Templater user manual

26.04.2016

Contents

About Templater

Templater is a minimal reporting library for Java and .NET which is used to generate reports based on DOCX and XLSX templates. Official site is: http://templater.info/

Templater takes a different approach with its minimal API than most reporting libraries, while remaining useful for 95%[1] reporting scenarios which are based on templates. Templater is not a generic reporting solution -- it does not support programmatic document manipulation except that based on removing, replacing and duplicating of existing tags in a document. It does not have an insert picture method - but if a tag exists in the document you can replace that tag with an Image, thus inserting a picture into the document. To be useful, Templater supports resizing of contexts – parts of document which are meant to be duplicated, such as rows in a table, list item, entire documents, excel sheets, named ranges in sheets and similar object types. Context can be nested which allows for generation of very complicated reports. Templater recognizes two tag formats: [[tag]] and {{tag}}.

Minimal API

Templater API has four parts:

- configuring plugins/initializing factory
- opening and closing/flushing document
- processing objects
- low level tag manipulation

JavaDoc is available at website: http://templater.info/apidocs/
.NET documentation is available in the XML provided with the DLL, but the API is almost identical to the Java API.

Various examples are available at website: http://templater.info/downloads

Supported documents

Templater works with Office Open XML format for Word and Excel. Supported extensions are:

- DOCX – standard Word 2007 format
- DOCM – marco-enabled Word 2007 format
- XLSX – standard Excel 2007 format
- XLSXM – macro-enabled Excel 2007 format
- TXT/CSV – text formats

Only primitive support is available for TXT files since they cannot really have any context except a row. This makes it useful only for most primitive use cases, while DOCX and XLSX support is feature rich.

---

1 yes, a made up statistical number

Configuring plugins

Configuration API supports default plugin initialization or initialization with external plugins. Same plugin interface is shared inside Templater, so users should be able to extend it in various ways. While plugins implemented by Templater are stateless, external plugins don't need to be. This can be used to implement more complicated features.
Example of plugin configuration looks like:

```
var factory =
       Configuration.Builder
       .Include<Questionnaire>(Process)
       .Include(FormatDate)
       .Build();
```

Where referenced functions are:

```
static bool Process(string prefix, ITemplater templater, Questionnaire q)
.
static object FormatDate(object value, string metadata)
```

Opening and closing documents

Opening API can work on documents in place if a filename is passed to it, but most scenarios will involve passing input and output streams with the appropriate extensions. While Templater could support auto-detection of extensions, there are currently no plans for such a feature.

In Java, input stream can be `FileInputStream` and output stream can be `FileOutputStream`. In .NET, the same stream can be used for both input and output, but due to LOH[2] (Large object heap issues) it's better to provide it with specialized streams such as `HttpResponse.OutputStream` and write directly into it. Internally, Templater will use an internal temporary stream, which can cause slightly increased memory usage for large documents[3], but will have better performance overall. Templater internal stream avoids LOH issues in .NET.

After processing is done, output can be populated, which is performed by the `flush` operation in Java or the `Dispose` operation in .NET.

While best practice in .NET is to avoid doing much work in `Dispose`[4], this allows for the API to remain small and consistent. An example usage is

```
using(var doc = Templater.Configuration.Factory.Open(input, output, "docx"))
{
      //processing
} //implicit call to Dispose
```

---

2 info about LOH can be found at https://connect.microsoft.com/VisualStudio/feedback/details/521147

3 XML parsers use most of the memory (both in JVM and .NET)
4 this is based on .NET framework guidelines which suggest having a Close method alongside Dispose method

A good practice is to cache the *IDocumentFactory* instance in a shared static field and reuse it for processing. While *ITemplateDocument* and *ITemplater* are not thread safe, *IDocumentFactory.Open* always provides a new instance.

