WIRTSCHAFTSUNIVERSITÄT WIEN
BAKKALAUREATSARBEIT

Titel der Bakkalaureatsarbeit:

Apache Velocity

Englischer Titel der Bakkalaureatsarbeit:

Apache Velocity

Verfasserin/Verfasser:	**Markus Auchmann**
Matrikel-Nr.:	**0451493**
Studienrichtung:	**J033 526 Bakkalaureat Wirtschaftsinformatik**
Kurs:	**1526 Projektseminar**
Textsprache:	**Englisch**
Betreuerin/Betreuer:	**Ao. Univ. Prof. Dr. Rony G. Flatscher**

Ich versichere:

dass ich die Bakkalaureatsarbeit selbstständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubten Hilfe bedient habe.

dass ich die Ausarbeitung zu dem obigen Thema bisher weder im In- noch im Ausland (einer Beurteilerin/ einem Beurteiler zur Begutachtung) in irgendeiner Form als Prüfungsarbeit vorgelegt habe.

dass diese Arbeit mit der vom Betreuer beurteilten Arbeit übereinstimmt.

Datum _____ Unterschrift _____

Table of Contents

Figures

1 Introduction

1.1 About this paper

This paper is about the Velocity Engine project by Apache and should give you an overview about how a Template Engine works and what the benefits and disadvantages are. As mentioned this paper will focus on the Velocity Engine project by Apache. At first you'll get an introduction about Template Engines and competing projects of Velocity. Afterwards you'll find a short introduction about Velocity and detailed code examples how to use it. The next points are some sample applications with Velocity in different application areas.

1.2 Terms and Definitions

1.2.1 Template Engine

"A template engine is a code generator that emits text using templates embedded with actions or data references. Engines generally espouse a model-view pattern in an attempt to separate the data source and template." [Antl06]

Template Engines encourage clean separation of content, graphic design, and program code. This leads to more modular, flexible, and reusable site architectures, shorter development time, and code that is easier to understand and maintain. Template Engines facilitate the construction of various formatted documents by allowing a static template to contain placeholders for dynamic output.

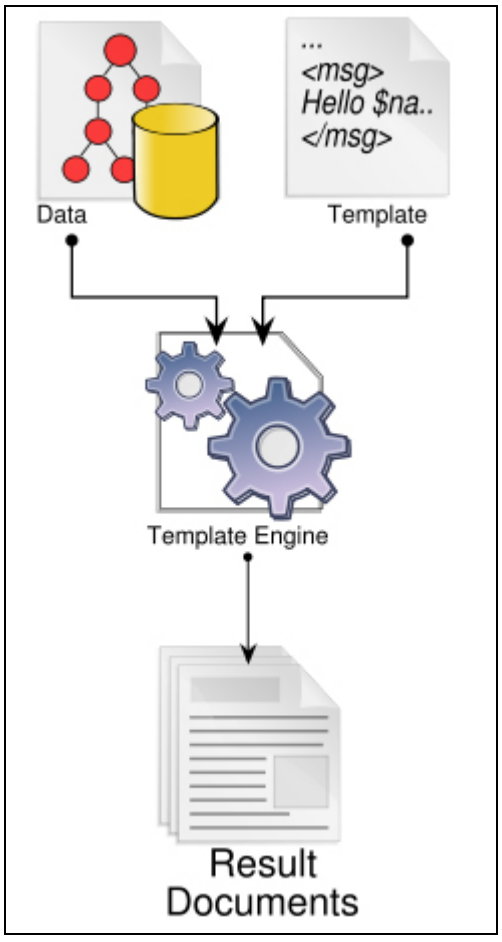
A Template processing system has several elements, but needs at least these:

- A data model which can be a Database, like MySQL, or a simple text file.
- A source template, which is the general layout of the result document without any data in. The data, which should be inserted in the result document, is described in an appropriate programming language.

- A Template Engine which is responsible for connecting to the data model, processing the code specified in the template and creating the result document.

- The result document is the Template file, merged by the Template Engine, with the data model. The result document can either be a web site, some code fragments, SQL source code or any other format specified in the template.

The general process of creating a result file is described in the figure below.



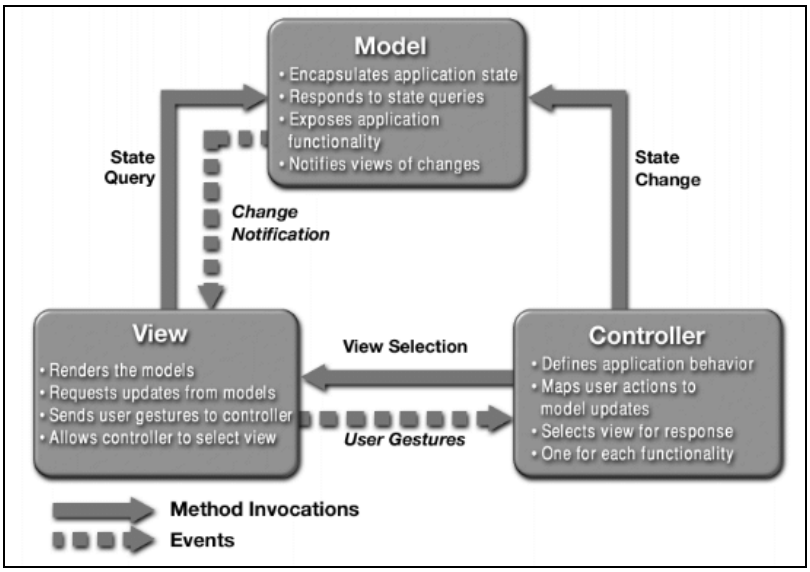
[Figure 1, Template Engine]

1.2.2 Web Template Engine

Web Template Engines are designed to produce web pages or web documents to be delivered over the internet.

1.2.3 Model View Controller

„By applying the Model-View-Controller (MVC) architecture to a Java™ 2 Platform, Enterprise Edition (J2EE™) application, you separate core business model functionality from the presentation and control logic that uses this functionality. Such separation allows multiple views to share the same enterprise data model, which makes supporting multiple clients easier to implement, test, and maintain.“ [Sun06]



[Figure 02, MVC]

The blueprint for the Model-View-Controller in Figure 02 is specific to Java Platforms. In other applications it could differ if it comes to the dependencies between the different layers. In general the Model View Controller is used to separate the three layers from each other to create code that is easy to maintain and also more difficult to break.

2 Web Template Engines

2.1 Motivation

The primary goal of Template Engines is the separation of application code from the presentation. The application code is created by programmers, who focus on the business logic. On the other hand designers work on the presentation to give the application a good look and feel. As designers are not familiar with the application code, separating the application code from the presentation prevents the breaking of the application by some unexperienced designers.

Furthermore designers don't want to mess around with coding the application. When using a Template, they can focus on the presentation of the application. In many Web Template Engines, designers just have to set some markers where the Template Engine inserts the desired data. In more sophisticated ones the designers can use loops and different data types to manipulate the output.

Every Template Engine supports one or more programming language, which it is able to interpret. Therefore the selection of a Template Engine depends on which programming language is used. In the next chapter Template Engines for the most popular web programming languages are introduced, except Velocity which will be described in more detail in Chapter 3.

2.2 Smarty – PHP

Smarty is a template and presentation framework written in Hypertext Preprocessor (PHP). *"Its focus is on quick and painless development and deployment of your application, while maintaining high-performance, scalability, security and future growth." [Smat06]* It lets the template and application designer share information of interest over an object. The Smarty framework is released under the LGPL[1], which means that users are allowed to receive the source code, dis-

[1] GNU - Lesser General Public License

tribute copies and change pieces of Smarty. *[SGNU06]*

2.2.1 Features

- Smarty provides a caching feature, which allows the programmer to choose if a section of a web page should be cached or not.
- It is possible to assign variables pulled from a configuration file. These variables are not accessible from the application code and therefore designers can store variables over several templates without the intervention from the programmer.
- While executing the template, designers have the opportunity to modify the variables with predefined functions, such as display all variables in upper-case, formatting dates or add spaces between characters. Furthermore there are predefined HTML generation functions, such as generating dropdowns, tables or lists. Smarty can also handle arrays in template files as it is possible to loop over them.
- Smarty has an easy to use plugin function, which allows programmers to create their own functions and use them either in the template or in the application code.
- A so called ‘debugging console’ is provided by Smarty which allows template designers to see all of the assigned variables and the programmer is able to investigate the rendering speed of the template.

[Smat06]

2.2.2 Sample

To use Smarty in an PHP application, the Smarty class file has to be included which should be downloaded from the official Smarty web page[2]. The next step is to create an instance of this class to get a Smarty Object. This Object can be filled with data, to access it from the Template. The Code in Listing 1 shows a simple implementation of the Smarty Object. In Line 7 the method assign of the Smarty object is used, to make the string hello `world` available to the Template. This String can be queried by using the identifier `hello` which is the first argument of this method. The next step is to tell the Smarty framework which Template file to use. In this case the file `index.tpl` is used, which, in our case, is in the same directory as the application code.

```
1 include('Smarty.class.php');
2
3 // create object
4 $smarty = new Smarty;
5
6 // assign some content
7 $smarty->assign('hello', 'Hello World');
8
9 // display it
10 $smarty->display('index.tpl');
```

[Listing 1]

The appropriate Template file is shown in Listing 2. The placeholder for variables stored in the Smarty objects must be written between angle brackets followed by a dollar sign. In Line 4 we query the variable `hello`, which we assigned to the Smarty object beforehand.

[2] http://smarty.php.net

```
1 <html>
2  <body>
3   <p>
4    {$hello}!
5   </p>
6  </body>
7 </html>
```

[Listing 2]

The output of this application is shown in Listing 3.

```
1 <html>
2  <body>
3   <p>
4    Hello World!
5   </p>
6  </body>
7 </html>
```

[Listing 3]

2.2.3 Summary

The big advantage of Smarty is the simple implementation, as you just need the Smarty classes in your PHP application. Furthermore the Smarty framework can be customized easily because of the powerful plugins. The template language is not very intuitive, but can be learned easily as it does not provide many options.

2.3 Contemplate – ASP, PHP, Perl

Contemplate is a Web Templating System which is able to run on server platforms which either are able to execute Active Server Pages (ASP), Hypertext Preprocessor (PHP) or Perl. *“Contemplate assembles text content, page layouts, and server-side or client-side scripts into dynamically generated or precompiled web pages based on arguments you pass into each page.“ [Cont06]*

Contemplate is distributed under the Apache Software License, which provides free distribution and use for any purpose. Users do not have direct access to

the development source code, but are allowed to modify the source for their own purpose.

2.3.1 Features

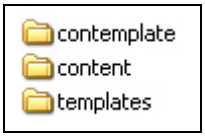
- Contemplate adds the idea of a person which is responsible for just creating the content. This person is called a content editor and is able to fill an extern content file with his contents.
- Furthermore these content pieces can be used in many templates, as the content file is accessible for all of them.
- Contemplate is compatible with many programming languages as listed beforehand and there is no need for an external database.
- Always a page is requested not the Template or the application code is invoked, but the so called assembler script of Contemplate is passed the arguments to create the requested page.
- It is possible to use server-side scripts such as PHP in Contemplate template files. This gives the designer the opportunity to use scripts to empower his design.

[Cont06]

2.3.2 Sample

To run Contemplate the first step is to download the software from the official contemplate web page[3]. Contemplate needs a specific folder structure in order to find the desired content and templates. The folder `contemplate`, shown in Figure 03, you find the source code of the Contemplate Templating System, the folder content contains HTML content files and the folder templates is filled with the HTML Templates.

[3] http://www.typea.net/software/contemplate/assembled/home.html



[Figure 03, Contemplate directory structure]

A content editor now creates a HTML file where he stores the text, which should later be displayed in our output file. In this case the content file is called `quotes.html`. Therefore a table has to be created with two cells. The first one holds the identifier (Line 5) and the second one the text (Line 8), like shown in Listing 4. This is one out of several ways a content editor can create his content files.

```
1 <html>
2  <body>
3   <table>
4    <tr><td>
5      ##hello
6    </tr></td>
7    <tr><td>
8      Hello World!
9    </tr></td>
10  </table>
11 </body>
12 </html>
```

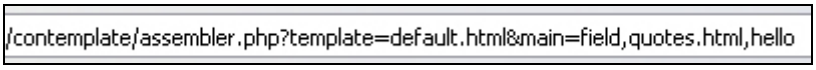
[Listing 4]

The designer has to create his HTML Template files in the templates directory. Here he can use a so called “embed tag” to access the data from the content files. These tags have a specific notation, which is “`<!--#embed identifier -->`“. According to Listing 5, the template designer created a blank HTML file with an identifier called `main` and some static text which comes directly from the Template.

```
1 <html>
2  <body>
3   <p><b>This is a static text</b><p>
4
5   <p>This not: <!--#embed main --><p>
6  </body>
7 </html>
```

[Listing 5]

What is really specific to Contemplate is the fact, that the reference to a template file is not “hard-coded” in the application code, but it is passed to the page through a url[4] parameter. Every page is created by requesting the assembler, in our case the assembler.php. The first parameter, called template, specifies the url to the HTML file where the Template is stored. This file is called `default.htm` and is stored in the templates directory. The next parameter tells the assembler, that the identifier `main` is a `field` which should be filled with some text out of the `quotes.html`, and which belongs to the identifier `hello`.



