Templater user manual

v8.4

September 21, 2026

New Generation Software Ltd
Rikard Pavelić



Content

About 4
New Generation Software Ltd. 4
Templater 4
Supported documents 5
Office Open XML 5
XML format 5
Text based formats 6
Supported languages 7
Microsoft .NET 7
Oracle Java 7
Google Android 8
Others 8
Getting started 9
Tags 9
Minimal API 10
Setting up a project 12
Thread safety 13
Built-in processors and analysis 13
Application Programming Interface 15
Low-level API 15
High level API 22
Configuration 27
Templater Editor for Microsoft Office 48
Installation and licensing 48
Schema 50
Tag analysis 53
Tag navigation 54
Tag listing 55
Running Templater 56
Debugging Templater 60
Word features 62

Mail merge 62
Resizable regions 63
Word specific features 80
Known issues 91
Excel features 93
Complex non-streaming documents 93
Resizable behavior 94
Excel specific features 113
Known issues 134
PowerPoint features 135
Ready-to-use presentations 135
Resizable behavior 135
PowerPoint specific features 145
Known issues 150
CSV/text features 151
Simple documents 151
Streaming documents 152
XML features 154
Simple documents 154
Streaming documents 155
Best practices 156
Complex documents 156
Performance/memory optimizations 163
Tag management 169
User defined plugins 171
F.A.Q. 172

About

New Generation Software Ltd.

N.G.S. is a software company founded in 2005 by Rikard Pavelić in Croatia. It focuses on making software development easier and more productive. Since its early days Templater-like solutions were used for reporting/document generation. Once we realized such an approach would be useful to others, N.G.S. released Templater v1.0 in 2011.

N.G.S. primary focus is on providing consulting around its software products.

Templater

Templater focuses on binding data with documents. This allows for separation of data and layout leading to customization of templates by business users/end clients.

Unlike other reporting libraries which focus on document layout through low level API, Templater in comparison only has high level API. This way document layout is not defined in code, but rather provided outside, often by designers, domain experts experienced with Excel or even end users of the software.

While Templater is not a generic reporting solution it can be used to create really complex documents, but often this will require knowledge of Word, Excel and PowerPoint to setup such documents. Enterprise companies can utilize Templater Editor for Microsoft Office to ease template management via smart analysis and quick iterations.

Templater is used by wide variety of companies and non-profits around the world, from large banks to small startups.

Supported documents

Office Open XML

Templater main focus is to support Microsoft Office formats based on XML. Word, Excel and PowerPoint support new XML based format which is standardized by ISO since 2007.

To support such formats Templater must understand many of the features supported by Word, Excel and PowerPoint. Templater is written in such a way that new Word, Excel and PowerPoint versions work out of the box for most old and new features. Only in specific scenario a new version is required to support some specific feature of the new Microsoft Office tools.

Supported extensions:

- docx - standard Word XML format
- docm - macro-enabled Word format
- xlsx - standard Excel XML format
- xlsm - macro-enabled Excel format
- pptx - standard Presentation XML format
- pptm - macro-enabled Presentation format

Some features require combination of documents/formats, e.g. chart in a Word/PowerPoint requires the use of xlsx embedded within the docx/pptx. Templater supports such features seamlessly. Support for Word embedded documents also works out of the box, such as other docx, XML or text files.

Templater has excellent performance and can create 100+ Word pages per second, generate huge Excel files (50MB+) while keeping memory under control.

Use of XML based formats requires a valid license; otherwise a watermark message will be left in the document.

XML format

To support OOXML, Templater needs to work closely with XML. Therefore, Templater has rather good XML support and can deal with large XML files in a performant manner.

Since v7, XML format is natively supported by Templater in streaming fashion which is convenient when it can cover a use case, such as integration, or some externally defined protocol.

If Templater is used without a valid license, it will inject comment at the top of the XML.

Text based formats

Unlike XML based formats which require a valid license, text-based formats can be used for free without buying a license.

Common use case for text-based formats are CSV (with streaming support), simple message generation (such as configurable SMS message) or white labeling specific HTML based outputs.

When passing in extensions to Templater, specific ones will be recognized:

- csv - Comma Separated Values format
- txt - text based format
- utf8 - text based format using UTF-8 encoding

All extensions use same processing method, with the only difference being that utf8 extension uses explicit UTF-8 encoding, while the others use system/input default encoding.

Supported languages

Source code is written in two different languages for different platforms. C# is used for Microsoft .NET, while Scala/Java is used for JVM. There is feature parity between the languages, as much as languages allow for it. Whenever a new version is released features are built for each language separately.

Since dynamic structures are natively supported, Templater is usable from command line or as JSON endpoint, which makes it easily usable from other languages.

Demo page provides a nice example of Templater support for JSON. While there is no in-built support for JSON, when JSON is transformed into appropriate lists and maps/dictionaries it can be processed by Templater.

Microsoft .NET

Templater is released for several Microsoft .NET versions:

- .NET v4.0
- .NET standard v2.0

This means Templater works on all relevant versions of .NET:

- legacy Microsoft .NET Framework - Windows only
- .NET (core) - new, cross platform version of Microsoft .NET

While Templater works on Mono, it is not officially supported on that platform.

Nuget can be used to add Templater dependency.

Oracle Java

Templater supports Java 8 and newer versions. There is no special use of internal APIs, so Templater works out-of-the box on new Java versions such as Java 11, 17 or newer. Templater for Java is a no dependency jar without any external dependency.

Maven can be used to add Templater dependency.

For JVM there is a separate Scala build for version 2.13.

When Scala builds are used, Scala specific data types can be used, such as Option and collections.

Google Android

Templater also works on Android, although it requires specific configuration setup. Both Java and Scala versions can be used. Since Android does not support awt package, during initialization, low level plugins need to be disabled, which will avoid loading the awt classes.

Minimum Android SDK version supported is 26, which is Android 8.

Others

JSON

Common use case is to pass in JSON to Templater. For this to work JSON must be transformed in appropriate data structures. In Java those are arrays, lists and maps and for .NET their counterparts: arrays, lists and dictionaries.

Maps/Dictionaries must have string as keys, meaning only:

- IDictionary<string, object>
- Map<String, Object>
- IDictionary/HashTable
- Map

with strings as keys are supported.

In .NET default JSON.NET conversion is not supported, but rather an alternative conversion method must be used.

Github example shows how Templater can be used from a command line by passing JSON with a template document to Templater for processing. It also has a .NET implementation for JSON -> Dictionary conversion which works as expected in Templater.

HTTP server

While there is no build of Templater for Javascript, it's rather easy to consume it through HTTP API. Templater demo page is a small application which provides various other functionalities and can be used as a starting point for building internal applications which use Templater though REST API.

Templater server application also shows how a third party application can be used to complement document generation by doing PDF conversion directly in the browser using Silurus or in the backend via Aspose, Spire or LibreOffice.

Getting started

Only a couple lines of code are required to use Templater API. While the main Templater API is very small, Templater can be configured with custom code which makes it highly customizable and allows support for all kinds of use cases.

Templater can be tested through the browser in an online demo which is just a simple application that shows few basic use cases.

All features of Templater are available for testing without buying a license in which case Templater will inject a watermark message into the document. Templater comes with a license designed for easy integration into third party applications. It is not allowed to remove the watermark message from the generated documents which will be removed once a valid license is used during the initialization.

Many examples are available at Github and customers will often be referred to the example relevant to their question.

Tags

Templater works by analyzing document and locating tags within the document. Tags can come in 3 different formats:

- [[TAG]]
- {{TAG}}
- <<TAG>>

Tag format can be customized/disabled during library initialization. This is explained in more detail later in the documentation.

Metadata

Tags can have metadata. Metadata is an additional info put alongside tag which is used by Templater to invoke some specific actions and often combined with custom code registered during library initialization. Metadata is defined by semicolon and a pattern. There can be multiple metadata on a tag. Special characters must be escaped with a backslash (\). Metadata are processed in order of the definition.

Examples of metadata:

- [[tag1]:format]
- {{tag2}:format(YYMMDD):padLeft(10)]
- <<tag3>:empty(value was missing)]

Navigation expressions

Special metadata can also be used in tags. Tags consist from navigation parts and each part can have additional metadata attached to it. This metadata can greatly change the way processing is handled. To enable this feature navigation metadata separator must be specified. Examples of navigation metadata with semicolon separator:

- [[instance.collection:top(10).account.number]]
- {{person:method(a,b)}}
- <<items:sort(name):at(1).description>>

Metadata on expressions can be combined, the same way metadata on tags can be combined.

Via Navigation expressions all Metadata features can be reproduced. This is not done by default to simplify learning curve.

Minimal API

There are only a handful of methods in Templater:

- Configuring library
  - **Factory** - for default configuration without any plugins. License file: *templater.lic* will be resolved from relevant places (resource in the project and the root folder)
  - **Builder** - for configuring library with various custom behavior
- Opening and closing/flushing the document
  - **Open**(file) - processing file in place
  - **Open**(input stream, extension, output stream, cancel token) - reading template from input stream and writing it to output stream. Depending on the extension and the usage additional in-memory stream can/will be used, alongside with continuous streaming to output while processing
- High level:
  - ***Process*****(**data**) -** accepts various data types and process it according to either build rules or via custom code defined for specific types
- Low level:
  - ***Resize*****(**tags, count) - resizes (duplicates) part of the document which contains all specified tags. When count = 0 part of the document will be removed
    - There is also special Resize where instead of strings, pairs of string and integers are accepted. This is highly specialized API for fine control of tags[1]
  - ***Replace***(tag, value) - replaces first matching tag in the document
  - ***Replace***(tag, index, value) - replaces tag at specified position
  - ***GetMetadata***(tag, all) - returns user defined metadata for the tag (either first or for all of them)

[1] This API is used to explicitly state which tags will be used. Unlike the string version which has various smart behavior built in, such as expanding the context and working on shared contexts, this version has much simpler behavior

- ***GetMetadata***(tag, index) - provides both user defined and internal metadata for the tag at specified index. Internal metadata is used by the library to detect appropriate parts of the document and relationship between tags
- ***Tags*** - lists all current tags detected in the document. Once a specific tag is processed it will be removed from this list

Most usage of Templater consists solely from using high level API. Low level API is mostly used by Templater internals or 3rd party plugins. In practice, Templater API should be used via application specific abstraction which does the setup of the library and just pass in the template and data for processing.

Builder/Configuration API is explained in more detail later in the documentation.

A short description of each method is available via native documentation. Javadoc can also be browsed online.

Pros and cons of minimal API

While high level API is deceptively simple, depending on the types of the data passed in, and on the plugins (either built-in or custom registered during initialization) Templater can perform various non-trivial operations on the document:

- image - picture will be inserted into the document
- SVG document – a SVG picture will be inserted into the document
- two-dimensional array/data table - dynamic resize feature of Templater will be invoked
- XML - raw XML can be inserted into the document or special actions can be performed
- URL - a link will be added into the document
- File – an embedded document will be added into Word
- ResultSet/IDataReader/Iterator/IEnumerator – document can be processed in a streaming way (without loading all data upfront/

While API is rather simple, to fully master the use of Templater various examples should be explored to learn all minor details which can be used to tweak behavior during processing.

Major benefit of minimal API is that once data is designed/defined it is no longer required to change interaction with the library. This simplifies change requests and increases code reuse. When dynamic data types are used, such as maps/dictionaries, often no code changes are required. Examples of this would be:

- ResultSet/Data table processing is independent of the SQL which populates them
- Dictionaries/Maps can be processed without any code changes
- data types used for presentation can be reused for reporting
- plugins can be used to transform between data types which allows for even more reuse

While benefits heavily outweigh the downsides, there are some and often developers need to unlearn some habits carried from other reporting libraries:

- its non-obvious that Image data type must be used to inject image into the document
- since there is no API for setting color of a font, such customizations require Templater specific way of doing things
- processing collections without for loops and start/stop definitions often require mind-shift for developers used to explicit imperative style of document generation

Setting up a project

While there are numerous project examples at Github, such as the most simple one it is rather trivial to setup the project.

.NET project example

1. In a C# project add a reference to Templater via Nuget: ***Install-Package Templater 8.1.0***
2. Prepare a Word file with tag [[tag]] and add it to the project
3. Initialize the library to create reusable factory:
```
var factory = Configuration.Factory;
```
4. Open the document for processing with using pattern:
```
using(var doc = factory.Open("template.docx"))
```
5. Process the document using high level API:
```
doc.Process(new { tag = "value" });
```
6. At the end of using there is an implicit call to finish the processing:
```
doc.Dispose();
```

During the call to Dispose Templater will flush the result of processing to the output stream, or in this case to the opened template.docx file.

Java project example

1. In Java Maven project add a dependency to Templater:
```
<dependency>
    <groupId>hr.ngs.templater</groupId>
    <artifactId>templater</artifactId>
    <version>8.1.0</version>
</dependency>
```
2. Prepare a Word file with tag [[tag]] and add it to project resources
3. Initialize the library to create reusable factory:
```
DocumentFactory factory = Configuration.factory();
```
4. Open the document for processing using try-with-resource pattern:
```
try (TemplateDocument doc = factory.open("template.docx")) {
```
5. Process the document using high level API:
```
doc.process(new HashMap<>(){{put("tag", "value");}});
```
6. At the end of using there is an implicit call with try-with-resource pattern, which can be also called manually, to finish the processing:
```
doc.close();
```

Providing license information

Before Templater can be used without a watermark message, valid license info must be specified during initialization. There are multiple ways to specify the license info:

- license can be embedded into the project as *templater.lic* file[2]
  - during initialization Templater will scan the project resources for the file with exact name. Resource file must be kept in the root folder, otherwise it will not be found
- license information can be specified during initialization via **Builder.Build**(customer, license)[3]
  - this is the recommended way of initializing the license, as it does not depend on project setup which sometimes change the resource behavior
- license file can be specified during initialization via **Builder.Build**(path)
  - path can specify location on the disk or in the project resource

Thread safety

Once factory was initialized it can be reused to open/process/flush templates. Factory can be reused concurrently across threads as it is thread safe. *ITemplateDocument* instance created from the factory is not thread safe and should not be concurrently used from different threads. Best practice is to initialize the factory once in a static field:

```
private static readonly IDocumentFactory Factory =
        NGS.Templater.Configuration.Builder.Build("Email", "License");
```

Templater supports cancellation pattern and its common practice to run processing on a separate thread while monitoring execution duration and memory usage. This can be used to guard against memory starvation since XML can be quite memory heavy.

Built-in processors and analysis

Various data types will work out of the box, such as:

- collections (arrays, lists, iterators, ...)
- maps and dictionaries
- result set/data reader
- data table/data set
- objects via reflection

When data is passed to high level API best match is found for the provided input. Once match is found data is processed by the appropriate processor. During processing Templater will navigate over objects/data and process parts of the object via other appropriate processors.

[2] .NET example: https://github.com/ngs-doo/TemplaterExamples/blob/0b21f41fb97163c0895100bc7114d169aa5d4c89/Beginner/WebExample%20(.NET)/TemplaterWeb.csproj#L98
[3] Java example: https://github.com/ngs-doo/TemplaterExamples/blob/master/Advanced/PowerQuery/src/main/java/hr/ngs/templater/example/PowerQueryExample.java#L21

Data is matched against the tags which were analyzed at the start of processing. Once document is analyzed, result of the analysis is available via various methods (**GetMetadata**) and properties (**Tags**).

Processors will try to use only the relevant tags, which mean that it's fine to call high level **Process** method multiple times which will only affect relevant parts of the document.

Processors can create new tags via **Resize** API or remove them via **Replace** API. Tags created via **Resize** API will be available for new processing. New tags can't be created[4] via the **Replace** API even if value is used which looks like a tag, e.g.: **Replace("tag", "[[newTag]]")**

Processing of streams can be chained and thus new tags created from the previous processing which is no longer available for processing can be processed in subsequent processing.

Built-in object processor will only use public fields and methods. While classes do not need to be public, in Java it is highly recommended that classes are public since otherwise performance overhead will be incurred or processing can fail due to lack of security permissions[5].

Java beans standard is supported for method names (which makes templates more readable).

[4] There is a way to create new tags by combining replace with a resize, but that is not a common API usage
[5] There is a configuration option to control this visibility behavior

Application Programming Interface

While there are not many methods in Templater API, they can be used in variety of ways and by combining them in specific patterns complex operations can be performed.

The API is small on purpose, to encourage generic data binding instead of programming document layout through code. Templater follows the language naming standards, thus interfaces in .NET are prefixed with an I, unlike in Java[6].

Low-level API

ITemplater (.NET)/Templater (Java) is often referred as low-level API since it works at the lowest level which is just an abstraction over the document format. There are only few methods, but with few overloads:

```
public interface ITemplater
{
        IEnumerable<string> Tags { get; }
        string[] GetMetadata(string tag, bool all);
        string[] GetMetadata(string tag, int index);
        bool Replace(string tag, object value);
        int Replace(string tag, int index, object value);
        bool Resize(IEnumerable<string> tags, int count);
        bool Resize(IEnumerable<TagPosition> tags, int count);
        IEnumerable<ITemplater> Clone(int count);
}
```

While low level API is rarely used there are use cases when it's useful to use it:

- writing custom data type processors
- writing custom handlers (especially for removing region of the documents)
- implementing custom streaming on objects
- analyzing document and rewriting it into a different processing format
- checking if all tags are processed at the end - it is often that templates are provided with typos or designed in some wrong way; in which case it's useful to explain the problem

Some plugins are invoked (low-level) when calling **Replace** method on the low-level API, while some of them are not (metadata, navigation). Special datatypes are recognized, meaning that call to **ITemplater.Replace**(tag, image) will still inject image into the document.

Tags

Once *ITemplater* is created, all tags are listed (only once) in this property. Once all occurrences of a tag have been processed this tag will no longer be listed. Since new tags can't be created with **Replace** method once processing has started, new values cannot appear in this property.

[6] Prior to v7 interfaces in Java were also prefixed by I

Tag sharing

It is expected that same tag is repeated multiple times. When same tag is defined on multiple places, depending on where the tags are detected they can enter into a special sharing mode. There are different aspects to tag sharing:

- tags repeated in the same row
  - tags share the same initial context
  - low level **Replace** will only replace a single tag; to replace the other tag **Replace** must be called again
- tags repeated in a different row, but part of the same context
  - tags have different initial context, but **Resize** was called in a setup which grouped them together
  - low level **Replace** will only replace a single tag; to replace the other tag **Replace** must be called again
- tags repeated in different tables, but part of the same context
  - when same tag is used in different table, depending on the **Resize** arguments tables can enter into a special sharing arrangement
  - low level **Replace** will only replace a single tag; to replace the other tag **Replace** must be called again
- tags repeated in different sheets/slides, but used in a collection
  - when same tag is used in different sheets or slides, but duplicated via **Resize** they will enter into a special sharing arrangement
  - low level **Replace** will only replace a single tag; to replace the other tag **Replace** must be called again
- tags bound to a custom XML (Word feature only)
  - when tag is bound to an XML, multiple instances of same tag point to the same underlying value
  - low level **Replace** will replace all occurrences of the relevant tag (tag can be repeated inside a collection, in which case only occurrences of the specific row will be replaced)
- tags repeated in embedded document (Word feature only)
  - tags in embedded documents will respect context rules of such documents (docx/csv/html/xml)

An important aspect of tag sharing is that tags will be shared as long as they have the same navigation path. If navigation expressions are used, tags which would traverse the same path will not be considered the same if their expressions are not the same. For example, expressions repeated in different tables:

{{employees.name}}

{{employees.name}:collapse}

will be considered the same, since they have the same navigation expression - employees, so when resized they will enter special sharing mode. To change this behavior so that tags are considered different, navigation expressions can be attached to the tags, e.g.:

{{employees:id(1).name}}

{{employees:id(2).name}:collapse}

Which will cause Templater to treat them differently due to different navigation expressions: employees:id(1) vs employees:id(2). Main use case for navigation expression is not so much to give distinction to paths, but rather to call into user defined plugins for navigation customization.

GetMetadata

Metadata is additional information defined alongside tag. There can be multiple metadata defined for a tag. They are processed in order of definition.

There are various use cases for metadata in tags:

- simple formatting
  - decimal places in numbers
  - date formatting
- type conversion
  - convert from string to an image (by looking up image from a specified location)
  - currency exchange rate conversion
- complex conversion
  - verbalizing numbers into text

Tag can be repeated in the document and they can have different metadata.