Processing objects

ITemplateDocument consists from a single process method and access to a low level API. Complete interface in .NET is:

```
/// <summary>
/// Template document with analyzed tags.
/// Modifies document in memory and saves changes on dispose.
/// </summary>
public interface ITemplateDocument : IDisposable
{
  /// <summary>
  /// Provides access to low level API.
  /// </summary>
  ITemplater Templater { get; }
  /// <summary>
  /// Modifies document according to rules for specified data type.
  /// Tags are resolved from properties, fields, methods, column names, etc.
  /// </summary>
  /// <typeparam name="T">Rules will be chosen from type</typeparam>
  /// <param name="data">Data for modifying tags</param>
  /// <returns>Itself</returns>
  ITemplateDocument Process<T>(T data);
}
```

There seems to be a bit of confusion on how exactly the `process` method is to be used. Efficient use of Templater consist from passing appropriate objects to process method and letting Templater find matching tags and contexts in the provided template. While `process(object)` is not really an intention revealing API, it allows for the same API across different documents and extensions without API changes.

Behind the `process` method there are various plugins available and the plugins are probed in registration order. Custom plugins are registered before default ones, so registration order is important.

For example in .NET, processing plugins are:

- data reader
- data row
- data set
- data table
- dictionary enumerable
- dictionary
- enumerable
- object

Java has only a subset of these plugins, since some objects don't really exist in Java

(DataSet, DataTable and DataRow).

Each plugin has its own specialized behavior, but in the end, except for passing objects to another plugin with the current navigation prefix, all they can utilize is the low level API available to the developer. If custom plugins are registered they can be invoked during probing (in object processor probing is also performed, so that the best processor is used for found value).

What this translates to in API usage is that if we have objects such as

```
class Receipt
{
	public String name;
	public List<Ingredient> ingredients;
}

class Ingredient
{
	public String name;
	public Decimal quantity;
	public Map<String, String> info;
}
```

and pass a collection of `Receipt` instances to process method, enumerable processor will be chosen and matched with the most appropriate context in the document. Enumerable processor works by duplicating detected context and invoking object processor on each part of the duplicated document.

It is interesting that if you send just an instance of `Receipt` to the process method, object processor will be chosen instead of enumerable, but same template can be used as before. The only difference is that the context will not be duplicated as it would be for a collection of `Receipts`.

When object processor is processing an instance of `Receipt` it will match the `[[name]]` tag with `String name` field from `Receipt` instance. Currently, Java bean standard is not supported, but you can use methods without arguments to read a value of an object, so in this case if `String getName()` method exists, `[[getName]]` tag can be matched.

During processing of the `Receipt` instance, object processor will find the `ingredients` field using reflection. The same cycle will begin, but now with the **"`Ingredients.`"** prefix. This means that the inner context containing `[[ingredients.name]]`, `[[ingredients.quantity]]` and `[[ingredients.info]]` will be detected and resized/duplicated based on numbers of items found in ingredients field. Each item in collection will be processed using the best matching processor –q in this case the object processor again. During processing of the `info` field, another best processor will be chosen, in this case a dictionary processor, with the **"`ingredients.info.`"** prefix. This means that if dictionary instance contains an "`author`" key `[[ingredients.info.author]]` tag can be matched.

An important distinction during processing is that if Templater finds `[[ingredients.size]]` tag, which corresponds to the `size()` method available on the `List` class, it will not be

considered part of the `Ingredient` class.

Specialized data types

Some types have special behavior, as in they will not just replace text with a string, but can perform some other actions. An example of this is the `Date` type in Java and `DateTime` type in .NET which, when used in Excel, will be replaced with an Excel representation of that value (except for dates before 1900-01-01 which are added as string values, since Excel doesn't recognize those dates).

Image is another example of a specialized data type which will replace a tag with the provided image.

Jagged arrays are another example of specialized data type which can be used to resize a table in both directions, not just rows (as with the standard collection processor). This feature is called dynamic resize and is available for few other appropriate data types (eg. two dimensional arrays, `DataTable`, `Icon` in .NET).

In Word XML is recognized as special data type (XElement in .NET and org.w3c.Element in JVM). In case when XML or XML collection is detected, they will be inserted into Word as-is which opens up a point for low level DOCX manipulation. This can be used for features such as injecting HTML into document, although it requires deep knowledge of the DOCX format.

While Templater can't create graphs or pivots, it can populate data sources which are used for graphs, pie charts and pivots. This makes it really easy to build visually impressive graphs and pie charts in Excel and have data source filled with Templater to produce a complete report.