[Figure 04, Contemplate assembler call]

The assembler now creates the main variable which we specified in Listing 5. The output of this specific page is shown in Listing 6.

```
1 <html>
2  <body>
3   <p><b>This is a static text</b><p>
4
5   <p>This not: Hello World!<p>
6  </body>
7 </html>
```

[Listing 6]

2.3.3 Summary

A big advantage of Contemplate is the fact, that it runs on platforms which either support PHP, ASP or Perl. Contemplate is a Web Template System which does not fully separate the application code from the designing part. It stores the application logic as well as the visual parts in the template files, which is counterproductive, as Template Engines should separate these. But Contemplate uses the idea of the Content Editor to provide a very useful feature for dynamically creating content files which can be accessed from various Templates. This speeds up the performance of this Web Template Engine.

[4] Uniform Resource Locator

2.4 Cheetah – Python

Cheetah[5] is a Python powered Template Engine that can generate any text-based format. *"It can be used stand alone or combined with other tools and frameworks. Web development is its principle use, but Cheetah is very flexible and is also being used to generate C++ game code, Java, sql, form emails and even Python code." [Chee06]* Cheetah runs under an open source license provided by the Open Source Initiative[6]. *[Open06]*

2.4.1 Features

- Cheetah is supported by every web framework which uses Python.
- It enriches the Python language with a simple to understand template language.
- *"Cheetah gives template authors full access to any Python data structure, module, function, object, or method in their templates." [Chee06]* But coders can restrict Template authors to access certain Python features.
- To make code reuse easy, Cheetah provides an object-oriented interface to templates that is accessible from the controller code or other Templates.
- Templates can consist of other Templates or parts of them.
- Cheetah supports caching mechanisms out of the box.
- Cheetah does not prevent architectures from using a Model View Controller pattern.

[Chee06]

[5] http://www.cheetahtemplate.org/
[6] http://www.opensource.org/

2.4.2 Sample

To use Cheetah Templates a minimum amount of code is needed. As shown in Listing 7 the controller code first creates a list of items in line 1. In this list we assign the string `Hello World` to the identifier `hello`. Next the necessary classes are imported and a template is instantiated. This happens in line 4 where the method `Template` is called which needs the template file and the object which should be shared. This object is the list which was created in line 1. Cheetah treats Templates as objects which can be queried for attributes, methods and so on.

```
1 DataModel = {'hello': 'Hello World! '}
2
3 from Cheetah.Template import Template
4 t = Template(file="HelloWorld.tmpl", searchList=[DataModel])
```

[Listing 7]

To make this application as simple as possible the template file (Listing 8) consists of one line, which is the placeholder for the identifier `hello`.

```
1 $hello
```

[Listing 8]

The output of this little application is shown in Listing 9.

```
1 Hello World!
```

[Listing 9]

[ONLa06]

2.4.3 Summary

As Cheetah treats Templates as objects there are many ways of working with them. This is definitely something which silhouettes Cheetah against other

Template Engines. On the other hand, Template designer are able to access all Python features which does not support the Model View Controller pattern. This can lead to problems for both, the Template designer and the controller programmer.

2.5 Embperl – Perl

Embperl[7] is a framework for building web sites with Perl, which is a programming language close to C and shell scripting. *"It delivers several features that ease the task of creating a websites, including dynamic tables, formfield-processing, escaping/unescaping, session handling, caching and more." [EmbP06]*

Embperl is directly integrated with Apache and mod_perl and so can achieve the best performance. It is optimized for delivering dynamic content online and therefore scales and performs very well for high end solutions. It can be used under the terms of either the GNU General Public License or the Artistic License[8]. *[EmbP06]*

2.5.1 Features

- Embperl lets you embed Perl code into HTML, XML or other text documents. The Perl code is evaluated at the server side by using for example Apaches mod_perl. The result is a fully rendered HTML file, which is displayable for all common browsers.

- It is possible to build reusable components, also in an object-oriented way, which can use components themselves.

- Embperl supports building Model-View-Controller applications as the application logic can be moved into an application object. The View is represented by the Templates.

[7] http://perl.apache.org/embperl/
[8] http://www.opensource.org/licenses/artistic-license.php

- Furthermore Embperl is able to interact with several Apache applications like PHP, JSP or CGI scripts.

- The output generation process is divided in steps which are handled by so called “pluggable providers”. Every provider can be configured to behave in a specific way

[EmbP06]

2.5.2 Sample

It is possible to use Embperl in many different ways, like including a requested HTML file in your script or create modular files. In this example we are using a way which is called “global variables”. If we want to assign a global variable we have to use the Embperl Object which can be filled with variables. Every time a request to a document is made a so called Request Object is generated, which will be destroyed after the request is done.

First we have to create a file containing the main template and the assignment of the global variable. This file must be named `base.epl` as the Embperl engine is looking for a so called file every time a request is made. Now we can use the Request Object to store variables, which we want to share between the documents on our web site. The `base.epl` is shown in Listing 10.

```
1 <HTML>
2  [-
3    $req = shift;
4    $req->{hello} = 'World!'
5  -]
6   <BODY>
7    [- Execute ('*') -]
8   </BODY>
9 </HTML>
```

[Listing 10]

This file represents the basic structure of the output file. In Line 3 the Request Object is stored in the variable `$req`. It is retrieved by using the shift statement, which gets it from the stack. In Line 4 the String `World!` is stored in the Request object.

Every time a request to a random HTML file is made, Embperl looks for a file called `base.epl`. Therefore the Statement in Line 7 tells Embperl to execute the HTML file which was originally requested by the Browser.

The corresponding HTML file must be located in the same directory as the `base.epl` script. Once again the Request Object has to be retrieved in the HTML file, which is shown in Listing 11. In Line 2 the variable `hello` is queried from the mentioned Object.

```
1 [- $req = shift; -]
2 Hello [+ $req ->{hello} +]!
```

[Listing 11]

The output of this simple application is shown in Listing 12.

```
1 <HTML>
6   <BODY>
7    Hello World!
8   </BODY>
9 </HTML>
```

[Listing 12]

[Perl06]

2.5.3 Summary

Embperl is a very powerful tool as it is directly integrated into the Apache web server, when using mod_perl. It gives you the opportunity of working in an “object-oriented” way when creating your own components. Once more there is the possibility to embed existing perl code to your code to enhance your web application. As you have the full perl power in your application, the “TIMTOWTDI”[9] motto of perl may confuse new programmers at the beginning.

[9] there is more than one way to do it

2.6 FreeMarker – Java

"FreeMarker is a 'template engine'; a generic tool to generate text output (anything from HTML to auto generated source code) based on templates. It's a Java package, a class library for Java programmers." [Free06]

FreeMarker[10] follows the Model View Controller approach, which strictly distinguishes between the application logic code and the presentation of the application. The tool is totally written in Java and uses Java for programming the application logic. To use FreeMarker in web applications, a servlet based Framework is needed to display FreeMarker created files (e.g. Struts[11]). FreeMarker is distributed under a BSD-style license. It is OSI Certified Open Source Software, which is a certification mark of the Open Source Initiative. *[Free06]*

2.6.1 Features

- With the use of FreeMarker any kind of text files can be created, like HTML, XML or Java source code.
- FreeMarker uses its own template language which provides usual directives (if/else/elseif), name-spaces and operations for specific data types like strings (uppercase, escaping …) or looping through arrays.
- FreeMarker can be used in servlets to replace JSP sites, but it also supports the JSP taglibs.
- It is possible to parse XML files within a template, using the XML processing capabilities. This feature lets FreeMarker be an alternative to XSLT transformations.
- The objects generated in the Controller code are available to the Template as variables through pluggable object wrappers. This helps the

[10] http://freemarker.sourceforge.net/
[11] http://struts.apache.org/

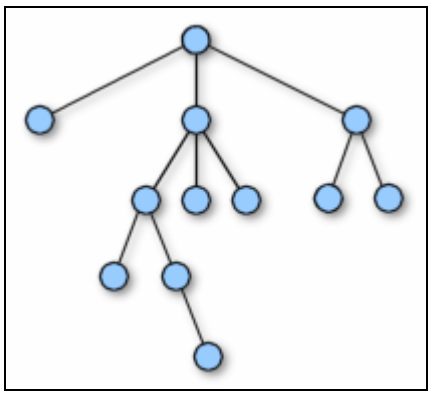
template designer to easily use the objects without bothering him with technical details.

[Free06]

2.6.2 Sample

Before making use of FreeMarker, you have to be sure that you have at least the J2SE 1.2 or higher.

First we can create our data model which is represented as a tree. This tree can either be filled with variables which represent a value (scalars) or variables which hold other subvariables (hashes). The topmost object is called the `root` object, where you can start to access the data model and go down along the edges to access other objects.



[Figure 05, FreeMarker data model tree]

According to Figure 05 we want to create a data model which looks like Listing 13, where we have a hash `helloWorld`. This hash holds two scalars, one scalar called `hello` with the value `Hello` and another scalar called `world` with the value `World!`.

```
1 (root)
2   |
3   +- helloWorld
4       |
5       + - hello = "Hello"
6       |
7       + - world = "World!"
```

[Listing 13]

The controller code has to be written in Java and is shown in Listing 14. First we have to import the freemarker engine (Line 1) and other utilities in order to create our data model and print out the result (Line 2 & 3). To make the Java code executable a class is needed. In our case we create the class `HelloWorld`. Therefore the filename of this Java application has to be `HelloWorld.java`. Line 7 tells the Java interpreter that this method should be executed whenever this application is invoked.

First of all a `configuration` instance has to be retrieved from FreeMarker. *"The `configuration` instance is a central place to store the application level settings of FreeMarker." [Free06]* We have to tell this instance where we store our Template files. In this case Line 11 tells the `configuration` instance that we store our Templates in a directory called `templates`. Furthermore an `ObjectWrapper` has to be passed to the `configuration` instance. This specifies how Templates will see the data model. This process is normally done only once in the whole application life cycle.

The next step is to get a Template instance from the just created `configuration` instance. This happens in Line 15 where we specify `helloWorld.txt` as our Template file.

The creation of the data model happens in Line 18 to 24 of Listing 14, where we create a Hashmap `root` which holds our data model. This Hashmap holds another Hashmap called `helloWorld` where the scalars are stored. This way it is possible to create a tree which looks like Figure 05.

As we have our data model and an instance of our Template, the next step is to merge these two together, to create the output. This happens in Line 23 where the function `process` of the template instance is called. It needs two parameters, the data model and the Writer, which tells the template how to show the result. In this case we will display the output in the console.

```
1 import freemarker.template.*;
2 import java.util.*;
3 import java.io.*;
4
5 public class HelloWorld {
6
7     public static void main(String[] args) {
8
9         Configuration cfg = new Configuration();
10         cfg.setDirectoryForTemplateLoading(
11                 new File("/templates"));
12         cfg.setObjectWrapper(new DefaultObjectWrapper());
13
14         /* Get a template from the configuration instance */
15         Template temp = cfg.getTemplate("helloWorld.txt");
16
17         /* Create the data model */
18         Map root = new HashMap();
19
20         Map helloWorld = new HashMap();
21         helloWorld.put("hello", "Hello");
22         helloWorld.put("world", "World!");
23
24         root.put("helloWorld", helloWorld);
25
26
27         /* Merge it together */
28         Writer out = new OutputStreamWriter(System.out);
29         temp.process(root, out);
30         out.flush();
31     }
32 }
```

[Listing 14]

The controller code is ready to process our template, which is shown in Listing 15. The data model can be accessed by using a dollar sign and two angle brackets.

```
1 ${helloWorld.hello} ${helloWorld.world}
```

[Listing 15]

The output of this sample is shown in Listing 16.

```
1 Hello World!
```

[Listing 16]

[JFre06]

2.6.3 Summary

FreeMarker is indeed the strongest rival of Velocity as both products use Java as controller code. What could be pointed out from Freemarker is the fact, that it is possible to use the JSP library in FreeMarker Templates. To find out the differences between Velocity and Freemarker, feel free to read on!