A common use case would be to repeat a DateTime field and show it as a separate date and time columns, e.g.:

Date	Time	User
[[event.on]:format(DMY)]	[[event.on]:format(HHMM)]	[[event.user]]

Using GetMetadata method on *event.on* tag will return:

- **GetMetadata**(“event.on”, false) - return the user defined metadata for the first tag. In this case this is: “format(DMY)”
- **GetMetadata**(“event.on”, true) - returns all user defined metadata for this tag. In this case this would be both “format(DMY)” and “format(HHMM)”
- **GetMetadata**(“event.on”, index) - returns both user defined and internal metadata for this tag at specific index. If invalid index was specified (negative or larger than the number of tags) null will be returned.

GetMetadata with an index is required to support advanced replacement scenarios where due to document layout tags are not replaced in order. A common use case would be to have a tag in multiple tables, but when resized only the first tag per table should be replaced.

Escaping special characters

Depending on which tag format is used, if such a character is used within the metadata is must be escaped with a backslash (\). Also, if semicolon has to be used, it also needs to be escaped with a backslash.

For example, to format a time escape code is required in certain cases: [[event.on]:format(HH\:MM)]

Internal metadata

When resize is used on set of tags, they become bound together via certain rules. Templater tracks this binding via internal metadata so it can process them appropriately.

Internal metadata have a "_ci:" prefix, which should be avoided for user defined metadata.

Replace

While **Tags** and **GetMetadata** are read-only operations, **Replace** method mutates the document. The document is mutated in memory[7] until it's flushed (or Resize is called in streaming mode).

Calling a replace will replace only a single tag (except when the tag is bound to XML in Word).

Replace will ignore the metadata specified on the tag and will just replace the specified tag with the provided value. Metadata is only processed when called from a high-level API.

Even if tag format is specified as a value, new tag will not be created without a new analysis of that document region. Analysis is only done at the start (whole document) and after a resize (region only affected by the resize).

While metadata handlers are not called on low level Replace methods, low level API handler are.

Specialized data types also behave as "expected", e.g.: an image passed directly to low level API will still be injected[8] into the document.

Special data types

Some formats use a specialized representation of a value, e.g.: Excel uses different representation for numbers, booleans, text and dates. Both Word and Excel support images which is also not just "a value".

Templater has special logic to handle few specific data types in OOXML formats:

- Image - will insert picture into the document at the location of the tag
    - there is a special Templater image type: ImageInfo[9]

[7] For Word and Excel Templater (.NET) uses a specialized stream to avoid LOH issues: https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/large-object-heap
[8] Inserting an image works only on OOXML formats. It will not be converted to ASCII art if inserted into a text format

        - by default awt images are supported: BufferedImage and ImageInputStream with low-level plugins which should be disabled on Android
        - .NET image types are supported: Image and Icon, but if used in modern .NET environment (such as .NET 7 or newer) only work on Windows and are disabled by default
    - If tag is located on *Alternative text* of an existing image, it will be replaced, but maintain existing setup (size, style, effects, …)
        - This is the only way to insert image in PowerPoint format
- SVG document – will insert SVG picture into the document at the location of the tag
    - Microsoft Office 2016 and later support vector images via SVG standard
    - .NET supported type: XDocument
    - Java supported type: w3c.doc.Document
    - Fallback image can be provided by registering SvgConverter during library initialization
- XML - will insert raw XML into the document
    - this is useful to directly provide values to the underlying format in a way it's not exposed through Templater
    - there are three internal metadata handlers for doing specific operations with the XML
        - replace-xml
        - merge-xml
        - remove-old-xml
        - It is also possible to specify the way how XML should be handled via XML attribute (templater-xml). This is often more convenient than expecting user to specify the way xml should be processed via tags
        - user defined plugins can be registered for further fine-grained control over XML operations
    - it's easy to corrupt the document if "invalid" XML is provided
    - .NET supported type: XElement
    - Java supported type: w3c.dom.Element
    - with multiple XML tag replacements in same paragraph, order might not be respected (it will use reverse order of processing, instead of tag location)
    - In Excel Templater supports various custom attributes which are used to specifying cell/row information. This can be used to specify background of a cell or height of a row:
        - templater-cell-style=CELL – will copy style from the specified CELL onto the cell where tag is located
        - templater-row-style=ROW – will copy style from the specified ROW onto the row where tag is located
        - templater-row-height – used to specify height of a row
        - templater-row-custom-height – used to specify custom height of a row

[9] It is common to transform the image before passing it to Templater (e.g.: resize, change DPI, etc...). A relevant example is on Github: https://github.com/ngs-doo/TemplaterExamples/tree/master/Intermediate/Pictures

        - templater-row-collapsed – used to specify collapsed state of a row
        - templater-row-custom-format – used to specify custom format used in a row
        - templater-row-hidden – used to specify hidden attribute of a row
        - templater-row-thick-bottom – used to specify thickness of row bottom
        - templater-row-thick-top – used to specify thickness of row top
- Jagged arrays and lists[10] with elements of same dimension - invokes a Dynamic resize feature of the Templater
    - all nested lists must be of the same size
- ResultSet/DataTable - invokes a Dynamic resize feature of the Templater
- URL/URI (in Word and PowerPoint only) - will create a simple hyperlink
    - Both Excel and Word have special hyperlink features which can have separate description and link. This feature is recognized/supported by Templater
- java.util.Date/java.time.*/DateTime (in Excel only) - will convert value into appropriate number (unless the value is before 1900-01-01)
    - This way dates can be formatted via Excel formatting features and used in other formulas as expected values
    - If custom Date objects are used (such as Joda) custom converter must be registered for to be recognized as native Excel values
- java.io.File/FileInfo (in Word only) – will create AltChunk element and import embedded document inside zip file
    - embedded document can be either of: txt, rtf, xml, htm, html, docx, docm or doc
    - tags from embedded documents will not be immediately available for usage[11], but stream can be closed and reopened for second processing which will allow for usage of those embedded tags
- java.util.Iterator/IEnumerable/IEnumerator – will process the data in streaming fashion (in case of IEnumerable only when the actual instance is not backed by ICollection) using the streaming size specified during the configuration (16k is the default). This allows for processing of large collections without loading the entire collections in memory

While there is no special type for HTML, it is possible to convert HTML into equivalent OOXML format (most of the time) via third party libraries. Word also supports HTML natively so HTML can be injected via File type as embedded document.

Low level converters

Types not supported by Templater (custom containers/monads or date types) can be converted into types which are recognized by Templater via plugins.

They are called by low level API before being passed to final replacement. They can be registered during library initialization.

[10] .NET also recognizes two dimensional arrays: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/arrays/multidimensional-arrays
[11] In similar manner as tags are not created by using replace("tag", "[[new tag]]") API

In Java there is low level converter registered by default - for converting awt images into Templater image data type. Similar conversion exists in .NET for converting System.Drawing.Image and Icon into Templater image data type. When library for legacy .NET is used, this conversion works out of the box. When library for modern .NET is used, this conversion must be activated during setup.

Resize

Another mutating API in Templater is resize which duplicates (when count > 1) or removes (when count = 0) parts of the document which contains all the specified tags.

Resize relies heavily on context detection - which is Templater specific way of detecting what part of the document should be affected by an operation. Unlike explicit imperative way to manage part of the document via start/stop commands and for loops, e.g.:

<for item in items> *${item.name}*	${item.price}	${item.total}
${item.description}		

</for>

Templater takes an approach of implied context based on the detected tags. Equivalent table looks in Templater as:

[[items.name]]	[[items.price]]	[[items.total]]
[[items.description]]		

This makes document much easier to design by non-developers and it reduces the amount of problems due to bad start/stop loop definition, since they are implied by the document structure and cannot be placed at an inappropriate location.

Special case of Resize(tags, 1) which neither duplicates nor removes the content is still useful since it's used to setup the relationship between tags which is used later on resizing only the subset of tags.

Region of the document which is affected by the resize is influenced by various features of the document it operates on, such as:

- lists
- tables
- sections
- repeatable content controls
- named ranges
- merge cells
- outline levels
- embedded documents

Resize returns boolean which is true if the tag was matched and resize could be performed (there are cases when resize can't be performed, such as requesting to do a resize on main Word document).

When complex documents are used with multiple collections for data sources, it can happen that the user sets up template in an incorrect way (either due to typo or misunderstanding of the behavior) which results in Resize not working on the expected context. While experienced developer with Templater will understand the problem by looking at the document, to help the average user Templater Editor has deep knowledge of rules and relationships and can guide the non-technical user towards the correct setup.

Clone

While Resize operates on the *best match* region of the document, clone always operates on the full document, or to be precise, part of the document managed by *ITemplater* instance, which for the default instance is the whole document[12].

It creates isolated low level *ITemplater* regions of the documents which can be independently modified, since they do not share tags once created.

With the various improvement to context detection Clone is used less and less, but still, sometimes it's useful to work around some current limitation of the Templater.

High level API

There is only a single high-level method and access to low level API. This way developer is encouraged to prepare the data for binding instead of trying to use low level API for manual document manipulation.

Process can be called multiple times, which is often the case even for a single argument object, since additional system wide info can be passed into every processing that way.

```
public interface ITemplateDocument : IDisposable
{
        ITemplater Templater { get; }
        ITemplateDocument Process<T>(T data);
}
```

Similarly to .NET version and the implicit Dispose()[13], Java version uses AutoClosable[14] try with resource pattern.

```
public interface TemplateDocument extends AutoCloseable {
    Templater templater();
    <T> TemplateDocument process(T data);
}
```

[12] Clone can be also called on previously cloned part of the document. Such operation will only affect part of the document, not the entire document

[13] .NET framework guidelines advise against doing much work in Dispose method, but this way there is no need for a Close method

[14] Prior to v6, Java Templater had flush API which was incorrectly named since it did not support multiple calls and thus should have been called close

Processors

High level API will analyze the provided object and call into appropriate processors. Additional processor can be registered during library initialization, but there are several built-in ones which cover wide area of use cases:

- object processor - works via reflection
  - will analyze class for public fields and methods without arguments
  - will call into other processors once navigation over property is detected
  - will respect internal **all** metadata
    - it instructs Templater that all tags should be replaced (this sometimes helps context detection which considers them unrelated)
- enumerable object processor - duplicates region of document with matching tags
  - best matching type is extracted from the values inside a collection - meaning signature can be an interface
  - types with known size will be processed as a whole (ICollection in .NET, Collection in Java and Seq in Scala)
  - streaming types will be processed in a streaming mode
    - chunk size can be configured in configuration API
    - streaming will only work on non-dictionary/map types. Templater must check if every element is a dictionary before it can process collection as a dictionary
  - will respect internal **clone** and **fixed** metadata
    - fixed metadata avoids call to resize and clears up any remaining tag after the processing
    - clone metadata calls into low level Clone instead of Resize while doing duplication
  - collection level methods (such as size) can be used (as long as they don't conflict with element level methods)
- dictionary processor - works on maps where strings are used as keys
  - can combine both dynamic keys and class fields/methods
  - will call into other processors once navigation over key is detected
  - supports keys with dot - which is not a navigation over keys
- enumerable dictionary processor - duplicates region of the document with matching tags
  - union of all keys is used to define context
  - will respect internal **clone** and **fixed** metadata
    - fixed metadata avoids call to resize and clears up any remaining tags after the processing
    - clone metadata calls into low level Clone instead of Resize
  - collection level methods often conflict with element level methods, so they are not as useful
- ResultSet/IDataReader processor - uses schema information for tag binding
  - works in streaming mode - resize is called in chunks[15] and then processed row by row

[15] streaming/chunking size can be configured during library initialization

    - respects the **fixed** metadata
        - will avoid call to the resize and clear up tags after the processing
    - calls into other processors when navigated over tag
    - supports multiple results sets (.NET only)
        - IDataReader does not support handling of unprocessed tags (it will not call into OnUnprocessed plugin) when in root prefix, but supports multiple result sets in that case. When value is available via some property and thus a navigation path, it supports handling of unprocessed tags but it does not support multiple result sets
- DataTable (.NET only) processor - uses schema info for tag binding
    - respects the **fixed** metadata
        - will avoid call to the resize and clear up tags after the processing
    - only a single resize is performed (when fixed metadata is not used)
    - calls into other processors when navigated over tag
- DataSet (.NET only) processor - uses schema info for tag binding
    - respects the **clone** metadata
        - will use Clone instead of Resize when doing duplication
    - supports relationships between tables
    - does not support handling of unprocessed tags (will not call into OnUnprocessed plugin)
    - will process tables based on their dependency order

Built-in processors support cancellation pattern and will quickly stop the processing if a token has been cancelled.

Processors only have access to low level API. There is no other internal API available to them. This makes them on same playing field as any of the custom processors registered during library initialization.

Object and dictionary processors can be combined: if class has both methods/fields and also implements dictionary (with all keys as strings) it will be processed for all tags (dictionary keys + methods/fields). When processed in such a way conflicting tags will be resolved to dictionary keys (in favor of methods/fields).

If same tag exists on a collection element and as a collection (e.g. Count/size), it will be resolved depending on the processor type:

- typed collection will always give preference to element methods
- dictionary collection will resolve tags in preference order:
    - dictionary keys
    - collection methods
    - element methods

Metadata formatters

Few built-in processors will iterate over metadata formatting plugins before passing value to low level replace API. This means that while direct call into low level Replace will not invoke certain plugins, call into high level Process will. An example would be:

tag = [[date]:format(YYYY)]

where we expect it to be replaced with a 4 digit year.

Call into **ITemplater.Replace**("date", date) will show the actual date, while the call to high level **ITemplateDocument.Process**(new { date = date }) will invoke the appropriate format plugin before passing it to low level API as a year value (of string).

Metadata handlers

Few built-in processors will iterate over generic metadata handler plugins before passing value to low level replace API. Unlike metadata handlers, they can stop further processing and instead do an unrelated change, such as removing tags from the document.

Collapse handlers are built into the library, but they can be disabled and/or custom handlers can be registered during initialization.

In practice explicit call to collapse is rarely needed, as Resize(tags, 0) should be called on empty collections. For advanced scenarios when tag sharing is used, Resize(TagPosition(tag, position), 0) can be used instead.

Navigation expressions

To allow for high degree of customizability v5 introduced navigation expressions for fine grained control over each part of tag navigation. While there are few built-in plugins, they can be disabled during initialization. In practice it is expected that user defined plugins take care of various customizations and specialization during path evaluation. Some common use cases for such expressions:

- [[items:at(2).name]] – access second element of the collection directly
- [[items:sortBy(name):top(5).description]] – display only first 5 elements sorted by name
- [[object:method(arg1, arg2).length]] – call arbitrary method on an object and provide custom arguments, since by default Templater allows navigation only over zero arguments methods
- [[items:top(5).person.responsibilities:sort(importance).description]] – use expressions multiple time during same path evaluation

Since expressions can become really user unfriendly, for improved experience they can be paired with aliases. E.g. alias can be defined for example as:

resp = items:top(5).person.responsibilities:sort(importance)

which would simplify the last example into [[resp.description]] which is much shorter and readable

Java/.NET differences

Due to Java erasure there are minor differences in behavior between Java and .NET implementation. When a collection is empty, unless the collection is an array Templater is unable to know the signature of the collection and has a hard time matching tag.

When this scenario happens during navigation, Templater will assume that all remaining tags are under current navigation prefix, but if this happens on top level tags, there are two basic approaches to the problem:

- automatic handling by using array instead of list
- adding collapse or some other metadata and handling it via OnUnprocessed API or at the end of processing
- manual handling by calling into the low level resize

In case of manual workaround code would look like:

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

where relevant tags would be specified manually. It's sufficient to pass in minimal number of tags which fully describe the affected area.

Processing document

Once *ITemplaterFactory* has been created it can be reused to create new *ITemplateDocument* for binding templates with data.

Open method on factory has several overloads:

- **Open**(string file) - will do a replace of the input file at the end of processing. This API is mostly used in some narrow cases and others, stream-based APIs should be used instead
- **Open**(InputStream input, String extension, OutputStream output, CancellationToken cancelToken) – should be used most of the time. Templater will not close provided streams; it will only flush to output at the end of processing. If CSV streaming is performed, Templater will also flush to output stream while doing Resize operations

Templater processing is CPU intensive and should not be done concurrently as *ITemplateDocument* is not parallelism friendly[16]. It intended to be used in a sequential manner. The only non-CPU intensive operation is when ResultSet/DataReader as being streamed/chunked, but even that is mostly CPU intensive (as data preparation will mostly be done in the background before processing starts).

[16] It is perfectly fine to concurrently process two different Templater requests for different instances of *ITemplaterDocument*. It's recommended to limit number of concurrent processing to somewhat below CPU count and check/guard for memory usage.

Process method returns *ITemplateDocument* to indicate that operations can be chained one after another. All operations on *ITemplateDocument* must be executed sequentially[17]. A common pattern in processing is to first process all fixed/static inputs and then send it the large collection, e.g.:

```
ITemplateDocument doc = ...;
doc.Process(new { filter = filter })
   .Process(new { system = new { user = username, at = DateTime.Now } })
   .Process(mainData);
```

Configuration

While there are various built-in plugins, for Templater to be truly useful in wide set of scenarios there must be a way to user defined behavior to be plugged in and customize the processing.

Therefore, Templater has several extensibility points for customizing the behavior and widening the feature set of the library. During library initialization such behaviors can be defined on *DocumentFactoryBuilder* accessible from the *Configuration.*builder() API.

Formatter plugins

Custom formatter can be registered via

```
DocumentFactoryBuilder include(Formatter formatter);

interface Formatter {
    Object format(Object value, String metadata);
}
```

where built-in plugins will iterate over all registered formatters (first custom, then built-in) and ask them to handle the value and metadata.

If there is no user defined metadata on the tag, this API will not be called. If there are multiple metadata on the tag iteration over them will repeat once for each metadata on the specific tag. There is alternative low-level API to handle cases when formatting should be performed even when there is no metadata.

Plugins are expected to match the metadata argument and return the formatted value when appropriate; otherwise they should just return the original argument. An example of the implementation looks like:

```
object Format(object value, string metadata)
{
  var ie = value as IEnumerable;
  if ((value == null || ie != null) && metadata.StartsWith("empty("))
  {
    var str = value as string;
    if (value == null || str != null && str.Length == 0
        || str == null && !ie.Cast<object>().Any())
      return metadata.Substring(6, metadata.Length - 5);
  }
```

[17] Since processing is CPU intensive it is not advisable to use async/await patterns or to call it from UI/IO thread

```
  return value;
}
```

This is a built-in plugin for **:empty**(value is empty) metadata. When provided value is null or empty it will show instead string within the metadata, which in this case is: *value is empty*.

The plugin matches the provided metadata and the type to check if it should be invoked.

Once invoked it checks if the value should be replaced and returns a different value. The returned value goes into next plugin for processing and so until all plugins are iterated over. This way value formatting can be chained across several plugins (in case of some complex transformation).

A common case in financial document creation is showing numbers as text. While it's fine to verbalize specific values in the domain, via a verbalize plugin conversion can be attached to any value by just marking it for verbalization, eg: [[invoice.total]:verbalize]

Instead of writing code for the actual verbalization, it's common to call into third party libraries for such conversion[18].