Low level tag manipulation

When Templater scans a template during initialization, it collects all the tags which are available in the `String[]` tags method/property. This collection only provides distinct tag names, but all metadata can be asked for each tag. Metadata is information contained in the template document along with the tag, such as `[[date]:format(yy-DD-mm)]` where date tag has a `format(yy-DD-mm)` metadata. Templater will sometimes inject special metadata used for context tracking purposes. It starts with _ci: and can be ignored. Since version 2.3 Templater has a **getMetadata(tag, index)** method which can be used to get metadata of each tag without first processing them. It will return null if tag doesn’t exists at that index.

`replace(tag, value)` method is the core of Templater API. Important thing about it is that it will not invoke processors, but specialized data types will still work, since they are implemented for each document type separately, while processors work on every document type. Replace will invoke replacement of only a single tag and return status if the tag was found/replaced. This is similar, but different from calling high level API with **process(new { tag = value })** which will also invoke processors, formatters and will often do multiple replacement if multiple tags are detected inside same context. Since version 2.3 Templater has additional **replace(tag, index, value)** method. This can be used to support more complex scenarios. While tags are ordered sequentially, there are some

exceptions to this; for example in Excel header tags are available only after every other tags.

`resize(tag collection, count)` is primary core API for duplication of contexts. Context is detected based on provided tags and resized/duplicated based on provided count. If 0 is used, context will be removed. In Java due to type erasure, Templater has difficulties removing context for empty collections, since it can't match tags for erased collection. In .NET type reification allows for such a feature.

Often, when collection is the primary input into the process method, check pattern can be used to remove the context, such as

```
if (collection.isEmpty()) {
	document.templater().resize(new String[]{"first", "last"}, 0);
} else {
	document.process(collection);
}
```

Another workaround for erasure is by using the collapse metadata which, if added on collection tag, will cause removal of appropriate context with matching tag even when the collection is empty.

Java arrays can also be used as a workaround for erasure. Templater will detect correct type even for empty arrays which means that

```
document.process(arrayCollection);
```

will work as expected.

For example, using previous model tag `[[ingredients]:collapse:hide]` can be added to a row to cause removal of it in Java even when ingredients field is null or empty. Metadata hide is used to remove the replaced value when collection is not empty, since Templater will replace tag `[[ingredients]]` with `toString` value of ingredients field, which is not really useful in this case.

Common pattern with collapse tag is conditional values used in a document. Collapse is just a registered plugin, which means that if custom behavior is required, users can implement their own collapse mechanism.

For model such as:

```
class Applicant
{
	public string Name { get; set; }
	public EmployedFromUntil EmployedFromUntil { get; set; }
	public EmployedFrom EmployedFrom { get; set; }
}
class EmployedFromUntil
{
	public string Employer { get; set; }
	public DateTime From { get; set; }
	public DateTime Until { get; set; }
}
class EmployedFrom
```

```
{
        public string Employer { get; set; }
        public DateTime From { get; set; }
}
```

where Applicant object is populated with either EmployedFromUntil or EmployedFrom properties, template can be prepared as such to cover both use cases. Collapse metadata can be inserted in specialized blocks, such as an invisible table:

Employer: [[EmployedFromUntil.Employer]] From: [[EmployedFromUntil.From]:format] - [[EmployedFromUntil.Until]:format] [[EmployedFromUntil]:collapse:hide]

Employer: [[EmployedFrom.Employer]] From: [[EmployedFrom.From]:format] [[EmployedFrom]:collapse:hide]

While those look like hacks, this is due to simplicity of tag definition, since there doesn't really exist separate information about tags within the document, except what is written in it. While Templater could have some editor to define tags in templates and store its metadata somewhere else, the only thing this would accomplish is removing that information from the tag, which is not really useful for the resulting document.

`clone(count)` is a somewhat distinct API from others, since others work on the same instance of `ITemplater`. `clone` is used when the whole document needs to be duplicated and each part of a duplicated document needs to be processed separately. `clone` is also available with `clone` metadata, which means that if you add `[[name]:clone]` tag to the document and provide collection of receipts to the process method, Templater will behave similarly as before but slightly different; context is now whole document by default, while previously it would be detected based on tag positions. If all tags are inside a table object, previously only new rows would be created, while whole document will now be replicated. Another difference is that even when whole document is detected as context, `clone` will insert a page break at the end of the document, while `resize` (which was invoked before) will not. Oftentimes `clone` is not necessary as same behavior can be obtained by tweaking the document a little.

