Templater user manual

Contents

About Templater

Templater is minimal reporting library for Java and .NET which is used for generating 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] scenarios used in reporting which are based on templates. Templater is not a generic reporting solution, as in, it does not support programmatic document manipulation except those based on removing, replacing and duplicating existing tags in a document. It doesn't have an insert picture method, but if a tag exists in a document, by replacing that tag with an Image, picture can be inserted 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.

Minimal API

Templater API has three parts:

- opening and closing/flushing document
- processing objects
- low level tag manipulation

Documents can be DOCX, XLSX or TXT files. Templater has only primitive support for TXT files since it doesn't really have context except row. So it's useful only for most primitive use cases, while DOCX and XLSX are feature rich.

JavaDoc is available at website: http://templater.info/apidocs/

.NET documentation is available in XML provided with the dll, but API is almost the same as Java.

Opening and closing documents

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

For example, in Java, input stream can be FileInputStream and output stream can be FileOutputStream. In .NET 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 HtppResponse.OutputStream and write directly into it. Internally Templater will use an internal temporary stream, which can cause increased memory usage for large documents, but will have better performance overall. Templater internal stream avoids LOH issue

---

1 yes, a made up statistical number

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

in .NET.

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

While best practice in .NET is not to do much work in Dispose[3], this way the API remains small and consistent. An example usage is

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

A good practice is to cache the Factory instance in a shared static field and reuse it for processing.

Processing objects

ITemplateDocument consists from a single process method and access to 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);
}
```

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

Behind the process method various plugins live and best plugin is matched against

---

3 this is based on .NET framework guidelines which suggest having a Close method alongside Dispose method

provided object before processing. For example in .NET processing plugins are:

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

Java has only subset of those 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 passing objects to another plugin with current navigation prefix, all they can utilize is low level API available to the developer.

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 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.

When object processor is processing an instance of Receipt it will match [[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 Receipt instance, object processor will find ingredients field using reflection. Same cycle will begin, but now with “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 best matching processor – in this case object processor again. During processing of the info field, another best processor will be chosen, in this case a dictionary processor, with “ingredients.info.” prefix. This means that if dictionary instance contains “author” key [[ingredients.info.author]] tag can be matched.

An important distinction during processing is 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 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 specialized data type which will replace tag with provided image.

Jagged arrays are another example of specialized data type which can be used to resize table in both directions, not just rows as with 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).

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 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 the tag, such as [[date]:format(yy-DD-mm)] where date tag has a format(yy-DD-mm) metadata.

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.

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.

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.

Common pattern with collapse tag is conditional values used in a document.

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 exists a separate information about tags in 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, only thing this would accomplish is removing those information from tag, which is not really useful for resulting document.

clone(count) is somewhat distinct API from others, since others work on the same instance of ITemplater. clone is used when whole document needs to be duplicated and each part of duplicated document needs to be processed separately. clone is also available with clone metadata, which means if you add [[name]:clone] tag to the document and provide collection of receipts to process method, Templater will behave similarly as before but slightly different. The difference is that context is now whole document by default, while previously it would be detected based on tag positions. Difference is that if all tags are inside a table object, only new rows would be created previously, while whole document will be replicated now. 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. But often, clone is not necessary since 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, Standard version of Templater doesn't offer extensibility to the developer. Extensibility comes in various flavors, some of which were covered previously be enumerating processors included in the Standard version. Extensibility is achieved by implementing API for that particular feature. In example of processors, Enterprise users have access to the 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 founds appropriate type.

Metadata plugins

Standard version includes various metadata plugins which are used to apply conversion on values. In practice this means that plugins are provided with a value and a 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 again a simple method

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

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

In .NET explicit registration doesn't really exist, since types are scanned and included

based on their signature. Currently in Scala version (JVM) plugins are registered by hand on startup along the lines of:

```
val formatters = Array[IDataFormatter](
  new BooleanFormatter
, new MyCustomFormattingPlugin
...
```

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 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 boolean value to 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

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 what 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 document.

While Templater could support other document formats natively, such as PDF, old DOC

and XLS formats, they are currently not supported and there aren't any plans to support them soon. ODF format was partly supported, but due to lack of interest, never finished.

While this makes Templater usability questionable, since most documents are exchanged as PDF-s, there are other ways to create PDF documents. Since this is mostly interesting doing on the server, running LibreOffice in headless mode is a popular way to do it. LibreOffice has "good enough" PDF conversion tool[4] which 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.

Processor plugins

As previously explained how processor works and it's single method plugin API, new processors can be easily added. While most processors work via reflection and only on public fields/methods/properties, custom specialized processor 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 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 few cases context detection can't detect correctly what context it should use - in such cases, unless we use more appropriate data type, there are workaround 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 document, once in a table and once in a list (or twice in two different tables), one way to process it in Templater is to call process method twice. But if the list is part of another object and we can’t really 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 must be used.

---

4 Using command line, small application or a tool such as JodConverter

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

Context detection

Probably the most important aspect of Templater is context detection; the ability to detect regions of document which needs 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 example of context in a Word document is a row in a table or a bullet in a list. In Excel an example of context is an explicit table or an implicit range. Context can also be named range, whole sheet, 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 correct level during processing 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.

Enterprise version

Standard version of Templater, should work for most developers. If a developer has some pattern which is useful to others, it can probably be included in the Standard version. But if developers have use-cases which are specific to them only and they need to add specialized plugin, they can do that in the Enterprise version.

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

Tests covering all features are available with 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 add it themselves easily in the Enterprise version.

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 license key into the application.

If you have an open source project and want to use Templater with it, you can ask for a license which you can include with yout project.[8]

---

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

8 Please provide us with a link or some official info about your project when asking for an open source license

Conclusion

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

By examining various available examples from 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 quite a 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.