3 Apache Velocity

3.1 History

The first beta Version of Jakarta Velocity was introduced in March 2001 under the umbrella of Jakarta projects. Since then the Velocity team has made many releases. The current stable version is Velocity 1.4, which is used in this thesis. Since October 2006 Velocity has been promoted by the Apache Software Foundation into an Apache Top Level Project (TLP). So Jakarta Velocity has now become Apache Velocity.

3.2 Introduction

"Velocity is a Java-based template engine. It permits anyone to use a simple yet powerful template language to reference objects defined in Java code." As Velocity provides a simple template language for designers, Velocity enforces a clear separation between the view, the model and the controller. Velocity acts as a bridge between these 3 entities and therefore provides a collection of Java-based classes. In web applications Velocity separates Java code from the web pages and provides an alternative to Java Server Pages or PHP.

[Velo06]

Velocity is distributed under the Apache License Version 2.0[12].

3.3 Features

- Using the VelocityViewServlet, creating web applications becomes quite easy. On the other hand Velocity is often used as controller code in web

[12] http://jakarta.apache.org/velocity/docs/license.html

frameworks as you can see in the impressive list of projects which use Velocity[13].

- Velocity can be used to generate output files in any format. This let Velocity act as a source code generator for Java, SQL or PostScript code.
- Velocity provides a Template language which is easy to use but nevertheless powerful.
- Velocity enables you to directly access Java objects of the controller code.

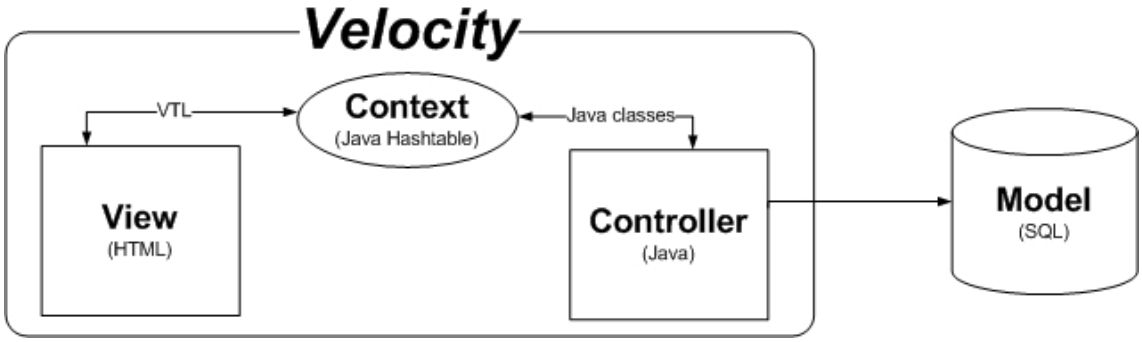
[Velo06]

3.4 How it works – Hello World!

3.4.1 Fundamentals

Velocity uses the Model View Controller concept to separate the View from the other components. In order to let the Controller share information with the View, an object called “context” exists, which acts like a broker. According to Figure 06 the View can access the Context using the Velocity Template Language. The controller is able to set or retrieve values from the Context using the provided Java classes. As the controller is written in Java code, he is able to access the model, which could be a SQL database. Note that the Model is not a part of Velocity itself, but is intended in the Model View Controller concept.

[13] http://wiki.apache.org/jakarta-velocity/PoweredByVelocity



[Figure 06, Velocity MVC]

In the following a Hello World applications is created, while explaining this concepts in detail. Let's consider that in our team there are working 2 people, the designer, which is responsible for the view component, and the programmer, which will take care of programming the controller. These 2 people will create our Hello World example.

3.4.2 View

First the designer begins to create the look and feel of the page. As we just want to create a quite simple application, he creates a HTML file. This file is the view or Template for our application, which is shown in Listing 17. The view can be any file format which is required, for example they could agree on creating a text file as view. There are no special requirements to Templates for Velocity.

```
1 <HTML>
2  <BODY>
3   $hello
4  </BODY>
5 </HTML>
```

[Listing 17]

The designer decides that the `$hello` should display the text which comes from the Context. The designer is now done with his work and therefore meets the guy, who creates the Controller code.

3.4.3 Context

The meeting won't last long as the designer tells the programmer that the Context should hold a variable `hello` with the text for our application.

„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 accessable via template elements called references.“ [VDev06]

The Context object, which is a hashtable that provides get and set methods for retrieving and setting objects, gives the designer the possibility to retrieve a object which is inserted by the programmer. But the designer is also able to insert objects into the Context or modify the Context. The context stores objects of type `java.lang.Object` keyed by objects of type `java.lang.string`. *[Velo06]*

Therefore the Context, which is defined as an interface in `org.apache.velocity.context`, provides five methods which are briefly described below:

- `boolean containsKey(java.lang.Object key)` returns true or false if a key exists in the Context or not.
- `java.lang.Object get(java.lang.String key)` returns the Object to the corresponding key.
- `java.lang.Object[] getKeys()` returns a list of all keys which are currently stored in the Context.
- `java.lang.Object put(java.lang.String key, java.lang.Object value)` lets the programmer insert a key, which is a string, and a corresponding Object.

- `java.lang.Object remove(java.lang.Object key)` removes a object stored in the Context by the corresponding key.

[VApi06]

3.4.4 Controller

As the programmer now has his task, he can start to create the Controller code. As our programmer is very experienced, he will easily be able to create a String with the value `Hello World!` as the designer wished for. This code is shown in Listing 18.

```
1 String hello = "Hello World!";
```

[Listing 18]

3.4.5 Initialize Velocity and merge

As the designer and the programmer have now done their job, Velocity comes into play. The programmer has to initialize Velocity and create the Context to merge the pieces they have created. The full code for this application is shown in Listing 19 and the major parts are described below.

```
1 import org.apache.velocity.app.Velocity;
2 import org.apache.velocity.VelocityContext;
3 import org.apache.velocity.Template;
4 import org.apache.velocity.exception.*;
5
6 import java.io.*;
7 import java.util.*;
8
9 public class Example {
10   public static void main (String[] args) {
11     try {
12
13       //init Velocity
14       Velocity.init();
15     } catch( Exception x) {
16       System.err.println( "init failed: " + x);
17       System.exit (1);
18       }
19
20     //obtain a template
21     Template template = null;
22       try {
23         template = Velocity.getTemplate("helloworld.vm");
24       } catch (ResourceNotFoundException e2) {
25         System.out.println ("no template found");
26       } catch (ParseErrorException e) {
27         System.out.println("Parse Error: " + e);
28       }  catch (Exception ee) {
29         System.out.println("Exception: " + ee);
30       }
31
32     //create the context
33     VelocityContext context = new VelocityContext();
34
35     //put something into the context
36     context.put("hello", "Hello World!");
37
38     //create a string writer to merge template and context
39     StringWriter writer = new StringWriter();
40
41       try {
42        template.merge(context, writer);
43       } catch (Exception x) {
44        System.err.println("Failed to merge:" + x);
45        System.exit(1);
46       }
47
48     System.out.println (writer.toString());
49   }
50 }
```

[Listing 19]

First of all the programmer has to import the necessary classes for Velocity and his application. Afterwards he creates a class including a `main` method, which will be called when the application starts. Note that the general sequence of Velocity applications consists out of six steps.

```
14        Velocity.init();
```

[Listing 20]

The first step is to initialize Velocity like we did in Listing 20. This initialization represents the Singleton model, which means that there is only one instance of the Velocity engine in the JVM that is shared by all parts of our application. The other possibility would be to create several instances of the Velocity engine.

```
21      Template template = null;
22        try {
23          template = Velocity.getTemplate("helloworld.vm");
24        }
```

[Listing 21]

The second step is to create a Template instance. This happens by invoking the method `getTemplate` of the Velocity class, which returns an instance of our Template. In this case we already tell Velocity where to find our Template file which is called `helloworld.vm`. The name of the file does not have to be `.vm`, it could be any file extension you would like to have.

```
33      VelocityContext context = new VelocityContext();
```

[Listing 22]

According to Listing 22, the next step is to create the Context. This happens by retrieving an instance of the Context from the Context class.

```
36      context.put("hello", "Hello World!");
```

[Listing 23]

Step number 4 is to fill the Context with the objects of desire. This is possible by calling the method `put`, which stores our `Hello World` string keyed by a String named `hello`.

```
39      StringWriter writer = new StringWriter();
40
41        try {
42         template.merge(context, writer);
43        }
```

[Listing 24]

Step number 5 needs two things to do. First we have to create an instance of a `writer` which enables us to store the output of the merge process. The next step is to merge the template with the Context. Therefore the method `merge` of our template instance has to be called, which needs two arguments. The first argument is the context object of our application, represented by the variable `context`. The second one is an Object which tells Velocity where the Template should be outputted. In this case we decided to output the result to the `StringWriter`.

The merge process now processes the template. This means *"if Velocity encounters a reference in your VTL template to `$hello`, it will search the Context for a corresponding value. "[VUse06]* In this case `$hello` will be replaced by our corresponding String `Hello World!`, if there is no entry in the Context the output would be `$hello`.

```
48      System.out.println (writer.toString());
```

[Listing 25]

Now as the merge is done by Velocity, the last step is to show the output to the console. This is done in Listing 25.

Keep this pattern in mind, as we will use it for the most of our examples.

3.4.6 Output

Now as the application is ready there are just 2 steps missing to our first up and running example. All we have to do now is to compile our application using the Java compiler. Then we have to invoke the application, and the output tells us

that everything worked out fine. The placeholder `$hello` was replaced by the String `Hello World!`. (Figure 07)

[Figure 07, HelloWorld example output]

4 Velocity Template Language

"The Velocity Template Language (VTL) is meant to provide the easiest, simplest, and cleanest way to incorporate dynamic content in a web page. Even a web page developer with little or no programming experience should soon be capable of using VTL to incorporate dynamic content in a web site." [VUse06]

The Velocity Template Language is grouped in three types of notations, called references, directives and macros.

Note that for all examples the code of Listing 19 is used to initialize Velocity and merge the Template with the Context. The only thing which differs is the Template itself and the objects stored in the Context.

4.1 References

"There are three types of references in the VTL: variables, properties and methods. As a designer using the VTL, you and your engineers must come to an agreement on the specific names of references so you can use them correctly in your templates." [VUse06]

References have a quite easy syntax, which is shown in Listing 26. The syntax is a dollar sign followed by the identifier, in this case a variable called `referenceName`.

```
1 $referenceName
```

[Listing 26]

4.1.1 Variables

Variables represent objects which are stored in the Context. *"Everything coming to and from a reference is treated as a String object. If there is an object that represents $foo (such as an Integer object), then Velocity will call its .toString() method to resolve the object into a String." [VUse06]* According to this let's create a Context which stores different Object types. In Listing 27 three Object

types are stored in the Context, a Java Object, a Integer with the value 99 and a String with the value String.

```
1 context.put("object", new Object());
2 context.put("integer", new Integer(99));
3 context.put("string", new String("String"));
```

[Listing 27]

To retrieve these values from the Context, the VTL provides the syntax shown in Listing 28, which is a dollar sign followed by the key in the Context.

```
1 Object:
2 $object
3
4 Integer:
5 $integer
6
7 String:
8 $string
```

[Listing 28]

As Velocity calls the `toString()` method of each object the output of the Java Object is his name. The values of the Integer and the String are displayed correctly, as both provide a `toString()` method which prints their values.

