Client and server-side templating with Velocity

Flexible templating engine provides a legacy-free alternative to JSP technology

Sing Li

February 10, 2004

Velocity is a versatile, open source templating solution that can be used standalone in report generation/data transformation applications, or as a view component in MVC model frameworks. In this article, Sing Li introduces Velocity and reveals how you can integrate its template-processing capabilities into your own client-side standalone application, server-side Web application, or Web services.

The manipulation and transformation of textual data in standard presentation or exchange formats such as HTML and XML is a frequent and often tedious activity in which every software developer engages. A templating engine can facilitate this process by maintaining the static portion of the output in templates, while dynamically generating and positioning the changing portions. Velocity is a highly functional, open source templating engine that you can easily integrate into either client-side or server-side applications.

For server-side applications, when integrated with Web tier containers compatible with Servlet 2.3+, Velocity provides a viable alternative to JSP technology that can enforce a clean separation of presentation logic from application business logic. In fact, the templating language supported by Velocity is simple enough and the resulting templates are sufficiently uncluttered that Web site designers and stylists can learn and maintain these templates.

In this article, we will examine Velocity's simple templating language, create some templates, and apply them in a standalone client application. We'll then integrate the template engine into the Struts MVC framework as a view component.

Basic template engine operation

The operation of a basic templating engine is quite straightforward. First consider the template in Listing 1:
Listing 1. A basic Velocity template

```html
<html><head><title>A Template Based Page</title></head><body><p>This is a page generated by $generatedBy.</p><p>The customer's name is $customerName.</p></body></html>
```

This template is a complete HTML file. You can create it using a text editor or a favorite graphical visual Web page editor. This ease of creation is the prime benefit and appeal of a template-based system.

The bold lines in the Listing 1 template will be replaced by actual data when the templating engine runs. The process of taking the data and combining it into a template is called merging. Consider the data represented in the script in Listing 2:

Listing 2. Setting data values for template merging

```bash
#set ($generatedBy = "Velocity")
#set ($customerName = "John Doe")
```

Now, if the Listing 1 template is merged with the Listing 2 data, the result will be as shown in Listing 3:

Listing 3. Data merged with template

```html
<html><head><title>A Template Based Page</title></head><body><p>This is a page generated by Velocity.</p><p>The customer's name is John Doe.</p></body></html>
```

You may recognize this feature as being similar to the mail merge feature in a word processor, in which form letters are merged with names and addresses from a mailing list. As with mail merge, the application is most useful when the data source being merged is large and varied.

Velocity is a templating engine in this pure sense. The output format of Velocity is limited only by what can be placed in a text template. This includes the most popular formats today (HTML, XML, SQL, and so on).

Creating templates with the Velocity Template Language

Velocity templates are text files (HTML, XML, and so on) containing:
• Static portions that will be merged as-is
• Placeholders that will be replaced with merged data
• Directives and instructions in a scripting language

The scripting language used with Velocity templates is known as the Velocity Template Language (VTL). The VTL syntax is relatively lean when compared to other scripting languages. For anyone with a programming background, learning VTL should be a very quick task.

Placeholders and references

A reference in VTL is a named element such as $customerName. References can act as data placeholders in Velocity templates. These placeholders are replaced with their textual value during the template merge to create the final output. For example, back in Listing 1, we can see two VTL references ($generatedBy and $customerName) used to create the final output. A variable is one type of reference in VTL. You can use the #set() directive to assign values to variables. Listing 4 shows some examples:

Listing 4. Variable typed VTL references

```
#set( $this = "Velocity")
#set( $numericBase = 999 )
#set( $booleanCondition = true )

This page is generated using $this.
There are ($numericBase + 1) pages in total.
```

Variable names must begin with a letter, which makes it easy for Velocity to distinguish variable names from currency notation within a template (for instance, $100 cannot be a variable name). All variables are converted to strings during the merge operation, which can create some interesting situations. Consider the text highlighted in red in Listing 4. The merged output is shown in Listing 5:

Listing 5. Variable with numeric value in merged template

```
This page is generated using Velocity.
There are (999 + 1) pages in total.
```

Because $numericBase is converted to a string during the merge operation, no arithmetic operation is performed. Because it's specialized for template operations and is not a general computing language, VTL needs to support only integer math (although plug-in tools can be used to extend this). The script below reveals how to work with this math capability:

```
#set( $newSum = $numericBase + 1)

There are $newSum pages in total.
```

The corresponding output after merging this template will be:

```
There are 1000 pages in total.
```
Property and method references

Other than variables that we set within the template, VTL references can also be object properties or methods. These objects are Java classes that are made available to the template (typically through the context, see Velocity contexts).

Properties of objects are accessed through similar JavaBean notations. For example, you can access the LastName property of the $customer object through the VTL reference $customer.LastName. Under the hood, Velocity uses an object's accessor method to obtain the property value (that is, the getLastName() method of the object is invoked).

You can invoke methods of an object, with or without a parameter list, with a familiar notation to property access. For example, you may call the getPhone() method of the $customer object to obtain a mobile phone number using the VTL reference $customer.getPhone("mobile").

Thus far, we have only dealt with scalar variables. To create an ArrayList variable consisting of multiple elements, use this syntax: 

```velocity
#set( $treeList = ["pine", "oak", "maple", "redwood"])
```

You can obtain the second item in the list by using $treeList.get(1).

After the assignment, $treeList is an ArrayList-based variable (as in the standard JDK collection class). You can access each of its elements directly using the notation $treeList.get(n), where n is a zero-based index ArrayList. For example, as indicated by the red highlighted line in Listing 6, $treeList.get(1) is used to select the second item in the ArrayList: oak. This syntax of invoking a method of the ArrayList class can be used to invoke methods of other available objects (see the Property and method references sidebar for more information).

Now, a word about placeholder substitution: Velocity will print any unrecognized reference as plain text, as illustrated by the next two bold lines in Listing 6:

### Listing 6. Placeholder substitution

The second item in the list is oak.

$notDeclared is an undeclared variable.

But $!notDeclared is invisible when not declared.

VTL supports a quiet reference notation to avoid rendering nonexistent or null references. If you use the quiet reference notation, as in $!notDeclared, then Velocity will render nothing to the output instead of the full reference name. Note the "!" before the variable name to signify the quiet reference notation. When the Listing 6 template is merged, both references are not assigned any value, but the first bold reference will be rendered as-is, while the second one will be invisible:

The second item in the list is oak.

$notDeclared is an undeclared variable.

But is invisible when not declared.

Selective rendering and repetition

You can use the #if... #then... #else... directive to conditionally render a certain portion of a template. Listing 7 shows an example:
Listing 7. Selective rendering using #if, #then, and #else

```velocity
#if $customer.GoldMember
   Thank you Mr. $customer.LastName, for flying with us.
   Your loyal patronage is greatly appreciated.
   This flight earns you an additional 5000 miles.
#else
   Thank you for flying with us.
   Please consider joining our frequent flyer program.
#endif
```

In the Listing 7 template, the boolean GoldMember property of the $customer object is used to determine which message to show on the final output. For a gold member, the first bold message will be rendered; otherwise, the second bold message will be displayed in the final output.

Repetition is required frequently in templates to format information in a tabular or list format. The data displayed is typically kept in an ArrayList reference. The only directive for handling repeated looping in Velocity is the #foreach directive. The Listing 8 template illustrates the use of the #foreach directive with our $treeList ArrayList variable. Of course, any other available collection-typed object reference, or an object property/method reference returning a collection, may also be used.

