

A Template Engine with T_EX and Friends

Paulo Roberto Massa Cereda

Abstract

This article presents some insights towards using template engines with T_EX and friends with the help of a command line tool written for the sole purpose of merging data sources and templates.

Sommario

Questo lavoro presenta alcune idee sull'uso dei template con il sistema T_EX, per mezzo di strumenti a riga di comando con lo scopo di unire i dati ai modelli di template.

1 Introduction

Repetition constitutes a fundamental part of one fair day's work for a fair day's pay. Very often, we end up with the very same tasks exhaustively executed. As a means to achieving time and resource optimizations, such repetitive patterns could be factorized and reused whenever needed.

Reports, letters and certificates are instances of document patterns that are repeated over and over. Office suites such as Libre Office and Microsoft Office offer features to merge documents containing special elements (also known as place holders) with external data sources (a database or a spreadsheet), resulting in new documents composed of both structure and data.

Beyond academic usage, T_EX and friends offer an interesting approach for providing document patterns. The advent of the great `datatool` package brought unlimited possibilities for creating documents based on the provided data (represented as a table in the CSV format) without worrying about the structure and layout. For example, consider the following code:

```
letter.tex
\documentclass{article}
\usepackage{datatool}
\DTLloaddb{db}{list.csv}

\begin{document}
\DTLforeach{db}{\name=name,\gift=gift}{
Dear Santa, I've been a good boy
this year, please bring me a \gift.

Merry Christmas from \name!
\newpage}
\end{document}
```

In this example, given a list of people and their respective gifts, it will be easy to generate letters

for Santa Claus (of course, the demand for gifts will increase greatly, but little we can do about it). A sample list in the CSV format is presented as follows:

```
list.csv
name,gift
Enrico,motorbike
David,pineapple pizza
Paulo,rubber duck
```

Based on both document structure and data, three letters to Santa will be generated in a single L^AT_EX run, one letter per page. However, I have a feeling that only two gifts from that list will be effectively delivered¹.

A T_EX-based solution is surely desirable in most cases. In this paper, however, I advocate for an alternative approach: template engines. I must point out that I am an avid user of `datatool` and the package really does wonders. The motivation for taking a different route relies on the fact that template engines are generic enough to be used beyond the T_EX world. Nicola Talbot provides an important note on the `datatool` manual cover:

The `datatool` bundle is provided to help perform repetitive commands, such as mail merging, but since T_EX is designed as a typesetting language, don't expect this bundle to perform as efficiently as custom database systems or a dedicated mathematical or scripting language. If the provided packages take a frustratingly long time to compile your document, use another language to perform your calculations or data manipulation and save the results in a file that can be input into your document.

— The `datatool` user manual

For this paper, the implementation and deployment details of a template engine from a programming language perspective will not be covered. Instead, I plan to keep the discussion solely at the user level. In order to make things easier, the narrative will be assisted by a command line tool, which I forgot I wrote it a couple of years ago! I wish you all a great reading.

1. I am sure David does not like pineapple pizza at all, but I cannot lose this opportunity of shocking Italian readers on his behalf.

2 My kingdom for a tool

Back in 2012, I was working with template engines for a report generation. So far, I only had come up with ad hoc solutions, but that mindset was about to change: what if I had a tool that merges a list of data sources into a template and generates the corresponding output? At first, my primary goal was a tool specifically designed for TEX and friends, but as time went by, the concept was extended to virtual any textual transformation.

Java was my language of choice at that time (and, not surprisingly, it still is), so the programming language in which the tool would be written was settled. But one question was still open:

Which template language is suitable for my needs? Would it cover most scenarios?

It would be desirable to be a template language with an easy syntax and potentially small footprint. Additionally, being Java-friendly would be a great bonus, as integration during the development phase had to be less traumatic as possible.

Being myself a fan of the Apache Foundation, I suddenly found the answer to my inquiries while browsing the list of active projects: I would use Velocity as my template language. This particular project aims at providing a simple yet powerful language known as Velocity Template Language (VTL), as a means to generate any sort of text-based content according to a predefined structure and corresponding data source.

