

A gentle introduction to \TeX plate: a document structure creation tool

Island of \TeX

Version 1.0.4 – July 27, 2021

Abstract

As the title implies, this document dares to be a gentle introduction to a rather unusual application, at least from our ordinary \TeX workflow: a tool for creating document structures based on templates. The application name is a word play on \TeX and *template*, so the purpose seems quite obvious: we want to provide an easy and straightforward framework for reducing the typical code boilerplate when writing \TeX documents. Also note that one can easily extrapolate the use beyond articles and theses: the application is powerful enough to generate *any* text-based structure, given that a corresponding template exists.

1 Introduction

A typical \TeX document usually contains a common preamble, the same code boilerplate we hold dear to keep our writing workflow efficient and, to an extent, satisfy our darkest typographic additions. A coherent preamble might significantly reduce the odds of having issues when compiling your document, but at the end of the day, it is always your fault when

something terribly wrong happens in your code¹. Anyway, consider the following \TeX document²:

```
\documentclass{article}

\usepackage[T1]{fontenc}
\usepackage[utf8]{inputenc}

\begin{document}
This is the law of the jungle:
Nutella pizza is good.
\end{document}
```

Modulo actual content, this structure is probably recurring on the majority of documents using the classical engines. Ultimately, we aim at providing a collection of text-based structures through a comprehensive command line interface³, so as a means to reduce the code boilerplate for typical documents.

¹This statement is suspiciously Carlislean. Also, since we are on the subject, make sure to never read documentation, as bad things happen.

²Apologies in advance to every pizza connoisseur out there (specially Italians) for the provocative statement in the document. That said, one must accept the truth that Nutella pizza *is* good.

³Granted, command line usage is not exactly as friendly, intuitive and comprehensive as one could have ever hoped. We have plans for a graphical interface in the near future, so stay tuned.

2 Concepts

Before introducing the command line application itself, we need to establish a common foundation and formally present a couple of concepts in which our tool is built. Consider this section as a primer to such elements.

2.1 Template

A template is basically a textual structure with certain entry points, in a sort of *fill in the blanks* fashion. For $\text{T}_{\text{E}}\text{X}$ plate, we rely on a template language provided by the Apache Foundation named VTL (or Velocity, for short), in its latest 2.0 specification, as of the writing of this manual.

Disclaimer

We do not intend to cover the language specification on its full glory in this manual. However, we might pinpoint and highlight some characteristics whenever relevant to the context.

For our purposes, the basic structure of any text-based document constitutes a template. For instance, the $\text{T}_{\text{E}}\text{X}$ document originally presented in Section 1, modulo actual content, can certainly act as a verbatim template:

```
\documentclass{article}

\usepackage[T1]{fontenc}
\usepackage[utf8]{inputenc}

\begin{document}

\end{document}
```

Note, however, that this template, as it is, does not hold any variables, con-

ditional flows or sophisticated language constructs. The next concepts will build upon variables in the text, as a means to enhance the template expressiveness.

2.2 Map

A map is a collection of variables and values instantiated at run time and available within the template context. Since these values are specified in the command line, $\text{T}_{\text{E}}\text{X}$ plate initially casts them as strings. It is worth noting, however, that these string values can be converted to arbitrary types later on through special functions named handlers in the template specification (see Section 4.3 for further details). Regarding syntax, a variable `bar` in the command line is referenced as a variable `$bar` inside the template context. Note that all methods from the corresponding type class are available in the variable scope, since a variable acts like a proper Java object. However, note that it is highly advisable to verify whether a variable is defined before invoking any method. For instance, consider the following excerpt:

```
#if($name)
Hello, my name is $name.
#else
Hello, my name is John.
#end
```

In this excerpt, observe that `$name` is a variable. If such variable is properly specified in the command line, it will be set accordingly in the template context and the corresponding conditional branch is handled. Please refer to the VTL specification for further details.

Disclaimer

Ultimately, Java objects will rely on the `toString()` method when the template is actually rendered, so there is no need to worry about converting arbitrary values back to strings.

The map is specified at run time, in the command line. It is important to observe that a variable might be marked as mandatory by the template (more on that later, when discussing the template specification in Chapter 4), so the absence of such variable in the map will trigger an exception and the tool will prematurely end.