There are several built-in formatter plugins:

- bool(Yes/No) and bool(Yes/No/Unknown) – useful for converting true/false/null into an appropriate message
  - example: for bool(Yes/No) false will be converted into: No
- format for Date (DateTime in .NET and java.util.Date in JVM) – will convert input value into short date format
- format(PATTERN) for Date – will convert input value into output using PATTERN. This is language/platform dependent string formatting of date (using SimpleDateFormat in JVM)
  - format(yy) will convert input value into 2 year digit format `2012-03-05` -> 12
- format(PATTERN) for other values – will convert input value into output using PATTERN. This is equivalent to calling String.format on value with provided pattern. It is language/platform dependent
  - format(N2) in .NET will convert number into two decimals string
  - format(%.2f) in JVM will convert number into two decimal string
- empty(MESSAGE) for string and collections – when input is empty or null, MESSAGE will be displayed in the output
- join(SEPARATOR) for collections – will combine collection elements with SEPARATOR in between elements
  - int[] { 1, 2, 3} with join(-) will be converted into string 1-2-3

[18] C# example can be found at: https://github.com/ngs-doo/TemplaterExamples/blob/0b21f41fb97163c0895100bc7114d169aa5d4c89/Intermediate/CollapseRegion/src/Program.cs#L99
Java example can be found at: https://github.com/ngs-doo/TemplaterExamples/blob/0b21f41fb97163c0895100bc7114d169aa5d4c89/Intermediate/CollapseRegion/src/main/java/hr/ngs/templater/example/CollapseExample.java#L113

- offset(EXPRESSION) for Date – will append specified offset to the actual Date value
    - this is language specific, as .NET will use TimeSpan to offset, while JVM supports only days
- padLeft(LENGTH) and padRight(LENGTH) – will append space on string representation of the value so that there are at least LENGTH number of characters with space populating left or right side of the string. This is useful for fixed length format when values need to be padded to meet the specification
    - 12345 with padLeft(8) will create output of: `  12345`
- padLeft(LENGTH,CHAR) and padRight(LENGTH,CHAR) – is similar to previous plugin, but instead of space custom character can be defined
    - 12345 with padLeft(8,0) will create output of: `00012345`
- substring(START) – is useful for cases when substring(N) needs to be called on the value. If START is greater than input length, empty string will be returned
- substring(START,LENGTH) – is useful for cases when substring with start and length arguments needs to be called on the value. Note that this implementation is aligned with substring behavior on .NET and is same across languages, so even JVM uses substring(START,LENGTH) pattern instead of Java default substring(START, END)
    - `Mr. Rodgers` with substring(0, 2) will be converted into `Mr`

Metadata handler plugins

Custom handlers can be registered via

```
DocumentFactoryBuilder include(Handler handler);

interface Handler {
    Handled handle(Object value, String metadata, String tag, int position,
Templater templater);
}
```

where Handled result is an enumeration of action which have happened in the plugin:

```
public enum Handled {
        NOTHING,
        OTHER_HANDLERS,
        THIS_TAG,
        NESTED_TAGS,
        WHOLE_OBJECT,
        CURRENT_CONTEXT;
}
```

While metadata formatter can transform input value, there are cases when more complex transformation must be performed. One such case is collapse (removal) of region in case of certain condition. There are few such built-in plugins:

- collapse -which will invoke resize 0 for the specified tag when it's null or empty
    - this is often useful to hide part of the document since the object is not available and thus document can't be populated with actual values

    - when collapse is encountered, Templater will execute templater.resize(tag, 0) on the tag which had the collapse plugin
    - works with tag sharing mode, by calling appropriate resize API when explicit position is specified
- collapse-to(other tag) - which will invoke resize 0 on two tags (original and one in argument)
    - in non-trivial documents a certain region of the document needs to be removed and it's not fully defined with a single tag. In that case specifying two tags for start and end of removal is often sufficient
    - when collapse-to is encountered, Templater will execute templater.resize(new string[] {tag, other-tag}, 0) where the tag represents the location of the collapse-to.
    - only works in non-sharing mode. If tag sharing is used handler will not be executed
- collapse-nested - will invoke resize 0 with all tags under specified path
    - most of the times tags which should be removed are nested under the specified path
    - when collapse-nested is encountered, Templater will execute templater.resize(all tags with the same prefix, 0) where tags with the same prefix will depend on the tag which has the collapse-nested defined
    - only works in non-sharing mode. If tag sharing is used handler will not be executed

Collapse pattern is Templater way of performing IF condition in documents. If region is not needed it can be removed during processing. Common alternative to *collapse* is a custom ***showIf(VALUE)*** plugin which works similar to collapse, just with inverse logic (keep this section if provided value matches the argument).

As with metadata formatter plugins, metadata handler plugins are expected to match the provided metadata and object value before invoking specific actions. With different Handled values, several options are available for continuing processing:

- Nothing[19] – indicates that plugin is not applicable and next plugin can be invoked
- OtherHandlers – should be used when plugin is matched, but ordinary processing can continue (replace operation on this tag and processing of nested/other tags). Primary use case for this option is to prevent other handlers to run
- ThisTag – when plugin is matched which resolves only current tag. Processing of nested tags can still continue
- NestedTags[20] – should be used when plugin has taken care of this and all nested tags (nested tags are tags which have the same navigation prefix, eg: [[prefix.name]] is a nested tag for [[prefix]] tag
- WholeObject – when plugin is applicable to whole object, not just nested tags, this return value can be used to skip processing of other tags in this level
- CurrentContext – should be used when other tags in same context are also handled by the plugin, so further processing of same tags in this context should be skipped

[19] Prior to v5.1 this was the behavior of returning false
[20] Prior to v5.1 this was the behavior of returning true

Example implementation of the collapse plugins looks like:

```
Handled Handle(object value, string metadata, string tag,
int position, ITemplater templater)
{
  if (metadata.Equals("collapse"))
  {
    var ie = value as IEnumerable;
    if (value == null || ie != null && !ie.Cast<object>().Any()
        || value is bool && (bool)value)
    {
      var resized = position == -1
        ? templater.Resize(tag, 0)
        : templater.Resize(new[] { new TagPosition(tag, position) }, 0);
      if (resized) return Handled.NestedTags;
    }
  }
  return Handled.Nothing;
}
```

The plugin checks if it matches the metadata, in which case it checks for null values, empty collections or true value for boolean property. When either of that is matched it invokes resize 0 on the low-level API which returns indication that it has process this and all nested tags when resize was performed causing stoppage of the current tag (and nested tags) processing.

While collapse plugin is sometimes useful, it's better to structure types and documents in a way so that collapse plugin is not required. In Word this is often done by using sections and having tag in the root of the document between sections. This way removal of the tag will use surrounding sections as boundaries for the operation, instead of having the same behavior implemented with collapse-to plugin.

For Word, since v6.1, sometimes a viable alternative to removing sections is to use embedded documents as a way to add custom sections during processing.

Processor plugin

Custom type processor can be registered via

```
<T> DocumentFactoryBuilder include(Class<T> manifest, Processor<T> processor);

interface Processor<T> {
    boolean tryProcess(String prefix, Templater templater, T value);
}
```

In rare cases when built-in processors can't handle specific type (either due to non-public visibility, custom naming rules or some other reason) a custom processor can be registered. All the built-in processors use the same API, to provide complex behavior.

Several examples of custom processor plugin can be found on Github, such as questionnaire example.

Low level replacer

While formatter plugin requires a metadata to match and is only invoked from high-level API, sometimes it's more useful to always to a type transformation. In that case a plugin for low level API can be registered via

```
DocumentFactoryBuilder include(LowLevelReplacer replacer);

interface LowLevelReplacer {
    Object replace(Object value, String tag, String[] metadata);
}
```

Low level replacer is invoked on every value sent to **Replace** method. Prior to Templater v6 Java version did not have support for Optional<X>. With v6 this is now built in and not necessary, but an implementation which could have added support for Java 8 Optional would look like:

```
public Object replace(Object value, String tag, String[] metadata) {
  return value instanceof Optional ? ((Optional)value).orElse(null) : value;
}
```

Replacers will be iterated in a similar way formatters are iterated over; result of previous transformation will be passed to next replacer until the final value is sent to the actual processor for final replacement.

While tag and metadata arguments are often not used, they were introduced to provide all information to user defined plugins. String array metadata argument represents user defined metadata for the tag which is currently being processed.

There are many use cases for low level replacers:

- having a default formatting for Dates into string, while format(PATTERN) plugin can still be used when different formatting of date is required.
- using custom locale for converting numbers. By default, JVM uses DOT (.) always when converting numbers into strings, while .NET uses locale dependent settings. If DOT needs to always be used in .NET, appropriate matching can be done with .ToString(CultureInfo.InvariantCulture) being called on the value
- converting rich types into images or XML. If special structure is used for image definition, e.g. map with special keys, appropriate steps can be taken to transform input into relevant format, such as base64 parsing and image loading
- quoting can be applied on strings when working with CSV

Navigation metadata separator

Navigation plugins are disabled by default, but can be enabled if metadata separator is defined via

```
DocumentFactoryBuilder navigateSeparator(char character, NavigationEnd
findEnding);
```

```
interface NavigationEnd {
    int endsAt(String input);
}
```

First parameter (character) is used to define start of navigation expression. Second parameter (findEnding) is used to find the ending of the remaining tag path. If second parameter is not provided it will try to match ending as parenthesis followed by navigation character or separator.

Simple implementation of find ending which would return first occurrence of either start of new navigation expression or next navigation over properties could look like:

```
input -> {
    int nc = input.indexOf("):");
    int ns = input.indexOf(").");
    if (nc != -1 && (nc < ns || ns == -1)) return nc + 1;
    return ns != -1 ? ns + 1 : input.length();
}
```

Everything between this separator and specified ending will be recognized as navigation metadata. Escaping is not supported, but by providing custom navigation ending function, complex expressions can be supported. Navigation separator must be included in the tag regex, or Templater will not recognize such tags.

Navigation expression plugins

Custom navigation expressions can be registered via

```
DocumentFactoryBuilder include(Navigate expression);
```

```
interface Navigate {
    Object navigate(Object parent, Object value, String member, String metadata);
}
```

To enable navigation expressions, first navigation metadata must be defined. Once defined, navigation expressions can fill many roles and greatly expand the customization options. There are two built-in plugins:

- at(INDEX) – will return element at specified index (zero based) from the input list. If index is larger than the size of the list, null will be returned
    - this is useful when working with collections and needing to access specific element (e.g. first) without expanding the collection. Often when working with dynamic structures, the input data does not match the expected structure and document, so it's helpful to being able to work around the data limitations
    - [object1, object2, object3] with at(1) will return object2 instead of collection, which will avoid the resize operation and Templater will continue working with object2 instance
- top(COUNT) – will return first COUNT elements in the collection
    - this is useful for reusing same collection multiple times, e.g.: when presenting top 3 elements at one place and top 10 elements at another

While it's preferable that input data matches expected layout, so that there are not many customization options in tags, but rather tags are used as is, with navigation expression many other use cases are supported (e.g. using semicolon navigation separator):

- navigating over other methods with arguments
  - [[object:method(X).value]] – by registering appropriate plugin custom methods can be called which require additional argument - X in this case. Parsing of such arguments must be performed during plugin call, so they can be transformed into appropriate types
- sorting lists based on specified order, followed by limiting number of elements
  - [[items:sort(property):top(5).description]] – if sort is not known in advance, it can be performed later via plugin. Custom navigations can be chained together to perform more complex operations and they will be evaluated in order of definition

Embedding such info in the tags, can make them quite unreadable. In that case it's a good practice to introduce aliases to shorten the tags. Most formatting plugins could be replicated as navigation plugins to provide consistent experience, but formatters were introduced much earlier and do not require activation via navigation option.

On unprocessed tags handler

Prior to v3 mismatch between input and template could cause Templater to consume large amounts of memory due to calling resize on a tag which was never processed. Since this could not be resolved in a universal way (removing such tags would hide typing errors) a new API was introduced to finally resolve this issue. Now Templater by default will "process" those tags, by appending **:unprocessed** metadata at their end. This way they will not influence resize anymore and typos will be left in the document (although a new processing will be required to detect them).

If tags with :unprocessed metadata were detected during resize operation, they will be ignored and skipped over.

This issue was amplified in dynamic structures, where JSON often did not have certain attributes at all, so Templater could not handle them consistently (they were sometimes resolved as the value was null).

Unprocessed tags handling can be customized via

```
DocumentFactoryBuilder onUnprocessed(UnprocessedTagsHandler handler);

interface UnprocessedTagsHandler {
    void onUnprocessed(String prefix, Templater templater, String[] tags, Object
value);
}
```

In scenario when there are unprocessed tags, Templater will invoke this API with:

- current navigation path
- low level API instance

- all the unprocessed tags
- object value (instance where the tags were mismatched, or the parent instance when the tags were missing due to null value)

A simple implementation which just removes all such tags would look like:

```
void OnUnprocessed(string prefix, ITemplater templater, IEnumerable<string> tags,
object value)
{
        foreach (var t in tags)
                templater.Replace(t, string.Empty);
}
```

It's useful to have validation of templates before they are sent for processing (e.g.: on template upload) in which case it would be useful to either leave the default implementation, or replace it with one which will be aware if there are such tags in the document.

There are still some cases in which OnUnprocessed API is not invoked, such as processing IDataReader without a prefix. It's good practice to always put some prefix in front of objects which will trigger expected handlers on most data types, e.g.:

```
templater.Process(new { prefix = sqlReader });
```

Instead of just passing in value:

```
templater.Process(sqlReader);
```

Resize limit

Prior to OnUnprocessed API one of the main ways to protect against faulty templates was the resize limit (and built-in guards for Excel row limits). Resize limits specify how many times can a tag be resized. This translates to the maximum nesting level. The default is 8.

In practice, even the most complex templates have nesting level up to 4. Nesting level of 4 means that there is a tag nested 4 collections deep. In rare cases when there is more than 8 level of collection nesting, this limit can be increased to the appropriate value via

```
DocumentFactoryBuilder resizeLimit(int limit);
```

XML user defined plugins

While there are several built-in plugins for various kind of XML merging options, for fine-grained control a new plugin can be registered via

```
interface XmlCombine {
   Element[] combine(Element location, Element[] input, String tag,
String[] metadata);
}
```

```
DocumentFactoryBuilder xmlCombine(String metadata, RemovalOption
tagRemoval, XmlCombine combine);

enum RemovalOption {
    BEFORE,
    AFTER,
    MANUAL;
}
```

When XML type is detected during processing and XML metadata is matched (via templater-xml attribute or tag metadata) defined plugin will be invoked. Tag handling also allows for fine tuning via:

- before – tag will be removed before processing
- after – tag will be removed after processing
- manual – tag will be left inside XML and its up to plugin code to decide what to do with the tag within XML

For location XML representing the top-level document node matching the input node will be provided, which represents the current state of document in OOXML format, while the input will be XML detected during processing. If just a single XML object was provided array will consist from a single element. Additional tag and metadata arguments are provided to allow plugin full context information so its able to implement required logic.

Developer can implement merging logic as it suits him/her best, with having several aspects in mind:

- if location is returned in result array, XML where tag was detected will still remain, although adjusted to match the new definition (if it was changed at all)
    - elements prior to location will be inserted before and elements after the location will be inserted after
- if location is not returned in result array, it will be removed from document and provided array will be inserted in its place
- tags defined within provided XML will not be analyzed (if tags need to be analyzed, new processing will be required)
- custom templater-xml attribute will be removed from the output XML not to cause “file corruption”
- correct OOXML node elements must be provided. Templater will not validate XML nodes for OOXML correctness
- its expected to mutate location argument as Templater will check it for instance reference and sync provided value to original XML

Java XML custom setup

Templater will use default Java XML library unless configured to use some other library. Even Java 11 specific APIs will be used to setup the default library[21]. Android requires specific dependencies before it can work, as it does not have support for javax.stream out of the box. This can be fixed by adding some relevant dependency, eg: `stax:stax:1.2.0`. Templater will use specific order of initialization:

[21] Templater is Java 8 compatible so it uses reflection to call into Java 11 API when available

- if Java 11 API is detected, XML will be configured via new API
- if Java specific version of Xalan library exists Templater will try to initialize itself via it since this is the only supported XML library
- when neither of those are available, default XML library will be used. It should be noted that custom dependencies often register their own XML library which can cause problems during the startup

There are two aspects to XML initialization:

- passing in fully configured factories - no further configuration will be performed by Templater and factories will be used as is
- passing in default factory instances and let Templater further configure them
    - Templater will perform various configurations regarding namespace use, security setup (to prevent XML attacks) and namespace validations
    - if a factory fails to be configured due to unsupported feature, this can be override on call by call basis in the provided factory instance

APIs for configuration are:

```
DocumentFactoryBuilder xmlBuilder(DocumentBuilderFactory builder, boolean
isFullyConfigured) throws ParserConfigurationException;

DocumentFactoryBuilder xmlTransformer(TransformerFactory transformer, boolean
isFullyConfigured) throws TransformerConfigurationException;

DocumentFactoryBuilder xmlReader(XMLInputFactory readerFactory, boolean
isFullyConfigured);

DocumentFactoryBuilder xmlWriter(XMLOutputFactory writerFactory, boolean
isFullyConfigured);
```

Type visibility requirements

Templater can navigate over public fields/methods. Objects passed into processing don't need to be public, but only public properties will be used for navigation. Due to Java reflection changes, this will stop working in future Java versions unless class is public. This can be enforced by changing default configuration to prevent non-public type usage via:

```
IDocumentFactoryBuilder TypeVisibility(bool onlyPublic);
```

If enabled, tags in non-public classes will not be recognized. Its best practice to enable this configuration, since it prevents performance degradation due to reflection visibility changes.

Streaming size

Templater supports streaming in multiple ways:

- streaming data types (collections without a known size or database readers)
- flushing output during text processing

IDataReader and ResultSet will use streaming size to process rows in chunks of specified size.

If a collection is used which does no implement appropriate interface:

- ICollection in .NET (for Count property)
- Collection/Array in Java (for size method)
- Seq in Scala (for size method)

as long as elements of the collection are not IDictionary or Map they will be processed in chunks.

Default chunk size is 16384. Custom chunk size can be configured via

```
DocumentFactoryBuilder streaming(int size);
```

Built-in navigation, metadata and handler plugins

While formatters, navigation expressions and handlers are only invoked on matching metadata, if they are not useful, they can be disabled via

```
DocumentFactoryBuilder builtInNavigation(boolean include);
DocumentFactoryBuilder builtInFormatters(boolean include);
DocumentFactoryBuilder builtInHandlers(boolean include);
```

While their overhead is not significant, if there is no use for them, they can be turned off.

Built-in low-level plugins

Due to platform specific image conversions, low level plugin for their conversion is enabled by default. For performance reasons this can be disabled and Templater image type can be used instead

```
DocumentFactoryBuilder builtInLowLevelPlugins(boolean include);
```

By default System.Drawing.Image In .NET and java.awt.image.BufferedImage will be converted into Templater specific ImageInfo type. When working with Android or special .NET platforms this needs to be disabled.

Custom navigation character

Default navigation character is DOT (.). In cases when some other character is more appropriate (so tags are more readable) this can be changed via appropriate API:

```
DocumentFactoryBuilder navigateUsing(char character);
```

For example if navigation character is changed from DOT into SLASH (/) tag equivalent to: [[collection.item.description]] would look like: [[collection/item/description]].

Tag regex customization

Templater matches tags via regex which can be customized. There are three built-in tag patterns which are recognized:

- [[tag]]
- {{tag}}
- <<tag>>

First format can't be used on some places (such as Excel sheet names), but if Templater detection needs to skip over some tags they can be disabled by modifying tag regex for specific pattern via

```
DocumentFactoryBuilder withMatcher(String regex, TagPattern pattern);

enum TagPattern {
    /** [[TAG]] pattern */
    BRACKETS,
    /** {{TAG}} pattern */
    BRACES,
    /** <<TAG>> pattern */
    CHEVRONS;
}
```

Default regex pattern is: [-+@\w\s.,!?/:()|]+

To disable a pattern unmatchable regex can be used:

```
builder.withMatcher("[^\\S\\s]", TagPattern.CHEVRONS)
```

There are two similar methods for configuring matchers. Main difference is that one is a convenience API which will specify regex for all formats, while the other is for configuring regex per format. Since by default Templater will recognize only latin (and some extra characters) when a language specific matching is required, they can be enabled by providing relevant regex, e.g.:

```
builder.withMatcher("[a-z\u0400-\u04FF]+")
```

Java bean configuration

By default, Java beans are enabled, but if they need to be disabled this can be done via

```
DocumentFactoryBuilder withJavaBeans(boolean useConvention);
```

Exact method match takes precedence over bean naming, so there is little reason to disable bean naming.

Member blacklisting

While Templater supports navigation over zero argument methods, fields and properties, sometimes it's useful to disable navigation over certain fields or methods. This can be done via blacklisting API by registering all the methods on which the navigation is disabled. Common use case for such a feature is to disable navigation to sensitive information such as **getClass.getProtectionDomain.getCodeSource.getLocation** which could reveal some sensitive information in SaaS products. Blacklisting is done via

```
DocumentFactoryBuilder blacklist(Member member);
```

SVG fallback conversion

Microsoft Office 2016 introduced support for vector graphics via SVG standard. While image will be displayed in new Office versions as expected by default, it will not work in older Office versions without a fallback image. To provide a fallback image SVG conversion must be registered during

initialization which will convert SVG document into an image so Templater can include both new SVG XML and old image format in the document. There are various 3rd party libraries for SVG conversions[22]. Conversion is registered via

```
DocumentFactoryBuilder svgConverter(SvgConverter converter);
```

Where *SvgConverter* is type with relevant method:

```
interface SvgConverter {
    ImageInfo convert(Document svg);
}
```

In .NET conversion is done via similar API:

```
IDocumentFactoryBuilder SvgConverter(Func<XDocument, ImageInfo> converter)
```

ImageInfo type is Templater specific image type which has a builder API in Java. An example of initializing can look like:

```
return ImageInfo.from(os.toByteArray())
        .extension("png")
        .height(t.getHeight())
        .width(t.getWidth())
        .build();
```

Spreadsheet specific configuration

While most configuration options work across all file types, sometimes there is a need for options specific to file type. In case of spreadsheets there are two configuration options:

- Limiting number of created sheets
- Explicitly ignoring all formula warnings
- Explicitly ignoring all number as text warnings

This spreadsheet specific options can be accessed via

```
ISpreadsheetConfigurationBuilder ConfigureSpreadsheet();
```

during configuration setup.