2.1 The Velocity language: a primer

A brief introduction to the Velocity Template Language is advisable. In short, VTL uses references to embed content in a document. Let us see how the traditional `Hello world` example would be:

```
#set( $foo = "Velocity" )
Hello $foo world!
```

Everything that starts with `#` is a statement. In our example, `set` is a directive that sets the value of a reference; `$foo` is a variable and gets the literal string `Velocity`. The output will be:

```
Hello Velocity world!
```

Observe that variables are preceded by `$` and are available after being declared. A variable can be explicitly declared with the `#set` directive or through a map of variables coming from the programming side. References `$foo` and `${foo}` denote the same variable. The second notation is advisable when handling complex operations.

Internally, a variable represents a Java object and thus all corresponding methods are available. For example, consider the following snippet:

```
#set( $foo = "Velocity" )
Hello ${foo.toUpperCase()} world!
```

Since `$foo` holds a string value, we can call the `toUpperCase()` method available in the `String` class from Java. The new output will be:

```
Hello VELOCITY world!
```

When rendering a reference, the value is automatically converted to a string. For instance, if `$foo` holds an object that represents an integer value, Velocity will call its `toString()` method and resolve the object into a string.

Variables can also be arrays, so an element can be accessed through its index, e.g., `$foo[0]` and so on. It is also important to observe that all array references are treated as if they are fixed length lists, meaning that all methods from the `List` class are available.

The `#if` directive allows for text to be included in the output on the conditional that the statement holds true. Consider the following example:

```
#set( $user = "Enrico" )
#if( $user == "Enrico" )
Forza Juve!
#else
Go Palmeiras!
#end
```

The variable `$user` is evaluated to determine whether the condition (that is, if `$user` holds the string value `Enrico`) is true. In our example, the condition is true, then the output will be:

```
Forza Juve!
```

The `#foreach` directive allows looping over the elements of a Java `Vector`, `Hashtable` or an array. For example, if the variable `$stooges` contains the names of the Three Stooges, the following code will iterate through them:

```
#set( $stooges = [ "Moe",
                  "Curly", "Harry" ] )
#foreach ( $stooge in $stooges )
$foreach.count - $stooge
#end
```

Note that there is a special variable `$foreach` holding several details about the particular looping execution. In our example, `$foreach.count` holds the current loop counter. The new output will be:

```
1 - Moe
2 - Curly
3 - Harry
```

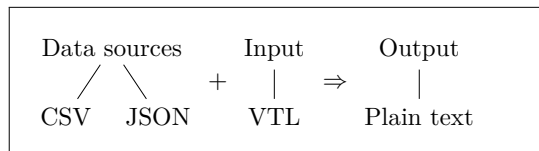
There is much more about VTL, but I believe this introduction covered all the bits we need to proceed in our narrative. More information about Apache Velocity can be found in the project website:

<https://velocity.apache.org>

2.2 The merging tool

Once the template language was chosen, it was a matter of time until I had a merging tool ready to be used in production: **duckity**. The culprit for the command line tool name is no one other than Enrico Gregorio himself, as he gave me the suggestion in the chat room of the TeX community at StackExchange. Figure 1 shows the default output of **duckity** when executed in the terminal without any command line parameters.

Let us start with the basics: **duckity** can merge any template written in VTL with any combination of CSV or JSON data sources. In short:



The data sources information (namely, file and identifier) can be described inside the input file through a specific JSON header, as seen in the following example:

```
{ "datasources": [
  {
    "file"       : "grades.csv",
    "identifier" : "students"
  }
] }
```

Observe that **datasources** is a list, so **duckity** expects as many pairs containing the file name and the identifier as the template requires for a proper merging. The identifier plays an important role in the template, as it will act as a reference to its corresponding file content (for example, **students** will become a variable named **\$students** that refers to the CSV data obtained from **grades.csv**). It is also important to note that the JSON format always encode entries as strings (both keys and values), and **duckity** will parse the file references accordingly based on the file extension.

Once the JSON header is defined, there is a special mark that divides header and template, and

duckity will always expect this mark (followed by a line break) to set up the template before merging:

```
[TEMPLATE]
```

Once the mark is found, **duckity** will consider every text after it to be written in VTL. Similarly, everything before the mark is ignored in the resulting output.

The JSON header can be omitted and replaced by command line flags representing the same information. The following command line flags represent the same information of our JSON header from the last example:

```
-d grades.csv -i students
```

There can be multiple command line flags representing multiple data sources and identifiers, and it is required that they always come in pairs (that is, the same number of identifiers and data sources). Note that the special mark is not needed in this scenario, as the entire file content will be considered as a VTL template.

