

- The user and developer community: join us via the [mail-lists](#). Mail list archives are available from that page, too.
- Wiki: <http://wiki.apache.org/velocity/> The Velocity wiki contains articles, sample code, and other community-written content.
- Frequently Asked Questions (FAQ): <http://wiki.apache.org/velocity/VelocityFAQ> please visit this page to read the latest FAQ and to contribute your own answers.
- source code: `src/java/...`: all the source code to the Velocity project
- application example 1: `examples/app_example1`: a simple example showing how to use Velocity in an application program.
- application example 2: `examples/app_example2`: a simple example showing how to use Velocity in an application program using the Velocity application utility class.
- logger example: `examples/logger_example`: a simple example showing how to create a custom logging class and register it with Velocity to receive all log messages.
- XML example: `examples/xmlapp_example`: a simple example showing how to use JDOM to read and access XML document data from within a Velocity template. It also includes a demonstration of a recursive Velocimacro that walks the document tree.
- event example: `examples/event_example`: An example that demonstrates the use of the event handling API.
- Anakia application: `examples/anakia`: example application showing how to use Velocity for creating stylesheet renderings of xml data

- documentation: docs: all the generated documentation for the Velocity project in html
- API documentation: docs/api: the generated Javadoc documentation for the Velocity project
- templates: test/templates: a large collection of template examples in our testbed directory, these are a great source of usage examples of VTL, the Velocity Template Language
- context example: examples/context\_example: two examples showing how the Velocity context can be extended. For advanced users.

All directory references above are relative to the distribution root directory.

## How Velocity Works

### 'The Fundamental Pattern'

When using Velocity in an application program or in a servlet (or anywhere, actually), you will generally do the following:

1. Initialize Velocity. This applies to both usage patterns for Velocity, the Singleton as well as the 'separate runtime instance' (see more on this below), and you only do this once.
2. Create a Context object (more on what that is later).
3. Add your data objects to the Context.
4. Choose a template.
5. 'Merge' the template and your data to produce the output.

In code, using the singleton pattern via the `org.apache.velocity.app.Velocity` class, this looks like

```
import java.io.StringWriter;
import org.apache.velocity.VelocityContext;
import org.apache.velocity.Template;
import org.apache.velocity.app.Velocity;
import org.apache.velocity.exception.ResourceNotFoundException;
import org.apache.velocity.exception.ParseErrorException;
import org.apache.velocity.exception.MethodInvocationException;

Velocity.init();

VelocityContext context = new VelocityContext();

context.put( "name", new String("Velocity") );

Template template = null;

try
{
    template = Velocity.getTemplate("mytemplate.vm");
}
catch( ResourceNotFoundException rnfe )
{
    // couldn't find the template
}
catch( ParseErrorException pee )
{
    // syntax error: problem parsing the template
}
catch( MethodInvocationException mie )
{
    // something invoked in the template
    // threw an exception
}
catch( Exception e )
{}

StringWriter sw = new StringWriter();

template.merge( context, sw );
```

That's the basic pattern. It is very simple, isn't it? This is generally what happens when you use Velocity to render a template. You probably won't be writing code exactly like this - we provide a few

tools to help make it even easier. However, no matter how to use Velocity the above sequence is what is happening either explicitly, or behind the scenes.

## To Singleton Or Not To Singleton...

As of Velocity 1.2 and later, developers now have two options for using the Velocity engine, the singleton model and the separate instance model. The same core Velocity code is used for both approaches, which are provided to make Velocity easier to integrate into your Java application.

### Singleton Model

This is the legacy pattern, where there is only one instance of the Velocity engine in the JVM (or web application, depending) that is shared by all. This is very convenient as it allows localized configuration and sharing of resources. For example, this is a very appropriate model for use in a Servlet 2.2+ compliant web application as each web application can have its own instance of Velocity, allowing that web application's servlet to share resources like templates, a logger, etc. The singleton is accessible via the `org.apache.velocity.app.Velocity` class, and an example of use:

```
import org.apache.velocity.app.Velocity;
import org.apache.velocity.Template;

...

/*
 * Configure the engine - as an example, we are using
 * ourselves as the logger - see logging examples
 */

Velocity.setProperty(
    Velocity.RUNTIME_LOG_LOGSYSTEM, this);

/*
 * now initialize the engine
 */

Velocity.init();

...

Template t = Velocity.getTemplate("foo.vm");
```

### Separate Instance

New in version 1.2, the separate instance allows you to create, configure and use as many instances of Velocity as you wish in the same JVM (or web application.) This is useful when you wish to support separate configurations, such as template directories, loggers, etc in the same application. To use separate instances, use the `org.apache.velocity.app.VelocityEngine` class. An example, which parallels the above singleton example, looks like:

```
import org.apache.velocity.app.VelocityEngine;
import org.apache.velocity.Template;

...

/*
 * create a new instance of the engine
 */

VelocityEngine ve = new VelocityEngine();

/*
 * configure the engine. In this case, we are using
 * ourselves as a logger (see logging examples..)
 */

ve.setProperty(
```

```

        VelocityEngine.RUNTIME_LOG_LOGSYSTEM, this);

    /*
     * initialize the engine
     */
    ve.init();

    ...

    Template t = ve.getTemplate("foo.vm");

```

As you can see, this is very simple and straightforward. Except for some simple syntax changes, using Velocity as a singleton or as separate instances requires no changes to the high-level structure of your application or templates.

As a programmer, the classes you should use to interact with the Velocity internals are the `org.apache.velocity.app.Velocity` class if using the singleton model, or `org.apache.velocity.app.VelocityEngine` if using the non-singleton model ('separate instance').

At no time should an application use the internal `Runtime`, `RuntimeConstants`, `RuntimeSingleton` or `RuntimeInstance` classes in the `org.apache.velocity.runtime` package, as these are intended for internal use only and may change over time. As mentioned above, the classes you should use are located in the `org.apache.velocity.app` package, and are the `Velocity` and `VelocityEngine` classes. If anything is missing or needed from those classes, do not hesitate to suggest changes - these classes are intended for the application developer.

## The Context

### The Basics

The concept of the 'context' is central to Velocity, and is a common technique for moving a container of data around between parts of a system. The idea is that the context is a 'carrier' of data between the Java layer (or you the programmer) and the template layer ( or the designer ). You as the programmer will gather objects of various types, whatever your application calls for, and place them in the context. To the designer, these objects, and their methods and properties, will become accessible via template elements called [references](#). Generally, you will work with the designer to determine the data needs for the application. In a sense, this will become an 'API' as you produce a data set for the designer to access in the template. Therefore, in this phase of the development process it is worth devoting some time and careful analysis.

While Velocity allows you to create your own context classes to support special needs and techniques (like a context that accesses an LDAP server directly, for example), a good basic implementation class called `VelocityContext` is provided for you as part of the distribution.

`VelocityContext` is suitable for all general purpose needs, and we strongly recommended that you use it. Only in exceptional and advanced cases will you need to extend or create your own context implementation.

Using `VelocityContext` is as simple as using a normal Java `Hashtable` class. While the interface contains other useful methods, the two main methods you will use are

```

public Object put(String key, Object value);
public Object get(String key);

```

Please note that the value must be derived from `java.lang.Object`. Fundamental types like `int` or `float`

must be wrapped in the appropriate wrapper classes.

That's really all there is to basic context operations. For more information, see the API documentation included in the distribution.

### Support for Iterative Objects for `#foreach()`

As a programmer, you have great freedom in the objects that you put into the context. But as with most freedoms, this one comes with a little bit of responsibility, so understand what Velocity supports, and any issues that may arise. Velocity supports several types of collection types suitable for use in the VTL `#foreach()` directive.