[Figure 08, VTL variables output]

4.1.2 Methods

"A method is defined in the Java code and is capable of doing something useful, like running a calculation or arriving at a decision. " [VUse06] To call a method of a class, an instance of the class has to be added to the context. This happens in Listing 29 where the Class `Friend` is created. This class has 3 meth-

ods which are called `setName`, `setAge` and `getFriend`. To access the class `Friend` the programmer decided to use the identifier `friend` in the Context.

```
1 public static class Friend {
2
3   private String name;
4   private Integer age;
5
6   public void setName (String name ) {
7      this.name = name;
8   }
9
10   public void setAge (Integer age) {
11      this.age = age;
12   }
13
14   public String getFriend() {
15      return ("The name of my friend is "+name+" and he is
16      "+age+" years old!");
17   }
18 }
19
20 [...]
21
22 context.put("friend", new Friend());
```

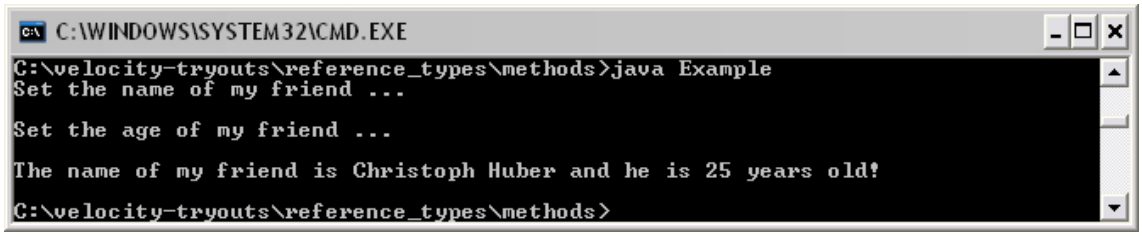
[Listing 29]

As the class is now exposed to the Template, we can simply call the methods like we do in Java. First the identifier has to be called as usual and the method is simply added after a dot. To send an argument to the method, the argument has to be between two brackets. In Listing 30 we call the method `setName` and `setAge` and pass arguments to them. The method in line 5 should return a String with the details we passed to the class before. When calling a method in VTL nothing is displayed where the method was placed.

```
1 Set the name of my friend ... $friend.setName("Christoph Huber")
2
3 Set the age of my friend ... $friend.setAge(25)
4
5 $friend.getFriend()
```

[Listing 30]

The output of this example is shown in Figure 09 where the correct String is returned from the class and displayed at the last line.



[Figure 09, VTL methods output]

4.1.3 Properties

Another possibility to access methods is the use of properties.

```
1 public static class Friend {
2
3   private String name;
4   private Integer age;
5
6   public void setName (String name ) {
7      this.name = name;
8   }
9
10   public void setAge (Integer age) {
11      this.age = age;
12   }
13
14   public String getName() {
15      return name;
16   }
17
18   public Integer getAge() {
19      return age;
20   }
21 }
22
23 [...]
24
25 context.put("friend", new Friend());
```

[Listing 31]

To make use of properties we created two more methods in Listing 31 where we just return the values of `age` and `name`.

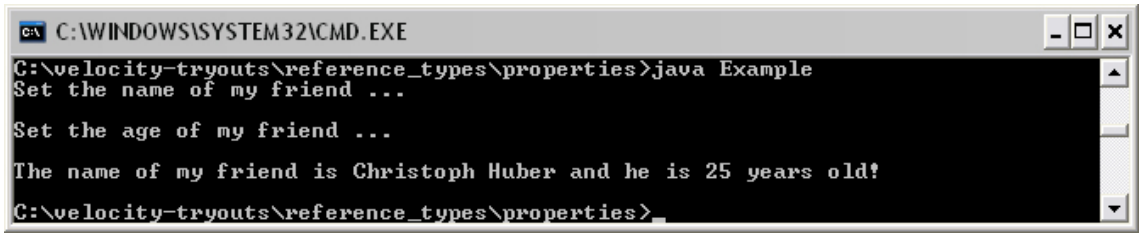
When using Properties the notation is once again a dollar sign, the identifier, a dot and then the method you want to call. Note that when using properties the name of the method in Java has to begin with `get` or `set` followed by the identifier. For example when calling `$friend.name` in Line 1 Velocity will look if there is a method called `getName()` or a map called `friend` with a key called

name. Note that Velocity is not case sensitive in this case `$friend.Age` also refers to the method `getAge()`.

```
1 Set the name of my friend ... $friend.setName("Christoph Huber")
2
3 Set the age of my friend ... $friend.setAge(25)
4
5 The name of my friend is $friend.name and he is $friend.Age
6 years old!
```

[Listing 32]

The output, after merging the Template in Listing 32 and the Controller code in Listing 31, is shown in Figure10.



[Figure 10, VTL properties output]

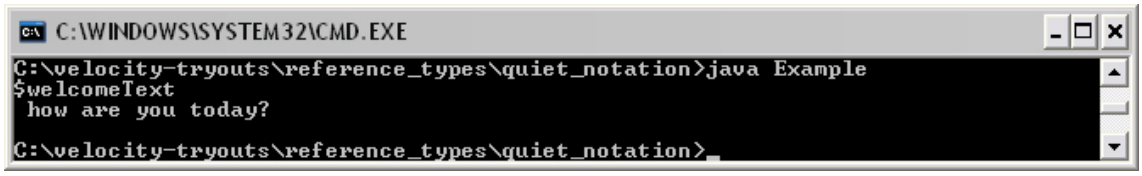
4.1.4 Quiet notations

If we process the template in Listing 33 and we do not put anything into the Context, Velocity can't resolve the variables. To avoid that these variables are shown in the output we can use quiet notations. Quiet notations do not show up if there is no corresponding entry in the Context. Quiet notations must have an exclamation point between the dollar sign and the identifier.

```
1 $welcomeText
2 $!hello how are you today?
```

[Listing 33]

So the output when processing this template is shown in Figure 11.



[Figure 11, VTL quiet notations]

4.2 Directives

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

[VUse06]

4.2.1 #set

The set directive let the designer put objects into the Context. Therefore we add a value to the context in the controller code.

```
1 context.put("value", "a value added by java");
```

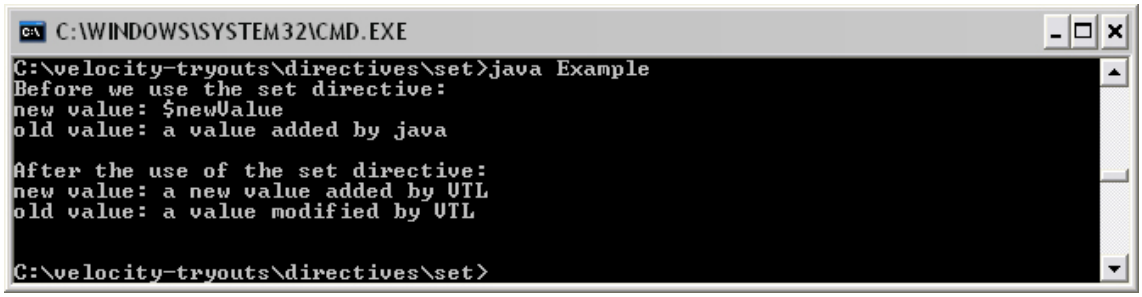
[Listing 34]

When a value is added using the set directive the Context is updated immediately. To show this, Listing 35 shows the values of the variable `$newValue` and `$value` before using the set directives. The first variable does not exist at this moment and the other one was added to the Context in Listing 34. In Line 5 the variable `$newValue` is added to the Context. Note that there is no need for initializing a variable. In Line 6 the value of the variable `$value` is modified.

```
1 Before we use the set directive:
2 new value: $newValue
3 old value: $value
4
5 #set($newValue = "a new value added by VTL")
6 #set($value = "a value modified by VTL")
7 After the use of the set directive:
8 new value: $newValue
9 old value: $value
```

[Listing 35]

The output of this example is shown in Figure 12.



[Figure 12, VTL set output]

Strings are not the only objects which can be added with the set directive. It is possible to set the following objects:

- Variable reference
- String literal
- Property reference
- Method reference
- Number literal
- Array List

As variable (at the left hand side of the set directive) a variable reference or a property reference can be used.

4.2.2 #if

„The #if *directive in Velocity allows for text to be included when the web page is generated, on the conditional that the if statement is true.“ [VUse06]*

A condition becomes true if it either is a Boolean data type which is set to true, or a reference that corresponds to a non null value. In these examples there is no need to put something to the context using the controller code, as we will add the values to the context using the set directive.

The use of the if directive is quite easy, as it begins with the `#if` statement and ends with an `#end` statement. The code between these two statements is being processed, if the condition turns out to be true. In Listing 36 we first create a variable called `condition`, which once is true and once not.

```
1 #set($condition = true)
2 #if($condition)
3 I am true! :)
4 #end
5
6 #set($condition = false)
7 #if($condition)
8 I am not true! :(
9 #end
```

[Listing 36]

The output of this example is shown in Figure 12, where the first block is processed (as the condition is true) and the second not (as the condition is false).

[Figure 13, VTL if output]

But the VTL provides much more powerful operators which are quite similar to Java. Listing 37 shows a set of various relational and logical operators. According to the output, shown in Figure 13, Line 5 shows that the block is processed if a String variable is set compared to Line 9 where a non existing variable turns

out false. The lines 13 and 17 show the use of the logical operators OR, AND and NOT.

Furthermore at Line 22 some more operators are introduced, which are also similar to Java. Line 22 shows the lesser than, line 24 the greater than, line 26 the lesser or equal than, line 28 the greater or equal than, line 30 the equal and line 32 once again the lesser than, which is the only one that is false.

```
1 #set($aString = "some text")
2 #set($trueCondition = true)
3 #set($falseCondition = false)
4
5 #if($aString)
6 1 - $aString
7 #end
8
9 #if($nonExistingVariable)
10 2 - hello?
11 #end
12
13 #if($trueCondition || $falseCondition)
14 3 - true
15 #end
16
17 #if(($trueCondition && !$trueCondition) || $falseCondition)
18 4 - false
19 #end
20
22 #if (0 < 1) 0 lesser than 1 #end
23
24 #if (1 > 0) 1 greater than 0 #end
25
26 #if (1 <= 1) 1 lesser or equal than 1 #end
27
28 #if (1 >= 1) 1 greater or equal than 1 #end
29
30 #if (1 == 1) 1 equals 1 #end
31
32 #if (1 < 1) 1 lesser than 1 #end
```

[Listing 37]

[Figure 14, VTL powerful if output]

4.2.3 #else

The else directive provides the possibility to jump into a block which is processed when the condition turns out to be false. Listing 38 shows that the `#else` statement has to be between the `#if` and the `#end` statement.

```
1 #set($condition = true)
2 #if($condition)
3 1 - I am true! :)
4 #else
5 2 - I am false! :(
6 #end
7
8 #set($condition = false)
9 #if($condition)
10 3 - I am true! :)
11 #else
12 4 - I am false! :(
13 #end
```

[Listing 38]

According to the output shown in Figure 14 the first directives jumps into the first block, as it is true, and the second into the else block, as it is false.

[Figure 15, VTL else output]

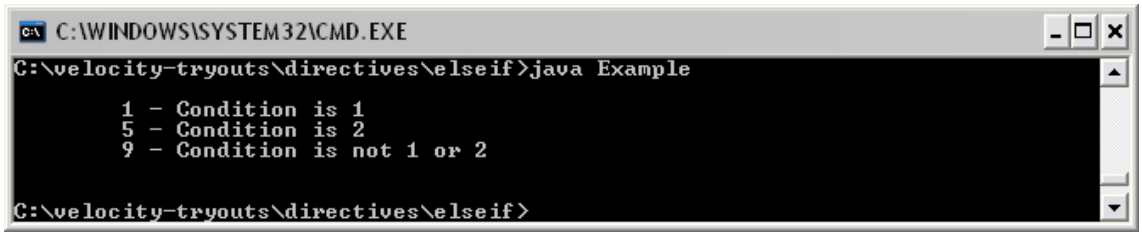
4.2.4 #elseif

The elseif directive adds the possibility of letting another condition to be checked, before jumping into the else block. This happens by adding an `#elseif` statement between the `#if` and the `#else` statement. Listing 39 shows a sample application for this. In line 1 a variable is set to 1. In the following if block it is checked whether the variable is 1, 2 or not. Depending on the value, the block is processed or not. After a block is processed i.e. a condition is true, Velocity stops processing the other statements. In this example every time a condition becomes true, the value of the variable is changed.

```
1 #set($condition = 1)
2
3 #if($condition == 1)
4       1 - Condition is 1 #set($condition = 2)
5 #elseif($condition == 2)
6       2 - Condition is 2
7 #else
8       3 - Condition is not 1 or 2
9 #end
10
11 #if($condition == 1)
12      4 - Condition is 1
13 #elseif($condition == 2)
14      5 - Condition is 2 #set($condition = 3)
15 #else
16      6 - Condition is not 1 or 2
17 #end
18
19 #if($condition == 1)
20      7 - Condition is 1
21 #elseif($condition == 2)
22      8 - Condition is 2
23 #else
24      9 - Condition is not 1 or 2
25 #end
```

[Listing 39]

Figure 15 shows the result of this application.