3 Command line

Our tool is a typical command line application, so we need to invoke it by simply typing `texplate` in the terminal:

```
$ texplate
```

Provided that `TeXplate` is properly available in the underlying operating system, we will get a lovely, colourful output in our terminal, along with a couple of scary messages telling that some required options are missing. Our tool provides four options, described as follows. Strictly speaking, there is only one mandatory option. The remainder might be optional, depending on certain scenarios.

3.1 Output

Referred as either `-o` (short representation) or `--output` (long representation), this option holds the output file in which the chosen template will be effectively written.

The following examples are valid entries to this option:

```
-o mydoc.tex  
-o=mydoc.tex  
--output mydoc.tex  
--output=mydoc.tex
```

The name is arbitrary, so you can use anything to your heart's desires. Keep in mind, however, that an existing file will be mercilessly overwritten. Also, make sure the path has the proper permissions for writing the output file.

3.2 Template

Referred as either `-t` (short representation) or `--template` (long representation), this option holds the reference to the template to be potentially merged with data and properly generated. The basic syntax is detailed as follows (note that `$article` refers to the template identifier):

```
-t article  
-t=article  
--template article  
--template=article
```

The provided string should match the template identifier (or file name), otherwise `TeXplate` will raise an error complaining about the missing reference. The template identifier will be discussed later on, in Section 4.1. For a discussion on file name lookup, see Section 4.2.

3.3 Map

Referred as either `-m` (short representation) or `--map` (long representation), this option holds a map entry, defined as a key=value ordered pair (mind the `=` symbol used as

entry separator). This option can be used multiple times. The following examples are valid entries to this option (note that `foo` and `bar` refer to an arbitrary map entry key and value, respectively):

```
-m foo=bar
-m=foo=bar
--map foo=bar
--map=foo=bar
```

The map entry denotes a variable in the template context, where the key is the variable name and the value is set to be the initial state of such variable. As the `--map` option can be used multiple times, consider the following scenario, where the map entry key is repeated:

```
--map foo=1 --map foo=2 --map foo=3
```

The final value associated to `foo` is set to be the rightmost occurrence of the corresponding pair in the command line.

3.4 Configuration

`TeXplate` also offers a configuration file in which the tool can read template data, for automation purposes. The configuration file is written in a text-based format named TOML⁴ and can hold at most two sections: the template name as a string (with the same behaviour of `--template` in the command line) and the contextual data map, with keys and associated values. A configuration file can have at least one of such sections, but never be empty. Consider the following configuration file:

⁴According to Wikipedia, TOML is a configuration file format intended to be easy to read and write due to obvious semantics and designed to map unambiguously to a dictionary. The syntax consists of entry pairs, sections and comments.

```
template = "article"

[map]
country = "Germany"
year = 2020
articles = [ "Die", "Der", "Das" ]
```

Observe that, contrary to the `--map` option in the command line, the `[map]` section in the configuration file accepts values other than strings (e.g., integers and lists of strings).

Disclaimer

Please keep in mind that map entries in the configuration file are not subjected to handlers (when available), as their command line counterparts are.

Command line options have priority over values originated from the configuration file. Given the previous configuration file, consider the following scenario:

```
$ texplate -c config.toml -t book
```

Even though `article` was the template set in the configuration, the `--template` option has higher priority and thus the template is set to `book`, as it is defined in the command line. The same logic is applied to map entries.

Disclaimer

Observe that, if a configuration file is being used and has no template key, the `--template` option automatically becomes mandatory.

Note that `TeXplate` has support for just one configuration file at run time. Support for multiple configurations might be added in the near future.

4 Template specification

\TeX plate uses predefined templates to generate text-based structures. Similar to the configuration structure presented in Section 3.4, the tool relies on text-based files written in TOML for holding the template specifications.

4.1 Naming scheme

For starters, a template has an identifier, a name that logically represents the text-based structure to be generated. Observe that the name corresponds to the reference used later on in the `--template` option in the command line. A template file name has to match this identifier, so e.g., the `article` reference is automatically linked to the `article.toml` file which contains the template specification.

Disclaimer

Observe that \TeX plate's lookup system is case-sensitive, so make sure to always reference the correct name.

Although there are no hard limitations on a template name, it is advisable to keep it short and concise, with no spaces whatsoever⁵. As a consequence, potential issues with extended characters in the Unicode range and command line escaping are avoided, and thus our beloved tool might just work as expected.