- `Object []` Regular object array, not much needs to be said here. Velocity will internally wrap your array in a class that provides an `Iterator` interface, but that shouldn't concern you as the programmer, or the template author. Of more interest, is the fact that Velocity will now allow template authors to treat arrays as fixed-length lists (as of Velocity 1.6). This means they may call methods like `size()`, `isEmpty()` and `get(int)` on both arrays and standard `java.util.List` instances without concerning themselves about the difference.
- `java.util.Collection` Velocity will use the `iterator()` method to get an `Iterator` to use in the loop, so if you are implementing a `Collection` interface on your object, please ensure that `iterator()` returns a working `Iterator`.
- `java.util.Map` Here, Velocity depends upon the `values()` method of the interface to get a `Collection` interface, on which `iterator()` is called to retrieve an `Iterator` for the loop.
- `java.util.Iterator` **USE WITH CAUTION:** This is currently supported only provisionally - the issue of concern is the 'non-resetability' of the `Iterator`. If a 'naked' `Iterator` is placed into the context, and used in more than one `#foreach()`, subsequent `#foreach()` blocks after the first will fail, as the `Iterator` doesn't reset.
- `java.util.Enumeration` **USE WITH CAUTION:** Like `java.util.Iterator`, this is currently supported only provisionally - the issue of concern is the 'non-resetability' of the `Enumeration`. If a 'naked' `Enumeration` is placed into the context, and used in more than one `#foreach()`, subsequent `#foreach()` blocks after the first will fail, as the `Enumeration` doesn't reset.
- Any public class with a public `Iterator iterator()` method that never returns null. As a last resort, Velocity will look for an `iterator()` method. This provides great flexibility and automatic support for Java 1.5's `java.util.Iterable` interface.

In the case of the `Iterator` and `Enumeration`, it is recommended that they are placed in the context only when it cannot be avoided, and you should let Velocity find the appropriate reusable iterative interface when that is sufficient and possible.

There are good reasons to use the `java.util.Iterator` interface directly (large data sets via JDBC, for example), but if it can be avoided, it might be better to use something else. By 'directly', we meant doing something like:

```
Vector v = new Vector();
v.addElement("Hello");
v.addElement("There");

context.put("words", v.iterator());
```

where the `Iterator` itself is placed into the context. Instead, if you simply did:

```
context.put("words", v);
```

then all would be fine: Velocity would figure out that `Vector` implements `Collection` (via `List`), and therefore will find the `iterator()` method, and use that to get a 'fresh' `Iterator` for its use each time

it needs to. With just a plain Iterator (the first snippet above...), once velocity has used it in a `#foreach()`, Velocity has no way of getting a new one to use for the next `#foreach()` it is used in. The result is no output from any subsequent `#foreach()` blocks using that reference.

This above isn't meant to give the impression that iterating over collections in Velocity is something that requires great care and thought. Rather, the opposite is true, in general. Just be careful when you place an Iterator into the context.

### Support for "Static Classes"

Not all classes are instantiable. Classes like `java.lang.Math` do not provide any public constructor, and yet may contain useful static methods. In order to access these static methods from a template, you can simply add the class itself to the context:

```
context.put("Math", Math.class);
```

This will allow you to call any public static method in `java.lang.Math` on the `$Math` reference in the template.

### Context Chaining

An innovative feature of Velocity's context design is the concept of *context chaining*. Also sometimes referred to as *context wrapping*, this advanced feature allows you to connect separate contexts together in a manner that makes it appear as one 'contiguous' context to the template.

This is best illustrated by an example:

```
VelocityContext context1 = new VelocityContext();
context1.put("name", "Velocity");
context1.put("project", "Jakarta");
context1.put("duplicate", "I am in context1");

VelocityContext context2 = new VelocityContext( context1 );
context2.put("lang", "Java" );
context2.put("duplicate", "I am in context2");

template.merge( context2, writer );
```

In the code above, we have set up `context2` such that it *chains* `context1`. This means that in the template, you can access any of the items that were put into either of the two `VelocityContext` objects, as long as there is no duplication of the keys used to add objects. If that is the case, as it is above for the key 'duplicate', the object stored in the nearest context in the chain will be available. In this example above, the object returned would be the string "I am in context2".

Note that this duplication, or 'covering', of a context item does not in any way harm or alter the covered object. So in the example above, the string "I am in context1" is alive and well, still accessible via `context1.get("duplicate")`. But in the example above, the value of the reference '\$duplicate' in the template would be 'I am in context2', and the template has no access to the covered string 'I am in context1'.

Note also that you have to be careful when you are relying on the template to add information to a context that you will examine later after the rendering. The changes to the context via `#set()` statements in a template will affect only the outer context. So make sure that you don't discard the outer context, expecting the data from the template to have been placed onto the inner one.

This feature has many uses, the most common so far is providing layered data access and toolsets.

As mentioned before, the Velocity context mechanism is also extendable, but beyond the current



scope of this guide. If you are interested, please see the classes in the package `org.apache.velocity.context` to see how the provided contexts are put together. Further, there are a few examples in the `examples/context_example` directory in the distribution which show alternate implementations, including [a goofy] one that uses a database as the backing storage.

Please note that these examples are unsupported and are there for demonstration/educational purposes only.

### Objects Created in the Template

There are two common situations where the Java code must deal with objects created at runtime in the template:

When a template author calls a method of an object placed into the context by Java code.

```
#set($myarr = ["a","b","c"] )
$foo.bar( $myarr )
```

When a template adds objects to the context, the Java code can access those objects after the merge process is complete.

```
#set($myarr = ["a","b","c"] )
#set( $foo = 1 )
#set( $bar = "bar")
```

Dealing with these cases is very straightforward, as there are just a few things to know:

- The VTL RangeOperator [ 1..10 ] and ObjectArray ["a","b"] are `java.util.ArrayList` objects when placed in the context or passed to methods. Therefore, your methods that are designed to accept arrays created in the template should be written with this in mind.
- VTL Map references are unsurprisingly stored as `java.util.Map`.
- Decimal numbers will be Doubles or BigDecimals in the context, integer numbers will be Integer, Long, or BigIntegers, and strings will be, of course, Strings.
- Velocity will properly 'narrow' args to method calls, so calling `setFoo( int i )` with an int placed into the context via `#set( )` will work fine.

### Other Context Issues

One of the features provided by the VelocityContext (or any Context derived from AbstractContext) is node specific introspection caching. Generally, you as the developer don't need to worry about this when using the VelocityContext as your context. However, there is currently one known usage pattern where you must be aware of this feature.

The VelocityContext will accumulate introspection information about the syntax nodes in a template as it visits those nodes. So, in the following situation:

- You are iterating over the same template using the same VelocityContext object.
- Template caching is off.
- You request the Template from `getTemplate()` on each iteration.

It is possible that your VelocityContext will appear to 'leak' memory (it is really just gathering more introspection information.) What happens is that it accumulates template node introspection information for each template it visits, and as template caching is off, it appears to the VelocityContext that it is visiting a new template each time. Hence it gathers more introspection information and grows. It is highly recommended that you do one or more of the following:

- Create a new VelocityContext for each excursion down through the template render process.



This will prevent the accumulation of introspection cache data. For the case where you want to reuse the VelocityContext because it's populated with data or objects, you can simply wrap the populated VelocityContext in another, and the 'outer' one will accumulate the introspection information, which you will just discard. Ex. `VelocityContext useThis = new VelocityContext( populatedVC );` This works because the outer context will store the introspection cache data, and get any requested data from the inner context (as it is empty.) Be careful though - if your template places data into the context and it's expected that it will be used in the subsequent iterations, you will need to do one of the other fixes, as any template `#set()` statements will be stored in the outermost context. See the discussion in [Context chaining](#) for more information.

- Turn on template caching. This will prevent the template from being re-parsed on each iteration, resulting the the VelocityContext being able to not only avoid adding to the introspection cache information, but be able to use it resulting in a performance improvement.
- Reuse the Template object for the duration of the loop iterations. Then you won't be forcing Velocity, if the cache is turned off, to reread and reparse the same template over and over, so the VelocityContext won't gather new introspection information each time.