[Figure 16, VTL elseif output]

4.2.5 #foreach

The foreach directive allows looping over elements which are arrays, hashtables, vectors or objects that implement a collection, enumeration, iterator or map interface. In this example we create a Vector in our Java code which we fill with some names. We put this Vector to our Context with the identifier `vector`, shown in Listing 40.

```
1 Vector vec = new Vector();
2 vec.add("Christoph Huber");
3 vec.add("Niki Lauda");
4 vec.add("Heidi Klum");
5
6 context.put("vector", vec);
```

[Listing 40]

In the Template we can easily loop over this element by the use of the foreach directive. This directive needs two arguments separated by the word `in`. The first argument is a variable in which Velocity stores the current value of the looped element. The second one is the element that the foreach directive should access. In the `foreach` block we can easily query the value of the current element by using the first argument we specified in the `foreach` statement. The variable `$velocityCount` stores the current loop counter.

```
1 Looping over the friends Vector:
2 #foreach($element in $vector)
3  $velocityCount $element is my friend!
4 #end
```

[Listing 41]

The output of this `foreach` directive is shown in Figure 16.

[Figure 17, VTL foreach output]

4.2.6 #include

„The #include *script element allows the template designer to import a local file, which is then inserted into the location where the* #include *directive is defined. The contents of the file are not rendered through the template engine.“ [VUse06]*

The file which is included must be in the same directory as the Template file. Let's create a text file with some text and a variable, to test if the variable will be resolved or not. The text file (test1.txt) is shown in Listing 42.

```
1 $hello from test1.txt
```

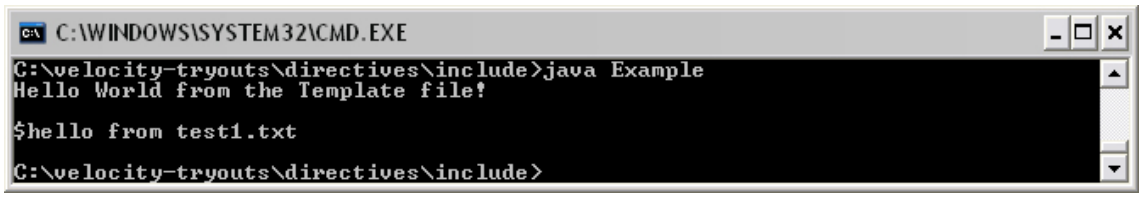
[Listing 42]

The corresponding Template file is shown in Listing 43. First we create a variable with the value `Hello World`. Next we output this variable following some text. Afterwards we use the include directive to get the text from the `test1.txt` text file.

```
1 #set($hello = "Hello World")
2 $hello from the Template file!
3
4 #include("test1.txt")
```

[Listing 43]

The output in Figure 16 shows, that the first hello variable in the Template file has been rendered and the variable from the text file not.



[Figure 18, VTL include output]

4.2.7 #parse

The parse directive also includes a file, but renders it. Let's just change the `include` of Listing 43 to `parse`. The parsed file is rendered and the variable `$hello` is changed to `Hello World`. The output is shown in Figure 16.

[Figure 19, VTL parse output]

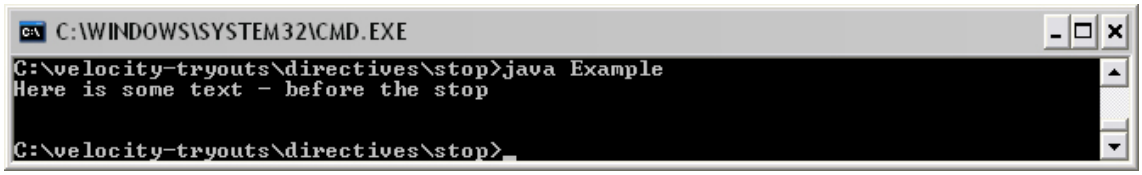
4.2.8 #stop

The stop directive simply stops the processing of the Template. Listing 44 shows some text before and some text after the stop directive.

```
1 #set($text = "Here is some text")
2 $text - before the stop
3 #stop
4 $text - after the stop
```

[Listing 44]

The output in Figure 16 shows that the processing stops at Line 3.



[Figure 20, VTL stop output]

4.3 Velocimacros

„The #macro *script element allows template designers to define a repeated segment of a VTL template. Velocimacros are very useful in a wide range of scenarios both simple and complex.“ [VUse06]*

A macro can be defined using the `#macro` syntax. After that a bracket is following with at least one argument. The first argument defines the name of the macro and the following argument(s) define the name(s) of the variables which can be passed to this macro.

Listing 45 shows a simple implementation of macros. In line 2 a macro with the name hello is created. To this macro it is possible to pass one variable, which is

available in the macro under the name `$hello`. This macro can be called like shown in Line 7. The call to a macro looks like a call to any other directive, but the name of the macro stands after the hash. Line 6 shows another macro, called `test` which is able to receive four arguments. When this macro is processed it calls another macro in Line 7, which proofs that it is possible to call macros out of macros. It is possible to pass the following objects to macros as arguments:

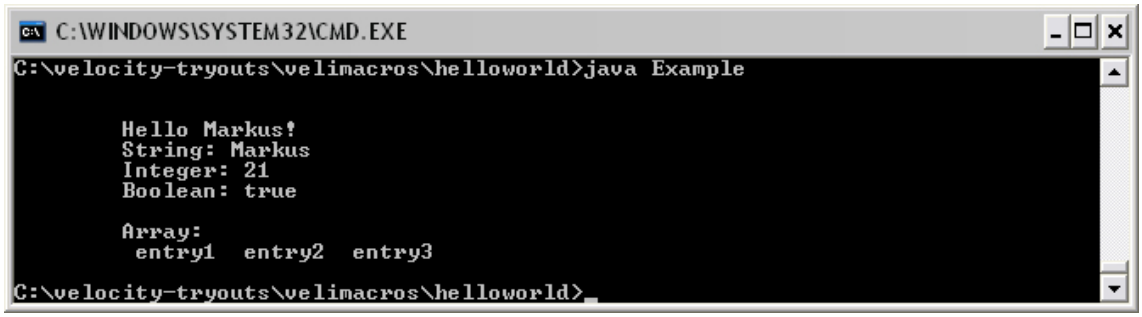
- References
- String literals
- Number literals
- Integer ranges
- Arrays
- Boolean values

For example in this macro an array is passed and it is processed in line 13.

```
1
2 #macro(hello $who)
3 Hello $who!
4 #end
5
6 #macro(test $string $integer $boolean $array)
7       #hello("Markus")
8       String: $string
9       Integer: $integer
10      Boolean: $boolean
11
12      Array:
13      #foreach($entry in $array) $entry #end
14 #end
15
16 #test("Markus" 21 true ["entry1", "entry2", "entry3"])
```

[Listing 45]

The output of this example is shown in Figure 18.



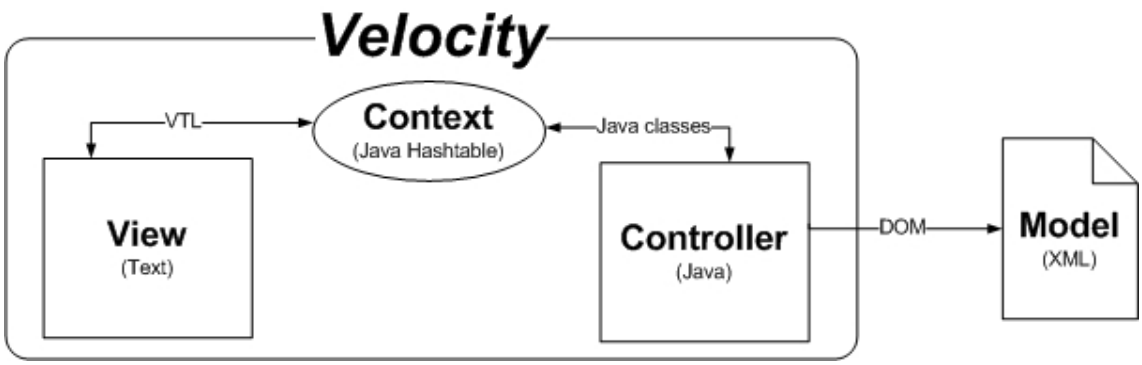
[Figure 21, VTL macro output]

To create a macro which is accessible from any template file, it is possible to insert it into a global macro file. This global macro file must be named `VM_global_library.vm` and it must be in the same directory as the Templates are. But it is possible to change the name and directory of the global macro file in the Velocity properties.

5 Sample Applications

5.1 Parsing XML

Parsing an XML file in Velocity is more a Java related topic. This application is a good example for understanding the Model View Controller. Our View will be a Velocity Template which will output Text. The controller is, as usual, our Java code and the model will be an XML file. As we already know, the view is able to access the Context by using the VTL and the Controller can access the Context by using the appropriate Java classes. The Controller will be able to access our Model by using the Document Object Model.



[Figure 22, parsing XML MVC]

5.1.1 Model

In this case the Model is our XML file, which is filled with a dataset of friends.

*„Extensible Markup Language (XML) is a simple, very flexible text format derived from SGML (ISO 8879). Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere.“ [*W3cX06*]*

```
1 <?xml version='1.0' encoding='utf-8'?>
2 <Friends>
3    <Friend>
4      <Name>Christoph Huber</Name>
5      <Age>21</Age>
6    </Friend>
7
8    <Friend>
9      <Name>Niki Lauda</Name>
10      <Age>57</Age>
11   </Friend>
12
13    <Friend>
14      <Name>Heidi Klum</Name>
15      <Age>33</Age>
16    </Friend>
17 </Friends>
```

[Listing 46]

5.1.2 View

The designer has the task to output all entries of the XML document, which have been put into the Context. Therefore he talks with the Controller programmer and they both agree that the entries will be in a vector. This vector has to have a specific format which is:

- The name of the person
- The age of the person
- The string `"break"`

The designer decided to separate the entries by the string `break` in order to be able to create breaks between the different people. As he now has all the information about what the Controller programmer will do, he can start to write the View which is shown in Listing 47.

```
1 My friends are:
2 #foreach($element in $Vector)
3  #if("break" == $element)
4
5  #else
6   $element
7  #end
8 #end
```

[Listing 47]

In the second Line the foreach directive is used to access the Vector and put the current element into the variable `$element`. In line 3 the if directive is used to check if the current element is the break element. If this is the case, a break is added in line 4. If this is not the case, the else directive is used to output the current element.

5.1.3 Controller

The Controller programmer has now the task to create the Vector in the format he agreed on with the designer. To parse the XML document we use the Document Object Model class `org.w3c.dom`. *"The Document Object Model is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. The document can be further processed and the results of that processing can be incorporated back into the presented page." [W3cD06]*

Note that the general code still is the same as in Listing 19 of the Hello World example. Now we just input something into the Context, but the pattern of initializing Velocity and merging stays the same.

Listing 48 shows how to use the `org.w3c.dom` in order to parse our XML file shown in Listing 46. In line 3 a new instance of the `DocumentBuilderFactory` has to be created, to get an instance of the `DocumentBuilder` from it. This `DocumentBuilder` can be used to parse our XML file. This happens in Line 7 and the parse method delivers a document object. This object can be queried for several nodes of the XML file. In Line 9 a so called `NodeList` is created which stores all nodes with the name `"Name"`. In Line 10 the same thing happens with all nodes with the name `"Age"`.

The next step is to create the Vector where all entries should be added. We can now iterate through the Node Lists to do so. In Line 14 a for directive is created which goes through the Node List step by step. In Line 15 we create a Node object which holds the value of the node (in our case the text). In Line 16 we are now able to get a String out of this node by using the `getNodeValue()` method. In Line 18 and 19 we do the same for the Age node, which provides us the second String we need.

We are now able to put both Strings into the Vector and we can add afterwards the `break` String. This will be carried out for all Nodes in the XML file.