Listing 8. Repetition using #foreach

```velocity
<table>
<tr><td>Tree Name</td></tr>
#foreach $name in $treeList
<tr><td>$name is a big tree!</td></tr>
#end
</table>
```

With the $treeList containing the list of tree names, the merged output of the Listing 8 template will look like Listing 9:

Listing 9. Merged output from the #foreach loop

```velocity
<table>
<tr><td>Tree Name</td></tr>
<tr><td>pine is a big tree!</td></tr>
<tr><td>oak is a big tree!</td></tr>
<tr><td>maple is a big tree!</td></tr>
<tr><td>redwood is a big tree!</td></tr>
</table>
```

When viewed from an HTML browser, of course, Listing 9 is a table of tree names.
Note that there is a "built-in" counter inside the body of a #foreach loop. It is accessible through the $velocityCounter reference within the body of the #foreach directive. By default, this counter starts at 1 and is incremented each time the loop is taken.

**Macros in Velocity**

One main feature of Velocity is its ability to easily define macros, called *Velocimacros*. Macros enable you to encapsulate template scripts and reuse them easily. By default, they are kept in a `VM_global_library.vm` file. For example, consider the Velocimacro called `#showTree()` in Listing 10:

### Listing 10. Defining a Velocimacro

```velocity
#macro (showTree)
    #if ($treeList )
        #foreach ($e in $treeList )
            $e
        #end
    #end
#end
```

You can invoke the `#showTree()` Velocimacro and use it to print out the `$treeList` ArrayList, provided the list is defined. The invocation syntax is simply `#showTree()`.

It is also possible for you to parameterize a macro. For example, we can modify the `#showTree()` macro to work on any list, as shown in Listing 11:

### Listing 11. A Velocimacro with parameters

```velocity
#macro (showList $val)
    #if ($val )
        #foreach ($e in $val )
            $e
        #end
    #end
#end
```

To invoke the Listing 11 macro with our `$treeList`, we can use `#showList($treeList)`. The output will be identical in either case, as shown in Listing 12:

### Listing 12. Merged output from Velocimacro

```
pine
oak
maple
redwood
```

**Other interesting VTL details**

Single-line comments or end-of-line comments are preceded by `##`. Multi-line comments are bracketed by the `#*` and `*#` pair.

When working with string data, you can use either double quotation marks or single quotation marks to delimit them. However, the use of double quotation marks allows evaluation of Velocity references, directives, or even Velocimacros *inside* the delimited string.
Velocity contexts

You can think of a context in velocity as a means to import Java objects for access within Velocity templates. This import must be done explicitly in Java coding. Unlike JSP code or JavaScript, there is no "natural" or "native way" for Velocity templates to access arbitrary Java objects. Only explicitly imported Java objects will be available within a Velocity template.

You can obtain a Velocity context by creating an instance of the org.apache.velocity.context.Context class. You can then attach objects that are to be imported for template use to the context using the put(key, value) method of the context. The key will be a string name that will appear as an available reference inside the templates. In production scenarios, graphic or Web designers may be in charge of creating and maintaining templates, while Java developers provide the set of objects that will be accessible inside the templates. In these cases, the designers and developers will collaborate by agreeing on the set of objects and their available properties. The attached attributes in Velocity context will be the main interfacing mechanism.

Accessing context attributes in templates

Take a look at the included example code (see Related topics) for a standalone parser. Look under the \code\src directory. For example, in our com.ibm.dvworks.velocity.VelocityParser class, we have created and added two attributes to the Velocity context, as shown in Listing 13:

Listing 13. Creating an instance of Velocity in our VelocityParser class

```java
public static void main(String[] args)    {
    VelocityParser velInstance = new VelocityParser(args[0]);
    velInstance.addToContext(  "treeFarm",
        new String [] { "redwood", "maple", "oak", "pine" });
    velInstance.addToContext( "title", "A Tree Farm");
    velInstance.addToContext( "date", new java.util.Date());
    velInstance.addToContext("fmtr",
        new org.apache.velocity.app.tools.VelocityFormatter(
            velInstance.getCurrentContext()));
    velInstance.processTemplate();
}
```

The attribute treeFarm is an ArrayList of tree names. The title attribute is a scalar string. Once attached to the context and passed in during the merge operation, these attributes are immediately usable within the Velocity templates. The template in Listing 14 uses these two attributes. You can find it in \code\app\treectx.vm.