## Using Velocity

If you are using [VelocityViewServlet](#) or other web frameworks, you may never call Velocity directly. However, if you use Velocity for non-web purposes, or create your own web framework you will need to directly call the Velocity Engine similar to [the fundamental pattern](#) shown earlier. One important additional thing to remember is to initialize the Velocity Engine before using it to merge templates.

### The Velocity Helper Class

Velocity contains an application utility class called Velocity ( `org.apache.velocity.app.Velocity` ). The purpose of this class is to provide the necessary methods required to initialize Velocity, as well as useful utility routines to make life easier in using Velocity. This class is documented in the project's javadoc, so please look there for definitive details. This documentation is intended to be of a tutorial nature; therefore for complete API information, the Javadoc is the definitive source.

The Velocity runtime engine is a singleton instance that provides resource, logging and other services to all Velocity users running in the same JVM. Therefore, the runtime engine is initialized only once. You can attempt to initialize Velocity more than once, but only the first initialization will apply. The rest of the attempts will be ignored. The Velocity utility class currently provides five methods used in configuration of the runtime engine.

The five configuration methods are:

- `setProperty( String key, Object o )`  
Sets the property key with the value o. The value is typically a String, but in special cases can also be a comma-separated list of values (in a single String, ex."foo, bar, woogie") as well as other things that will arise.
- `Object getProperty( String key )`  
Returns the value of the property key. Note that you must be aware of the type of the return value, as they can be things other than Strings.
- `init()`  
Initializes the runtime with the default properties provided in the distribution.(These are listed below in the section pertaining to properties.)
- `init( Properties p )`  
Initialize the runtime with the properties contained in the `java.util.Properties` object

passed as an argument.

- `init( String filename )`  
initializes the runtime using the properties found in the properties file filename

Note that in each case, the default properties will be used as a base configuration, and any additional properties specified by the application will replace individual defaults. Any default properties not overwritten will remain in effect. This has the benefit that only the properties you are interested in changing need to be specified, rather than a complete set.

Another thing to note is that the `init()` calls may be called more than once without harm in an application. However, the first call to any of the `init()` functions will configure the engine with the configuration properties set at that point, and any further configuration changes or `init()` calls will be ignored.

The most common approaches to initializing Velocity will be something like:

1. Setup the configuration values you wish to set in a file in the same format as `org/apache/velocity/runtime/defaults/velocity.properties` (the default set), or in a `java.util.Properties`, and then call either `init( filename )` or `init( Properties )`
2. Set the configuration values individually using `setProperty()` and then call `init()`. This method is generally used by more advanced applications that already have their own configuration management system - this allows the application so configure Velocity based upon values it generates at runtime, for example.

Once the runtime is initialized, you can do with it what you wish.. This mostly revolves around rendering templates into an output stream, and the Velocity utility class allows you to do this easily. Currently, here are the methods and a brief description of what they do:

- `evaluate( Context context, Writer out, String logTag, String instream )`  
`evaluate( Context context, Writer writer, String logTag, InputStream instream )`  
These methods will render the input, in either the form of `String` or `InputStream` to an output `Writer`, using a `Context` that you provide. This is a very convenient method to use for token replacement of strings, or if you keep 'templates' of VTL-containing content in a place like a database or other non-file storage, or simply generate such dynamically.
- `invokeVelocimacro( String vmName, String namespace, String params[], Context context, Writer writer )`  
Allows direct access to Velocimacros. This can also be accomplished via the `evaluate()` method above if you wish. Here you simply name the vm you wish to be called, create an array of args to the VM, a `Context` of data, and `Writer` for the output. Note that the VM args must be the 'keys' of the data objects in the `Context`, rather than literal data to be used as the arg. This will probably change.
- `mergeTemplate( String templateName, Context context, Writer writer )`  
Convenient access to the normal template handling and rendering services of Velocity. This method will take care of getting and rendering the template. It will take advantage of loading the template according to the properties setting for the file resource loader, and therefore provides the advantage of file and parsed template caching that Velocity offers. This is the most efficient way to access templates, and is recommended unless you have special needs.
- `boolean templateExists( String name )`  
Determines if a template name is able to be found by the currently configured resource loaders.

Once we know about these basic helpers, it is easy to write Java program that uses Velocity. Here it is:

```

import java.io.StringWriter;
import org.apache.velocity.app.Velocity;
import org.apache.velocity.VelocityContext;

public class Example2
{
    public static void main( String args[] )
    {
        /* first, we init the runtime engine.  Defaults are fine. */
        Velocity.init();

        /* lets make a Context and put data into it */
        VelocityContext context = new VelocityContext();

        context.put("name", "Velocity");
        context.put("project", "Jakarta");

        /* lets render a template */
        StringWriter w = new StringWriter();

        Velocity.mergeTemplate("testtemplate.vm", context, w );
        System.out.println(" template : " + w );

        /* lets make our own string to render */
        String s = "We are using $project $name to render this.";
        w = new StringWriter();
        Velocity.evaluate( context, w, "mystring", s );
        System.out.println(" string : " + w );
    }
}

```

When we run this program, and have the template `testtemplate.vm` in the same directory as our program (because we used the default configuration properties, and the default place to load templates from is the current directory...), our output should be:

```

template : Hi!  This Velocity from the Jakarta project.
string : We are using Jakarta Velocity to render this.

```

where the template we used, `testtemplate.vm`, is

```

Hi!  This $name from the $project project.

```

That's all there is to it! Note that we didn't have to use both `mergeTemplate()` and `evaluate()` in our program. They are both included there for demonstration purposes. You will probably use only one of the methods, but depending on you application requirements, you are free to do what you wish.

This appears to be a little different from the 'fundamental pattern' that was mentioned at the beginning of this guide, but it really is the same thing. First, you are making a context and filling it with the data needed. Where this examples differs is that in the part of the above example where `mergeTemplate()` is used, `mergeTemplate()` is doing the work of getting the template and merging it for you, using the lower-level calls in the Runtime class. In the second example, you are making your template dynamically via the String, so that is analgous to the 'choose template' part of the process, and the `evaluate()` method does the merging for you using lower level calls.

So the example above sticks to the same simply pattern of using the Velocity template engine, but the utility functions do some of the repeated drudge work, or allow you other options for your template content other than template files.

### Exceptions

Velocity may throw one of several exceptions during the parse / merge cycle. These exceptions extend `RuntimeException` and do not need to explicitly caught, although each includes specific properties that

may help in presenting useful error messages to the end user. The exceptions are found in the package `org.apache.velocity.exception` and are:

1. `ResourceNotFoundException`  
 Thrown when the resource management system cannot find a resource (template) that was requested.
2. `ParseException`  
 Thrown when a VTL syntax error is found when parsing a resource (template).
3. `TemplateInitException`  
 Thrown during the first pass of template parsing; reports problems with macro and directive initialization.
4. `MethodInvocationException`  
 Thrown when a method of object in the context thrown an exception during render time. This exception wraps the thrown exception and propagates it to the application. This allows you to handle problems in your own objects at runtime.

In each case, a message is put into the runtime log. For more information, see the Javadoc API documentation.

### Miscellaneous Details

While the above example used the default properties, setting your own properties is very simple. All you have to do is make a properties file somewhere and pass the name of that file to the `init(String)` method of the Velocity utility class, or make a `java.util.Properties` object, add the desired properties and values, and pass that to the `init(Properties)` method. The latter method is convenient, because you can either fill it directly from a separate properties file via the `load()` method, or even better, you can fill it dynamically from your own application / framework's property set at runtime. This gives you the freedom to combine all of the properties for your app into one properties file.