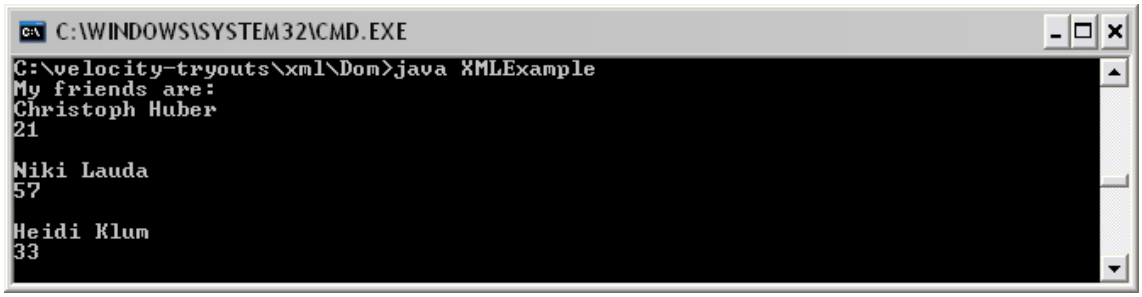
Finally we can put the Vector into the Context.

```
1 try {
2
3   DocumentBuilderFactory factory  =
4   DocumentBuilderFactory.newInstance();
5   DocumentBuilder builder  =
6   factory.newDocumentBuilder();
7   Document document = builder.parse(new File("myFriends.xml"));
8
9   NodeList ndListName = document.getElementsByTagName("Name");
10  NodeList ndListAge = document.getElementsByTagName("Age");
11
12  Vector friendsVector = new Vector();
13
14  for( int i=0; i<ndListName.getLength(); i++ ) {
15     Node nodeDataName = ndListName.item(i).getFirstChild();
16     String sDataName = nodeDataName.getNodeValue();
17
18     Node nodeDataAge = ndListAge.item(i).getFirstChild();
19     String sDataAge = nodeDataAge.getNodeValue();
20
21     friendsVector.add(sDataName);
22     friendsVector.add(sDataAge);
23     friendsVector.add("break");
24   }
25 }
26
27 context.put("Vector", friendsVector);
```

[Listing 48]

5.1.4 Output

The output for this example is shown in Figure 19.



[Figure 23, parsing XML output]

5.2 Using the Bean Scripting Framework

"Bean Scripting Framework (BSF) is a set of Java classes which provides scripting language support within Java applications, and access to Java objects and methods from scripting languages. BSF allows one to write JSPs in languages other than Java while providing access to the Java class library. In addition, BSF permits any Java application to be implemented in part (or dynamically extended) by a language that is embedded within it. This is achieved by providing an API that permits calling scripting language engines from within Java, as well as an object registry that exposes Java objects to these scripting language engines." [JBSF06]

In this chapter three different nutshell examples of how to use BSF are provided. The used programming languages are the Tool Command Language (Tcl), Java Script and IBM's Restructured eXtended eXecutor (Rexx). There will be two simple examples for Tcl and JavaScript and a more sophisticated one for BSF4Rexx.

5.2.1 Jacl – Tcl

To let the Bean Scripting Framework understand this programming language, Jacl has to be used. Jacl provides the "translation" of the Tcl commands for the Bean Scripting Framework. It can be downloaded at the official Jacl webpage[14].

[14] http://tcljava.sourceforge.net/docs/website/index.html

In this example we will execute a Tcl script out of Java without assigning the result to our Template. This lets us see how Velocity behaves when an output is coming directly from the controller code.

In Listing 49 we have a Tcl code which assigns a number to a variable (line 1), prints a String (line 3) and at least prints another string including a variable (line 4).

```
1 set number 5
2
3 puts "greets from Tcl via Jacl supported by BSF"
4 puts "the number is: $number"
```

[Listing 49]

The Template code is shown in Listing 50. It just outputs a small message that this text is coming from Velocity.

```
1 This is Velocity speaking!
```

[Listing 50]

The code for executing the script is shown in Listing 51. First we have to create an instance of the `BSFManager`, which handles all scripting execution engines running under its control. The next step is to read in the whole script using the File Reader. In Line 5 the `getStringFromReader` returns a String containing the script. This script can be passed to the `exec` method which just executes the script. The first argument is the name of the scripting language, the second one the source file, the third and the fourth are the line and the column numbers in the source for expressions. The last argument is the String with the script which we read in before. The script is now executed by the `BSFManager`.

[JBSF06]

```
1 BSFManager mgr = new BSFManager ();
2
3 try {
4   FileReader in = new FileReader("test.jacl");
5   String script = IOUtils.getStringFromReader(in);
6   mgr.exec("jacl", "test.jacl", -1, -1, script);
7 }
```

[Listing 51]

The output is shown in Figure 20, where we can see that first the Velocity Template is executed and afterwards the Tcl code.

[Figure 24, BSF Tcl output]

5.2.2 Rhino – JavaScript

To use JavaScript in the Bean Scripting Framework Rhino has to be used, which can be downloaded from the official Rhino webpage[15]. *"Rhino is an open-source implementation of JavaScript written entirely in Java. It is typically embedded into Java applications to provide scripting to end users." [Rhin06]*

In this example we will evaluate a JavaScript script which is shown in Listing 52. This script calls a function which adds 1 to the argument it received. After that it returns the sum.

```
1 function f(x){
2  return x+1
3 }
4
5 f(99)
```

[Listing 52]

[15] http://www.mozilla.org/rhino

As we can see in Listing 53 our Template file requires a variable called `calc` in the Context, which should represent the result of the JavaScript.

```
1 This is Velocity speaking!
2
3 JavaScript calculated: 99 + 1 = $calc
```

[Listing 53]

To evaluate a JavaScript the first thing we have to do is to create and enter a Context (note: this has nothing to do with the Velocity Context) as shown in Listing 54 – Line 1. The Context stores information about the execution environment of a script. Next we have to initialize the Standard Object which we have to use later. The next steps (Line 5-6) are similar to Listing 51 as we just put the JavaScript file into a String. In Line 8 the Context is used to evaluate a string. The scope is the Object where Rhino looks for variables. The second argument is our JavaScript and the third argument tells Rhino where to output errors. The fourth argument tells Rhino at which line number the error output should start.

Now the script is evaluated and we store the result in an object. This object is converted into a String and inserted into the Velocity Context by the identifier calc.

```
1 Context cx = Context.enter();
2 Scriptable scope = cx.initStandardObjects();
3
4 try {
5   FileReader in = new FileReader ("test.js");
6   String script = IOUtils.getStringFromReader (in);
7
8   Object result = cx.evaluateString
9   (scope, script, "<cmd>", 1, null);
10
11  context.put("calc", cx.toString(result));
12 }
```

[Listing 54]

The output of this example is shown in Figure 21.

[Figure 25, BSF JavaScript output]

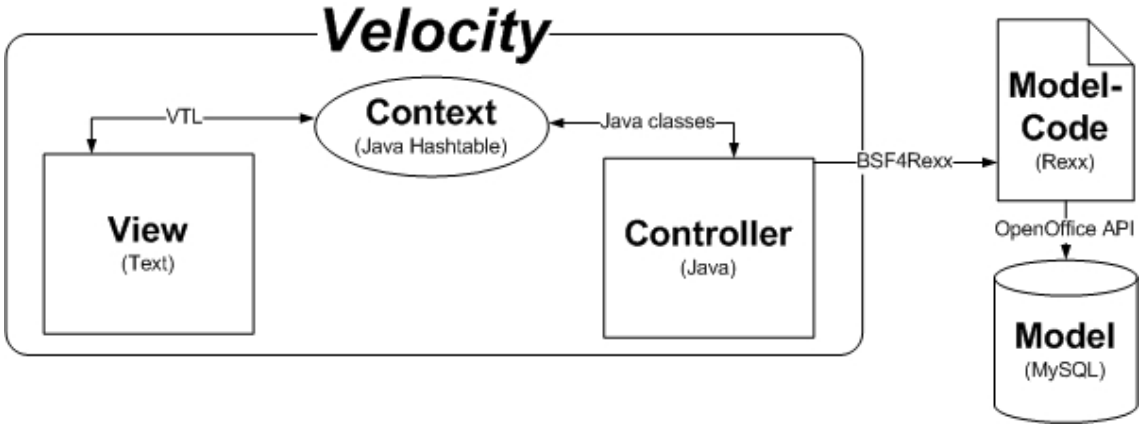
5.2.3 BSF4Rexx – ObjectRexx

BSF4Rexx is the Bean Scripting Framework for ObjectRexx. In this example ObjectRexx is used to be the model code of our application. To grant ObjectRexx access to a database once again the BSF4Rexx engine is used. *„Because OpenOffice.org provides a Java interface (the JavaUNO) and ObjectRexx can be interpreted to Java via BSF4Rexx, ObjectRexx can be used for OpenOffice.org automation.“ [Aham05]*

OpenOffice provides the possibility to connect to databases. The way this works is not explained in this thesis, so please refer to the bachelor thesis of Stefan Schmid[16].

In this application we want to retrieve a value of a MySQL table and ouput it in the Template. The model code should be able to be called with arguments, so that we can choose which row to query. Figure 22 shows the way we would like to create our application. As explained we will use BSF4Rexx in the Controller to run the Model code which is written in ObjectRexx. ObjectRexx itself will then use BSF4Rexx to get access to the database with the OpenOffice API.

[16] *[Stef06]*



[Figure 26, BSF4Rexx example MVC]

In this example a MySQL database is used which is called `test` and has a structure shown in Figure 23.

[Figure 27, BSF4Rexx example MySQL structure]

The necessary ObjectRexx code to retrieve the text attribute by the corresponding ID is shown in Listing 55. The code will be explained very sparsely so, once again, if you are interested in automating OpenOffice via BSF4Rexx please refer to the works of Mr. Ahammer and Mr. Schmid. *[Aham05, Stef06]*

Important for this application is Line 26 where the query is build. We query the table `test` for the attribute `text` where the first argument which we provided to this script matches the `ID` column. At Line 40 the result of this query is returned.

```
1 xContext = UNO.connect()  -- connect to server and retrieve the
2 XContext object
3 XMcf = xContext~getServiceManager
4
5 oDriverManager = -
6 xMcf~createInstanceWithContext( -
7 "com.sun.star.sdbc.DriverManager", xContext)
8 xDriverManager = oDriverManager~XDriverManager
9
10 URL = "jdbc:mysql://localhost:3306/test"
11
12 props = bsf.createArray(.UNO~PropertyValue, 3)
13 props[1] = .UNO~PropertyValue~new
14 props[1]~Name  = "user"
15 props[1]~Value = "root"
16 props[2] = .UNO~PropertyValue~new
17 props[2]~Name  = "xXxXxX"
18 props[2]~Value = "goforit"
19 props[3] = .UNO~PropertyValue~new
20 props[3]~Name  = "JavaDriverClass"
21 props[3]~Value = "org.gjt.mm.mysql.Driver"
22
23 xConnection = xDriverManager~getConnectionWithInfo(url, props)
24 xStatement = xConnection~createStatement
25
26 query = "SELECT text FROM test WHERE ID = "arg(1)
27
28 xResult = xStatement~executeQuery(query)
29 xRow = xResult~XRow
30
31 result = ""
32
33 IF xResult~isBeforeFirst = 0 THEN
34    result = "no results!"
35 ELSE DO
36    DO WHILE xResult~next > 0
37      result = xRow~getString(1)
38    END
39 END
40 return result
41 ::requires UNO.CLS
```

[Listing 55]

The Controller code now has the task to call the ObjectRexx script and pass the ID we want to query. The appropriate code is shown in Listing 56. In Line 2 and 3 the BSF4Rexx engine is initialized. After that a Vector is created which holds the arguments we want to pass to the ObjectRexx script. In this case we decide to pass “3” to the ObjectRexx script. Afterwards we put the whole script of Listing 55 into a String. In Line 12 the `apply` method of the BSF4Rexx Engine is called which runs the script and passes it the arguments Vector. This method returns an Object which should hold the result of the query. In Line 15 we put

this Object into the context, as Velocity uses the `toString` method the result should be displayed in the Template.

```
1  try {
2    BSFManager mgr         = new BSFManager();
3    BSFEngine  rexxViaBSF = mgr.loadScriptingEngine("rexx");
4
5    Vector vArgs = new Vector();
6
7    vArgs.add("3");
8
9    FileReader in = new FileReader("select.rex");
10   String script = IOUtils.getStringFromReader(in);
11
12   Object obj = rexxViaBSF.apply(null, 0, 0, script, null,
13   vArgs);
14
15   context.put("RexxValue", obj);
16 }
```

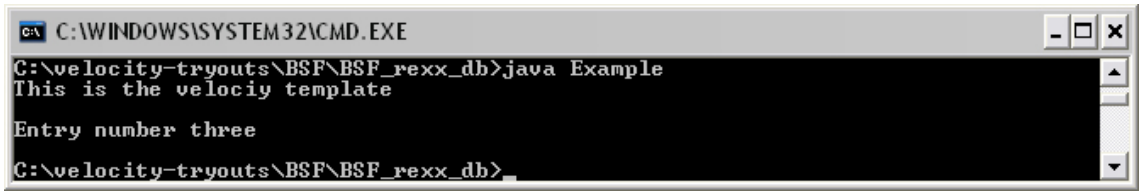
[Listing 56]

The Template which outputs the result is shown in Listing 57.

```
1 This is the velociy template
2
3 $RexxValue
```

[Listing 57]

The output of this example is shown in Figure 24. Indeed the correct result of the query was added to our Context and outputted in the Template file.