Listing 14. Using the $treeFarm context attribute reference

```
<table>
<tr><td>$title</td></tr>
#foreach $name in $treeFarm
<tr><td>$name is a big tree!</td></tr>
#end
</table>
```

After merging, the output looks like Listing 15:
Listing 15. Merged output from the template

```html
<table>
<tr><td>A Tree Farm</td></tr>
<tr><td>redwood is a big tree!</td></tr>
<tr><td>maple is a big tree!</td></tr>
<tr><td>oak is a big tree!</td></tr>
<tr><td>pine is a big tree!</td></tr>
</table>
```

Note that the syntax of using the $treeFarm context attribute reference is identical to the $treeList variable reference examined earlier.

**Initializing the template engine**

Examine the `main()` method listing of the `VelocityParser` class (see Listing 13). `VelocityParser`'s constructor creates the parser and loads the template, then attributes to be used by the template engine are added, and finally `processTemplate()` is called to merge the data and template. We will examine each of these methods in turn.

You can use the static methods in the `org.apache.velocity.app.Velocity` class to initialize Velocity and load a template file. The methods to use are `init()` and `getTemplate()`, respectively. The call to the `init()` method occurs within the constructor of our `VelocityParser` class, as shown in Listing 16:

**Listing 16. Initializing the template engine in the constructor of our VelocityParser class**

```java
public VelocityParser(String templateFile) {
    try {
        Velocity.init("velocity.properties");
        mainTemplate = Velocity.getTemplate(templateFile);
    } catch( Exception ex ) {
        System.out.println("Error processing template file: " + templateFile );
    }
}
```

In Listing 16, the call to `init()` creates a single instance of the Velocity engine. If your application needs to create and manage multiple instances of the Velocity template engine, use the `org.apache.velocity.app.VelocityEngine` class instead.

**Context chaining**

An immediately usable Velocity context is created by simply calling the trivial constructor of the `org.apache.velocity.VelocityContext` class.
Velocity context can be **chained** (wrapped one inside another). This can be quite useful when you want to control the visibility or availability of certain object references in templates.

Velocity will search the attributes of all the chained contexts for object references. If there are duplicate names, the outer-most attribute will be used, and the inner attribute(s) with the same name will not be accessible.

To chain Velocity contexts, the context to be chained is passed as an argument to the constructor of a new context. The overloaded `addToContext()` method in our `VelocityParser` class, shown in Listing 17, illustrates this:

**Listing 17. Adding context attribute or context chaining with the addToContext() method**

```java
public void addToContext(String key, Object value) {
    if (mainContext == null)
        mainContext = new VelocityContext();
    mainContext.put(key, value);
}

public void addToContext(VelocityContext chainCtx) {
    mainContext = new VelocityContext(chainCtx);
}
```

In the `processTemplate()` method, the template’s `merge()` method is called to combine the context information with the template and generate the output stream, as illustrated in Listing 18:

**Listing 18. Merging templates in the processTemplate() method**

```java
public void processTemplate() {
    try {
        BufferedWriter writer = writer = new BufferedWriter(
            new OutputStreamWriter(System.out));
        if (mainTemplate != null)
            mainTemplate.merge(mainContext, writer);
        writer.flush();
        writer.close();
    } catch( Exception ex )    {
        ex.printStackTrace();
    }
}
```

**Velocity as a standalone parser**

To compile the example standalone parser, use the `compile.bat` file in the `\code\app` directory of the distribution. To try out the parser, use the `process.bat` batch file, which contains:

```
set VEL_HOME=\jdk1.4\vel14rc1
java -classpath ..\classes;%VEL_HOME%\velocity-dep-1.4-rc1.jar
com.ibm.dvworks.velocity.VelocityParser %1 %2 %3
```

Note you must set the `VEL_HOME` environment variable in both `compile.bat` and `process.bat` to the directory where Velocity is installed. There are two different JAR files included with Velocity distribution: `velocity-dep-????.jar` (where `???` is the release number info), and `velocity-????.jar`. 
The `velocity-dep-???.jar` file includes all external dependencies (Jakarta common-collections, Avalon Logkit, and ORO regular expression library) and can be used immediately. If you already have some of these libraries in your classpath, you may want to use the `velocity-???.jar` file instead. If neither of these JAR compositions suits your needs exactly, you can easily rebuild Velocity to be just the way you need it. The Velocity distribution includes an ant script to build seven different JAR configurations for different application scenarios.