If we wanted to use a different directory than the current directory to load our template from, we could do something like this:

```
...
import java.util.Properties;
...
public static void main( String args[] )
{
    /* first, we init the runtime engine. */

    Properties p = new Properties();
    p.setProperty("file.resource.loader.path", "/opt/templates");
    Velocity.init( p );

    /* lets make a Context and put data into it */

    ...
}
```

And, assuming you have a directory `/opt/templates` and the template `testtemplate.vm` is in there, then things would work just fine. If you try this and have a problem, be sure to look at the `velocity.log` for information - the error messages are pretty good for figuring out what is wrong.

If you need to place objects into the Velocity properties then you cannot use the `Velocity.init(Properties p)` method. Instead you should create a new instance of the `org.apache.commons.collections.ExtendedProperties` class, copy all properties from an existing `Properties` object into the `ExtendedProperties` and then add new properties with your objects to the `ExtendedProperties` object.

```
...
```

```

VelocityEngine velocityEngine = new VelocityEngine();
ExtendedProperties eprops = null;
if (props==null) {
    eprops = new ExtendedProperties();
} else {
    eprops = ExtendedProperties.convertProperties(props);
}

// Now set the property with your object instance
eprops.setProperty("name", object);

...
velocityEngine.setExtendedProperties(eprops);
velocityEngine.init();
...

```

You may want to also consider using the Application Attributes feature described in the following section.

## Application Attributes

*Application Attributes* are name-value pairs that can be associated with a `RuntimeInstance` (either via the `VelocityEngine` or the `Velocity` singleton) and accessed from any part of the Velocity engine that has access to the `RuntimeInstance`.

This feature was designed for applications that need to communicate between the application layer and custom parts of the Velocity engine, such as loggers, resource loaders, resource managers, etc.

The Application Attribute API is very simple. From the application layer, there is a method of the `VelocityEngine` and the `Velocity` classes:

```
public void setApplicationAttribute( Object key, Object value );
```

through which an application can store an `Object` under an application (or internal component) specified key. There are no restrictions on the key or the value. The value for a key may be set at any time - it is not required that this be set before `init()` is called.

Internal components can access the key-value pairs if they have access to the object via the `RuntimeServices` interface, using the method

```
public Object getApplicationAttribute( Object key );
```

Note that internal components cannot set the value of the key, just get it. If the internal component must communicate information to the application layer, it must do so via the `Object` passed as the value.

## Configuring Event Handlers

Velocity contains a fine-grained event handling system that allows you to customize the operation of the engine. For example, you may change the text of references that are inserted into a page, modify which templates are actually included with `#include` or `#parse`, or capture all invalid references.

All event handler interfaces available in Velocity are in the package `org.apache.velocity.app.event`. You may create your own implementation or use one of

the sample implementations in the package `org.apache.velocity.app.event.implement`. (See the javadocs for more details on the provided implementations).

*org.apache.velocity.app.event.IncludeEventHandler*

The IncludeEventHandler can be used to modify the template that is included in a page with `#include` or `#parse`. For example, this may be used to make all includes relative to the current directory or to prevent access to unauthorized resources. Multiple IncludeEventHandler's may be chained, with the return value of the final call used as the name of the template to retrieve.

```
public IncludeEventHandler extends EventHandler
{
    public String includeEvent( String includeResourcePath,
                               String currentResourcePath,
                               String directiveName );
}
```

Available implementations include:

- `org.apache.velocity.app.event.implement.IncludeNotFound`
- `org.apache.velocity.app.event.implement.IncludeRelativePath`

*org.apache.velocity.app.event.InvalidReferenceEventHandler*

Normally, when a template contains a bad reference an error message is logged and (unless it is part of a `#set` or `#if`), the reference is included verbatim in a page. With the InvalidReferenceEventHandler this behavior can be changed. Substitute values can be inserted, invalid references may be logged, or an exception can be thrown. Multiple InvalidReferenceEventHandler's may be chained. The exact manner in which chained method calls behave will differ per method. (See the javadoc for the details).

```
public InvalidReferenceEventHandler extends EventHandler
{
    public Object invalidGetMethod( Context context,
                                    String reference,
                                    Object object,
                                    String property,
                                    Info info);

    public boolean invalidSetMethod( Context context,
                                    String leftreference,
                                    String rightreference,
                                    Info info);

    public Object invalidMethod( Context context,
                                 String reference,
                                 Object object,
                                 String method,
                                 Info info);
}
```

Available implementations include:

- `org.apache.velocity.app.event.implement.ReportInvalidReferences`

*org.apache.velocity.app.event.MethodExceptionEventHandler*

When a user-supplied method throws an exception, the MethodExceptionEventHandler is invoked with the Class, method name and thrown Exception. The handler can either return a valid Object to be used as the return

value of the method call or throw the passed-in or new Exception, which will be wrapped and propagated to the user as a `MethodInvocationException`. While `MethodExceptionHandler`'s can be chained only the first handler is actually called -- all others are ignored.

```
public interface MethodExceptionHandler extends EventHandler
{
    public Object methodException( Class clazz,
                                   String method,
                                   Exception e )
                                   throws Exception;
}
```

Available implementations include:

- `org.apache.velocity.app.event.implement.PrintExceptions`

*`org.apache.velocity.app.event.ReferenceInsertionEventHandler`*

A `ReferenceInsertionEventHandler` allows the developer to intercept each write of a reference (`$foo`) value to the output stream and modify that output. Multiple `ReferenceInsertionEventHandler`'s may be chained with each step potentially altering the inserted reference.

```
public interface ReferenceInsertionEventHandler extends EventHandler
{
    public Object referenceInsert( String reference,
                                   Object value );
}
```

Available implementations include:

- `org.apache.velocity.app.event.implement.EscapeHtmlReference`
- `org.apache.velocity.app.event.implement.EscapeJavascriptReference`
- `org.apache.velocity.app.event.implement.EscapeSqlReference`
- `org.apache.velocity.app.event.implement.EscapeXmlReference`

## Registering Event Handlers

You may register event handlers in either of two manners. The easiest way to register event handlers is to specify them in `velocity.properties`. (Event handlers configured in this manner are referred to as "global" event handlers). For example, the following property will escape HTML entities in any inserted reference.

```
eventhandler.referenceinsertion.class =
org.apache.velocity.app.event.implement.EscapeHtmlReference
```

Most event handler interfaces will also permit event handlers to be chained together. Such a chain may be in a comma separated list or as additional lines with a property/value pair. For example, the following event handler properties install two `ReferenceInsertionEventHandler`'s. The first will apply to references starting with "msg" (for example `$msgText`) and will escape HTML entities (e.g. turning `&` into `&amp;`). The second will escape all references starting with "sql" (for example `$sqlText`) according to SQL escaping rules. (note that in these examples, the first two properties given relate to the event handler configuration while the second two properties are used by the specific event handler implementation).

```
eventhandler.referenceinsertion.class =
org.apache.velocity.app.event.implement.EscapeHtmlReference
eventhandler.referenceinsertion.class =
org.apache.velocity.app.event.implement.EscapeSqlReference
```



```
eventhandler.escape.html.match = /msg.*/
eventhandler.escape.sql.match = /sql.*/
```

Event handlers may also be attached to a context via an `EventCartridge`. This allows event handlers to be tied more closely to a specific template merge or group of merges. The event handler will automatically be injected with the current context if it implements the `ContextAware` interface. (Due to thread-safety reasons this is not possible with global event handlers).

The following code shows how to register an event handler with an `EventCartridge` and a context.

```
...
import org.apache.velocity.app.event.EventCartridge;
import org.apache.velocity.app.event.ReferenceInsertionEventHandler;
import org.apache.velocity.app.event.implement.EscapeHtmlReference;
import org.apache.velocity.app.event.implement.EscapeSqlReference;
...
public class Test
{
    public void myTest()
    {
        ....

        /**
         * Make a cartridge to hold the event handlers
         */
        EventCartridge ec = new EventCartridge();

        /**
         * then register and chain two escape-related handlers
         */
        ec.addEventHandler(new EscapeHtmlReference());
        ec.addEventHandler(new EscapeSqlReference());

        /**
         * and then finally let it attach itself to the context
         */
        ec.attachToContext( context );

        /**
         * now merge your template with the context as you normally
         * do
         */

        ....
    }
}
```

## Velocity Configuration Keys and Values

Velocity's runtime configuration is controlled by a set of configuration keys listed below. Generally, these keys will have values that consist of either a `String`, or a comma-separated list of `Strings`, referred to as a `CSV` for comma-separated values.