For example, if we want to send a collection to process and resize the whole document, but are not using clone metadata, we can still have page breaks by either adding page break at the end of the document or at the beginning of the document (which is preferred since then we will not have one last empty page).

Extensibility

While Templater was built to be extensible, as of version 2.0 Templater offer extensibility to the developer even in standard version. Extensibility comes in various flavors, some of which were covered previously like enumerating default processors. Extensibility is achieved by implementing API for that particular feature. In example of processors, it's expected to provide implementation for data processor interface which is just a

```
bool TryProcess(string prefix, ITemplater templater, T data)
```

method. By implementing this signature and registering it alongside other processors, process method can now choose this processor as best one and object processor can invoke it too when it finds an appropriate type.

Metadata plugins

Templater includes various metadata plugins which are used to apply conversions to values. In practice this means that plugins are provided with a value and metadata and they decide if they want to act upon it and transform it somehow. An example of this would be `bool` metadata which can be used as `bool(Yes/No/Maybe)` and boolean metadata plugin will convert Boolean object to Yes, No or Maybe string depending on its value (true, false, null).

API for metadata plugins is a simple method

```
object Format(object value, string metadata)
```

which needs to be implemented and registered alongside other formatting plugins.

In .NET registration is done by providing `Func<object, string, object>` signature during configuration. In JVM version expected IFormatter interface implementation is expected. In Scala this would look like:

```
val factory = Configuration.builder().include(new
IDocumentFactoryBuilder.IHandler {
  override def handle(value: AnyRef, metadata: String, property: String,
templater: ITemplater) :Boolean = {
    ...
  }
}).build()
```

An example of useful metadata plugin would be dynamic conversion of field to Image. For example, if a field contains the location of an Image, we could load that Image using a custom plugin and return it as a result. Currently Image objects need to be added directly into the model for Templater to pick it up:

```
class MailMerge
{
        public string Title;
        public DateTime Date;
        public Image Signature;
}
```

README.txt file contains list of all plugins available in the Standard version. Here are some of them:

- boolean formatter (mentioned before) - which converts a boolean value to an appropriate string. This is useful for localization of yes/no answers in reports.
- date time formatter - `format` - which is useful in .NET since it doesn't have explicit Date object such as LocalDate often used in Java. It will convert DateTime value to

short string date value if it doesn't have a time part

- date/time formatter - `format(XXXXX)` - which can be used to present a value in some specific format. While string property can be added to the model which the exact format, sometimes it is more appropriate to specify custom format in the template. Often it's much easier to customize it on the fly, instead of recompiling code.
- empty formatter - `empty(XXX)` - which will replace null value or empty collection with defined string
- value formatter - `format(XXX)` - which can be used on various types to invoke platform specialized formatting for that type

Nice thing about formatting plugins is that they can override default behavior and in such a way work sometimes around built-in limitations. Custom plugins are called before built-in ones, so this can be used to provide different behavior.

Example of such plugin is bool(true/false) which can be improved with better parsing method to extract messages from it.

Document plugins

Templater comes with three document formats: TXT, DOCX and XLSX, which means it has three document plugins. Document plugins are implemented using low level API: `ITemplater`.

That's everything that needs to be implemented to support some new document format. While small on the surface, often the implementation is quite big since it needs to understand and be able to manipulate target documents.

While Templater could support other document formats natively, such as PDF, old DOC and XLS formats, they are currently not supported and there are no plans to support them soon. ODF format was partly supported, but was never finished due to lack of interest.

Since most documents are exchanged as PDFs, there are a couple of ways to create PDF documents from Templater's output. One popular way of doing it is on the server, by running LibreOffice in headless mode. LibreOffice has a PDF conversion tool which is "good enough" and can be used to convert DOCX documents to PDF. On Windows servers Microsoft Word Service tool[5] can be used, since MS Office doesn't really work in server environments.