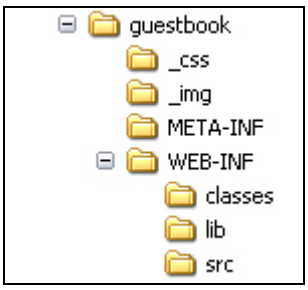
[Figure 28, BSF4Rexx example output]

5.3 Guestbook using Servlets

This example shows how to use Velocity in web applications. The target of this example is to create a Guestbook using Velocity. To do so you need to have a servlet engine installed. In this example the servlet engine Tomcat[17] is used. For hints how to install and run tomcat please refer to the provided Tomcat website. In the following the steps to create this web application are described.

5.3.1 Create a web application in Tomcat

The folder structure of a web application in Tomcat has to have the same one as shown in Figure 25. The `_css`, `_img` and `META-INF` folders are optional. In the root of the application (`/guestbook/`) the Template file is located. The `WEB-INF` directory has sub directories. In the source directory we have to put the Java source code, in the lib directory necessary libraries have to be put and in the classes directory we have to put the compiled Java classes.



[Figure 29, Tomcat directory structure]

In the `WEB-INF` directory itself we have to put the web.xml. This xml file is needed by Tomcat to know how the servlet is composed. It is shown in Listing 58. In line 5 we tell Tomcat that we will use the VelocityViewServlet for processing our servlet requests. It will be discussed in the next chapter. In line 8 we tell the VelocityViewServelt to load the toolbox.xml which is located in the `WEB-INF` directory. This is part of the VelocityViewServlet and will be described as well in the next chapter. The load-on-startup in line 11 means that the servlet will be loaded initially when the server starts and not when the client requests it.

Now Tomcat has to know what we want to display if someone requests our application. This is shown in line 13 where we map Tomcat to the .vm files in the root directory. When the user first requests the application he will be forwarded to the index.vm which we defined in line 18.

```
1 <web-app>
2  <servlet>
3    <servlet-name>velocity</servlet-name>
4    <servlet-class>
5       org.apache.velocity.tools.view.servlet.VelocityViewServlet
6    </servlet-class>
7    <init-param>
8      <param-name>org.apache.velocity.toolbox</param-name>
9      <param-value>/WEB-INF/toolbox.xml</param-value>
10    </init-param>
11    <load-on-startup>10</load-on-startup>
12  </servlet>
13  <servlet-mapping>
14    <servlet-name>velocity</servlet-name>
15    <url-pattern>*.vm</url-pattern>
16  </servlet-mapping>
17  <welcome-file-list>
18    <welcome-file>index.vm</welcome-file>
19  </welcome-file-list>
20 </web-app>
```

[Listing 58]

To run a web application it is possible to create a web archive. This web archive holds all information about our servlet, including the web.xml and all libraries which we want to use. If you are interested in trying this example yourself, you can use the provided ant code in the appendix.

5.3.2 VelocityViewServlet

In this example we will use the VelocityViewServlet, which is provided in the VelocityTools[18], which are not part of the Velocity core. *"VelocityView provides support for rapidly and cleanly building web applications using Velocity templates as the view layer. The project is designed with the Pull-MVC Model in mind and works well in conjunction with web application frameworks that act as*

[17] http://tomcat.apache.org/
[18] http://jakarta.apache.org/velocity/tools/index.html

the controller (e.g. Struts), but can be used quite effectively on its own for those creating simpler applications.“ [VVie06]

“The Pull-MVC Model entails the ability to create an object that is able to "pull" the required information out at execution time within the template.“ [Turb06]

We will use the VelocityViewServlet as a stand alone application. In this case we do not have to add any details about the Template or Context in the Controller code. All we have to do is to add the whole Controller code class to the Context of the servlet. In the VelocityViewServlet the Context is configured in the toolbox.xml.

So let’s take a closer look at this file, which is shown in Listing 59. To let the View access the Controller class we have to create a new `tool`. This tool has a key which is the identifier in the Context, a scope (could be application, session or request) which determines the lifecycle of the tool and the class. The class property is the same as we have to use in our Controller code. A big advantage of the VelocityViewServlet is the possibility to implement tools. In this case we use the ParameterParser written by Nathan Bubna to get the POST array or the GET data.

```
1 <toolbox>
2  <tool>
3     <key>guestbook</key>
4     <scope>request</scope>
5     <class>Guestbook</class>
6  </tool>
7  <tool>
8   <key>params</key>
9   <scope>request</scope>
10  <class>
11     org.apache.velocity.tools.view.tools.ParameterParser
12   </class>
13  </tool>
14 </toolbox>
```

[Listing 59]

5.3.3 Planning process

Let’s consider we have three people involved in our project. According to the MVC concept we have a designer who will code the View, a Java programmer

who will create the Controller code and a Java programmer who will create the code for accessing the Model. In a meeting they agree on which methods to use and what they have to return in order to guarantee a smooth workflow.

5.3.4 Controller

The programmer has to create the Guestbook class which is shown in Listing 60. In Line 5 the programmer creates a method which returns the result Vector of the database query. In Line 9 the method `newEntry` receives the data from the View and sends it to the Model code. This is a perfect example for what the Controller code has to do. It is the bridge between the View and the Model and has the task to respond to actions made in the View. On the other hand it is also used to provide some functions to the View. This is shown in Line 20 where a method is created which tells the View if a new entry was successfully added or not.

```
1 public class Guestbook {
2
3   private boolean falseNewEntry;
4
5   public Vector getEntries() {
6      return select();
7   }
8
9   public void newEntry(String name, String mail,
10  String homepage, String text) {
11     text = text.replace("\r\n", "<br>");
12
13     insert(name, mail, homepage, text);
14  }
15
16  public void setfalseNewEntry(boolean falseNewEntry)  {
17     this.falseNewEntry = falseNewEntry;
18  }
19
20  public boolean getfalseNewEntry() {
21     if (this.falseNewEntry) {
22     return true;
23  } else {
24     this.falseNewEntry = true;
25     return false;
26  }
27 }
```

[Listing 60]

The Controller code is very short as we don’t have to make complex calculations for the Guestbook example.

5.3.5 Model

In this example a MySQL database is used which is shown in Figure 26. There are five fields which will be filled with the Guestbook entries. The primary key is the `ID` which increments automatically. This is the basis for the Model programmer. His task is now to create methods which are able to insert data into the database and query data from the database.

Field	Type	Null	Key	Default	Extra
ID	int(3)	NO	PRI	NULL	auto_increment
name	varchar(50)	YES		NULL	
homepage	varchar(50)	YES		NULL	
mail	varchar(50)	YES		NULL	
text	text	YES		NULL	

[Figure 30, Guestbook MySQL]

The Model programmer has to use the methods he agreed on with the Controller programmer. In order to establish a connection to the database, a database Driver is needed. It can be easily downloaded from the MySQL webpage[19]. In Listing 61 the code for inserting a new entry is shown. Important for this application is that this piece of code is able to insert the new entry into the database. Therefore the Controller programmer passes the necessary data to this method. In Line 13 to 15 the SQL statement is created which inserts the data. As long as this piece of code works, all other people involved in this project don't have to have any idea about accessing a database from Java.

[19] http://www.mysql.com

```
1 public void insert(String name, String mail,
2 String homepage, String text) {
3
4   Connection con;
5   Statement stm;
6
7    try {
8      DriverManager.registerDriver(new com.mysql.jdbc.Driver());
9      con = DriverManager.getConnection(
10      "jdbc:mysql://localhost:3306/guestbook", "root", "pw");
11      stm = con.createStatement();
12
13      stm.executeUpdate("INSERT INTO guestbook VALUES(DEFAULT,
14      '"+ name +"','"+ homepage +"','"+ mail +"',
15      '"+ text +"')");
16
17      } catch (Exception x) {
18              System.out.println("Error: "+x);
19      }
20
21 }
```

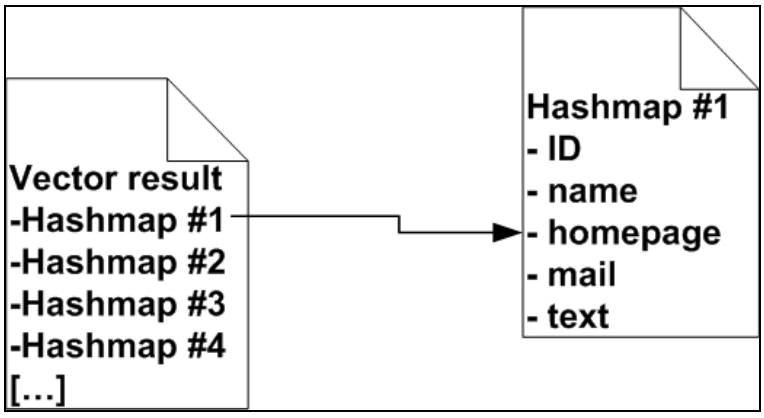
[Listing 61]

Just for reasons of completeness the code for querying the database is shown in Listing 62. Once again; the reason why someone uses Velocity in his project is to separate the Model, Controller and the View code. So in this case it is not important for Velocity how accessing the database works. Therefore the Model code is not discussed in detail. Important for us is that this method returns a Vector which itself holds Hashmaps. These Hashmaps are filled with the different entries which are coming from the database.

```
1 public Vector select() {
2
3   Connection con;
4   Statement stm;
5   String message = "";
6
7   Vector result = new Vector();
8
9    try {
10      DriverManager.registerDriver(new com.mysql.jdbc.Driver());
11      con = DriverManager.getConnection(
12     "jdbc:mysql://localhost:3306/guestbook", "root", "pw");
13      stm = con.createStatement();
14
15      ResultSet rset = stm.executeQuery(
16      "select * from guestbook");
17
18      while (rset.next()) {
19      Map dbResult = new HashMap();
20       dbResult.put("id", rset.getString("ID"));
22       dbResult.put("name", rset.getString("name"));
23       dbResult.put("homepage", rset.getString("homepage"));
24       dbResult.put("mail", rset.getString("mail"));
25       dbResult.put("text", rset.getString("text"));
26       result.add(dbResult);
27      }
28  } catch (Exception x) {
29      System.out.println("Error: "+x);
30  }
31
32  return result;
33 }
```

[Listing 62]

To illustrate this Figure 27 shows the design of this Vector. This is the only thing the View programmer has to know, as he has to work with the Vector object.



[Figure 31, Guestbook Vector design]

5.3.6 View

Let's review what happened so far. The Controller programmer and the Model programmer have done their job. But the designer don't has to care about how they did it, he just has to have the following information:

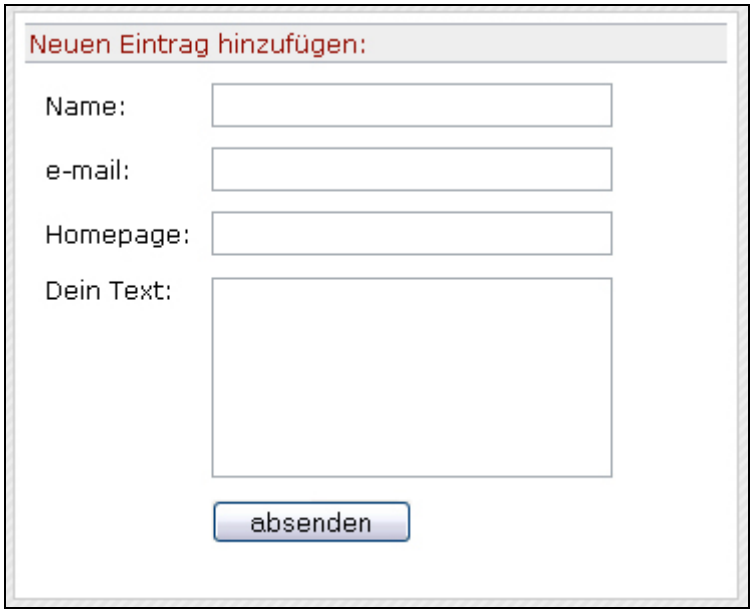
- If he wants to get all entries he has to call the method `getEntries` of the guestbook class. This returns him a Vector which includes many Hashmaps, which themselves represent the entries.
- If he wants to insert a new entry he has to call the method `getEntry` of the guestbook class.
- If he wants to know if the insert was successful or not, he has to call the method `getfalseNewEntry` of the guestbook class, which returns him a Boolean value.