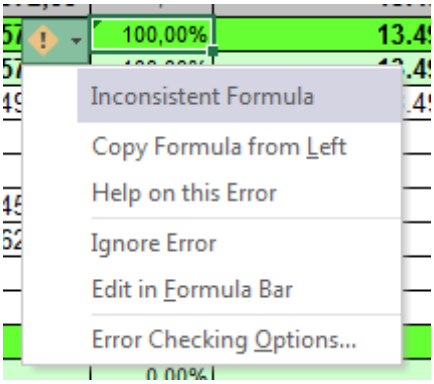
By default, there is no limit on number of newly created sheets, but when there is a need to restrict it, this can be done using

```
ISpreadsheetConfigurationBuilder NewSheetsLimit(int maximum);
```

If a resize operation detects that there were more newly created sheets than specified via this limit an ApplicationException/RuntimeException will be thrown with a relevant message.

[22] Pictures example: https://github.com/ngs-doo/TemplaterExamples/tree/master/Intermediate/Pictures contains code for SVG conversion via external libraries

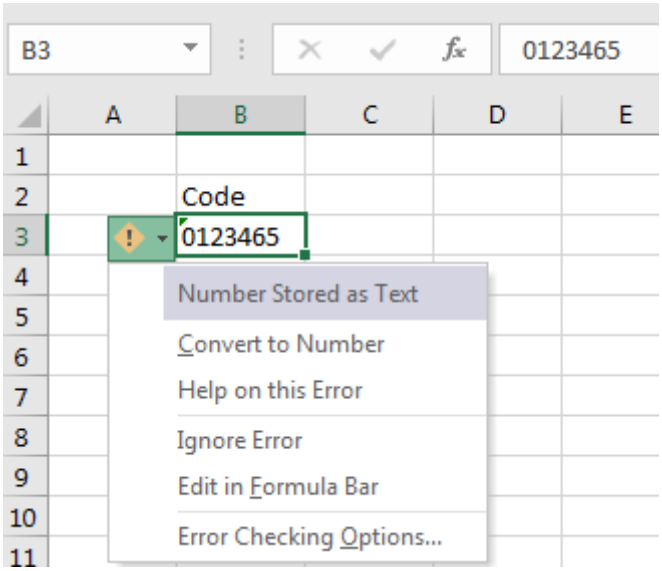
When formulas are moved around/adjusted they might end up in a state which Excel considers inconsistent. If this is a false positive, such warnings can be turned off via specific configuration option:




```
ISpreadsheetConfigurationBuilder FormulaWarnings(bool ignore);
```

In case when ignore is set to true, Templater will enable ignoring formula warning on all sheets containing formulas, which will prevent Excel from reporting warning on them.

Another common Excel warning is when text has the number entered as value. Since its common mistake that people put in numbers as text, Excel will often warn about this too.



To turn off this suggestion, there is relevant configuration option:

```
ISpreadsheetConfigurationBuilder NumberAsTextWarnings(bool ignore);
```

Document signing

Reporting team and Enterprise license allow for document signing. Office documents can be signed with user defined certificates. Common use case for signing would be to show document authenticity

by having proof of origin. Signature will be valid as long as signing certificate is recognized and document was not modified after the signing[23]. Signing requires certificate with private key information and is done via

```
DocumentFactoryBuilder sign(X509Certificate mcertificate, PrivateKey privateKey);
```

and in .NET via

```
IDocumentFactoryBuilder Sign(X509Certificate2 certificate);
```

If certificate is not recognized it will be shown as recoverable since it is not trusted.



Templater Editor integration

Reporting team and Enterprise license allow for Templater Editor integration. This is useful for embedding schema of the model into the document which then allows for tag listing within the Templater Editor. This greatly simplifies document preparation, as available tags are listed within the Microsoft Office, plus additional validations can be performed due to known schema and relationships. Templater Editor integration is configured via

```
EditorConfigurationBuilder configureEditor();
```

where specific configuration options can be set-up. At the end of the editor configuration it should be specified if schema is being embedded in the document or not via

```
DocumentFactoryBuilder configure(boolean embedSchema);
```

[23] Excel often recalculates cell values/tables/pivots which cause signature invalidation. For this reason Templater will change formula evaluation to manual after processing

When schema is being embedded into the document, instead of changing the document via resize and replace API, tag list will be constructed based on the calls to high level API and embedded into the document. When there is infrastructure for testing if the uploaded document is valid (checking if used tags are available), this can be reused for embedding schema into the document. When providing template to the user or administrator it's expected to first run schema preparation.

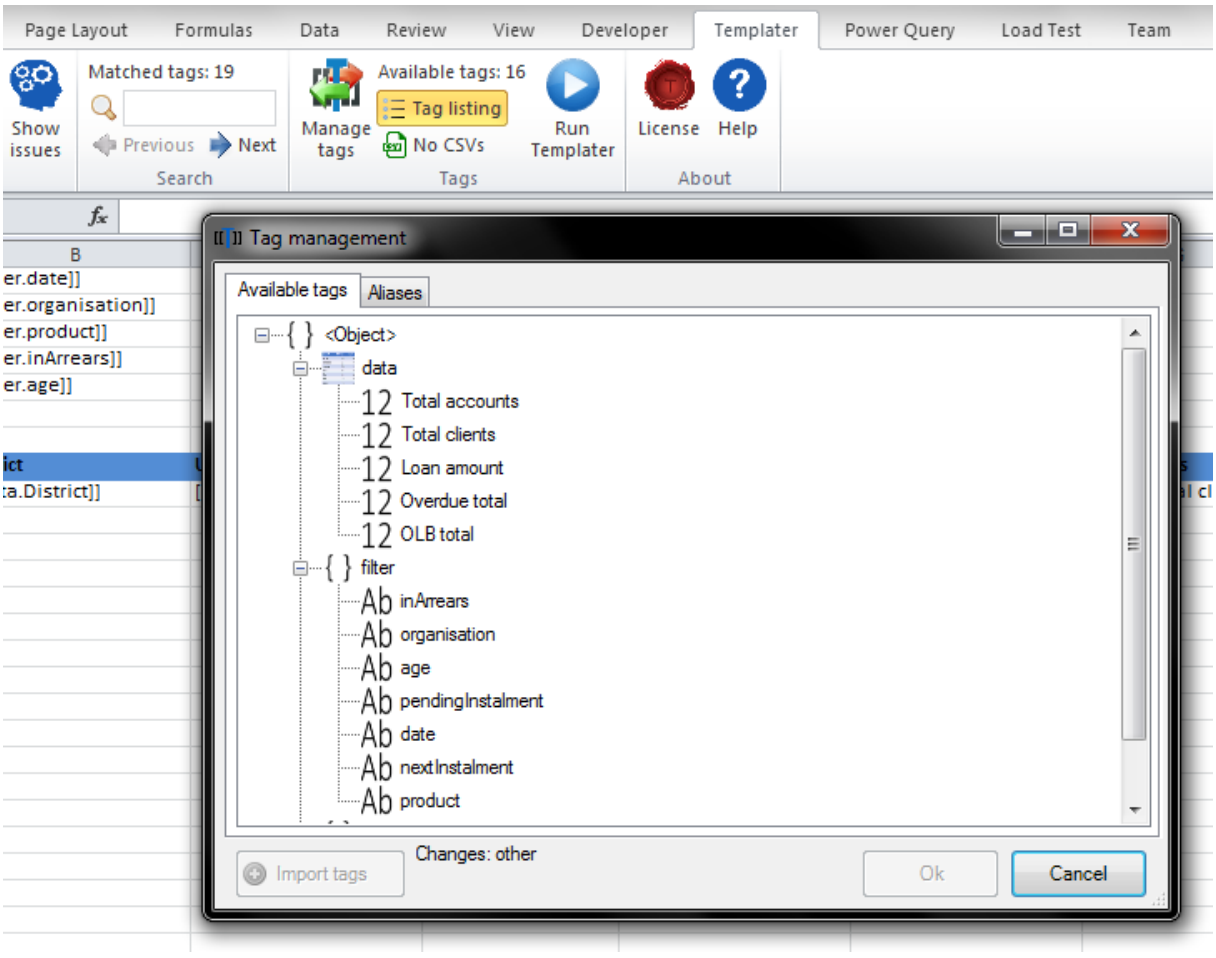
When schema is not being embedded into the document, previously defined schema will be removed and thus tag list will not be available. If Editor is not configured during initialization, schema will not be considered during processing, meaning if it exists in the document, it will remain in the document unchanged.

Tag management

Templater Editor allows for user defined tags via Manage tags button. This can be disabled via

```
EditorConfigurationBuilder tagManagement(boolean allow);
```

Once disabled, user can only work with previously defined tags, which ought to be embedded via embedding schema step. It is recommended to disable tag management when schema is being embedded. When tag management is not allowed, Import tags button will be disabled

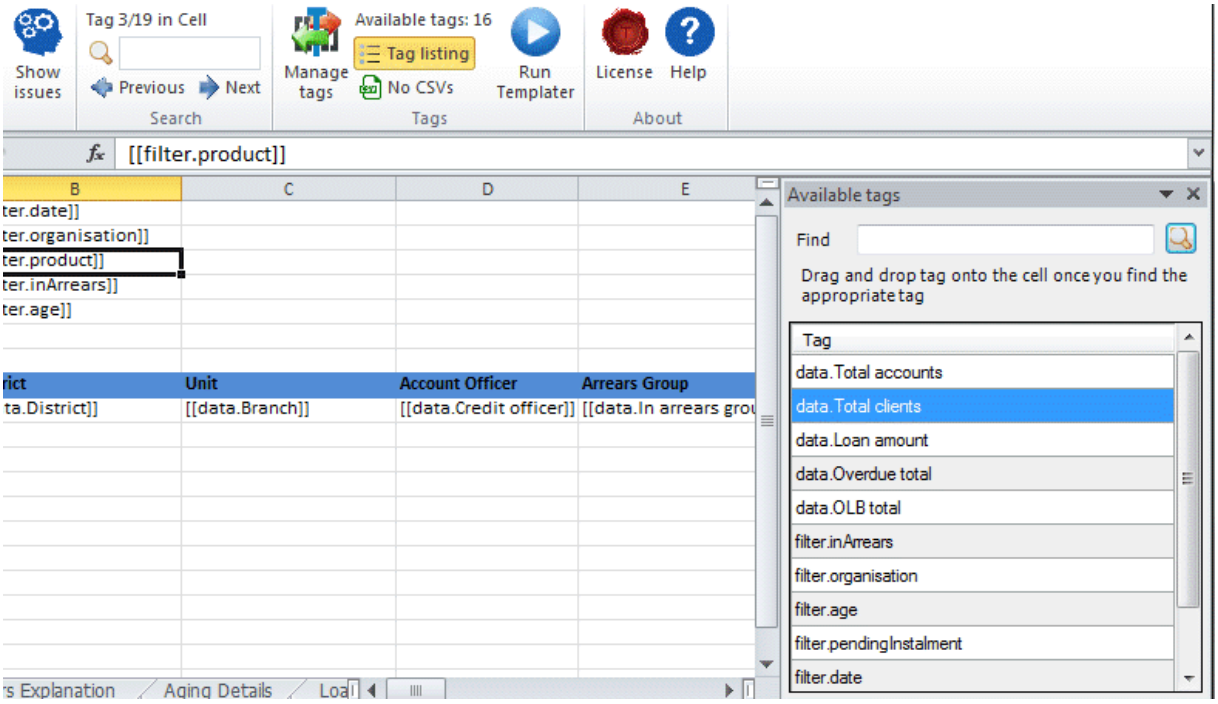


Tag listing

Templater Editor can list available tags within the Office editor, which allows for easy drag-drop from the list onto the document. This reduces the possibility for errors and allows exploration of which tag is available. This can be shown via

```
EditorConfigurationBuilder tagListing(boolean show);
```

Once enabled, user will see available tags pane which also have search capabilities for easier location of tags when there are many tags to choose from. It is recommended to enable this option.



Tag detection

Tags in templates are only recognized when tag detection is enabled. When enabled Templater Editor will continuously scan document on changes to check if there are new tags. For large documents this can take some time, but for templates it should be rather quick. This can be enabled via

```
EditorConfigurationBuilder tagDetection(boolean enabled);
```

Once enabled, Templater Editor will analyze document for tags, report how many tags were detected in the document and allow navigation over tags. When paired with issue detection, it will provide excellent user experience as problems will be detected and explained so that user is able to resolve them. Best practice suggestion and common warnings will also indicate possible improvements in the template. It is recommended to enable this option.

Issue listing

When tag detection is enabled, issues detected by Templater Editor can be listed so they can be addressed. Issues will be listed when enabled via

```
EditorConfigurationBuilder issuesListing(boolean show);
```

Templater will scan documents for various known issues. Some are common, while some might be rather specific. Issues will be listed in the Issues pane so that user can address them. It is recommended to enable this option.

Debugging integration

Templater Action Log can be viewed in the Templater Editor. By default, debugging is disabled, but can be enabled via

```
EditorConfigurationBuilder debugLog(boolean capture);
```

Templater will embed entire action log which can be replayed step by step in the Editor. When this option is activated, output document will look like the original template, but it will contain action log of all captured operations.

Alias definition

Tag aliases can be defined via Templater Editor. When aliases need to be defined for a template, this can be done via

```
EditorConfigurationBuilder addAlias(String prefix, String alias);
```

Using aliases simplifies tag usage as it can turn long cryptic tags into something short and readable. It is more common to define such aliases in the template itself, but when appropriate this can also be done via API.

Tag metadata resolution

While ideally tag names should convey the purpose, often it is not clear what they mean and when they should be used just by reading tag name. To make tags easier to consume tags can be extended with additional metadata:

- Status of the tag – by default tags are active, but if some tags need to be removed in the future, they can be marked as deprecated. Inactive tags will not be shown in the tag listing, unless specific option is enabled to list them (which is disabled by default)
- Type of the tag – while Templater can pick up tag types in most cases, there are cases where it does not have enough information to provide actual type used in the application. For such cases actual type can be sent via metadata which will be used instead of the detected one
- Description – longer explanation of the tag purpose can be provided via description metadata. This is very useful to explain when to use it, how it should be used and various other application specific information

- Example – example values for the tag can be shown via example metadata. While this information can be included in the description, this metadata is specific for the purpose of showing example data
- Category – tags can be grouped in different categories which should simplify exploration of tags and location of the appropriate tag

Metadata provider is registered via

```
EditorConfigurationBuilder metadataResolver(MetadataProvider customResolver);
```

```
interface MetadataProvider {
    Map<String, TagMetadata> lookup(Object source);
}
```

and should provide dictionary of relevant metadata for properties/attributes of the requested source. Source can be various things:

- signature of the class
- table metadata
  - ResultSetMetadata/DataTable
- instance of the object being analyzed
  - ResultSet/IDataReader/DataTable
  - Dictionary/Map
  - Collection

During analysis Templater will call into MetadataProvider and when metadata is returned will be used instead of detected one (such as type signatures or property status) or will complement information (with descriptions, examples, etc…)

Metadata can be defined via builder API:

```
val tableInfo = new MetadataProvider {
 override def lookup(source: Any): util.Map[String, TagMetadata] = {
  source match {
   case ResultSetMetaData =>
    val desc = new util.LinkedHashMap[String, TagMetadata]()
    desc.put(
      "Col B",
      TagMetadata.builder()
       .status(TagStatus.DEPRECATED)
       .actualType(classOf[Integer])
       .example("100")
       .build())
    desc.put(
     "Col C",
     TagMetadata.builder()
      .actualType(classOf[BigDecimal])
      .example("100.0")
      .description("decimal number")
      .build())
    desc
```

```
    case _ =>
     null
   }
  }
}
```

This is considered highly advanced usage of the integration, but provides the best experience for the user.

Templater Editor for Microsoft Office

While Templater does not need special editor, as templates can be prepared in any Office editor which supports Open Office XML format, such as Microsoft Office or LibreOffice, using Templater Editor provides additional safety from within familiar Microsoft Office interface. It's also very easy to test Templater behavior directly from within Word, Excel and PowerPoint by just passing in data via Templater Editor user interface.

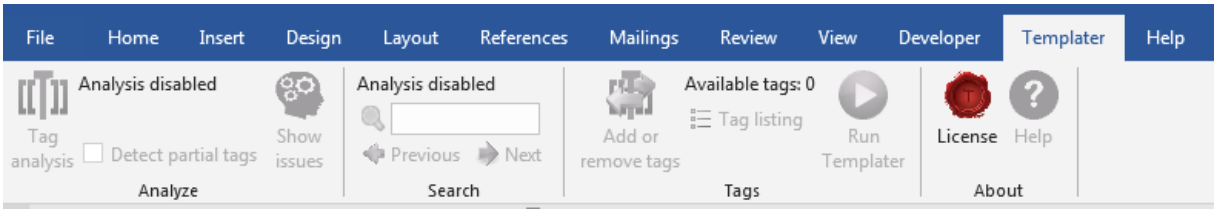
Templater Editor works on any Microsoft Office 2007 or later version which runs on Microsoft Windows.