To provide a pleasant out-of-the-box experience, Velocity comes with a set of default configuration properties that are reasonable and acceptable for most applications. This frees the developer from tinkering with complex configuration options at the initial get-go and allows him or her to experiment immediately with the template engine.

**Velocity vs. JSP technology on the server**

On the server side, you can use Velocity to process templates and generate dynamic content (HTML, XML, and so on). This is very similar in purpose to JSP technology. However, the JSP model provides unhindered access to the underlying Servlet API and Java programming language. In fact, to avoid access to these native features, you will have to exercise great discipline in coding (using only EL, tag libraries, and the like). Essentially, this is a wide-open access model.

Contrast this with Velocity. Being a completely self-contained template engine and script interpreter, Velocity has a completely closed model. Access to anything specific to the system and/or the Java programming language must be explicitly enabled. By default, there is no access within a Velocity template to any facet of the Java programming language. This closed model allows Velocity to provide a decoupled templating presentation layer, cleanly separated from any application business logic or data management code.

Let's now see server-side Velocity in action by integrating the templating engine to the latest Tomcat 5 release.

**Deploying Velocity with Tomcat 5**

The Velocity distribution comes with an `org.apache.velocity.servlet.VelocityServlet` library class that you can extend to quickly create a template processing servlet. Any templates tested on the standalone client application can be deployed on the server using `VelocityServlet`. Moving your standalone Velocity templates to a Web application is relatively easy. Simply perform the following steps:

1. Derive a Servlet class from the `org.apache.velocity.servlet.VelocityServlet` class.
2. Override and implement one of the `handleRequest()` methods.
3. In the implementation of `handleRequest()`, add data or tools (see the Tools in Velocity sidebar) that you want to use within the template as attributes to the context.
4. In the implementation of `handleRequest()`, fetch the template from a file or a resource (such as a JAR file) and return it.

---

**Tools in Velocity**

Tools are utility Java objects that are made available to templates through the Velocity context. While you can attach such objects manually to the context, the toolbox manager...
of VelocityViewServlet can do this in a flexible and structured manner. Typically, tools are used inside templates by invoking their methods. Both VelocityViewServlet and VelocityStruts provide valuable pre-tested tools that are extremely useful when deploying Velocity as a view technology.

In the example code distribution, com.ibm.dvworks.velocity.VelTestServlet is a servlet that we have created following the above steps. You can examine the code under the webapps\vServlet \WEB-INF\src directory. If you modify the code, make sure you recompile it using the compile.bat batch file.

The deployment descriptor (the web.xml file) defines the servlet and maps it to the /vServlet URL pattern, as shown in Listing 19:

Listing 19. Tomcat deployment descriptor for custom Velocity-based servlet

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
 PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <servlet>
    <servlet-name>vServlet</servlet-name>
    <servlet-class>com.ibm.dvworks.velocity.VelTestServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>vServlet</servlet-name>
    <url-pattern>/vServlet</url-pattern>
  </servlet-mapping>
</web-app>
```

The template to be loaded and processed is placed in the webapps\vServlet directory. In our case, it is called variables.vm. To test, make sure you have the velocity-dep-???.jar file placed in the webapps\vServlet\WEB-INF\lib directory, then start up Tomcat 5 and access the http://localhost:8080/vservlet/Servlet URL.

Deploying VelocityViewServlet

For extensive templating work in a Web application, you should use VelocityViewServlet from the Velocity tools collection. Velocity tools is a subproject of Velocity (see Related topics for the URL to download the latest version). This servlet provides more complete support for using Velocity as a view layer technology, either working in conjunction with or instead of JSP technology. Using VelocityViewServlet can save you significant redundant coding, since it provides:

- Direct template access to the request object and attributes, the session object and attributes, and the servlet context and attributes
- A formalized, externally configurable "toolbox" for adding custom tools that can be used by the template (tools in this sense are just compiled Java classes that have public methods)
- A library of versatile, tested, ready-to-use tools

To integrate VelocityViewServlet into your Web application, take a look at the sample velview Web application (in the webapps\velview directory). This application includes the template that we
have been working with throughout this article. In addition, it also displays attributes from request, session, and servlet context objects. The integration steps are as follows:

First, make sure that the `velocity-tools-view.jar` file is in the application's `lib` directory. Of course, the velocity JAR files should also be located here.

In the deployment descriptor `web.xml` file, include `VelocityViewServlet`. The initialization parameter is a toolbox description XML file. The servlet is mapped to handle all files with a `.vm` extension, as shown in Listing 20:

**Listing 20. Tomcat deployment descriptor (web.xml) for VelocityViewServlet**

```xml
<?xml version="1.0" encoding="ISO-8859-1"?><!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd"><web-app>
  <servlet>
    <servlet-name>velocityView</servlet-name>
    <servlet-class>org.apache.velocity.tools.view.servlet.VelocityViewServlet</servlet-class>
    <init-param>
      <param-name>org.apache.velocity.toolbox</param-name>
      <param-value>/WEB-INF/toolbox.xml</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>velocityView</servlet-name>
    <url-pattern>*.vm</url-pattern>
  </servlet-mapping>
</web-app>
```

In the toolbox descriptor (toolbox.xml) file of this example, two generic tools from the available Velocity tools library are included and made accessible within the template: `DateTool` and `MathTool`. These tools enable us to format date and time information, and perform floating point math within our templates, as shown in Listing 21:

**Listing 21. Toolbox descriptor including DateTool and MathTool**