There is a set of default values contained in Velocity's jar, found in `/src/java/org/apache/velocity/runtime/defaults/velocity.defaults`, that Velocity uses as its configuration baseline. This ensures that Velocity will always have a 'correct' value for its configuration keys at startup, although it may not be what you want.

Any values specified before `init()` time will replace the default values. Therefore, you only have to configure velocity with the values for the keys that you need to change, and not worry about the rest. Further, as we add more features and configuration capability, you don't have to change your configuration files to suit - the Velocity engine will always have default values.

Please see the section above **Using Velocity** for discussion on the configuration API.

Below are listed the configuration keys that control Velocity's behavior. Organized by category, each

key is listed with its current default value to the right of the '=' sign.

### Runtime Log

`runtime.log = velocity.log`

Full path and name of log file for error, warning, and informational messages. The location, if not absolute, is relative to the 'current directory'.

`runtime.log.logsystem`

This property has no default value. It is used to give Velocity an instantiated instance of a logging class that supports the interface `org.apache.velocity.runtime.log.LogChute`, which allows the combination of Velocity log messages with your other application log messages. Please see the section **Configuring Logging** for more information.

`runtime.log.logsystem.class =  
org.apache.velocity.runtime.log.AvalonLogChute`  
Class to be used for the Velocity-instantiated log system.

`runtime.log.invalid.references = true`

Property to turn off the log output when a reference isn't valid. Good thing to turn off in production, but very valuable for debugging.

`runtime.log.logsystem.avalon.logger = name`

Allows user to specify an existing logger *name* in the Avalon hierarchy without having to wrap with a LogChute interface. **Note:** You must also specify `runtime.log.logsystem.class = org.apache.velocity.runtime.log.AvalonLogChute` as the default logsystem may change. There is **no** guarantee that the Avalon log system will remain the default log system.

### Character Encoding

`input.encoding = ISO-8859-1`

Character encoding for input (templates). Using this, you can use alternative encoding for your templates, such as UTF-8.

`output.encoding = ISO-8859-1`

Character encoding for output streams from the VelocityServlet and Anakia.

### #define() Directive

`define.provide.scope.control = false`

Used to turn on the automatic provision of the \$define scope control during #define() calls. The default is false. Set it to true if you want a local, managed namespace you can put references in when within a #define block or if you want it for more advanced #break usage.

### #evaluate() Directive

`evaluate.provide.scope.control = false`

Used to turn on the automatic provision of the \$evaluate scope during #evaluate() or Velocity[Engine].evaluate(...) calls. The default is false. Set it to true if you want a local, managed namespace you can put references in during an evaluation or if you want it for more advanced #break usage.

### #foreach() Directive

`foreach.provide.scope.control = true`

Used to control the automatic provision of the \$foreach scope during #foreach calls. This gives access to the foreach status information (e.g. \$foreach.index or \$foreach.hasNext). The default is true. Set it to false if unused and you want a tiny performance boost.

```
directive.foreach.maxloops = -1
```

Maximum allowed number of loops for a #foreach() statement.

```
directive.foreach.skip.invalid = true
```

Tells #foreach to simply skip rendering when the object it is iterating over is not or cannot produce a valid Iterator.

### #include() and #parse() Directive

```
directive.include.output.errormsg.start = <!-- include error :
directive.include.output.errormsg.end = see error log -->
```

Defines the beginning and ending tags for an in-stream error message in the case of a problem with the #include() directive. If both the .start and .end tags are defined, an error message will be output to the stream, of the form '.start msg .end' where .start and .end refer to the property values. Output to the render stream will only occur if both the .start and .end (next) tag are defined.

```
directive.parse.max.depth = 10
```

Defines the allowable parse depth for a template. A template may #parse() another template which itself may have a #parse() directive. This value prevents runaway #parse() recursion.

```
template.provide.scope.control = false
```

Used to turn on the automatic provision of the \$template scope control during #parse calls and template.merge(...) calls. The default is false. Set it to true if you want a secure namespace for your template variables or more advanced #break control.

### Resource Management

```
resource.manager.logwhenfound = true
```

Switch to control logging of 'found' messages from resource manager. When a resource is found for the first time, the resource name and classname of the loader that found it will be noted in the runtime log.

```
resource.manager.cache.class
```

Declares the class to be used for resource caching. The current default is org.apache.velocity.runtime.resource.ResourceCacheImpl. When resource.manager.defaultcache.size is set to 0, then the default implementation is the standard java ConcurrentHashMap. Otherwise, a non-zero cache size uses an LRU Map. The default cache size is 89. Note that the ConcurrentHashMap may be better at thread concurrency.

```
resource.manager.defaultcache.size
```

Sets the size of the default implementation of the resource manager cache size. The default is 89.

```
resource.loader = <name> (default = file)
```

*Multi-valued key. Will accept CSV for value.* Public name of a resource loader to be used. This public name will then be used in the specification of the specific properties for that resource loader. Note that as a multi-valued key, it's possible to pass a value like "file, class" (sans quotes), indicating that following will be configuration values for two loaders.

```
<name>.loader.description = Velocity File Resource Loader
```

Description string for the given loader.

```
<name>.resource.loader.class =
org.apache.velocity.runtime.resource.loader.FileResourceLoader
```

Name of implementation class for the loader. The default loader is the file loader.

```
<name>.resource.loader.path = .
```

*Multi-valued key. Will accept CSV for value.* Root(s) from which the loader loads templates. Templates may live in subdirectories of this root. ex. homesite/index.vm This configuration key applies currently to the FileResourceLoader and JarResourceLoader.

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

Controls caching of the templates in the loader. Default is false, to make life easy for development and debugging. This should be set to true for production deployment. When 'true', the `modificationCheckInterval` property applies. This allows for the efficiency of caching, with the convenience of controlled reloads - useful in a hosted or ISP environment where templates can be modified frequently and bouncing the application or servlet engine is not desired or permitted.

`<name>.resource.loader.modificationCheckInterval = 2`

This is the number of seconds between modification checks when caching is turned on. When this is an integer  $> 0$ , this represents the number of seconds between checks to see if the template was modified. If the template has been modified since last check, then it is reloaded and reparsed. Otherwise nothing is done. When  $\leq 0$ , no modification checks will take place, and assuming that the property `cache` (above) is true, once a template is loaded and parsed the first time it is used, it will not be checked or reloaded after that until the application or servlet engine is restarted.

To illustrate, here is an example taken right from the default Velocity properties, showing how setting up the FileResourceLoader is managed

```
resource.loader = file

file.resource.loader.description = Velocity File Resource Loader
file.resource.loader.class =
org.apache.velocity.runtime.resource.loader.FileResourceLoader
file.resource.loader.path = .
file.resource.loader.cache = false
file.resource.loader.modificationCheckInterval = 2
```

**Velocimacro**

`velocimacro.library = VM_global_library.vm`

*Multi-valued key. Will accept CSV for value.* Filename(s) of Velocimacro library to be loaded when the Velocity Runtime engine starts. These Velocimacros are accessible to all templates. The file is assumed to be relative to the root of the file loader resource path.

`velocimacro.permissions.allow.inline = true`

Determines if the definition of new Velocimacros via the `#macro()` directive in templates is allowed. The default value is true, meaning any template can define and use new Velocimacros. Note that depending on other properties, those `#macro()` statements can replace global definitions.