Installation and licensing

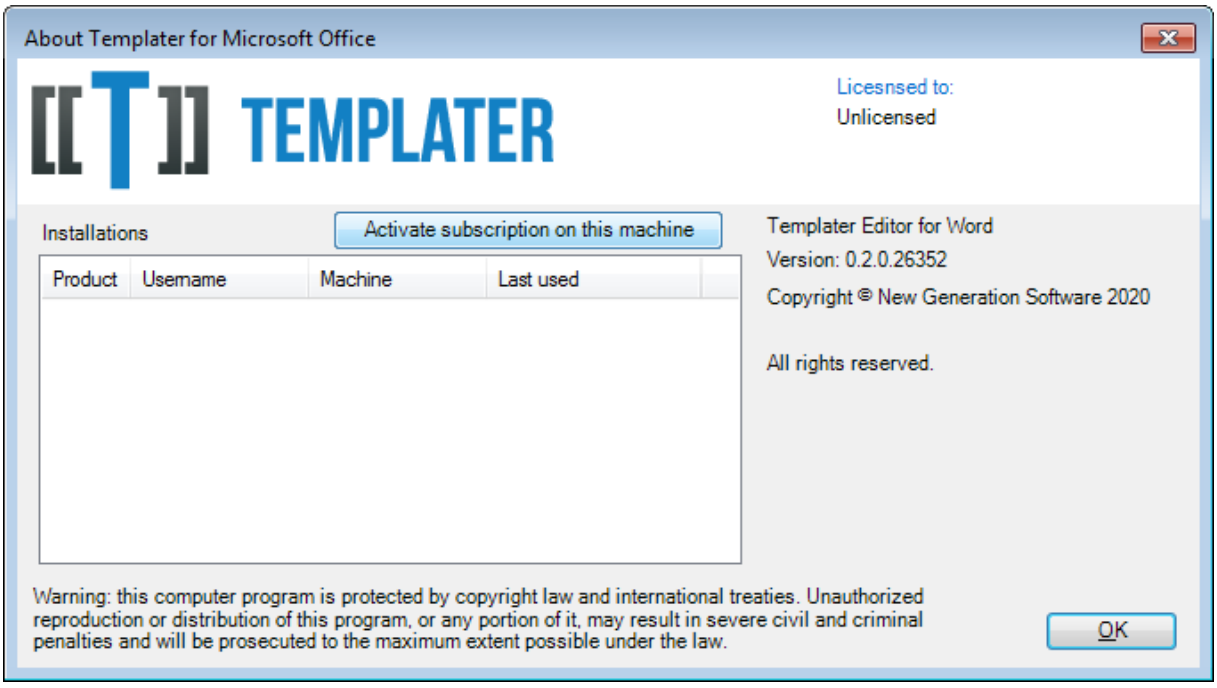
Templater Editor has a separate subscription license which is available only for customers with Reporting Team or Enterprise licenses with active support subscription. Templater Editor can be installed via download links on Downloads page:

- Microsoft Word Add-In
- Microsoft Excel Add-In
- Microsoft PowerPoint Add-In

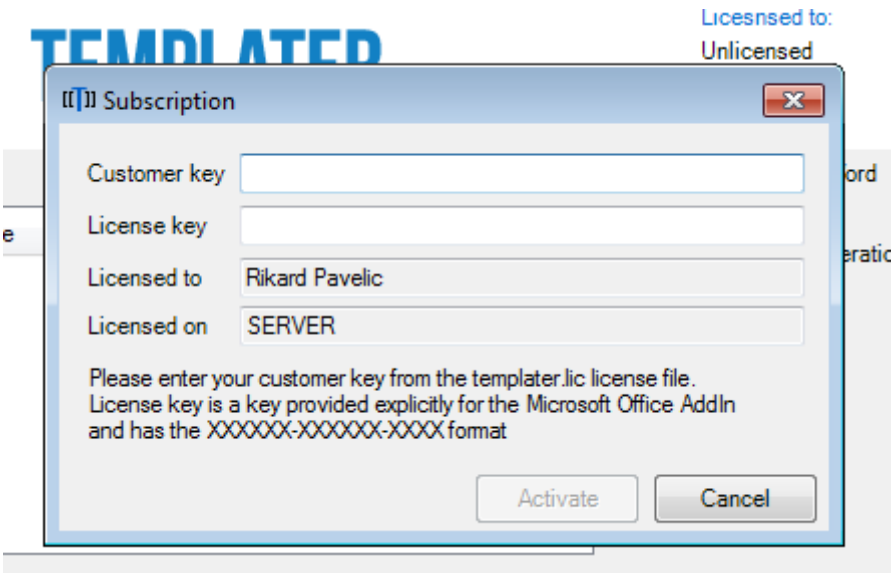
Once installed, a new tab will appear in Office ribbon:



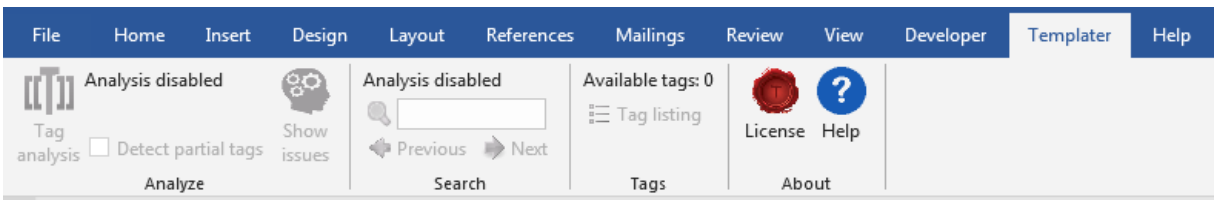
Only License button will be enabled at that point, until a valid license is entered in the Dialog which appears after pressing the License button:



To activate the license, press the Activate subscription button and enter relevant license information:



If correct license information is entered, Help button will be available, with others enabled depending on the opened document:



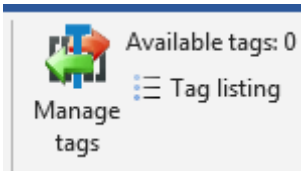
Templater Editor needs to be connected to Internet to activate the subscription. It does not require active Internet connection after, but it needs to validate the license at least every 30 days via active Internet connection.

If support is renewed, Templater Editor will prolong expiration date for 1 year until next support end date. If support is not renewed, Templater Editor will stop working after expiration date.

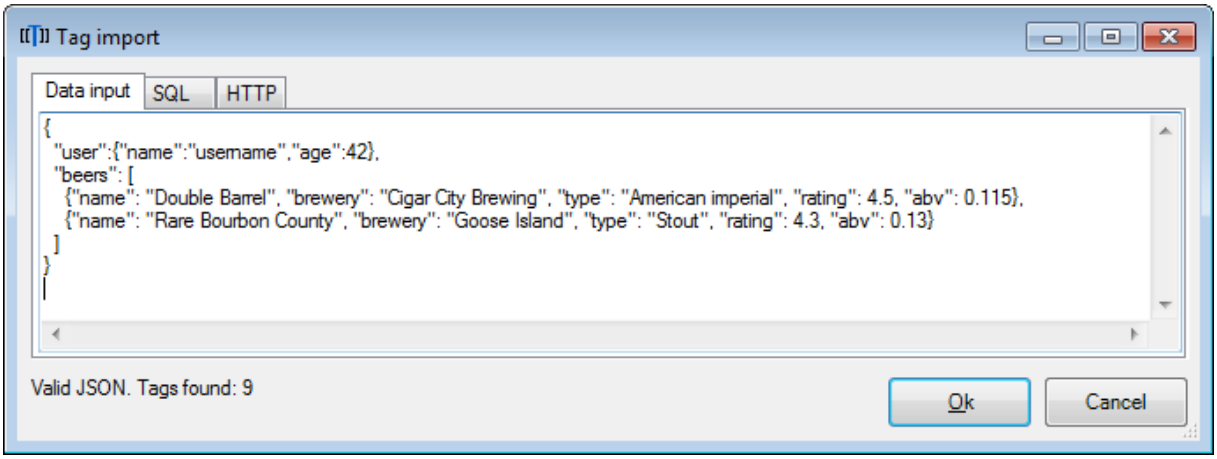
Schema

To get the most of the Templater Editor, a known schema is required. While Templater does not require schema upfront before processing, it does discover one during processing which can be used to embed such information into the document.

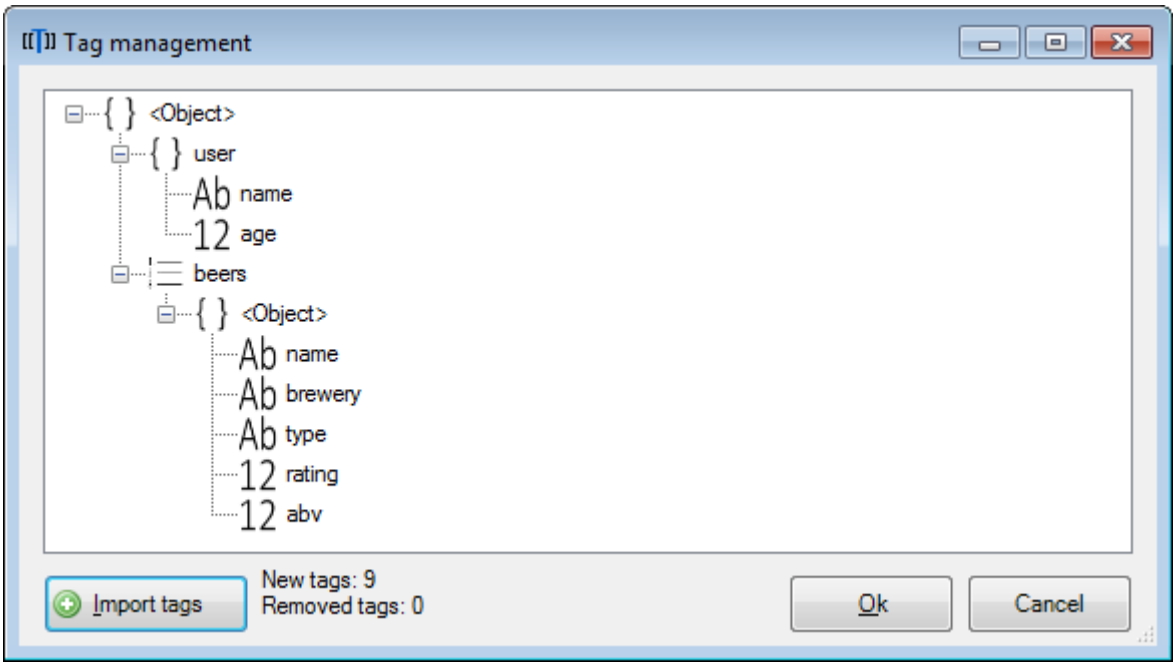
Schema can be defined manually via **Manage tags** button:



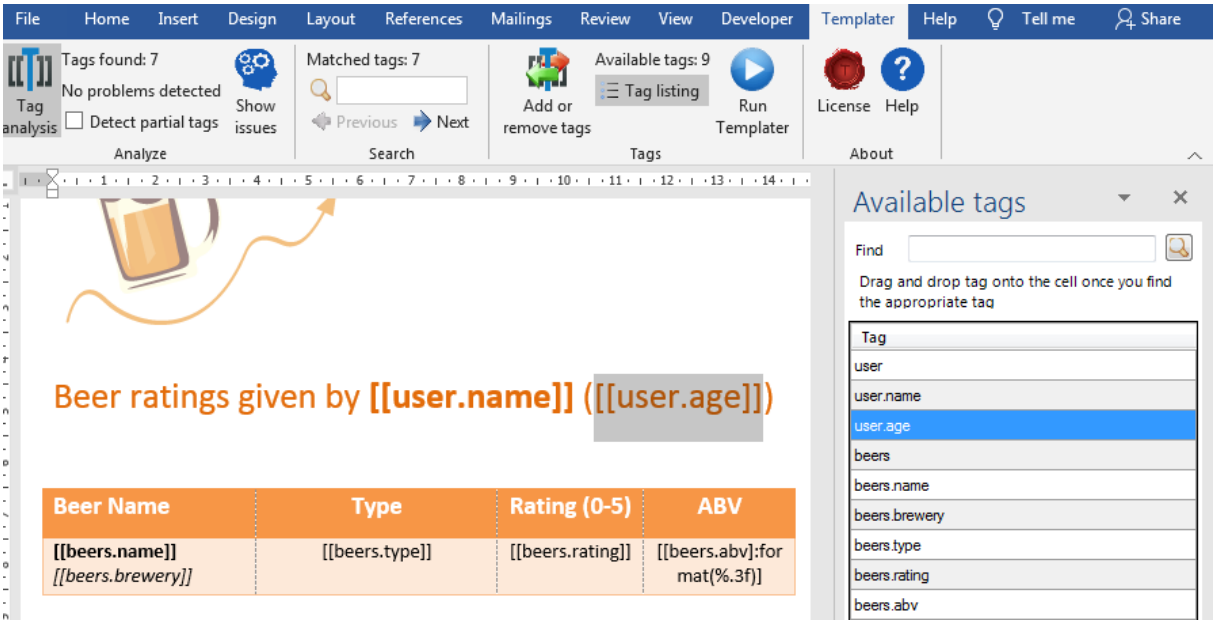
after which new tags can be defined via **Import tags** button. The easiest way to define schema/tags is to write relevant JSON into the editor:



After which they can be added to existing schema by confirming import:



Known tags can be listed via Available tags pane by toggling **Tag listing** button:



Listed tags can be dragged dropped onto the document for easy template setup[24].

Schema builder

While schema can be defined manually, it is expected that schema is defined via Templater integration using configuration options in the builder API.

To embed schema into the document Templater configuration needs to be configured via

[24] Some controls (such as WordArt) do not allow drag-drop, so old method of writing tag manually must be used

```
EditorConfigurationBuilder configureEditor();
```

which needs to be setup for schema embedding via

```
DocumentFactoryBuilder configure(boolean embedSchema);
```

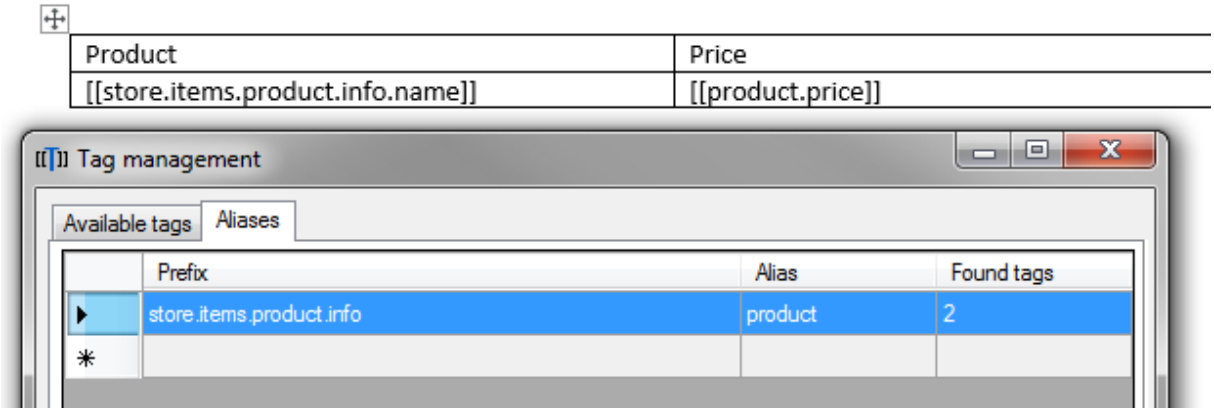
Suggested way to setup such factory is to have separate factory configuration specialized for embedding schema into the document which would look something like:

```
DocumentFactory factory = Configuration.builder()
    .configureEditor()
      .tagListing(true)
      .configure(true)
    .build();
```

Reference on how to setup schema embedding can be found in the Github examples.

Aliases

Templater Editor allows alias definition which is useful when dealing with long tags (or tags with long navigation) in Word tables. By default, table cell will try to accommodate all text within the cell, which often means will stretch too wide.



To work around this problem, a shorter tag can be used which will be auto-expanded by Templater during processing.

Tag aliases are especially useful when dealing with complex tag paths which leverage navigation expressions. Instead of writing explicit tags which includes sorting, limiting and various other options, e.g.:

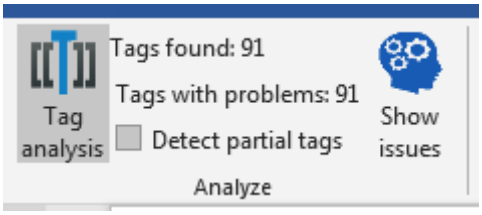
[[store.items:sort(price):top(10).product.info.price]]

It is much easier to work with tag such as:

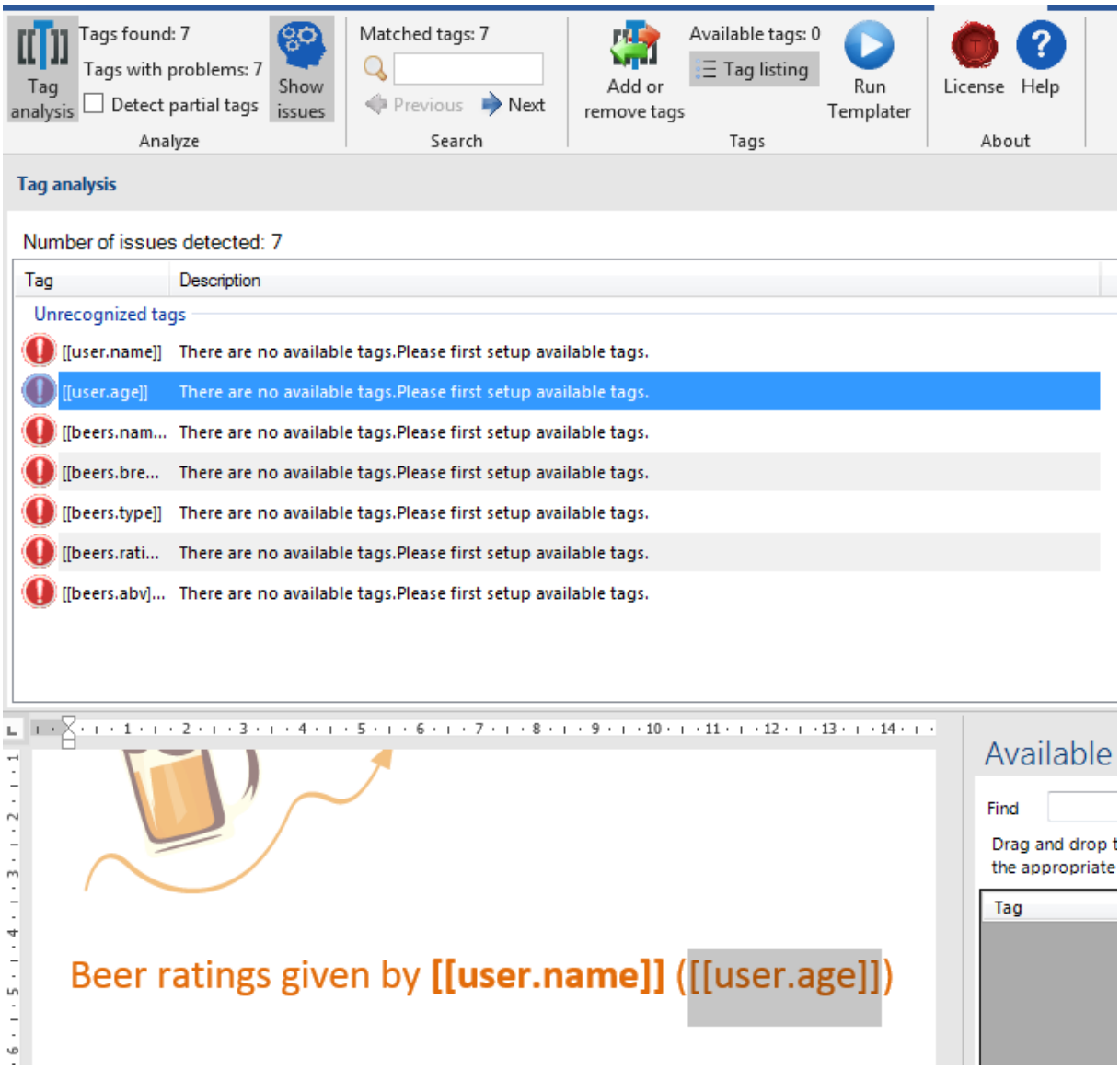
[[product.price]] which hides the technical details of sorting and limiting only first 10 elements behind the alias.

Tag analysis

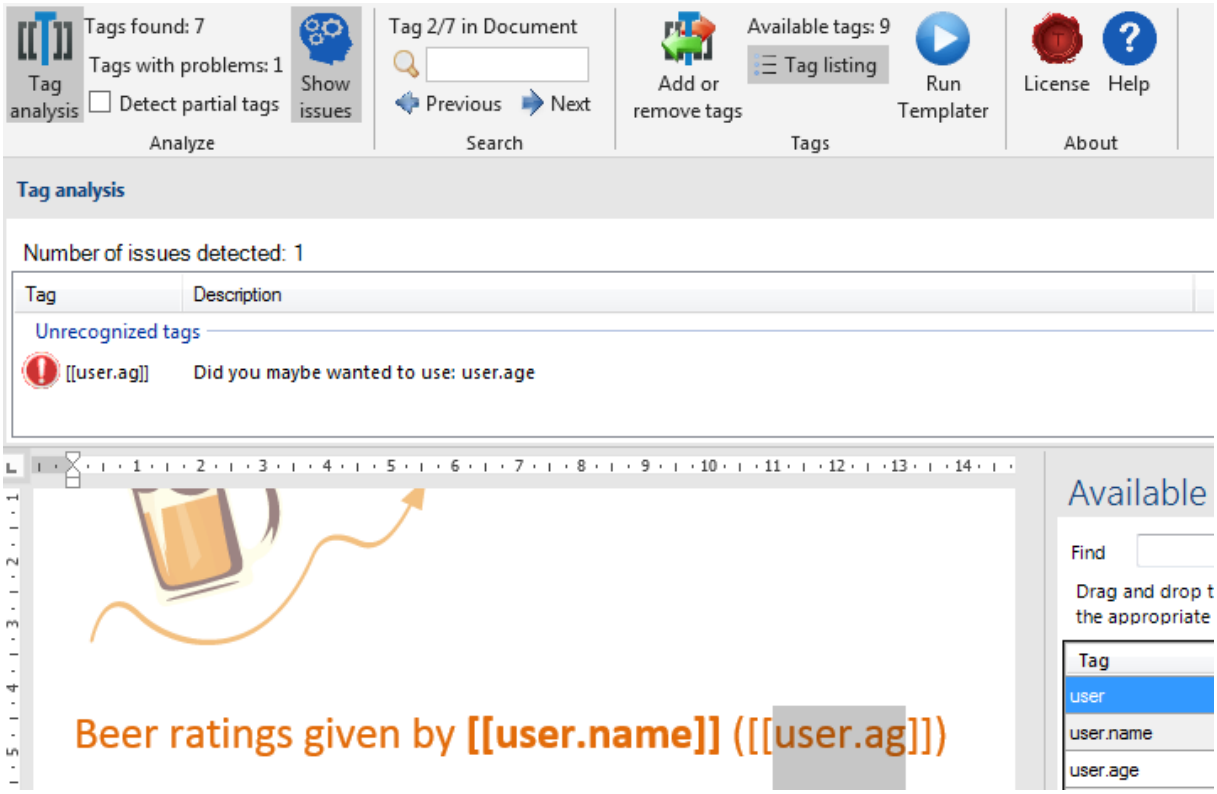
Main feature of Templater Editor is tag analysis and issue detection. This is available via first group in the Ribbon menu. If Tag analysis is enabled, Templater will continuously check document for tags and various issues with tag setup:



For analysis to be truly useful Templater must be aware of schema which is available. Otherwise all detected tags will be reported as unknown:



With schema, only actual issues will be reported, such as typos:



Templater Editor will recognize all kind of issues and suggest resolutions to them.