```xml
<?xml version="1.0"?><toolbox>
  <tool>
    <key>date</key>
    <scope>application</scope>
    <class>org.apache.velocity.tools.generic.DateTool</class>
  </tool>
  <tool>
    <key>math</key>
    <scope>application</scope>
    <class>org.apache.velocity.tools.generic.MathTool</class>
  </tool>
  ...
</toolbox>
```

A set of commonly used standard tools is available with `VelocityViewServlet`, as shown in Table 1:

**Table 1. Standard tools available with VelocityViewServlet**
<table>
<thead>
<tr>
<th>Tool Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LinkTool</td>
<td>Works with URIs. Very frequently used. You will need this tool whenever you</td>
</tr>
<tr>
<td></td>
<td>create a clickable link within your template. It can generate the context-</td>
</tr>
<tr>
<td></td>
<td>dependent portion of the URI.</td>
</tr>
<tr>
<td>CookieTool</td>
<td>Enables templates to create or access browser-cached cookies.</td>
</tr>
<tr>
<td>ParameterParser</td>
<td>Simplifies the parsing of incoming request parameters.</td>
</tr>
</tbody>
</table>

There are also two highly specialized, less frequently used tools, as shown in Table 2:

**Table 2. Specialized VelocityViewServlet tools**

<table>
<thead>
<tr>
<th>Tool Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ViewRenderTool</td>
<td>Enables templates to parse strings that contain VTL.</td>
</tr>
<tr>
<td>AbstractSearchTool</td>
<td>Provides a skeletal tool (must be extended with your own custom Java code)</td>
</tr>
<tr>
<td></td>
<td>for the implementation of online searches and search result pagination.</td>
</tr>
</tbody>
</table>

You can use the [http://localhost:8080/velview/variables.vm](http://localhost:8080/velview/variables.vm) URL to test the velview application. You should examine the template source code to see the Velocity engine, the LinkTool, and the CookieTool at work.

**Interoperating with the Struts framework**

Struts is a popular Web application building framework based on the MVC model. The default view component technology for Struts is JSP technology. However, you can easily integrate Velocity as the view component. Figure 1 illustrates this specific use of Velocity:

**Figure 1. Integrating Velocity with the Struts MVC framework**

![Diagram of Struts MVC framework with VelocityViewServlet integration](image-url)
It is important to note that Velocity does not displace JSP technology in this composition. Instead, JSP technology and Velocity templates can work alongside one another. To integrate Velocity, configure `VelocityViewServlet` to process .vm templates, as we described in the Deploying `VelocityViewServlet` section. This means that .jsp files will continue to be processed by the container (that is, Jasper in Tomcat 5), while any .vm templates are passed to Velocity.

The `VelocityStruts` component of the Velocity Tools subproject (see Related topics) has everything you need to integrate Velocity with Struts. `velocitystruts` provides a set of specialized Velocity tools to access Struts-specific resources and information within a Velocity template. Table 3 provides a brief list of the most frequently used tools:

**Table 3. Tools for VelocityStruts integration**

<table>
<thead>
<tr>
<th>Tool Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StrutsLinkTool</td>
<td>Specialized version of the LinkTool for working with Struts targets. Provides <code>setAction()</code> and <code>setForward()</code> to access the pre-configured action mappings.</td>
</tr>
<tr>
<td>FormTool</td>
<td>Accesses Struts's form beans.</td>
</tr>
<tr>
<td>ErrorsTool</td>
<td>Works with Struts error messages, including support for internationalization.</td>
</tr>
<tr>
<td>MessageTool</td>
<td>Provides access to Struts's internationalization support, more specifically the language-dependent message resources.</td>
</tr>
</tbody>
</table>

There is also a set of tools available specifically for working with the new features in Struts 1.1, as shown in Table 4:

**Table 4. Specialized Struts 1.1 access tools**

<table>
<thead>
<tr>
<th>Tool Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SecureLinkTool</td>
<td>Works with Struts 1.1's secured link (SSL) extension.</td>
</tr>
<tr>
<td>ActionMessagesTool</td>
<td>Provides access to Struts 1.1's new <code>ActionMessages</code> object.</td>
</tr>
<tr>
<td>TilesTool</td>
<td>Provides access to Struts 1.1's Tiles extension support.</td>
</tr>
<tr>
<td>ValidatorTool</td>
<td>Provides access to Struts 1.1's Validator extension, generating code to validate form input fields.</td>
</tr>
</tbody>
</table>

In the `webapps\struts-example` directory, you will see an example of using Struts instead of JSP technology to create Struts pages. In this case, we replace the first title page of the example Web application distributed with Struts. You may want to try your hand at replacing other pages. The following list describes the steps involved.

1. Copy the Velocity libraries into the `WEB-INF\lib` directory of the struts-example application.
   Using Tomcat 5 (5.0.16 is latest at the time of writing) and Struts 1.1, you will need to copy the following JAR files into the `webapps\struts-example\WEB-INF\lib` directory:
   - `velocity-tools-1.1-beta1.jar`
   - `velocity-1.4-rc1.jar`
2. Next, in the Struts configuration file (WEB-INF\struts-config.xml), the Struts action mapping is set to forward to the index.vm file instead of the index.jsp file, as shown in Listing 22:

```
Listing 22. Modifying Struts action forward to \index.vm

<action path="/logoff"
type="org.apache.struts.webapp.example.LogoffAction">
    <forward name="success" path="/index.vm"/>
</action>
```

3. In the deployment descriptor WEB-INF\web.xml file, configure VelocityViewServlet to process .vm files. Also set the welcome file to index.vm instead of index.jsp, as shown in Listing 23:

```
Listing 23. Modifying the deployment descriptor of the struts-example Web application