4.2 Directory lookup

\TeX plate employs a lookup system that basically searches two locations for template

⁵Another Carlislean statement would remind us that people who put spaces in their file names deserve no sympathy. At all.

files (specifications written in the TOML format), in order of priority:

1. `~/.texplate/templates` which refers to a path structure from the user home directory.
2. The application's resources which refers to a files within the JAR file. You can use a ZIP viewer to look at the templates there.

For instance, as a means to illustrate the concept of directory lookup, consider the following command line command:

```
$ texplate -t article -o mydoc.tex
```

\TeX plate will look for a template file named `article.toml` in the path structure from the user home directory, as seen in #1. If the file is found, the search ends and the tool proceeds to the template generation. Otherwise, \TeX plate will attempt to look in its own template folder for a match. If found, the tool will use this file reference. If the template file is not found in any of the two locations, the tool reports the issue and prematurely ends.

if the argument provided to `--template` holds no `.toml` extension, \TeX plate will handle it as a string and fallback to the default template lookup, previously described. Otherwise, the tool will handle it as a path reference and read the template from it. Consider, for instance, these scenarios:

```
--template article
```

In this scenario, \TeX plate searches for `article.toml` in the default directories (home and application, in that order).

```
--template article.toml
```

In this scenario, `TeXplate` searches for `article.toml` in the working (current) directory.

```
--template dir/article.toml
```

In this scenario, `TeXplate` searches for `article.toml` in the `dir/` subdirectory. Note that absolute and relative paths are supported.

4.3 Handlers

A map entry, when obtained from the command line through the `--map` option, holds a string value. `TeXplate` provides a straightforward way to convert string values to arbitrary types through special functions named handlers in the template specification. The following handlers are available for type conversion:

- `to-csv-list`: this handler, as the name implies, converts a string to a list of comma-separated values. The implementation attempts to respect quoted elements, when available.
- `to-boolean`: this handler converts a string to a boolean value, based on the following patterns: `true` and `false`, `1` and `0`, and `yes` or `no`.
- `to-string-list-from-file`: this handler, as the name implies, converts a string into a file reference and attempts to read its contents into a list of lines. If the file reference does not exist or is somehow invalid, an empty list is assigned instead.

When type conversion is needed in the template logic, handlers are assigned to

map keys in the corresponding specification, detailed later on, in Section 4.4.

Disclaimer

Handlers are internal functions provided by `TeXplate` that map `String` values to arbitrary types.

When referencing handlers, make sure to write the correct name in the template specification, so the lookup and subsequent call work as expected.

4.4 Template structure

The template structure holds at most two sections: the template itself and an optional `[handlers]` section holding the handlers to be applied to certain map entries. The following keys are required for the first section (i.e., the template definition):

- `name`: this key holds a string value that denotes the template name. Observe that `TeXplate` requires this key to hold the exact value of the template identifier, as described in Section 4.1.

```
name = "mytemplate"
```

- `description`: this key holds a string value that provides a short yet meaningful description for the template being written. It is usually suitable to use the `"""` (triple double quotes) notation for this key, as it allows long strings to span multiple lines (and ignore line breaks).

```
description = """  
A simple template used to  
illustrate the template  
structure.
```

```
''''
```

- **authors:** this key, as the name implies, holds a list of strings denoting the template authors, for obvious blaming purposes, in case anything goes terribly wrong⁶.

```
authors = [ "Alice", "Bob" ]
```

- **requirements:** this key holds a list of strings denoting potential requirements for the template. For instance, the template might require some variables to be specified at run time, either through the command line or configuration file:

```
requirements = [ "names" ]
```

If the requirements are not complied, \TeX plate will report the issue and prematurely end. When the template imposes no requirements at all, make sure to provide an empty list:

```
requirements = [ ]
```

- **document:** this key, as the name suggests, holds the actual text-based document, written using the VTL specification (or any subset of it). It is usually suitable to use the ''' (triple single quotes) notation for this key, as it allows raw strings (i.e., literal reproduction with no need of escaping characters) and multiple lines while respecting breaks.

```
document = '''  
These are the names:
```

```
\begin{itemize}  
#foreach ($name in $names)  
  \item $name  
#end  
\end{itemize}  
'''
```

The [handlers] section, as the name indicates, holds potential handler references to be applied to certain map entries. These references are set as strings. For instance, \$names is actually a list of strings, so a type conversion has to be employed:

```
[handlers]  
names = "to-csv-list"
```

Observe that, if there is no need for handlers in the template structure, this section might safely be omitted.