When **Detect partial tags** checkbox is enabled, Templater will also perform search for common tag typos, e.g.: writing [[user.age] with last bracket missing, or even [user.age] with both first and last bracket missing.

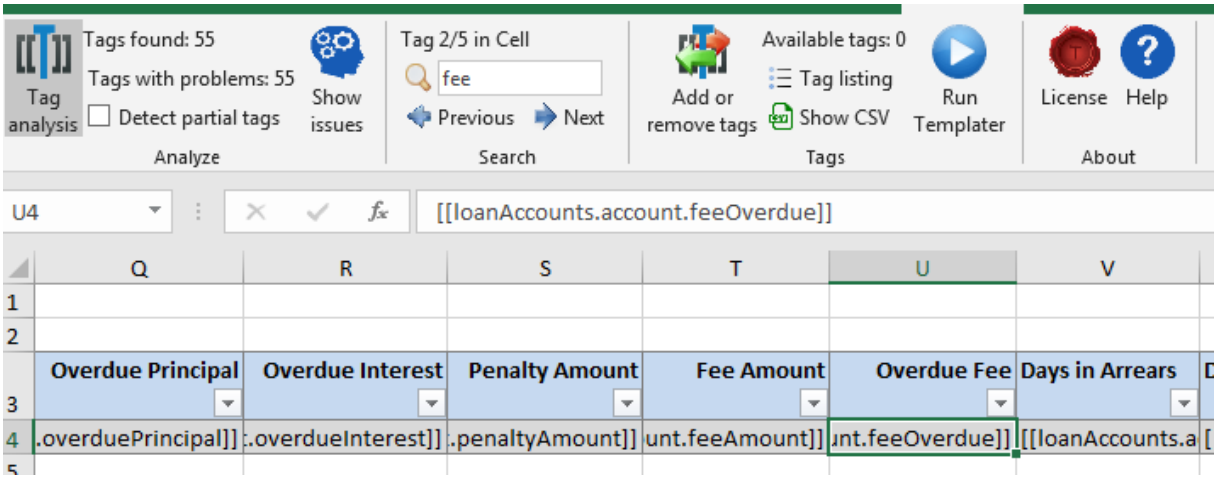
Some of the issues/warnings/suggestions which will be recognized by Templater Editor:

- Typos – with suggestion for alternative tag which is similar to the typo
- Partial tags – tags which are missing second bracket or similar problem
- Usage of multiple collections within a same table/chart – this will result in cartesian product which is most likely not wanted
- Alias suggestions – when alias is defined, it is suggested to use it on tags which match the alias prefix
- Bad tag placement – in some cases tags are placed in a location which will cause problems
- Wrong headers/footers – aggregate tags should be used in such locations, instead of tags used for resizing
- And many others…

Tag navigation

By clicking on the issue in the list, Templater will position the cursor on the problematic tag which allows for easy issue resolution.

Previous and **Next** buttons will allow navigation over all detected tags in the document. Search bar can be used to narrow down the search for a subset of tags:



With many tags present in the document, toggling **Tags with problems** button will instead search only through tags with some issues, although that is similar to going over issue list[25].

Some tags are embedded within the xlsx/docx document:

- csv tags used for PowerQuery/Get&Transform in Excel
- xlsx tags used for charts in Word

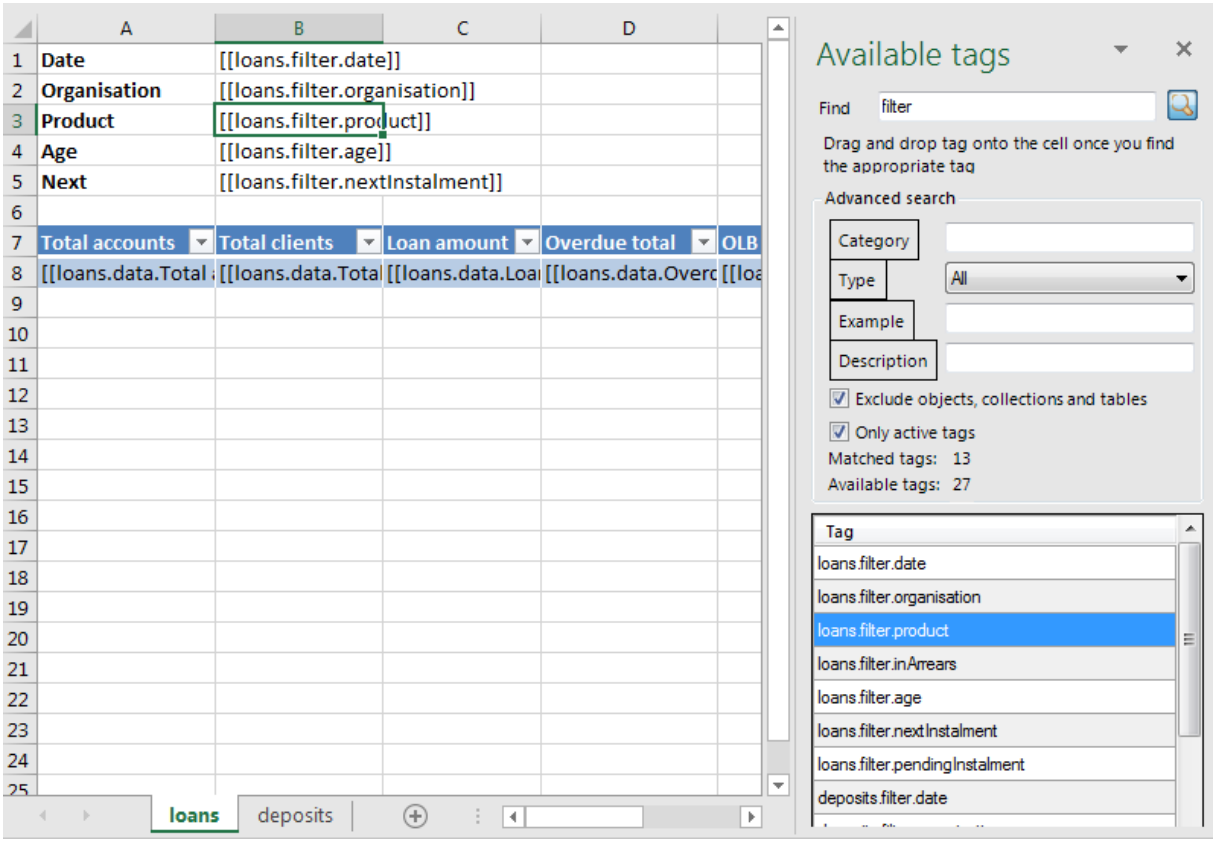
While navigating over tags, Templater will open up embedded document and show tags. Embedded CSV tags be read only, while chart tags can be edited.

Tag listing

With known schema, Templater can show listing of common tags[26]. Users can use this list to search for relevant tags and then drag/drop them onto the document.

Searching for available tags can be expanded by toggling the search button which will show the advanced search options:

[25] Single tag can have multiple issues and can be reported multiple times in Issues list
[26] Not all possible tags can be listed, as navigation can be done over recursive types and thus list would be indefinitely long



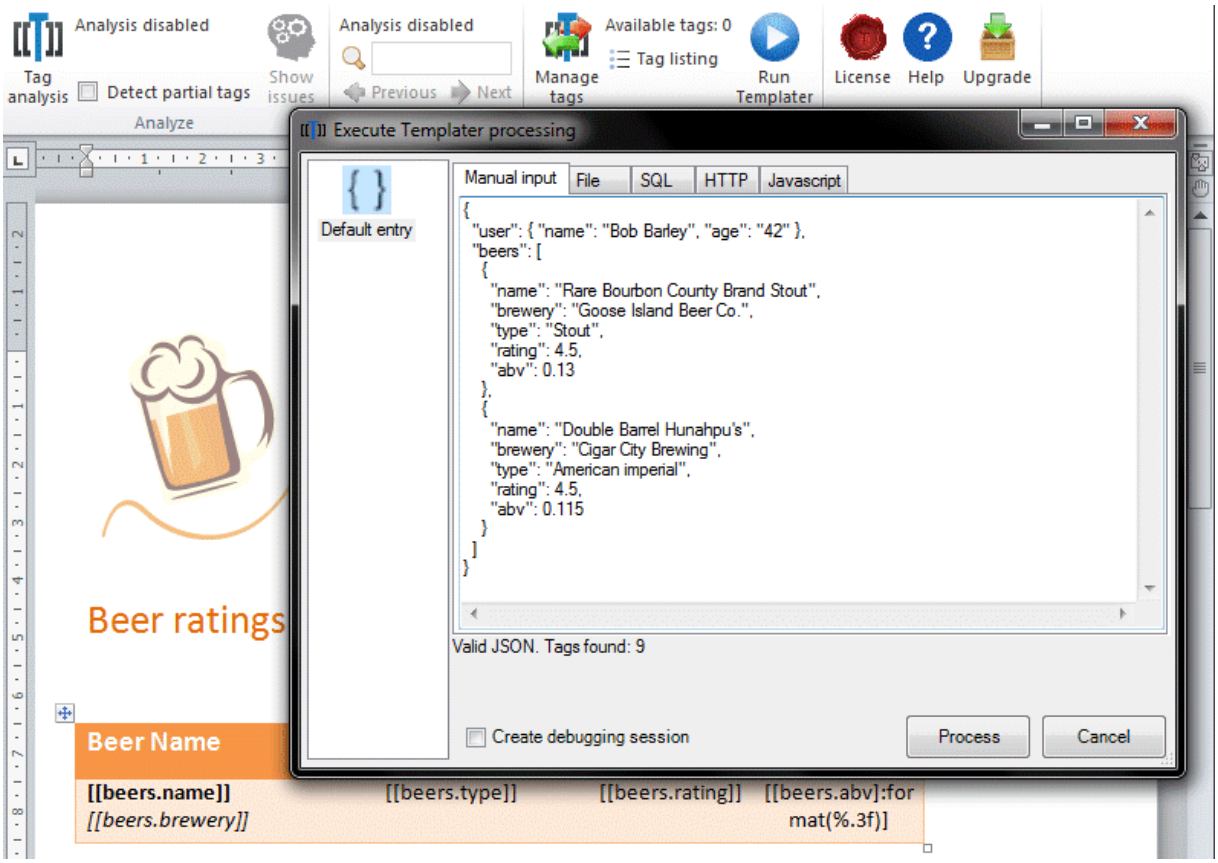
Additional information can be embedded in the schema for each property/tag which will then be available in the search window. This is done via

```
EditorConfigurationBuilder metadataResolver(MetadataProvider resolver);
```

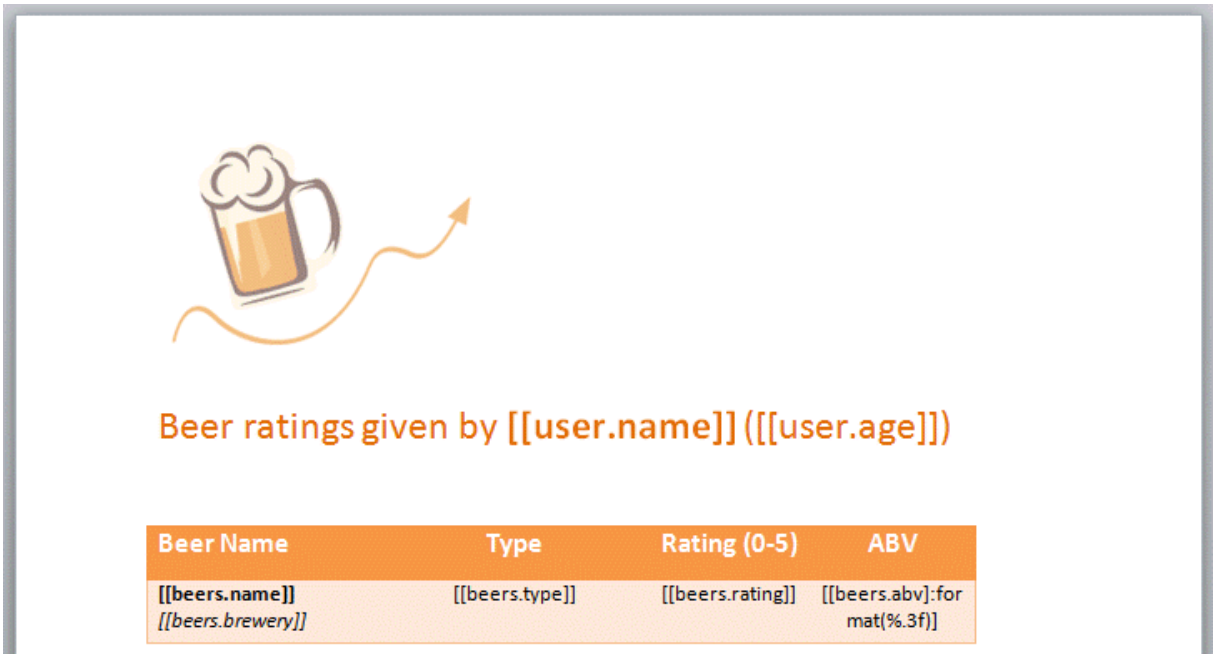
explained in the Configuration section.

Running Templater

It is very easy to test templates by running Templater from within the Office UI. Templater can be run without predefined schema, but it does need input which will be used for processing. Currently the easiest way to run Templater is to paste relevant JSON into the text box, or reference the json file locally on disk:



Upon processing Templater Editor will show the processed file which in this case should look like:



Processing options

There are several options for processing:

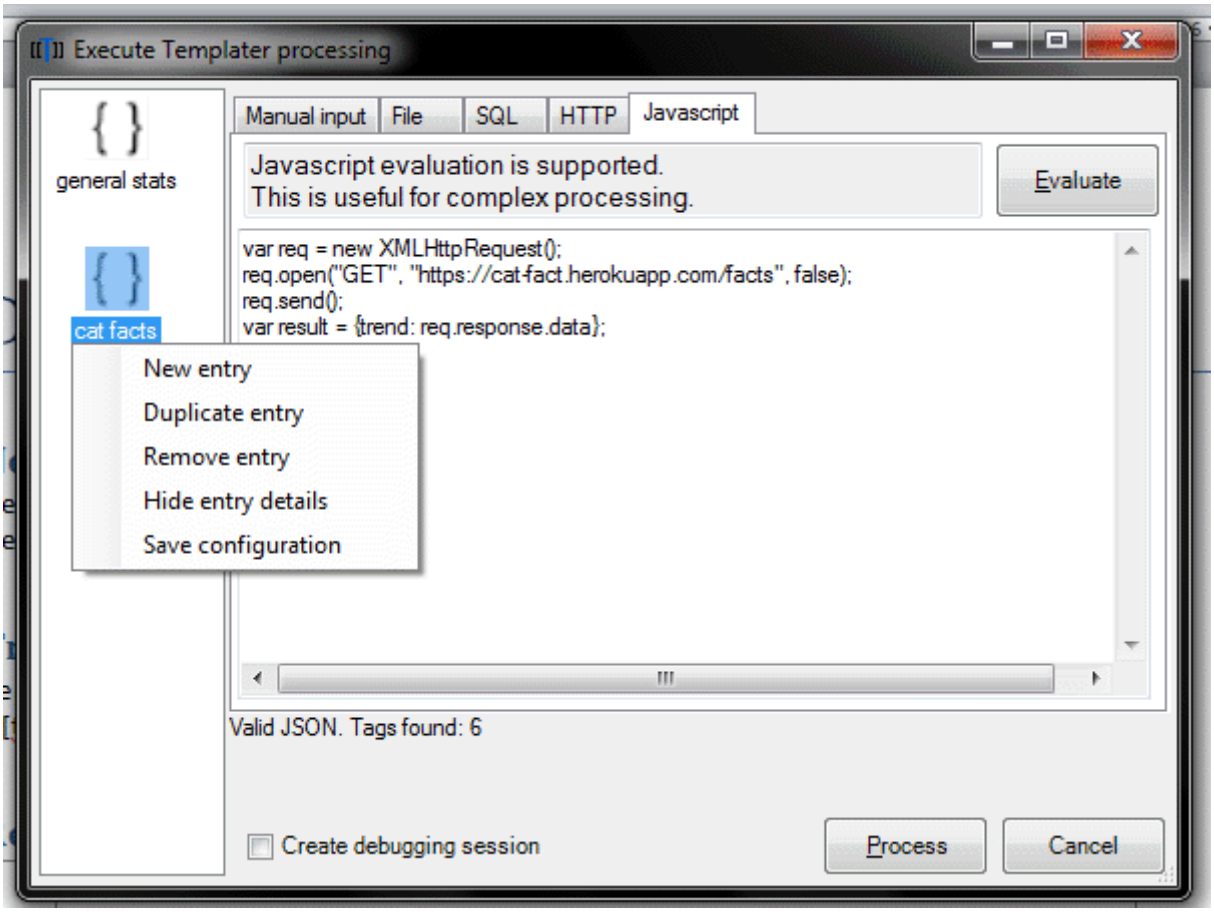
- manual JSON or XML input
- selection of file with JSON/XML content

- SQL query
- HTTP request
- Javascript code (which can use XMLHttpRequest for API requests)

Using right click on entry list will provide additional options:

- adding/removing entries
- showing/hiding entry details
- saving entries configuration into document

Entry configuration will be saved encrypted into document which allows saving of sensitive credentials.



Custom plugins

Templater Editor can be set up with user defined plugins, which add relevant formatters, navigation expressions and all other customization options.