`velocimacro.permissions.allow.inline.to.replace.global = false`

Controls if a Velocimacro defined 'inline' in a template can replace a Velocimacro defined in a library loaded at startup.

`velocimacro.permissions.allow.inline.local.scope = false`

Controls 'private' templates namespaces for Velocimacros. When true, a `#macro()` directive in a template creates a Velocimacro that is accessible only from the defining template. This means that Velocimacros cannot be shared unless they are in the global or local library loaded at startup. (See above.) It also means that templates cannot interfere with each other. This property also allows a technique where there is a 'default' Velocimacro definition in the global or local library, and a template can 'override' the implementation for use within that template. This occurs because when this property is true, the template's namespace is searched for a Velocimacro before the global namespace, therefore allowing the override mechanism.

`velocimacro.library.autoreload = false`

Controls Velocimacro library autoloading. 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.arguments.strict = false`

When set to true, will throw a `ParseException` when parsing a template containing a macro with an invalid number of arguments. Is set to false by default to maintain backwards compatibility with templates written before this feature became available.

`velocimacro.body.reference = false`

Defines name of the reference that can be used to get the body content (an AST block) given for a block macro call (e.g. `#@myMacro()` has a body `#end`). The default reference name is "bodyContent" (e.g. `$bodyContent`). This block macro feature was introduced in Velocity 1.7.

`macro.provide.scope.control = false`

Used to turn on the automatic provision of the `$macro` scope control during `#macro` calls. The default is false. Set it to true if you need a local namespace in macros or more advanced `#break` controls.

`<somebodymacro>.provide.scope.control = false`

Used to turn on the automatic provision of the `$<nameofthmacro>` scope control during a call to a macro with a body (e.g. `#@foo #set($foo.a=$b) ... $foo.a #end`). The default is false. Set it to true if you happen to need a namespace just for your macro's body content or more advanced `#break` controls.

### Strict Reference Setting

`runtime.references.strict = false`

New in Velocity 1.6, when set to true Velocity will throw a `MethodInvocationException` for references that are not defined in the context, or have not been defined with a `#set` directive. This setting will also throw an exception if an attempt is made to call a non-existing property on an object or if the object is null. When this property is true then 'directive.foreach.skip.invalid' defaults to true, but explicitly setting 'directive.foreach.skip.invalid' will override this default. For a complete discussion see [Strict References Setting](#).

`runtime.references.strict.escape = false` Changes escape behavior such that putting a forward slash before a reference or macro always escapes the reference or macro and absorbs the forward slash regardless if the reference or macro is defined. For example `"\$foo"` always renders as `"$foo"`, or `"#foo()"` is always rendered as `"#foo()"`. This escape behavior is of use in strict mode since unintended strings of characters that look like references or macros will throw an exception. This provides an easy way to escape these references. However, even in non-strict mode the developer may find this a more consistent and reliable method for escaping.

### String Interpolation

`runtime.interpolate.string.literals = true`

Controls interpolation mechanism of VTL String Literals. Note that a VTL `StringLiteral` is specifically a string using double quotes that is used in a `#set()` statement, a method call of a reference, a parameter to a VM, or as an argument to a VTL directive in general. See the VTL reference for further information.

### Math

`runtime.strict.math = false`

Affects all math operations in VTL. If changed to true, this will cause Velocity to throw a `MathException` whenever one side of a math operation has a null value (e.g. `#set( $foo =`

`$null * 5 ))` or when trying to divide by zero. If this value is left `false`, then rendering will continue and that math operation will be ignored.

### Parser Configuration

```
parser.pool.class = org.apache.velocity.runtime.ParserPoolImpl
```

This property selects the implementation for the parser pool. This class must implement `ParserPool`. Generally there is no reason to change this though if you are building a high volume web application you might consider including an alternate implementation that automatically adjusts the size of the pool.

```
parser.pool.size = 20
```

This property is used by the default pooling implementation to set the number of parser instances that Velocity will create at startup and keep in a pool. The default of 20 parsers should be more than enough for most uses. In the event that Velocity does run out of parsers, it will indicate so in the log, and dynamically create overflow instances as needed. Note that these extra parsers will not be added to the pool, and will be discarded after use. This will result in very slow operation compared to the normal usage of pooled parsers, but this is considered an exceptional condition. A web application using Velocity as its view engine might exhibit this behavior under extremely high concurrency (such as when getting Slashdotted). If you see a corresponding message referencing the `parser.pool.size` property in your log files, please increment this property immediately to avoid performance degradation.



### Pluggable Introspection

```
runtime.introspector.uberspect =  
org.apache.velocity.util.introspection.UberspectImpl
```

This property sets the 'Uberspector', the introspection package that handles all introspection strategies for Velocity. You can specify a comma-separated list of Uberspector classes, in which case all Uberspectors are chained. The default chaining behaviour is to return the first non-null value for each introspection call among all provided uberspectors. You can modify this behaviour (for instance to restrict access to some methods) by subclassing `org.apache.velocity.util.introspection.AbstractChainableUberspector` (or implementing directly `org.apache.velocity.util.introspection.ChainableUberspector`). This allows you to create more interesting rules or patterns for Uberspection, rather than just returning the first non-null value.

## Configuring Logging

Velocity has a few nice logging features to allow both simplicity and flexibility. Without any extra configuration, Velocity will setup a file-based logger that will output all logging messages to a file called `velocity.log` in the 'current directory' where Velocity was initialized. For more advanced users, you may integrate your current logging facilities with Velocity to have all log messages sent to your logger.

Starting with version 1.3, Velocity will automatically use either the [Jakarta Avalon Logkit](#)  logger, or the [Jakarta Log4j](#)  logger. It will do so by using whatever it finds in the current classpath, starting first with Logkit. If Logkit isn't found, it tries Log4j.

To utilize this feature, simply use the 'non-dependency' Velocity jar (because Logkit is baked into the jar with dependencies) and place either the logkit or log4j jar in your classpath.

In general, you have the following logging options:

- **Default Configuration**  
By default, Velocity will create a file-based logger in the current directory. See the note above

regarding automatic detection of Logkit or Log4j to use as the default logging system.

- **Existing Log4j Logger**

Starting with version 1.3, Velocity will log its output to an existing Log4j Logger setup elsewhere in the application. To use this feature you must

1. Make sure that the Log4j jar is in your classpath. (You would do this anyway since you are using Log4j in the application using Velocity.)
2. Configure Velocity to use the Log4JLogChute class by specifying the name of the existing Logger to use via the 'runtime.log.logsystem.log4j.logger' property.

Note that this support for Logger is in version 1.5 of Velocity. Further, in version 1.5 we removed the now-ancient and very deprecated original Log4JLogSystem class and replaced with the current Log4JLogChute class which uses the Logger class. We apologize for the confusion, but we needed to move on.

- **Custom Standalone Logger**

You can create a custom logging class - you just need to implement the interface `org.apache.velocity.runtime.log.LogChute` and then simply set the configuration property `runtime.log.logsystem.class` with the classname, and Velocity will create an instance of that class at init time. You may specify the classname as you specify any other properties. See the information on the [Velocity helper class](#) as well as the [configuration keys and values](#). Please note that the old `org.apache.velocity.runtime.log.LogSystem` interface has been deprecated for v1.5 in favor of the new LogChute interface. This is due to significant upgrades to our logging code that could not be supported by the LogSystem interface. But don't worry, if you specify a custom class that implements the LogSystem interface, it will still work. However, it will generate deprecation warnings. You should upgrade your custom logger to implement LogChute as soon as possible.

- **Integrated Logging**

You can integrate Velocity's logging capabilities with your applications existing logging system, simply by implementing the `org.apache.velocity.runtime.log.LogChute` interface. Then, pass an instance of your logging class to Velocity via the `runtime.log.logsystem` configuration key before initializing the Velocity engine, and Velocity will log messages to your application's logger. See the information on the [Velocity helper class](#) as well as the [configuration keys and values](#).