And that is it, the tool has a very straightforward behavior: get the data sources information, retrieve the corresponding contents, process the template and write the output (that is, the merged template) to a file. In Section 3, we will take a look at some examples.

3 Examples

For our first example, let us consider a table of student grades in the CSV format. The table is presented as follows:

```
grades.csv
Alice,8.0,7.3
Bob,2.2,6.7
Carl,10.0,9.3
David,9.2,10.0
```

Now, let us focus on the table presentation. We need to iterate through this table (a list of rows) and we can use the **#foreach** statement previously seen to achieve this goal:

```
#foreach( $student in $students )
Notes of $student[0]:
- First exam: $student[1]
- Second exam: $student[2]
#end
```

Note that the indices represent the columns (starting from zero) in our table. The template is a plain text, but we can easily make it TeX-aware:

```
[paulo@nineveh ~] $ duckity
  _|      _| o _|_
(_| | _| (_| |< | | _| \ /
                               /

duckity 1.1 - The template helper
Copyright (c) 2012, Paulo Roberto Massa Cereda
All rights reserved.

usage: duckity [file [--datasource D --identifier I]*
        --output file | --help | --version]
-d,--datasource <arg>    set the datasource
-h,--help                print the help message
-i,--identifier <arg>    set the identifier
-o,--output <arg>        set the output file
-v,--version              print the application version
```

FIGURE 1: The default output of `duckity` when executed without any command line parameters.

```
\begin{tabular}{lSS}
\toprule
{Student} & {Exam 1} & {Exam 2} \\
\midrule
#foreach( $st in $students )
$st[0] & $st[1] & $st[2] \\
#end\bottomrule
\end{tabular}
```

Cool, we got our first template done (note that the S column type comes from `siunitx`). Now let us see the entire template (plus header), ready for merging:

```
ex1.tex
{ "datasources": [
  {
    "file"      : "grades.csv",
    "identifier" : "students"
  }
] }
[TEMPLATE]
\documentclass{article}
\usepackage[T1]{fontenc}
\usepackage[utf8]{inputenc}
\usepackage{siunitx}
\usepackage{booktabs}
\begin{document}
\begin{table}[h]
\begin{tabular}{lSS}
\toprule
{Student} & {Exam 1} & {Exam 2} \\
\midrule
#foreach( $st in $students )
$st[0] & $st[1] & $st[2] \\
#end\bottomrule
\end{tabular}
\end{table}
```

```
\end{table}
\end{document}
```

And that is it! Now `duckity` can process `ex1.tex` and merge `grades.csv` with the underlying template. Do not worry with the `$` syntax of VTL, as they will not mess with your mathematical formulas! The execution of `duckity` is as follows, it only suffices to provide the output flag:

```
[paulo@nineveh ~] $ duckity ex1.tex \
-o out1.tex
  _|      _| o _|_
(_| | _| (_| |< | | _| \ /
                               /

Done.
```

Checking `out1.tex`, we can observe the merging was successful, as the grades were correctly typeset! Highlighting just the important bits:

```
\begin{tabular}{lSS}
\toprule
{Student} & {Exam 1} & {Exam 2} \\
\midrule
Alice & 8.0 & 7.3 \\
Bob & 2.2 & 6.7 \\
Carl & 10.0 & 9.3 \\
David & 9.2 & 10.0 \\
\bottomrule
\end{tabular}
```

Now let us add a different scenario: what if we want to display only the students that did not fail on the first exam? The solution is simple, it is a matter of adding a simple check inside the `#foreach` statement:

```
#foreach( $st in $students )
#if ( $math.toFloat($st[1]) > 5.0 )
- $st[0]
#end
#end
```

Observe that we had to make use of a helper object (referenced by `$math`), since the CSV and JSON parsers resolve all entries to strings. A T_EX version of the plain text excerpt is presented as follows:

```
\begin{itemize}
#foreach( $st in $students )
#if ( $math.toFloat($st[1]) >= 5.0 )
\item $st[0]
#end
#end
\end{itemize}
```