To setup Templater Editor with custom plugins, two prerequisites are required:

- zip file containing dll(s) which will be used (or a single dll)
- location where zip/dll will be hosted

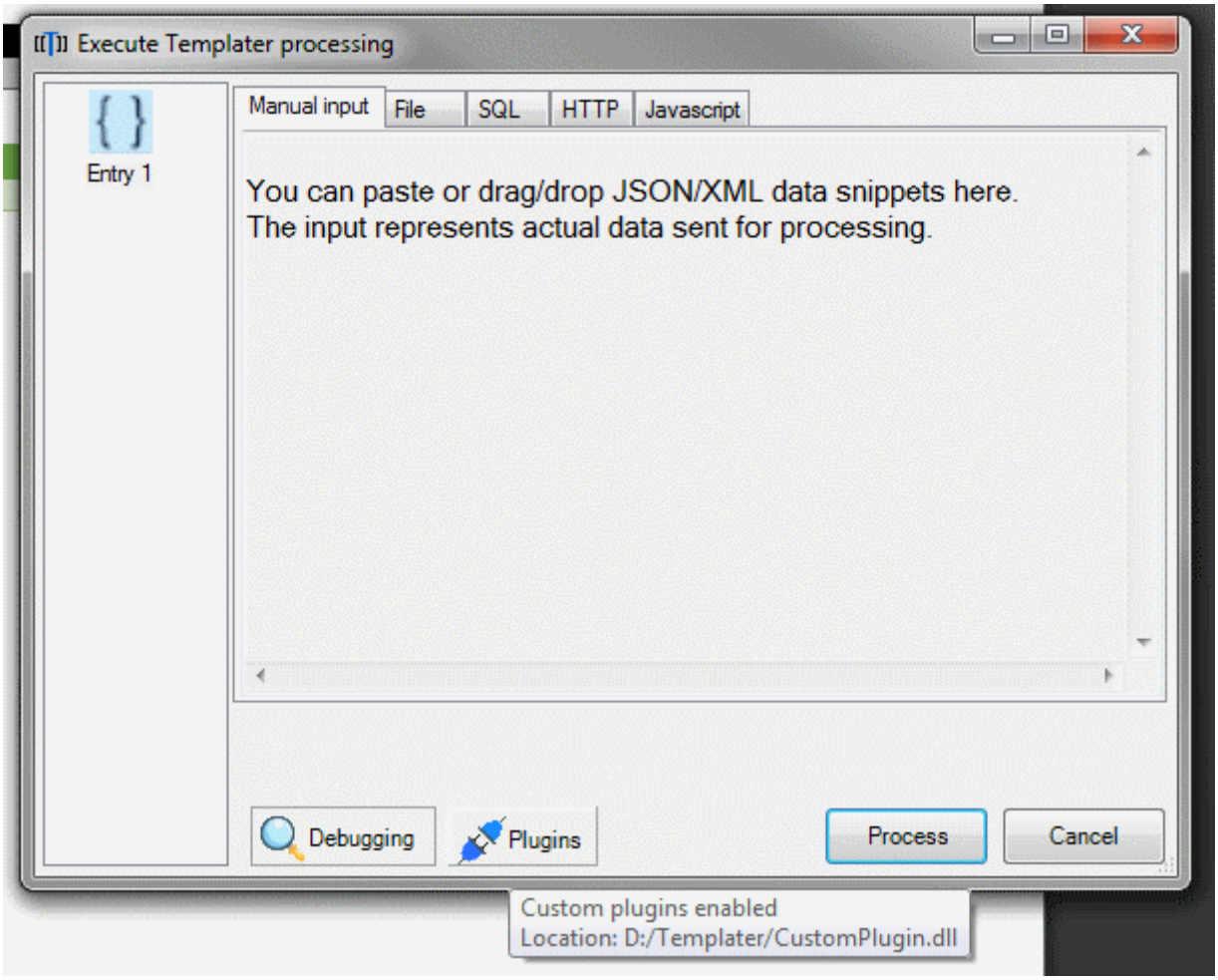
Project should be set-up in “legacy” .NET Framework 4.5+. Editor configuration must be performed via class which looks like:

```
public class TemplaterEditor
{
        public TemplaterEditor()
        {
        }

        public void Configure(IDocumentFactoryBuilder builder)
        {
                //setup the build with custom plugins
        }
```

IDocumentFactoryBuilder is the same interface as the one in Templater, with one minor difference, that is has no **Build** methods. If plugin configuration requires configuration per extension, instead of empty constructor, constructor with string argument can be used. Templater Editor will scan for public class named TemplaterEditor and call it before running Templater processing.

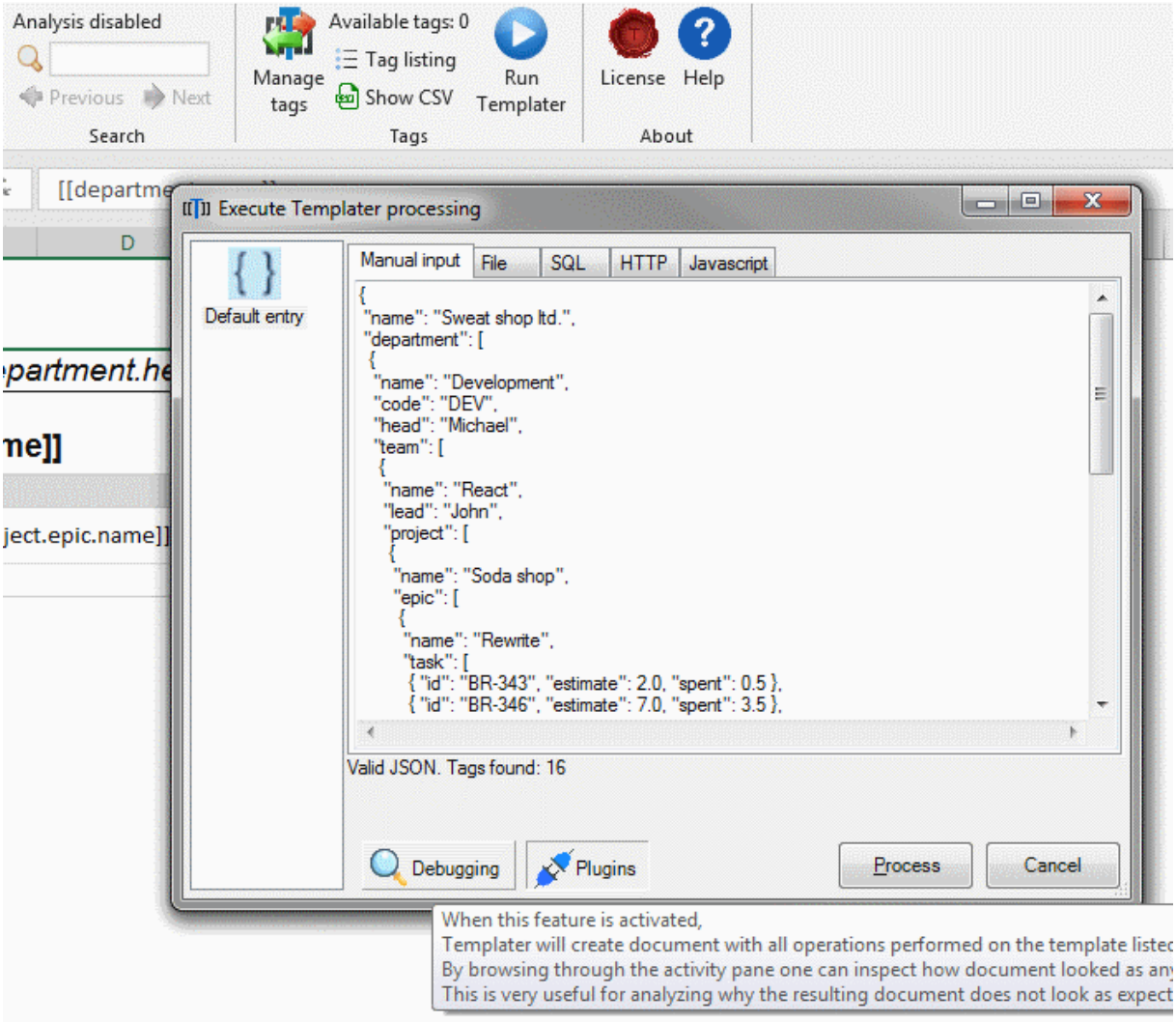
When license is configured with plugin location (which can be http(s), ftp or location on disk/network) a new button will be visible in the UI:



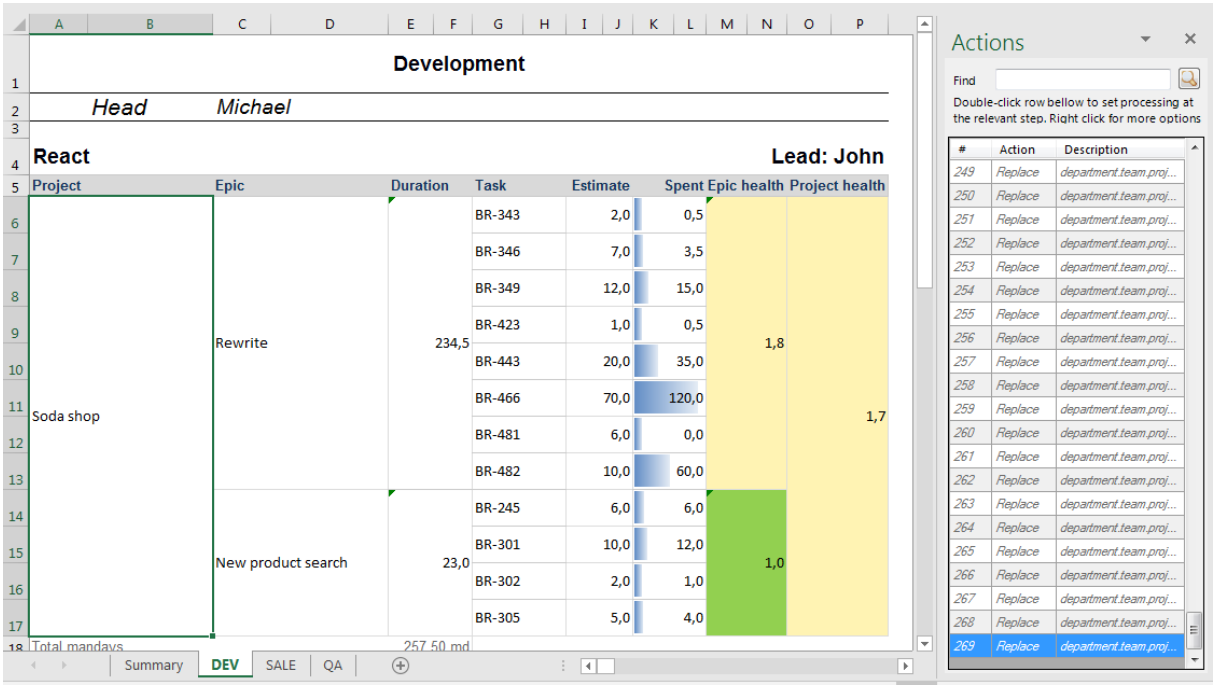
Debugging Templater

While Templater has only two basic operations: resize and replace, when there are many such operations, it helps being able to compare documents across them. For this purpose, Templater has a debugging capability which allows inspection how document looked at any point in time during processing.

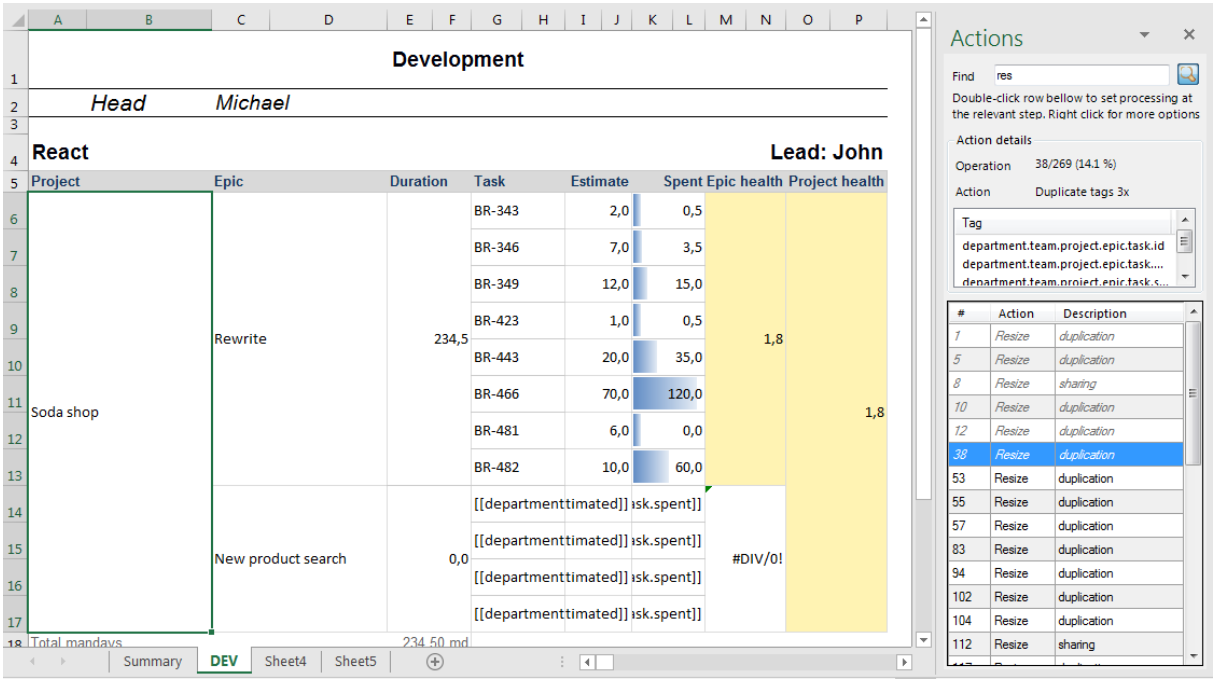
Debugging session can be created directly from the Templater Editor by selecting the **Debugging** toggle on the Run Templater screen:



Once processing is finished, a new document will open with Actions pane which allows to recreate document at any point in time:



Search filter can be used to find appropriate action. By expanding details more information will be displayed about selected action:



This kind of inspection into Templater processing allows for easy problem resolution and quick development loop.

Word features

Templater has extensive support for various Word features, but there are still few advanced ones missing. Various features are supported out-of-the box without any special code, while some require special handling and are introduced over time.

Mail merge

On surface Templater looks just like a mail merge solution. You can put tags on specific places in the document and replace them later with actual values. One could wonder why a library would even be required for that, as OOXML is just a ZIP file with an XML files which can be easily edited/manipulated.

But even in such a simple use case there are obstacles, as Word tends to split text into paragraphs so even a simple text such as [[TAG]] often looks like

```
<w:r w:rsidRPr="00A42204">
  <w:rPr>
    <w:lang w:val="en-US"/>
  </w:rPr>
  <w:t>[[</w:t>
</w:r>
<w:proofErr w:type="spellStart"/>
<w:r w:rsidRPr="00A42204">
  <w:rPr>
    <w:lang w:val="en-US"/>
  </w:rPr>
  <w:t>TAG</w:t>
</w:r>
<w:proofErr w:type="spellEnd"/>
<w:r w:rsidRPr="00A42204">
  <w:rPr>
    <w:lang w:val="en-US"/>
  </w:rPr>
  <w:t xml:space="preserve">]] </w:t>
</w:r>
```

which contains various “useless” Word specific information not really relevant for the original [[TAG]] text. There are also various Word specific rules such as **xml:space=”preserve”** which must be respected during processing.

Once tables and lists start to get used, replacing a tag is no longer: “*just locate and replace tag value in XML*”. With the addition of images, special Word objects, such as charts which are implemented as an embedded Excel file within the Word zip changing tags requires extensive knowledge of the Word behavior, format and rules. Therefore, a library which copes with those adjustments can be of quite a big help to the developer, even if his is quite familiar with the OOXML format.

Resizable regions

Templater uses various Word features as indicators for the duplication behavior. The simplest example would be to use row in a table as resizing region - a context. More complicated example would be a list in a table surrounded by section breaks. To understand resizing behavior of Templater few rules must be understood. When Resize(tags, count) is called Templater will

- find the best matching region of the document which encapsulates all specified tags (first occurrences of such tags)
    - regions will be limited to the rows in a table (match for starting and ending row)
        - table region can span multiple rows
    - relevant list levels will be matched
        - list levels can match the hierarchical structure of the model
    - Repeating Section Content Control resizable object can act as a container for group of tags
    - Listable Content Controls (ComboBox and DropDown list) – will behave as a simple list like object
    - embedded document will act as a natural grouping for tags, although it will act as resizable regions on its own
    - when region includes top level document (there is a tag which is neither in list or a table) sections around the region will be looked up
    - if section is not detected, start/end of the document will be used
- if all tags are inside tables/lists or other resizable elements, instead of duplicating the objects, tables and lists will be resized instead
    - this means when a same collection is repeated both in a table and a list, that those tables and a list will get new rows instead of new tables and lists being created in the document
- when count = 0 indicating removal of the content part of the document will be removed
    - with the exception when tags were detected at the top level and no sections were found - which would result in whole document removal
    - Word has a special behavior for removing “drawing” objects such as TextBox, images and WordArt. When resize with count = 0 is called on them, as long as only they are referenced, instead of finding the best section, Templater will just remove those objects

This way built-in Word features can be used to indicate the expected resizing context for the Templater. Common use cases include:

- use of a table without borders to group elements together
    - table is visible in Word, but not in the printed document
- use of lists without bullet indicators to simulate paragraph duplication
    - Word and Templater will consider it a list, but it will look like a plain paragraph
- nesting list (or tables) inside a table for fine tuning nested elements layout
- using table header repeating (and various other features) for tuning the listing behavior
- using Repeating Section Content Controls as a simpler way to define regions

- using sections (with or without page breaks)
    - an important aspect of sections is that section settings for current region are defined at the end of that region/section, which gets copied on resize

Tags will be detected in almost any part of Word document, such as:

- header or footer
- Word arts
- embedded Excel, Word, html or txt files
- bounded custom XMLs
- hyperlink descriptions
- and various others

Tables

Most basic resizable object in Word is a table. Table can be with or without header; it can have various options attached to it, such as:

- borders
- spacing
- repeating of header row
- cell/row breaking
- alignments
- automatic resizing
- styles
- cell merging
- text direction

which makes it really useful to design complex layouts.

Resizing a table is quite intuitive in Templater. When table like:

Column A	Column B
[[collection.columnA]]	[[collection.columnB]]

is matched with an appropriate input, e.g.:

```
{
 "collection": [
  {"columnA": "value A1", "columnB": "value B1"},
  {"columnA": "value A2", "columnB": "value B2"}
 ]
}
```

The result will look quite intuitive:

Column A	Column B
value A1	value B1

value A2	value B2

A really important aspect of such transformation is:

- it is implied by the document structure
- there are no loop or start/end constructs in the document
- it matches against the input "intuitively" by using dot (.) for navigation

Multi-row context

Over the years Templater context detection and manipulation improved significantly[27]. This allowed for context use over multiple rows, such as:

Product	Price
[[items.name]]	[[items.price]:format(N2)]
[[items.description]]	

when matched with an appropriate input, e.g.:

```
{
 "items": [
  {"name": "Product A", "price": 99.99,"description":"Nice useful tool"},
  {"name": "Product B", "price": 120,"description":"Spans\nmultiple\nrows"}
 ]
}
```

Produces an expected table which looks like:

Product	Price
Product A	99.99
Nice useful tool	
Product B	120.00
Spans *multiple* *rows*	

and has several non-trivial features:

1. context is no longer a single row, but two rows, since tags were defined across several rows
2. simple number formatting can be used to tweak the output into expected format
3. bolding, italics and other text features were preserved
4. newlines in text input resulted in newlines in cell values

Common use case with displaying collections is that they include information from the parent object. Templater can cope with various setups, even when provided data "is out of order".

[27] Various table examples can be found at: https://github.com/ngs-doo/TemplaterExamples/tree/master/Intermediate/WordTables

Product	Price
[[items.name]]	[[currency.signBefore]][[items.price]][[currency.signAfter]]
[[item.picture]:image]***[[items.description]]***	

when matched with an appropriate input, e.g.:

```
{
 "items": [
  {"name": "Flashlight", "price": 19.99,"description":"Let there be light","picture":"flashlight.png"},
  {"name": "Helmet", "price": 42.25,"description":"Protects the head","picture":"helmet.png"}
 ],
 "currency": {"signBefore":"", "signAfter":"€"}
}
```

will result in an expected output:

Product	Price
Flashlight	19.99**€**
 ***Let there be light***	
Helmet	42.25**€**
 ***Protects the head***	

This example includes two specific features:

- coping with out of order tags ("currency" was defined after the "items" part)
- inserting the image (via custom images plugin[28])

In this example currency was defined on top level object (as sign which can go in front of the number and as a sign which can go after the number). If "currency" was processed before the "items" it would be a simple case of replacing the tags with **€** and duplicating replaced value per rows. But in this case currency was defined after and it was expected that all rows have the same value, which Templater detected and replaced them accordingly.

Tag for image was defined as [[items.picture]:image] which expects that there exists a plugin which knows how to handle the **image** metadata by loading the image from the appropriate place and putting it at the place of the tag. This way complex interaction between custom code (converting

[28] There is no such builtin plugin in Templater, but developer can easily create/add their own

„flashlight.png“ into an actual image) and Templater (injecting the image into the document) worked together to produce a complex output.

Dynamic resize

A special feature of Templater is processing specific input types (two dimensional collections and DataReader/ResultSet) in a specialized way.

A basic use case for Dynamic resize would be to transform table template into a final output, e.g.:

[[table]]

when matched with an appropriate input, e.g.:

```
{
 "table": [
  ["A", "B", "C"],
  ["A-1", "B-1", "C-1"],
  ["A-2", "B-2", "C-2"],
  ["A-3", "B-3", "C-3"]
 ]
}
```

it will be transformed into a table with 3 equal columns and 4 rows:

A	B	C
A-1	B-1	C-1
A-2	B-2	C-2
A-3	B-3	C-3

While this is useful for some scenarios, usually explicitly defined table templates are used since they allow for more fine-grained tuning.