Example of PDF conversion using LibreOffice:

C:\Program Files\LibreOffice4\program\soffice.exe -norestore -nofirststartwizard -nologo -headless -convert-to pdf document-example.docx

will result in a document-example.pdf.

5 Code example is available at http://msdn.microsoft.com/en-us/library/ff181518.aspx

Processor plugins

As previously explained, the processor works as a single method plugin API, and new processors can be easily added. While most processors work via reflection and only on public fields/methods/properties, custom specialized processors can work without such restrictions. They often use other methods directly on other processors which they have as a dependency.

For example, DataTable processor has DataRow processor as a dependency and it calls its methods directly and not via plugin API. This allows for greater flexibility than just using the low level API to process some complex document by reusing other existing processors.

Some processors understand specialized metadata, which are mentioned in the README.txt file, some of which are:

- collection metadata - `fixed` - useful when we want to use a table with predefined tags, but don't want to resize that table. For example we want to have a table with 10 fixed rows, all containing `[[description]]` tag, but if we provide collection with 3 items, only first three will be replaced with provided values, while others will be replaced with an empty string
- object metadata - `all` - in a few cases context detection cannot correctly detect what context should be used - in such cases, unless we use a more appropriate data type, there are workarounds with specialized metadata such as all which tells object processor to replace all found tags, even in different contexts.
- repeatable collection metadata - `repeat` - which can be used to force Templater to reuse same collection on multiple parts of the document. For example, if we have a list which we want to use two times on a document, once in a table and once as a list (or twice in two different tables), one way to process it in Templater would be to call process method twice. But if the list is part of another object and we can't send that object twice, we can either reuse same property under different method name (such as `getList` and `getList2`) or force Templater to reuse collection multiple times in same context. This will probably be improved in the future, but for now those workarounds can be used.

Context detection

Probably the most important aspect of Templater is context detection; the ability to detect regions of document which need to be duplicated. Most reporting solutions have editors where designers can explicitly declare context regions, while Templater needs to guess the best context for provided tags and data.

Context detection is specialized for each document, since they all have different rules and models. Simple examples of contexts are a row in a table or a bullet in a list in Word. In Excel an example of context is an explicit table or an implicit range. Context can also be a named range, a whole sheet, the whole document and similar concepts.

But context detection is not just that simple, let's say for example that we are using Word document which has a table and we want context to be two rows, not just one row.

If we use class Ingredient from above and have a table as such

Name	Quantity
{{name}}	
	{{quantity}}

Templater will assume context of two rows since it will search for best context using name, quantity and info tags, and by analyzing the result it will conclude that duplication of Ingredient context must span both lines.

This is even more important in Excel when context is calculated around cells, where it will detect range starting from `{{name}}` cell and ending on `{{quantity}}` cell, which means it also has a width.

Besides duplicating context, style is maintained, which makes it possible to retain colors, fonts, bolding and various other features.

When recursive structure is used in combination with appropriate object in template, complex templates can be built.

For example if we have

```
public class Nest
{
	public String name;
	public Nest[] nested;
}
```

we can use lists such as

1. [[name]]
    1.1. [[nested.name]]
        1.1.1. [[nested.nested.name]]

and Templater will work on the correct level during processing of such a pattern.

DataSet relations can be used as master-detail binding meaning that structures such as DataTable with column level1 and DataTable with columns level1 and level2 joined with ChildRelations using level1 column can be used to match contexts in table or in list such as

1) [[level1]]

a) [[level2]]

with appropriate behavior.

Contexts can be nested, meaning they can be a table in a table with a list in a row and, by using property navigation and collections, complex report can be created.

In some scenarios context need to retain state so nested context manipulation can be done correctly. This is especially visible in Excel where resizing can often stretch merged cells, but rules for stretching are dependent on previous resize. Also, pushdown rules need

to take into account previously defined context, so that correct range is pushed bellow currently used one. For example Excel document such as:

[[Simple.Name]]				
	[[Simple.Items.Name]]	[[Simple.Items.A]]	[[Simple.Items.B]]	[[Simple.Items.Total]]
			Total	**[[Simple.Total]]**

need to remember total range from [[Simple.Name]] to [[Simple.Total]] when resizing [[Simple.Items.*]] context, since if they only push cells bellow their maximum range, [[Simple.Name]] tag which were copied beneath it will not get pushed bellow (as it should).