Now let us see a complete example based on the previous code. It is important to note that if every student failed on exam 1, the generated T_EX file will contain an error: an empty `itemize` environment! In this case, the solution would be populating an auxiliary variable (through the `#set` directive) with the students who succeeded on exam 1 and then check the length of this variable. That will be left as an exercise to the reader.

```
ex2.tex
{ "datasources": [
  {
    "file"      : "grades.csv",
    "identifier" : "students"
  }
] }
[TEMPLATE]
\documentclass{article}
\usepackage[T1]{fontenc}
\usepackage[utf8]{inputenc}
\begin{document}
\begin{itemize}
#foreach( $st in $students )
#if ( $math.toFloat($st[1]) >= 5.0 )
\item $st[0]
#end
#end
\end{itemize}
\end{document}
```

After running `duckity` on `ex2.tex`, the list of students that succeeded on exam 1 is correctly generated:

```
\begin{itemize}
\item Alice
\item Carl
```

```
\item David
\end{itemize}
```

If `ex2.tex` did not have the JSON header (nor the special template mark), the command line invocation would be:

```
[paulo@nineveh ~] $ duckity ex2.tex \
-d grades.csv -i students -o out2.tex
_| _| _| _| _| _|
(_| | _| ( _| < | | _| \ /
/
Done.
```

So far, our examples contemplated data sources in the CSV format. For the sake of completeness, consider the following file specified in the JSON format (note that the square brackets denote a list of elements):

```
person.json
{
  "name"      : "Paulo",
  "location"  : "Brazil",
  "interests" : [ "ducks", "Pringles" ]
}
```

While elements from CSV references are indexed by positional values (that is, integers representing the columns starting from zero), elements from JSON references are indexed by their corresponding identifiers (also known as keys). If the data source identifier is `person` (thus the reference will be `$person`), in order to access the `name` element, it suffices to invoke `$person.name` in VTL. The following example makes use of such keys in order to build a sentence:

```
ex3.tex
{ "datasources": [
  {
    "file"      : "person.json",
    "identifier" : "person"
  }
] }
[TEMPLATE]
My name is $person.name, I am from
$person.location and my interests
include #foreach( $i in
$person.interests )$i#if(
$foreach.hasNext() )#if(
$foreach.count ==
$person.interests.size() - 1)
and #else, #end#end#end.
\bye
```

Observe the presence of `$foreach.hasNext()`, which indicates if the loop still have elements to iterate, plus an evaluation on the list size. This

is a trick to display elements from a list, so the last element is preceded by the word **and** and not by a comma (the result looks more natural and visually appealing). Velocity statements can be nested in order to generate an elaborated output (also note that, in some cases, spaces do matter!). The generated TEX code is presented as follows:

```

out3.tex
My name is Paulo, I am from Brazil
and my interests include ducks
and Pringles.
\bye

```

We can have multiple data sources in the same template, provided that they have different identifiers. The next example covers this scenario:

```

ex4.tex
{ "datasources": [
  {
    "file"      : "grades.csv",
    "identifier" : "students"
  },
  {
    "file"      : "person.json",
    "identifier" : "person"
  }
] }
[TEMPLATE]
I am $person.name and my friend
is $students[0][0]!
\bye

```

The first row from the CSV file is directly accessed through `$students[0]` (no need for a `#foreach` statement here), and the first index gives the student name. The output is as follows:

```

out4.tex
I am Paulo and my friend is Alice!
\bye

```

It is worth mentioning that version 4.0 of **arara** has support for VTL through a **velocity** rule. The corresponding directive takes the following parameters:

- **input**: optional, it refers to the input file containing the VTL template. If not specified, **arara** will assume the current file being processed holds the template.
- **output**: mandatory, it refers to the output file, generated from the template merging.
- **context**: mandatory, it contains a map that represents all references to be included in the template. It is very similar to a JSON file, but only the keys are expected to be strings, values can be anything that maps to a Java object (from primitives to complex data structures).