5 A complete example

Now that all concepts are formally introduced in the previous chapters, it is time to glue everything together. The following steps cover the basics from creating a template to merging data into it:

1. First and foremost, let us create a template named mytemplate, inspired on the previous content presented in this user manual. The naming scheme, as well as the directory structure, must be respected, so open your favourite editor⁷ and create the following file:

⁶Of course, it is important to observe that an error, exception or issue is definitely *not our fault*.

⁷Your favourite editor should be vim, of course. Anything else is simply wrong and unacceptable.

```
$ mkdir -p ~/.texplate/templates/  
$ cd ~/.texplate/templates/  
$ vim myarticle.toml
```

2. Now, add the following content to the newly created mytemplate.toml file:

```
name = "mytemplate"  
description = ""  
A simple template used to  
illustrate the template  
structure.  
""  
authors = [ "Alice", "Bob" ]  
requirements = [ "names" ]  
document = '''  
These are the names:  
  
\begin{itemize}  
#foreach ($name in $names)  
  \item $name  
#end  
\end{itemize}  
'''  
  
[handlers]  
names = "to-csv-list"
```

3. Done, the template is ready to be used! Now, simply call `TeXplate` in the command line and provide the required map entry, as seen as follows:

```
$ texplate -t mytemplate -m  
names=John,Jane -o list.tex
```

The output (i.e., the template merged with the provided data) will be written to a text-based file named `list.tex`.

4. When running `TeXplate`, this is the expected output to be displayed in the command line (note that the layout is slightly modified due to space constraints in this user manual):

```
TeXplate 1.0.4, a document  
structure creation tool  
Copyright (c) 2020, Island of TeX  
All rights reserved.
```

```
Configuration file  
mode disabled ..... [ DONE ]  
Entering full  
command line mode ..... [ DONE ]
```

Please, wait...

```
Obtaining reference ... [ DONE ]  
Composing template .... [ DONE ]  
Validating data ..... [ DONE ]  
Merging template  
and data ..... [ DONE ]
```

```
Done! Enjoy your template!  
Written: 78 B
```

5. Great, everything worked as expected! Now, let us check the contents of the newly generated `list.tex` file:

```
$ cat list.tex  
These are the names:  
  
\begin{itemize}  
  \item John  
  \item Jane  
\end{itemize}
```

As seen in the previous output, the template and provided data were successfully merged.

6. Let us reproduce the same output with a configuration file. Observe, however, that handlers are not applied to map entries in the configuration file, so we have to explicitly set names as a list of strings. Let us create a configuration file named `config.toml` in the current directory:


```
$ vim config.toml
```

Now, add the following content to the newly created `config.toml` file:

```
template = "mytemplate"

[map]
names = [ "John", "Jane" ]
```

- The new call to `TEXplate`, given the aforementioned configuration file, is slightly different than the previous one (step 3), so write the following entry in the command line:

```
$ texplate -c config.toml -o
  list.tex
```

- The output is pretty much the same obtained in the previous call (step 4), except for the following lines acknowledging the configuration file mode:

```
Checking
configuration ..... [ DONE ]
Adjusting variables
from file ..... [ DONE ]
```

The final lines from `TEXplate` indicate everything worked as expected:

```
Done! Enjoy your template!
Written: 78 B
```

- The generated `list.tex` file has exactly the same contents as illustrated in step 5. It is important to remember that an existing output file will be mercilessly overwritten.

The previous steps described how `TEXplate` works, from creating a template to merging data into it, as a means to generating a text-based document. Chapter 6

presents text-based templates shipped with our tool, as well as the available variables in the document context.