### Using Log4j With Existing Logger

Here is an example of how to configure Velocity to log to an existing Log4j Logger.

```
import org.apache.velocity.app.VelocityEngine;
import org.apache.velocity.runtime.RuntimeConstants;

import org.apache.log4j.Logger;
import org.apache.log4j.BasicConfigurator;

public class Log4jLoggerExample
{
    public static String LOGGER_NAME = "velexample";

    public static void main( String args[] )
    {
        throws Exception
        {
            /*
             *  configure log4j to log to console
             */

            BasicConfigurator.configure();

            Logger log = Logger.getLogger( LOGGER_NAME );

            log.info("Log4jLoggerExample: ready to start velocity");

            /*
             *  now create a new VelocityEngine instance, and
             *  configure it to use the category
             */

            VelocityEngine ve = new VelocityEngine();
```



```

        ve.setProperty( RuntimeConstants.RUNTIME_LOG_LOGSYSTEM_CLASS,
            "org.apache.velocity.runtime.log.Log4JLogChute" );

        ve.setProperty("runtime.log.logsystem.log4j.logger",
            LOGGER_NAME);

        ve.init();

        log.info("follows initialization output from velocity");
    }
}

```

Note that the above example can be found in `examples/logger_example`.

### Simple Example of a Custom Logger

Here is an example of how to use an instantiation of your class that implements Velocity's logging system as the logger. Note that we are not passing the name of the class to use, but rather a living, existing instantiation of the class to be used. All that is required is that it support the `LogChute` interface.

```

import org.apache.velocity.runtime.log.LogChute;
import org.apache.velocity.runtime.RuntimeServices;
...

public class MyClass implements LogChute
{
    ...

    public MyClass()
    {
        ...

        try
        {
            /*
             * register this class as a logger with the Velocity singleton
             * (NOTE: this would not work for the non-singleton method.)
             */
            Velocity.setProperty(Velocity.RUNTIME_LOG_LOGSYSTEM, this );
            Velocity.init();
        }
        catch (Exception e)
        {
            /*
             * do something
             */
        }
    }

    /**
     * This init() will be invoked once by the LogManager
     * to give you the current RuntimeServices instance
     */
    public void init(RuntimeServices rsvc)
    {
        // do nothing
    }

    /**
     * This is the method that you implement for Velocity to
     * call with log messages.
     */
    public void log(int level, String message)
    {
        /* do something useful */
    }

    /**
     * This is the method that you implement for Velocity to
     * call with log messages.
     */
    public void log(int level, String message, Throwable t)
    {
        /* do something useful */
    }

    /**
     * This is the method that you implement for Velocity to
     * check whether a specified log level is enabled.
     */
    public boolean isLevelEnabled(int level)

```

```
{
    /* do something useful */
    return someBooleanValue;
}
...
}
```

## Configuring Resource Loaders


### Resource Loaders


One of the fundamental and important parts about Velocity is the resource management system and the resource loaders. They are referred to as 'resources' here rather than 'templates' because the resource management system will also handle non-template resources, specifically things that are loaded via the `#include()` directive.

The resource loader system is very flexible, allowing one or more resource loaders to be in operation at the same time. This allows tremendous flexibility in configuration and resource management, and further allows you to write your own resource loaders for your special needs.

There are currently four kinds of resource loaders that are included with Velocity, each described below. Note that in the example configuration properties given, a common name for the loader is shown (ex. 'file' in `file.resource.loader.path`). This 'common name' may not work for your configuration. Please read the section on [resource configuration properties](#) to understand how this system works. Also, each of these loaders is located in the package `org.apache.velocity.runtime.resource.loader`.

- **FileResourceLoader** : This loader gets resources from the filesystem. Its configuration properties include:
  - `file.resource.loader.path` = <path to root of templates>
  - `file.resource.loader.cache` = true/false
  - `file.resource.loader.modificationCheckInterval` = <seconds between checks>

This is the default loader, and is configured, by default to get templates from the 'current directory'. In the case of using Velocity with servlets, this can be a problem as you don't want to have to keep your templates in the directory from which you start your servlet engine. See the documentation for your servlet or web framework (for example [VelocityViewServlet](#) ) for more info on how to configure the location of the Velocity templates.

- **JarResourceLoader** : This loader gets resource from specific jar files. It is very similar to the FileResourceLoader, except that you have the convenience of bundling your templates into jars. The properties are identical, except for `jar.resource.loader.path`, where you provide the full location of the jar(s) you wish to load resources from. To specify a jar for the loader.path you use the standard JAR URL syntax of `java.net.JarURLConnection`.
- **ClasspathResourceLoader** : This loader gets resources from the classloader. In general, this means that the ClasspathResourceLoader will load templates placed in the classpath (in jars, for example) While the classpath is a source of great pain and suffering in general, it is a very useful mechanism when working on a Servlet Spec 2.2 (or newer) compliant servlet runner. [Tomcat](#)  is an example of such. To use this loader effectively, all you must do is jar your templates, and put that jar into the WEB-INF/lib directory of your webapp. There are no configuration options to worry about, nor is the absolute vs. relative path an issue, as it is with Jar and File resource loaders. Again, please note that the ClasspathResourceLoader is not only for use with a servlet container, but can be used in any application context.
- **URLResourceLoader** : This loader gets resources from a URL connection. Its configuration properties include:
  - `url.resource.loader.root` = <root URL path of templates>
  - `url.resource.loader.cache` = true/false

- `url.resource.loader.modificationCheckInterval` = <seconds between checks>

This loader simply downloads resources from configured URLs. It works much like the `FileResourceLoader`, however, it can pull templates down from any valid URL to which the application can create a connection.

- **DataSourceResourceLoader** : This loader will load resources from a `DataSource` such as a database. This loader is only available under JDK 1.4 and later. For more information on this loader, please see the javadoc for the class `org.apache.velocity.runtime.resource.loader.DataSourceResourceLoader`.

## Configuration Examples

Configuring the resource loaders for Velocity is straightforward. The properties that control the are listed in the [resource configuration](#) section, for further reference.

The first step in configuring one or more resource loaders is to 'declare' them by name to Velocity. Use the property `resource.loader` and list one or more loader names. You can use anything you want - these names are used to associate configuration properties with a given loader.

```
resource.loader = file
```

That entry declares that we will have a resource loader known as 'file'. The next thing to do is to set the important properties. The most critical is to declare the class to use as the loader:

```
file.resource.loader.class =
org.apache.velocity.runtime.resource.loader.FileResourceLoader
```

In this case, we are telling velocity that we are setting up a resource loader called 'file', and are using the class `org.apache.velocity.runtime.resource.loader.FileResourceLoader` to be the class to use. The next thing we do is set the properties important to this loader.

```
file.resource.loader.path = /opt/templates
file.resource.loader.cache = true
file.resource.loader.modificationCheckInterval = 2
```

Here, we set a few things. First, we set the path to find the templates to be `/opt/templates`. Second, we turned caching on, so that after a template or static file is read in, it is cached in memory. And finally, we set the modification check interval to 2 seconds, allowing Velocity to check for new templates.

Those are the basics. What follows are a few examples of different configurations.