The next example illustrates how **arara** handles templates using VTL. Observe that, contrary to **duckity**, there is only one data source, that is, the map represented by the **context** parameter (similar to JSON syntax, but it is still YAML). Map entries are directly mapped inside the template with no prefix.

```

ex5.tex
% arara: velocity: {
% arara: --> output: out5.tex,
% arara: --> context: { name: "Paulo" }
% arara: --> }
Hello, my name is $name.
\bye

```

Now, it is just a matter of invoking **arara** on `ex5.tex` and the merged file `out5.tex` (set as output in the **velocity** directive) will be automatically generated:

```
[paulo@nineveh ~] $ arara ex5.tex
```

The resulting file `out5.tex` is as follows. Observe that the directives from `ex5.tex` were transposed to the merged file as well, as they are also part of the VTL template:

```

out5.tex
% arara: velocity: {
% arara: --> output: out5.tex,
% arara: --> context: { name: "Paulo" }
% arara: --> }
Hello, my name is Paulo.
\bye

```

We can exploit the use of conditionals in order to make **arara** process those two files differently, even if directives are transposed. Consider the new example presented as follows:

```

ex6.tex
% arara: velocity: {
% arara: --> output: out6.tex,
% arara: --> context: { name: "Paulo" }
% arara: --> } if currentFile().
% arara: --> getName() == "ex6.tex"
% arara: pdftex if currentFile().
% arara: --> getName() == "out6.tex"
Hello, my name is $name.
\bye

```

The new **velocity** directive has a conditional now, so **arara** will only execute the corresponding rule if, and only if, the current file being processed is `ex6.tex`. Otherwise, the tool will simply ignore this directive and move on. We also included a **pdftex** directive using the same trick, now checking if the current file is the proper one

```
[paulo@nineveh ~] $ arara ex7.tex
```

```
/ _ ' | , _ / _ ' | , _ / _ ' |  
| ( _ | | | ( _ | | | ( _ |  
\ _ , - | _ \ _ , - | _ \ _ , -
```

Processing 'ex7.tex' (size: 178 bytes, last modified: 09/17/2017
13:08:39), please wait.

(Velocity) The Velocity engine SUCCESS
(PDFTeX) PDFTeX engine SUCCESS

Total: 0.37 seconds

FIGURE 2: Execution of `arara` in the command line.

(i.e., `out6.tex`). In this scenario, two independent runs of `arara` are required, one for each file.

However, we can do better than this. Let us forget about conditionals and use the `files` parameter on our `pdfTeX` directive and get our result with a single run. Consider the following example:

```

_____ ex7.tex _____
% arara: velocity: {
% arara: --> output: out7.tex,
% arara: --> context: { name: "Paulo" }
% arara: --> }
% arara: pdftex: {
% arara: --> files: [ out7.tex ]
% arara: --> }
Hello, my name is $name.
\bye

```

A nice improvement indeed! On a single run of `arara`, we get `out7.tex` from our template and compile it using the `pdftex` rule. The output of this particular run is shown in Figure 2.

It is worth mentioning that **arara** is not exactly the best way of handling templates regarding data sources. For this matter, **duckity** is far superior, as it offers two data source formats, namely CSV and JSON. However, both tools are powered by VTL, which is a fantastic template language, regardless of which back-end is used. Your mileage may vary. For **arara**, I might include support for data sources in the future, so please let me know if this feature would be interesting. For now, **duckity** is the best option if you want to take a direct approach to generating any sort of text-based content according to a predefined structure and corresponding data source. For a general approach, **arara** might help.

Unfortunately, **arara** 4.0 is not officially released yet, as the user manual is being written at the moment. Hopefully, the tool will be available soon. For now, you can easily build a development version, provided that you have an instance of a Java Development Kit (also known as JDK) and a tool

named Maven (from the Apache Foundation). The source code is hosted at GitHub:

<https://github.com/cereda/arara>

Hopefully, these examples covered most of the common scenarios when handling data sources with templates. It is highly advisable to take a closer look at the VTL guide in order to learn the advanced features that particular scripting language has to offer.

4 Final remarks

This article discussed the use of template engines as an alternative approach to \TeX -based solutions on reading data sources. Template engines are generic enough in order to cover different domains of application. The discussion was kept at the level user, with the assistance of a command line tool named **duckity**.

The command line tool is open source and released under the New BSD license. Java binaries, as well as the source code, are available in the project repository at GitHub:

<https://github.com/cereda/duckity>

Happy T_FXing with templates!

Acknowledgments

The author wishes to thank Claudio Beccari, Enrico Gregorio, Carla Maggi and all friends from GUT for the opportunity of writing this humble article for *ArXiv*.

▷ Paulo Roberto Massa Cereda
Università di San Paolo, Brasile
paulo dot cereda at usp dot br