Resizable elements in Word

Depending on where the tag is detected, Templater will behave differently. Specific elements are considered resizable:

- Table
- List

which means if tag is detected only within those elements it will try to create context around it. Then resize will duplicate minimum context which is defined around all specified tags. Lists or tables can be nested inside other tables. Table with invisible border can be used as custom control for defining applicable range.

Low level metadata *page* can be used to instruct Templater that actual range is page, even when tag is defined within table/list.

Resize tuning in Excel

Excel is based on cells and as such, cell regions spanning all specified tags are considered context. Specific elements have custom behavior and can change drastically behavior and definition of the context:

- Table
- Named range
- Merge cell

Similarly to Word, Excel tables are recognized as special resizable element. Using tables provides some benefits, such as formula definition based on column names. But using tables also incurs some restrictions such as disallowance of merge cells. Tables are often convenient as data source for pivot tables and charts, since they can be referenced with their name. Tables in Excel are always regarded as a single context. When doing resize(0) on a table, table will not be removed, since table with zero rows are not allowed in Excel.

Named ranges are convenient way to manually define context for Templater. This is often necessary since Templater will use minimum spanning context, while it ought to use somewhat larger one. By defining appropriate named range around those specific tags, Templater will use that instead. We can still use nested ranges inside that named range, in which case it will stretch to accommodate for the new elements within. During

pushdown/push-right named ranges will be considered as region definitions (along with tables and merged cells) which influence the calculation for push range.

Merge cells are useful visual construct which Templater respects during resize.

Enterprise version

Source code is available with Enterprise version (C# for .NET or Scala for JVM)[6]

Tests covering all features are available with the Enterprise version.[7]

Templater could have more complex features in metadata, such as conditional evaluation (runtime evaluation of code defined in metadata or similar specialized commands), but this goes slightly against its core principals, which are that object model should be mostly prepared before sending it for processing and only small tweaks should be utilized in the actual document. If developers are very interested in such a feature, they can easily add it themselves by writing custom plugins and registering them during startup.

License

Templater can be evaluated without purchasing a license, in which case it will insert a message at the start of the document. It tries to be a low cost solution, licensed per developer, with an option to get a source code using Enterprise license.

To remove the message developers need to include the license information during Factory initialization or by embeding the license key into the application.

Known issues

Templater uses a lot of herustics to pick the best context. While this works out-of-the-box for trivial documents, on more complicated ones, close matching of model with the document structure is required.

Java XML provider works best with latest version of Xerces and Xalan. While Templater might work ok for most scenarios with earlier versions, or with some other provider, in case of some strange errors it's best to redirect provider factories to „standard ones“. Change log specifies specific System properties for customization.

Non ASCII chars in tags. Templater supports only subset of characters in tags. It's best to stick with letters and numbers for tag definition.

Pushdown/Push-right might break the document in unexpected ways. Its rules might be non-obvious but are not too complicated.

---

6 Customers need to specify wheater they want a .NET or JVM source. If both sources are wanted, two Enterprise licenses must be bought

7 The only thing which is not available is the private encryption key used for license validation

Conclusion

Templater takes quite a different approach than most reporting libraries, since it doesn't have `addParagraph`, `insertPicture` and `importTable` methods -- it tries to unify document processing under a minimal API which is not document dependent. While sometimes this doesn't work, most of the time it works really well. Context detection, while mostly heuristics, works quite well and is able to understand the intentions of the document designer. The ability to template documents in Word or Excel makes customization by the non-developers a possibility. PDF export, while not ideal, is often good enough when native documents are not desirable.

By examining various available examples available on the website, developers should get a hang of the expected behavior (after maybe a few initials hurdles caused by expectations from other reporting libraries) and should be able to build fairly complex documents by using various tricks such as collapse as conditions for hiding parts of the documents and nested collections for automating duplications of various parts. In the end, probably the most important thing is to be able to give the template to some non-developer and let him tweak it without the need for code changes.