**Do-nothing Default Configuration:** As the name says, there is nothing you have to do or configure to get the default configuration. This configuration uses the `FileResourceLoader` with the current directory as the default resource path, and caching is off. As a properties set, this is expressed as:

```
resource.loader = file

file.resource.loader.description = Velocity File Resource Loader
file.resource.loader.class =
org.apache.velocity.runtime.resource.loader.FileResourceLoader
file.resource.loader.path = .
file.resource.loader.cache = false
file.resource.loader.modificationCheckInterval = 0
```

**Multiple Template Path Configuration:** This configuration uses the `FileResourceLoader` with several directories as 'nodes' on the template search path. We also want to use caching, and have the

templates checked for changes in 10 second intervals. As a properties set, this is expressed as:

```
resource.loader = file

file.resource.loader.description = Velocity File Resource Loader
file.resource.loader.class =
org.apache.velocity.runtime.resource.loader.FileResourceLoader
file.resource.loader.path = /opt/directory1, /opt/directory2
file.resource.loader.cache = true
file.resource.loader.modificationCheckInterval = 10
```

**Multiple Loader Configuration :** This configuration sets up three loaders at the same time, the FileResourceLoader, the ClasspathResourceLoader, and the JarResourceLoader. The loaders are set-up such that the FileResourceLoader is consulted first, then the ClasspathResourceLoader, and finally the JarResourceLoader. This would allow you to quickly drop a template into the file template area to replace one of the templates found in the classpath (usually via a jar) without having to rebuild the jar.

```
#
# specify three resource loaders to use
#
resource.loader = file, class, jar

#
# for the loader we call 'file', set the FileResourceLoader as the
# class to use, turn off caching, and use 3 directories for templates
#
file.resource.loader.description = Velocity File Resource Loader
file.resource.loader.class =
org.apache.velocity.runtime.resource.loader.FileResourceLoader
file.resource.loader.path = templatedirectory1, anotherdirectory, foo/bar
file.resource.loader.cache = false
file.resource.loader.modificationCheckInterval = 0

#
# for the loader we call 'class', use the ClasspathResourceLoader
#
class.resource.loader.description = Velocity Classpath Resource Loader
class.resource.loader.class =
org.apache.velocity.runtime.resource.loader.ClasspathResourceLoader

#
# and finally, for the loader we call 'jar', use the JarResourceLoader
# and specify two jars to load from
#
jar.resource.loader.description = Velocity Jar Resource Loader
jar.resource.loader.class =
org.apache.velocity.runtime.resource.loader.JarResourceLoader
jar.resource.loader.path = jar:file:/myjarplace/myjar.jar,
jar:file:/myjarplace/myjar2.jar
```

Note that the three names 'file', 'class', and 'jar' are merely for your convenience and sanity. They can be anything you want - they are just used to associate a set of properties together. However, it is recommended that you use names that give some hint of the function.

Note that while all three require very little configuration information for proper operation, the ClasspathResourceLoader is the simplest.

### Pluggable Resource Manager and Resource Cache

The Resource Manager is the main part of the resource (template and static content) management system, and is responsible for taking application requests for templates, finding them in the available resource loaders, and then optionally caching the parsed template. The Resource Cache is the mechanism that the Resource Manager uses to cache templates for quick reuse. While the default versions of these two facilities are suitable for most applications, for advanced users it now is possible to replace the default resource manager and resource cache with custom implementations.

A resource manager implementation must implement the `org.apache.velocity.runtime.resource.ResourceManager` interface. A description of the requirements of a resource manager is out of scope for this document. Implementors are encouraged to review the default implementation. To configure Velocity to load the replacement

implementation, use the configuration key:

```
resource.manager.class
```

This key is also defined as a constant `RuntimeConstants.RESOURCE_MANAGER_CLASS`

A resource cache implementation must implement the `org.apache.velocity.runtime.resource.ResourceCache` interface. As with the resource manager, a description of the requirements of a resource manager is out of scope for this document. Implementors are encouraged to review the default implementation. To configure Velocity to load the replacement implementation, use the configuration key:

```
resource.manager.cache.class
```

This key is also defined as a constant `RuntimeConstants.RESOURCE_MANAGER_CACHE_CLASS`

A resource cache implementation may want to limit the cache size (rather than providing an unbounded cache which could consume all available memory). To configure Velocity to set the size for your cache, use the configuration key:

```
resource.manager.cache.size
```

This key is also defined as a constant `RuntimeConstants.RESOURCE_MANAGER_CACHE_SIZE`

## Template Encoding for Internationalization

Velocity allows you to specify the character encoding of your template resources on a template by template basis. The normal resource API's have been extended to take the encoding as an argument:

`org.apache.velocity.app.Velocity:`

```
public static Template getTemplate(String name, String
encoding)

public static boolean mergeTemplate( String templateName,
String encoding, Context context, Writer writer )
```

The value for the *encoding* argument is the conventional encoding specification supported by your JVM, for example "UTF-8" or "ISO-8859-1". For the official names for character sets, see [here](#).

Note that this applies only to the encoding of the template itself - the output encoding is an application specific issue.

## Velocity and XML

Velocity's flexibility and simple template language makes it an ideal environment for working with XML data. [Anakia](#) is an example of how Velocity is used to replace XSL for rendering output from XML. The Velocity site, including this documentation, is generated from XML source using Anakia. The Jakarta site is also rendered using Anakia.

Generally, the pattern for dealing with XML in Velocity is to use something like [JDOM](#) to process your XML into a data structure with convenient Java access. Then, you produce templates that access data directly out of the XML document - directly through the JDOM tree. For example, start with an XML document such as:

```
<document>
  <properties>
    <title>Developer's Guide</title>
    <author email="geirm@apache.org">Velocity Doc Team</author>
  </properties>
</document>
```

Now make a little Java program that includes code similar to:

```
...
SAXBuilder builder;
Document root = null;

try
{
    builder = new SAXBuilder(
        "org.apache.xerces.parsers.SAXParser" );
    root = builder.build("test.xml");
}
catch( Exception ee)
{}

VelocityContext vc = new VelocityContext();
vc.put("root", root.getRootElement());

...
```

(See the Anakia source for details on how to do this, or the Anakia example in the examples directory in the distribution.) Now, make a regular Velocity template:

```
<html>
  <body>
    The document title is
    $root.getChild("document").getChild("properties")
    .getChild("title").getText()
  </body>
</html>
```

and render that template as you normally would, using the Context containing the JDOM tree. Of course, this isn't the prettiest of examples, but it shows the basics - that you can easily access XML data directly from a Velocity template.

One real advantage of styling XML data in Velocity is that you have access to any other object or data that the application provides. You aren't limited to just using the data present in the XML document. You may add anything you want to the context to provide additional information for your output, or provide tools to help make working with the XML data easier. Bob McWhirter's [Werken Xpath](#) is one such useful tool - an example of how it is used in Anakia can be found in `org.apache.velocity.anakia.XPathTool`.

One issue that arises with XML and Velocity is how to deal with XML entities. One technique is to combine the use of Velocimacros when you need to render an entity into the output stream:

```
## first, define the Velocimacro somewhere

#macro(xenc $sometext)$tools.escapeEntities($sometext)#end

## and use it as

#set( $sometext = " < " )
<text>#xenc($sometext)</text>
```

where the `escapeEntities()` is a method that does the escaping for you. Another trick would be to create an encoding utility that takes the context as a constructor parameter and only implements a method:

```
public String get(String key)
{
    Object obj = context.get(key)
    return (obj != null)
        ? Escape.getText( obj.toString() )
        : "";
}
```

Put it into the context as "xenc". Then you can use it as:

```
<text>$xenc.sometext</text>
```

This takes advantage of Velocity's introspection process - it will try to call `get("sometext")` on the `$xenc` object in the Context - then the `xenc` object can then get the value from the Context, encode it, and return it.


Alternatively, since Velocity makes it easy to implement custom Context objects, you could implement your own context which always applies the encoding to any string returned. Be careful to avoid rendering the output of method calls directly, as they could return objects or strings (which might need encoding). Place them first into the context with a `#set()` directive and the use that, for example:

```
#set( $sometext = $jdomElement.getText() )
<text>$sometext</text>
```

The previous suggestions for dealing with XML entities came from Christoph Reck, an active participant in the Velocity community.

## Summary

We hope this brief guide was a helpful introduction to using Velocity in your Java projects, and thank you for your interest in Velocity. We welcome any and all comments you may have about this documentation and the Velocity template engine itself.

Please submit all detailed, thoughtful and constructive feedback through our [mailing lists](#) .