With this information he can now begin to create the View code. In the following Listings only the VTL is discussed. The look and feel of the application is not as important as discussing it at this point. Listing 63 shows how to get the entries out of the guestbook and how to display them. In Line 1 the Vector which holds the HashMaps is accessed and in Lines 2 to 6 we get the values out of the HashMaps by directly querying the values by their keys.

```
1 #foreach($item in $guestbook.getEntries())
2   ID: $item.id
3   Name: $item.name
4   Homepage: $item.homepage
5   Mail: $item.mail
6   Text: $item.text
7 #end
```

[Listing 63]

To add new entries the designer decides to create a nice looking form which is shown in Figure 28. It lets the user input all necessary data to create a new guestbook entry. When the user hits the submit button the data he inputted is POSTed to the next site.



[Figure 32, Guestbook Form]

This site is called newEntry.vm and its task is to add new entries to the corresponding method. As the data in this form is POSTed, we have to access the POST array to get the values. This is the point where the Designer can use the ParameterParser tool. Listing 64 shows the code of the newEntry.vm. First the Designer checks if the name and the text fields have been filled out by the user. If not he sets the `addEntry` variable to false. This has the effect that the `newEntry` method of the guestbook class is not called. This is to avoid that user type in nothing and that an empty guestbook entry is inserted in the database. If these two values are filled with some content, the `newEntry` method is called in line 12 and the parameters of the ParameterParser are passed. The ParameterParser uses the variable `params` to let the Designer access the POST array. In Line 19 the file which shows the guestbook entries is parsed so that the new entry is displayed.

```
#set($addEntry = true)

#if($params.name == "" || !$params.name)
  #set($addEntry = false)
#end

#if($params.text == "" || !$params.text)
  #set($addEntry = false)
#end

#if($addEntry)
     $guestbook.newEntry($!params.name,
$!params.mail, $!params.homepage,
$!params.text)
#else
     $guestbook.setfalseNewEntry(true)
#end

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

[Listing 64]

The whole look and feel of this application is shown in Figure 29.

[Figure 33, Guestbook application look and feel]

5.4 Powering Ajax Applications

5.4.1 Fundamentals

Ajax is short for Asynchronous JavaScript and XML. Ajax isn't a single technology; rather it is a collection of four technologies that complement one another.

- JavaScript is the language in which Ajax applications are written.
- Cascading Style Sheets are used to style the application.
- *"The XMLHttpRequest allows web programmers to retrieve data from the web server as a background activity. The data format is typically XML." [Ajax06]*
- The Document Object Model is used to parse the XML file retrieved from the web server.

Ajax is used to retrieve data from the Server without reloading the whole page, using the XMLHttpRequest object. Ajax applications need a XML document to deliver the necessary data from the server to the client. That's where Velocity comes into play.

Velocity is able to create any file format, so it is possible to create XML files. In order to make these files available to the Ajax application a Tomcat server is needed. In this example we are using once again the VelocityViewServlet to generate the output files. As this example is only a proof of concept it is very simple, but can be extended easily.

[Ajax06]

5.4.2 Controller

To see that the creation of the XML file is dynamic, every time the method `getResponse` is called a random number is returned.

```
1 public class easyAjax {
2
3   public Integer getResponse(){
4      Random generator = new Random();
5      int number = generator.nextInt(6) + 1;
6      return number;
7   }
8 }
```

[Listing 65]

5.4.3 Toolbox

The toolbox exposes the class created in the controller to the Template by the name `easyAjax`. Furthermore it holds a String called `message` with the text `This is Velocity speaking`.

```
1   <tool>
2     <key>easyAjax</key>
3     <scope>request</scope>
4     <class>easyAjax</class>
5  </tool>
6  <data type="string">
7    <key>message</key>
8    <value>This is Velocity speaking</value>
9  </data>
```

[Listing 66]

5.4.4 View

The view is our XML file, so it is called index.xml. In Line 4 the Node `number` node has the text which should show the output from Velocity, by calling the method `getResponse` from the `easyAjax` class.

```
1 <?xml version='1.0' encoding='ISO-8859-1' ?>
2 <document>
3  <number>
4   $message - your number is: $easyAjax.getResponse()
5  </number>
6 </document>
```

[Listing 67]

5.4.5 The velocity.properties configuration file

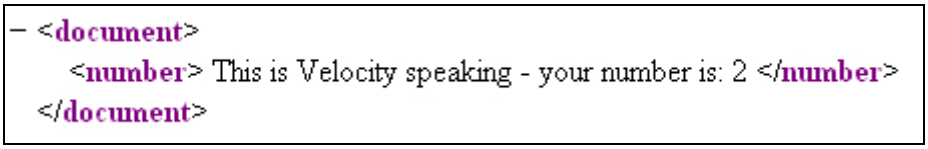
The only thing which has to be modified is the content Type of the xml document, as it is text/html by standard. Therefore a file named `velocity.properties` in the `WEB-INF` directory has to be created which is shown in Listing 68. It just sets the default Content Type of our documents to text/xml, so the Document Object Model, which is used to parse XML files in JavaScript, can now do his work.

```
1 default.contentType = text/xml
```

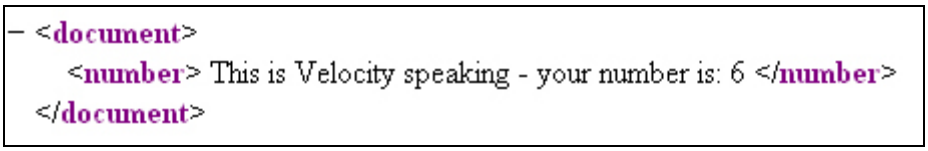
[Listing 68]

5.4.6 Output

The output of 2 separate calls to this file is shown in Figure 29 and Figure 30.



[Figure 34, Ajax output 1]



[Figure 35, Ajax output 2]

6 Summary

Velocity is an easy to use tool which is widely accepted in the community. The Velocity Template Language perfectly meets the needs of Designer which can use this easy to learn language, to access the Context. In web applications Velocity performs very well using the VelocityViewServlet. Another big advantage of Velocity compared to other Templating engines is the fact, that it uses Java. This lets the programmer open up new dimensions compared to scripting languages.

Velocity becomes reasonable even if it handles small projects. But Velocity can be much more powerful when it comes to big applications, where hundreds of people are involved. In this case Velocity can help the developers to make the maintenance of their code much easier. But today the probability to find developers who are familiar with Velocity is not very high.

Therefore Velocity is a good approach, but needs to get promoted better to increase the acceptance.

7 List of references

[Aham05] Andreas Ahammer: OpenOffice.org Automation, Bachelor Course Paper, Department of Business Informatics (Prof. Dr. Rony G. Flatscher), Vienna University of Economics and Business Administration, November 06, 2005

[Ajax06] Dave Crane, Eric Pascarello: Ajax in Action, Manning, 2006

[Antl06] Terence Parr, University of San Francisco: Java Template Engine, http://www.antlr.org/stringtemplate/index.tml, retrieved on 2006-10-26

[Chee06] Cheetah, http://www.cheetahtemplate.org/, retrieved on 2006-10-31

[Cont06] Contemplate Web Templating System http://www.typea.net/software/contemplate/assembled/home.html, retrieved on 2006-10-30

[EmbP06] Embperl, http://perl.apache.org/embperl/, retrieved on 2006-11-04

[Free06] Freemarker, http://freemarker.sourceforge.net/, retrieved on 2006-11-29

[JBSF06] Bean Scripting Framework, http://jakarta.apache.org/bsf/, retrieved on 2006-25-12

[JFre06] Vincent Dibartolo: FreeMarker: An open alternative to JSP, www.javaworld.com/javaworld/jw-01-2001/jw-0119-freemarker.html, retrieved on 2006-11-29

[ONLa06] Andrew Glover: Python-Powered Templates with Cheetah, http://www.onlamp.com/pub/a/python/2005/01/13/cheetah.html, retrieved on 2006-10-31

[Open06] Open Source Initiative, http://www.opensource.org/, retrieved on 2006-10-31

[Perl06] Neil Gunton: Creating Modular Weg Pages With EmbPerl, http://www.perl.com/pub/a/2001/03/embperl.html, retrieved on 2006-11-04

[Rhin06] Rhino: JavaScript for Java, http://www.mozilla.org/rhino/, retrieved on 2006-25-12

[SGNU06] Smarty GNU, http://www.gnu.org/copyleft/lesser.html#SEC1, retrieved on 2006-11-20

[Smar06] Smarty Template Engine, http://smarty.php.net/, retrieved on 2006-10-30

[Stef06] Stefan Schmid: OpenOffice.org Database, Bachelor Course Paper, Department of Business Informatics (Prof. Dr. Rony G. Flatscher), Vienna University of Economics and Business Administration, 2006

[Sun06] Sun Microsystems Inc., Java Blueprints, Model-View-Controller, http://java.sun.com/blueprints/patterns/MVC-detailed.html, retrieved on 2006-10-26

[Turb06] Turbine: Pull vs. Push, http://jakarta.apache.org/turbine/further-reading/pullmodel.html, retrieved on 2006-12-27

[VApi06] Apache Velocity API, http://jakarta.apache.org/velocity/docs/api/index.html, retrieved between 2006-10 and 2007-01

[VDev06] Apache Velocity Developer Guide, http://jakarta.apache.org/velocity/docs/developer-guide.html, retrieved between 2006-10 and 2007-01

[Velo06] Apache Velocity, http://jakarta.apache.org/velocity/index.html, retrieved between 2006-10 and 2007-01

[VUse06] Apache Velocity User Guide, http://jakarta.apache.org/velocity/docs/user-guide.html, retrieved between 2006-10 and 2007-01

[VVie06] VelocityView, http://jakarta.apache.org/velocity/tools/view, retrieved on 2006-12-27

[W3cD06] World Wide Web Consortium: Document Object Model, http://www.w3.org/DOM/, retrieved on 2006-12-12

[W3cX06] World Wide Web Consortium: Extensible Markup Language, http://www.w3.org/XML/, retrieved on 2006-12-12

8 Appendix:

8.1 Setting up Velocity

To get the newest Version of Velocity go to the Velocity web page[20]. Download the latest version and unzip into a folder you like. Please note that it is recommended to choose a directory without spaces.

Velocity requires a Java development kit and a Java platform, which can both be downloaded from the sun page.

- If you want to build Velocity by yourself you need the tool ant. Go to the build directory ant type `ant jar`, to create a Java archive file. This can then be included into your classpath.
- If you use the binary version of Velocity just insert the classpath located in the root directory into your classpath.

If you have any problems please refer to the Velocity manual.

8.2 Setting up ant

Ant is needed to build Velocity form the source. If you want to do so go to the web page of ant and download the current version (http://ant.apache.org).

- Unzip the archive to a directory of your choice
- Add a new path called `ANT_HOME` to your pathes:
  ```
  set ANT_HOME=c:\ant_directory
  ```
- Add a new path called `JAVA_HOME` to your pathes which points to your Java development kit:
  ```
  set JAVA_HOME=c:\path_to_java\jdk1.5.x
  ```

- Add the bin directory of the ant directory to your path:
  `set PATH=%PATH%;%ANT_MOE&\bin`

If you have any problems please refer to the ant manual

8.3 Ant script for creating web app archives

If you want to use ant to create a web application archive file for tomcat, put this script into the root directory of your webapp under the name `build.xml`.

```
<project name="Guestbook powered by Velocity" default="war"
basedir=".">

  <property name="SRC"      value="${basedir}/WEB-INF/src"/>
  <property name="CLASSES"  value="${basedir}/WEB-INF/classes"/>
  <property name="LIB"      value="${basedir}/WEB-INF/lib"/>

  <!-- Construct compile and javadoc classpath -->
  <path id="classpath">
    <fileset dir="${LIB}">
      <include name="*.jar"/>
    </fileset>
  </path>

  <target name="compile">
    <mkdir dir="${CLASSES}"/>
    <!-- Compile the java code from ${SRC} into ${CLASSES} -->
    <javac srcdir="${SRC}"
           includes="*/**"
           destdir="${CLASSES}">
       <classpath refid="classpath"/>
    </javac>
  </target>

  <target name="war" depends="compile">
    <jar jarfile="${basedir}/../guestbook.war"
         basedir="${basedir}"
         excludes="**/MANIFEST.MF,**/servlet.jar"/>


  </target>

  <target name="clean">
    <!-- remove old class files -->
    <delete dir="${CLASSES}"/>
  </target>

</project>
```

[build.xml]

[20] http://jakarta.apache.org/velocity/

If you have the same folder structure as described in this thesis, just type `ant` in the root directory of your webapp to let ant create a web application archive file.