<!-- Action Servlet Configuration -->
<servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
        <param-name>config</param-name>
        <param-value>/WEB-INF/struts-config.xml,
            /WEB-INF/struts-config-registration.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet>
    <servlet-name>velocity</servlet-name>
    <servlet-class>org.apache.velocity.tools.view.servlet.VelocityViewServlet</servlet-class>
    <init-param>
        <param-name>org.apache.velocity.toolbox</param-name>
        <param-value>/WEB-INF/toolbox.xml</param-value>
    </init-param>
    <init-param>
        <param-name>org.apache.velocity.properties</param-name>
        <param-value>/WEB-INF/velocity.properties</param-value>
    </init-param>
</servlet>

<!-- Action Servlet Mapping -->
<servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>

<servlet-mapping>
    <servlet-name>velocity</servlet-name>
    <url-pattern>*.vm</url-pattern>
</servlet-mapping>

<!-- The Welcome File List -->
<welcome-file-list>
    <welcome-file>index.vm</welcome-file>
</welcome-file-list>
```

4. Last but not least, copy the toolbox.xml and velocity.properties files from this article's source code download (see Related topics) to the WEB-INF directory.

The new index.vm file is shown in Listing 24. You may want to compare this with the original index.jsp file.
Listing 24. Struts interoperation through use of the index.vm Velocity template

```html
<html>
<head>
<title>$msg.get("index.title")</title>
</head>
<body bgcolor="white">

#if ( !$application.database)
  <font color="red">
    ERROR: User database not loaded -- check servlet container logs for error messages.
  </font>
  <hr>
#end

<h3>$msg.get("index.heading")</h3>
<ul>
<li>
  <a href="$link.setURI("editRegistration.do").addQueryData("action","Create")">
    $msg.get("index.registration")
  </a>
</li>
<li>
  <a href="$link.setURI("logon.jsp")">
    $msg.get("index.logon")
  </a>
</li>
</ul>
<p>&nbsp;</p>
<a href="$link.setURI("tour.do")">
  $msg.get("index.tour")
</a>
<p>&nbsp;</p>
<img src="$link.setURI("powered-by-logo.gif")" alt="Powered by Velocity"/>

</body>
</html>
```

In index.vm, the message tool in $msg is used throughout this template to access Struts's locale-dependent, internationalized resources. This approach eliminates most hard-coded strings in the template, localizing changes to the resource bundle containing the internationalized strings.

You can use the conditional #if directive of VTL to directly check for the existence of the database attribute in the servlet context. The $application reference can be used to access any attribute in the servlet context ($request, $response, and $session are also available to access attributes of other Servlet API objects).

The setURI() method of the LinkTool is used to generate a server-side URI link to Struts's actions, as well as to the "Powered by Velocity" logo image. Note the use of the addQueryData() method of the LinkTool to append additional action information to a resulting URI.

You can test the Velocity page by starting Tomcat 5 and accessing the http://localhost:8080/struts-example/ URL. Note how it works identically to the original JSP version.

**Conclusions**

The Velocity template processor can be directly integrated into your own Java language application, immediately providing report-generation or template-processing features.
Extending the template engine to Web applications, VelocityServlet can be used to process Velocity templates that generate HTML output dynamically. The Velocity tools project provides further support for componentized Web tier application development with VelocityViewServlet. VelocityViewServlet provides a convenient view layer for construction of Web-based UI based on templates.

When designing complex Web applications using MVC model frameworks, Velocity as a view/templating technology -- in the form of VelocityViewServlet -- can be readily plugged into the framework. In the case of the popular Jakarta Struts MVC framework, Velocity can work alongside JSP-based view technology and interoperates with any choice of model technology.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>j-velocity-code.zip</td>
<td>23 KB</td>
</tr>
</tbody>
</table>
Related topics

- For the latest version of the Velocity templating engine, associated documentation, mailing lists, and community news, visit the official Velocity site.
- The Struts MVC framework is a very popular open-source framework for creating Web applications. Velocity can be used as the view technology for Struts, alongside JSP technology.
- Turbine is an open-source MVC framework designed from scratch to work well with Velocity as the view component technology.
- To use WebSphere Studio V5 to create Struts-based Web applications, check out this series of articles by Colin Yu and Jane Fung (November 2003).