6 Included templates

`TEXplate` ships with the following text-based templates, automatically available from the application directory:

article

This reference holds a simple template for the default article class, with support for new engines (with `fontspec` fallback), babel languages, geometry options, generic packages, and `TikZ` and corresponding libraries. There are no requirements for this template. Available variables are described as follows:

- `xetex`: boolean value, changes the default behaviour to accommodate the `XeTEX` engine. Typical `fontenc` and `inputenc` packages are replaced by `fontspec` when this variable holds true (semantically equivalent).

```
-m xetex=true
```

- `luatex`: boolean value, changes the default behaviour to accommodate the `LuaTEX` engine. Typical `fontenc` and `inputenc` packages are replaced by `fontspec` when this variable holds true (semantically equivalent).

```
-m luatex=true
```

- `options`: string value, holds the document class options, when applied.

```
-m options=12pt,a4paper
```

- **babel**: string value, holds a sequence of languages supported by the babel package. Keep in mind that the last entry in the sequence is set to be the default language.

```
-m babel=english,italian
```

- **geometry**: string value, holds the options for the geometry package. It is important to note that order matters.

```
-m geometry=margins=2cm
```

- **packages**: list of strings, holds a list of packages to be included in the document preamble, in the specified order.

```
-m packages=longtable,array
```

- **tikz**: boolean value, checks whether the document should include support for TikZ in the preamble.

```
-m tikz=true
```

- **libraries**: list of strings, holds a list of TikZ libraries to be included in the document preamble, in the specified order. It is important to observe that this variable has no effect whatsoever if the tikz variable is either not set or does not hold true.

```
-m libraries=automata,positioning
```

standalone

This reference holds a simple template for the standalone class, with support for new engines (with fontspec fallback), babel languages, geometry options, generic packages, and TikZ and corresponding

libraries. There are no requirements for this template. Available variables are described as follows:

- **xetex**: boolean value, changes the default behaviour to accommodate the XeTeX engine. Typical fontenc and inputenc packages are replaced by fontspec when this variable holds true (semantically equivalent).

```
-m xetex=true
```

- **luatex**: boolean value, changes the default behaviour to accommodate the LuaTeX engine. Typical fontenc and inputenc packages are replaced by fontspec when this variable holds true (semantically equivalent).

```
-m luatex=true
```

- **options**: string value, holds the document class options, when applied.

```
-m options=12pt,a4paper
```

- **babel**: string value, holds a sequence of languages supported by the babel package. Keep in mind that the last entry in the sequence is set to be the default language.

```
-m babel=english,italian
```

- **geometry**: string value, holds the options for the geometry package. It is important to note that order matters.

```
-m geometry=margins=2cm
```

- **packages**: list of strings, holds a list of packages to be included in the document preamble, in the specified order.

```
-m packages=longtable,array
```

- `tikz`: boolean value, checks whether the document should include support for TikZ in the preamble.

```
-m tikz=true
```

- `libraries`: list of strings, holds a list of TikZ libraries to be included in the document preamble, in the specified order. It is important to observe that this variable has no effect whatsoever if the `tikz` variable is either not set or does not hold true.

```
-m libraries=automata,positioning
```

7 Final remarks

This document aimed at being a gentle introduction to \TeX plate, a tool for creating document structures based on templates. We invite you to contribute to this project by submitting feature requests, issues and new templates:

```
gitlab.com/islandoftex/texplate
```

Happy \TeX ing with \TeX plate!

License

\TeX plate is licensed under the New BSD License. Please note that the New BSD License has been verified as a GPL-compatible free software license by the Free Software Foundation, and has been vetted as an open source license by the Open Source Initiative.

Changelog

1.0.4 (current)

Fixed

- Resolve outdated dependency with vulnerability.

1.0.3 (2020-08-07)

Added

- New `to-string-list-from-file` handler added. This handler converts the map entry value into a File reference and attempts to read its contents into a list of lines. If the file reference does not exist or is invalid (e.g, not a file or with insufficient permissions), an empty list is assigned instead.

Changed

- \TeX plate template path resolution has changed. Use `-t article` to get the default article template. If you want to specify a file instead, use `-t article.toml` or `-t /my/path/to/file.toml`. Relative paths will be resolved against the working directory.

- Updated dependencies.

1.0.2 (2020-02-02)

Fixed

- \TeX plate now finds its templates even on Windows.

Changed

- TeXplate now finishes its transition to Kotlin. We did not change any functionality in the course of this change.
- Templates are now provided as resources from the JAR instead of a separate folder on the hard drive.

1.0.1 (2020-01-17)

Changed

- TeXplate will now distribute only non-generic template file names. In the system's template directory, we search for `texplate-<name>.toml` as well.

1.0.0 (2020-01-15)

Added

- Base functionality and default templates.
- User manual.