Dynamic resize can be combined with “standard” table templates which allows for best of both worlds, as most of the table can be predefined, but some specific parts can still allow for dynamicizm:

[[names.a]]	[[names.b]]	[[columns]]
[[row.a]]	[[row.b]]	[[row.dynamic]]

when matched with an appropriate input, e.g.:

```
{
 "names": {"a":"Column A", "b": "Column B"},
 "columns": [["Column X", "Column Y"]],
 "row": [
  {"a":"A1", "b":"B1", "dynamic":[["X1","Y1"]]},
  {"a":"A2", "b":"B2", "dynamic":[["X2","Y2"]]}
 ]
}
```

will result in table which is partly dynamic:

Column A	Column B	Column X	Column Y
A1	B1	X1	Y1
A2	B2	X2	Y2

Cell merging

While cells can be merged in the template, there are use cases when they need to be merged during table generation/population. For this reason, there are two built-in metadata plugins:

- merge-nulls - invokes horizontal cell merging when cell value is null
- span-nulls - invokes vertical cell merging when cell value is null

Cell merging works both in Dynamic resize and standard table resize. Table such as:

Column A	Column B	Column C
[[nulls.a]:merge-nulls]	[[nulls.b]:merge-nulls]	[[nulls.c]:merge-nulls]

when paired with input such as:

```
{
  "nulls": [
    {"a":"A1", "b":null, "c":null},
    {"a":"A2", "b":"B2", "c":null},
    {"a":null, "b":null, "c":null},
    {"a":"A4", "b":null, "c":"C4"}
  ]
}
```

will result in table with merged cells:

Column A	Column B	Column C
A1		
A2	B2	
A4		C4

Existing merge cells

If there are existing merge cells in the table, Templater has specialized behavior for dealing with them:

- if tag range does not touch start of a merge cell or goes beyond end of merge cells, merge cell will be stretched
- otherwise context will be expanded to include merge cell(s)
- if tag is contained within the merge cell, context will include merge cells in minimum spanning range

Stretching merge cells in a table looking like:

Merge cell	Col A	Col B
	[[num]]	[[txt]]

When matched with an appropriate input, e.g.:

```
[
 { "num": 1, "txt":"A" },
 { "num": 2, "txt": "B" }
]
```

Will result in table with merge cell stretched:

Merge cell	Col A	Col B
	1	A
	2	B

If duplication instead of stretching is desirable behavior, but there are no tags in the first row, one can add helper tag which will be hidden after processing, e.g. [[text]:hide]. This way minimum context range can include additional rows and thus instruct Templater to behave differently in this specific case.

This could look like:

Merge cell [[txt]:hide]	Col A	Col B
	[[num]]	[[txt]]

And would result in:

Merge cell	Col A	Col B
	1	A
Merge cell	Col A	Col B
	2	B

Removing a table

When Resize(tags, 0) is called on a table, relevant rows will be removed. Sometimes this means that entire table will be removed, but often for table with headers which don't have any tags the header remains at the end of the resizing. In case when there is a separate header without tags a common workaround is to add special tag on the header with collapse and hide metadata:

Product[[items]:collapse:hide]	Price
[[items.name]]	[[items.price]]

This way when items collection is empty a separate resize 0 will be called just for the header row. When collection is not empty hide metadata will take care of not showing any text in place of the tag.

There is also a common pattern to show a different table when there are no rows, but this is explored in more detail later in sections part.

Lists

The second basic resizable elements in Word are lists. All lists types are supported:

- bullets
- numbered
- multi-level

Sometimes it's useful to tweak the layout of the list so it looks like a regular paragraph, because Templater will consider list a resizable element, while paragraph is not[29], e.g.:

[[text]]

matched with an appropriate input:

```
[
  {"text":"first row"},
  {"text":"second row"},
  {"text":"third row"}
]
```

will result in list which looks like an ordinary paragraph (list alignment is moved to the right):

first row
second row
third row

It is common to nest lists inside tables and while it is not common to nest tables within lists, that also works as expected.

Nesting

Common use case for lists is pairing it with deep nesting or even with recursive structures. When specialized data structure is used, such as:

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

it is rather easy to pair it with nested list by predefining maximum nesting level, e.g.:

[29] Paragraph can behave as resizable elements if there are continuous sections around it in which case everything between the sections will be duplicated

1. [[value]]
    1.1. [[nested.value]]
        1.1.1.[[nested.nested.value]]
            1.1.1.1. [[nested.nested.nested.value]]

Based on the input, resulting list will match the nesting levels and values, e.g. for input as:

```
[
  { "value":"Level A-1", "nested":[
     { "value":"Level A-2a", "nested":[] },
     { "value":"Level A-2b", "nested":[
        { "value":"Level A-3", "nested":[
           { "value":"Level A-4a", "nested":[]},
           { "value":"Level A-4b", "nested":[]}
        ]}
     ]}
  ]},
  { "value":"Level B-1", "nested":[
     { "value":"Level B-2a", "nested":[] },
     { "value":"Level B-2b", "nested":[] }
  ]}
]
```

a matching list will be created:

1. Level A-1
    1.1. Level A-2a
    1.2. Level A-2b
        1.2.1.Level A-3
            1.2.1.1. Level A-4a
            1.2.1.2. Level A-4b
2. Level B-1
    2.1. Level B-2a
    2.2. Level B-2b

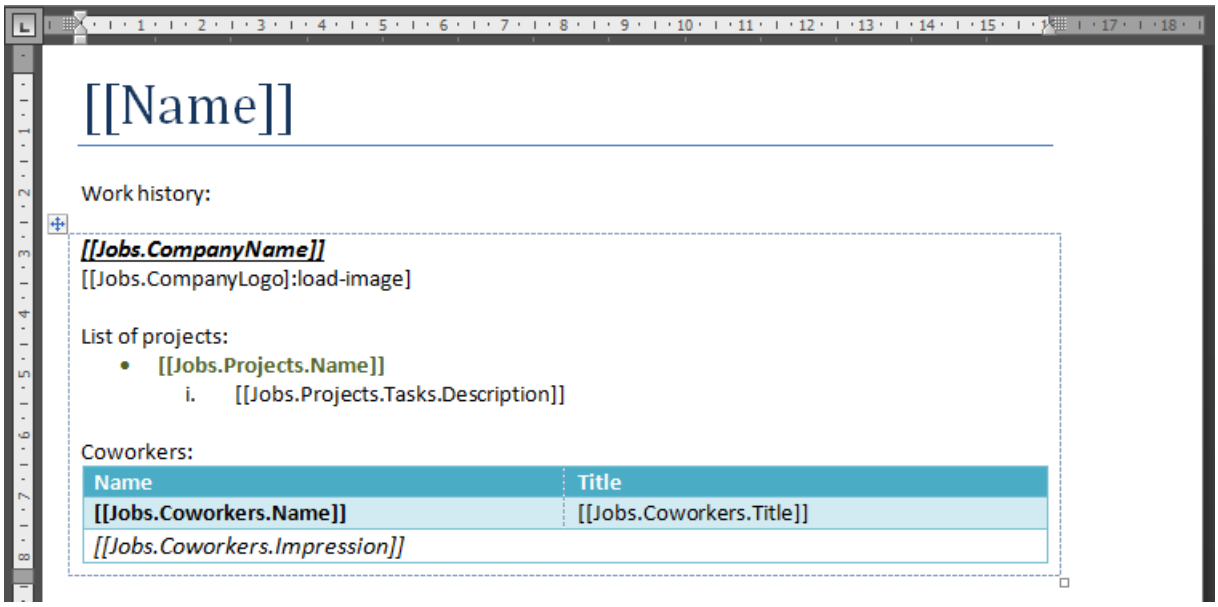
Since style is defined on the list, while Templater only binds the data with the list, complex list representations can be easily constructed.

Lists in table

A very common use case is to have lists inside a table. The nesting can be arbitrary deep (up to the resize limit configuration option).

Templater will take care of duplicating lists and renumbering them appropriately (when numbered lists are used).

A common “trick” with nesting lists in a table is to use “invisible” tables - tables without border. They will be visible in the editor, but not in the resulting document, e.g.:



Multilevel lists in a table

List duplicated within a table will continue with numbering. If multiple lists are used to simulate nesting Templater might not behave as expected unless the layout of the list is configured to match the expected behavior.

Template configured with two lists:

1	2	3
	1. [[sports.name]]	
1.	[[sports.events.name]]	[[sports.events.description]]

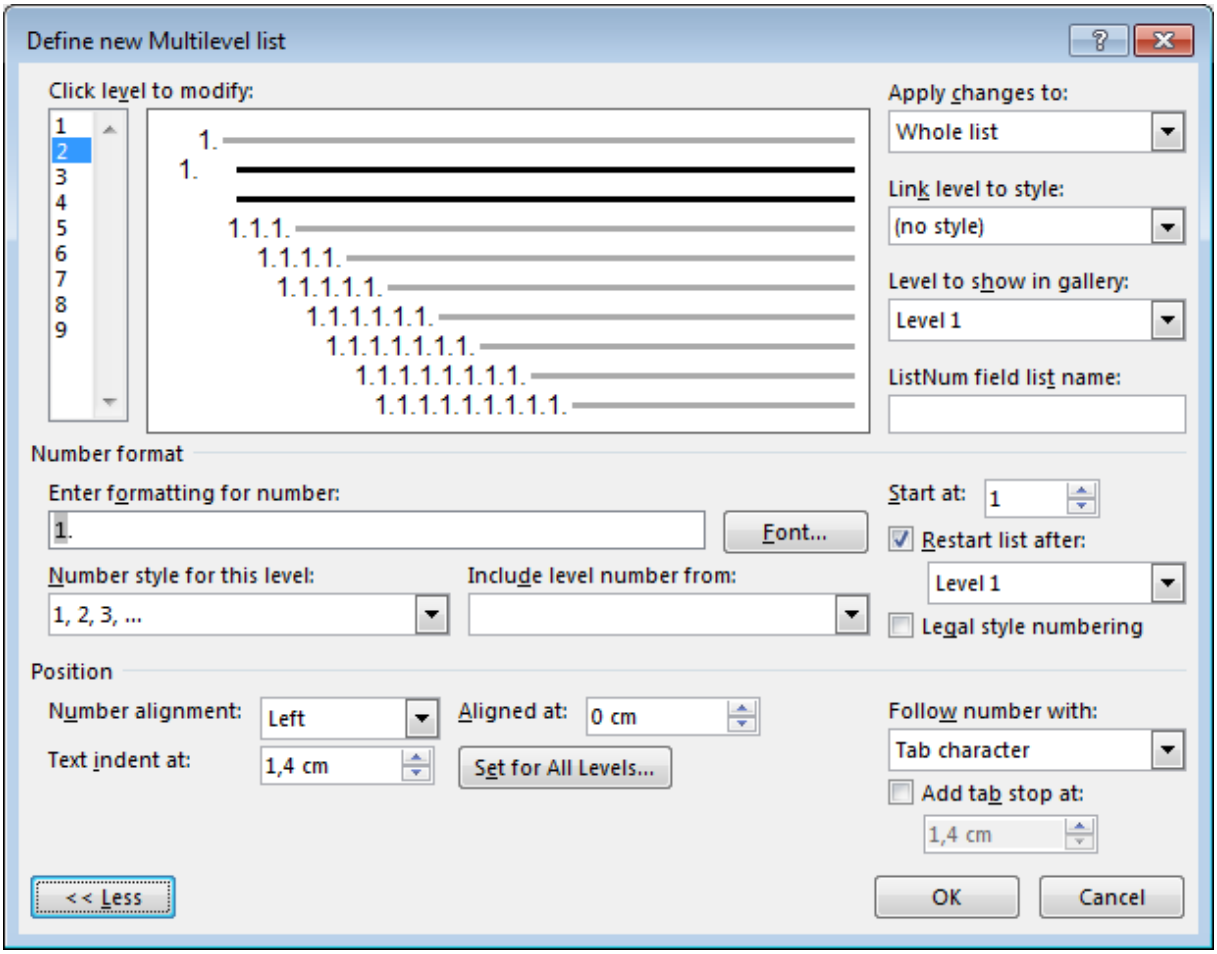
when paired with matching input:

```
{
   "sports":[
      { "name":"Football", "events":[
         { "name":"World Cup", "description": "FIFA World Cup is international tournament for national
teams"},
         { "name":"European Champions League", "description": "UEFA club competition"}
       ]},
      { "name":"Basketball", "events":[
         { "name":"NBA", "description": "Men professional basketball league in North America"},
         { "name":"Olympics ", "description": "Olympic basketball tournament "}
       ]}
 ]
}
```

will result in output where both lists increase their numbering:

1	2	3
	1. Football	
1.	World Cup	FIFA World Cup is international tournament for national teams
2.	European Champions League	UEFA club competition
	2. Basketball	
3.	NBA	Men professional basketball league in North America
4.	Olympics	Olympic basketball tournament

To get the desired output a single multilevel list can be used. With some alignment adjustment via multilevel list settings page



This way Templater will produce expected output:

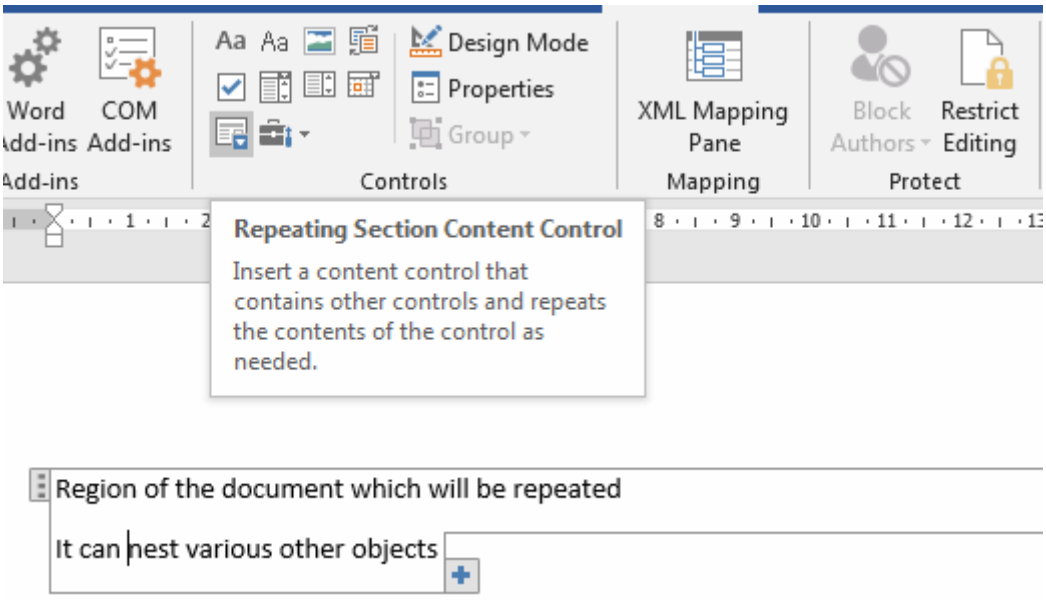
1	2	3
	1. Football	
1.	World Cup	FIFA World Cup is international tournament for national teams
2.	European Champions League	UEFA club competition
	2. Basketball	
1.	NBA	Men professional basketball league in North America
2.	Olympics	Olympic basketball tournament

Removing lists

Unlike table which often have special header row, lists are much easier to remove and there is no need for any special workarounds. Calling Resize(tags, 0) on them should remove relevant part of the document (and tweak the document when necessary so it doesn't become corrupted).

Repeating Section Content Control

Since v6.1 Templater supports usage of specific Content Control intended for repeating regions of the document:

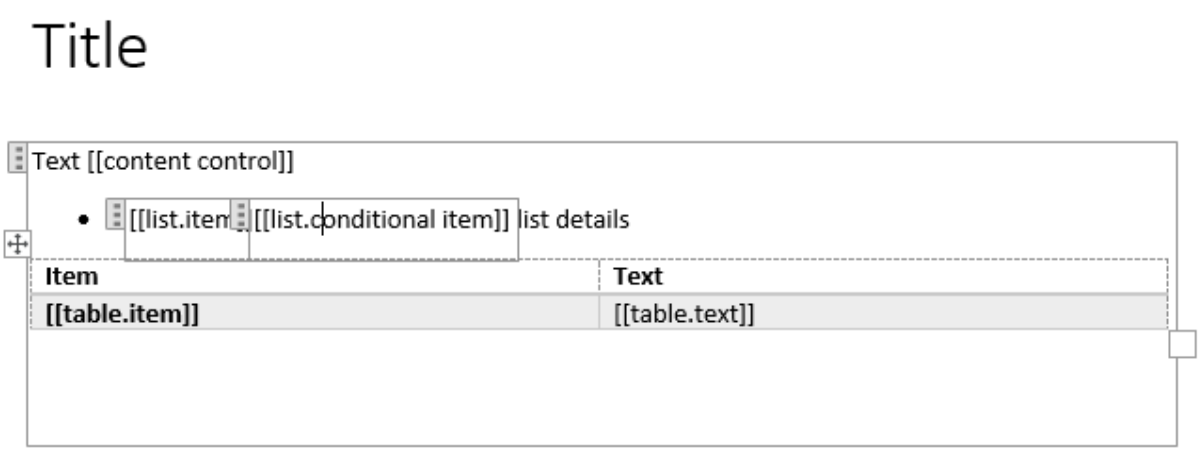


This Content Control is most useful when region need to be duplicated multiple times. If the main purpose is conditional display (e.g., remove region when not necessary) than other ordinary Content Controls can serve the purpose.

It is common to implement duplication of paragraphs this way, as Templater does not consider paragraphs resizable regions, but when placed inside Repeating Section, they start behaving as resizable region.

This Content Control can be quite useful when combined with nesting; as tables, list and other resizable objects can be used inside.

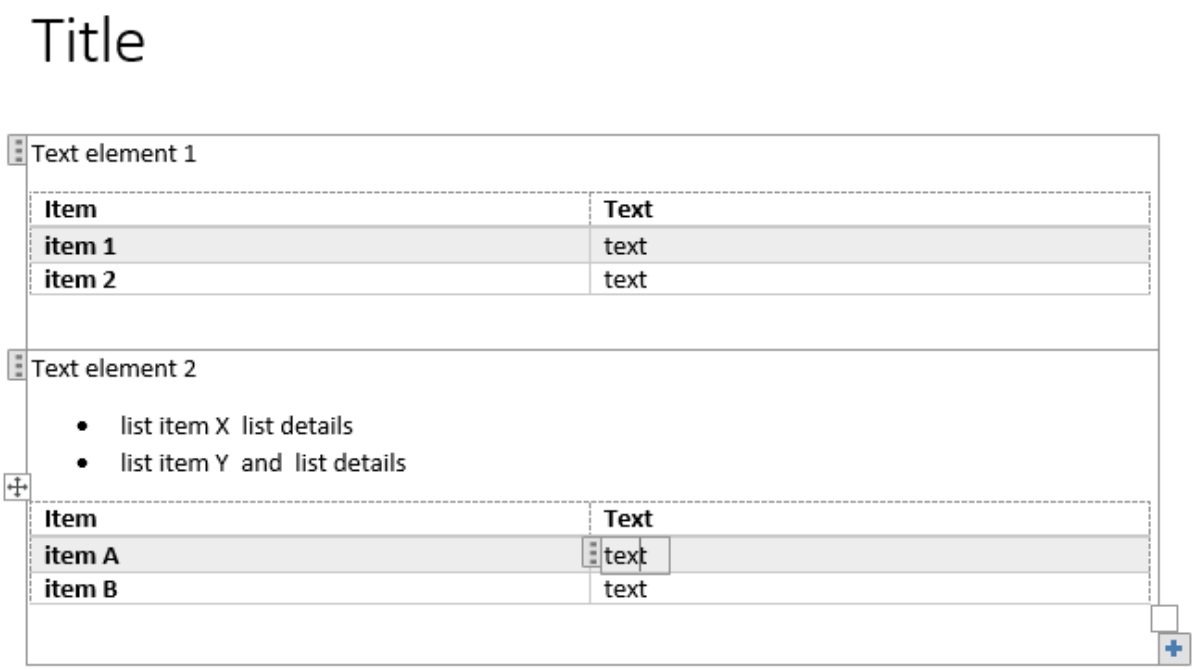
For example, template such as:



When paired with example JSON:

```
[{
   "content control": "element 1",
            "list":[],
            "table":[
                        {"item":"item 1", "text":"text 1"},
                        {"item":"item 2", "text":"text 2"}
            ]
},{
   "content control": "element 2",
            "list":[
                        {"item":"list item X", "conditional item":""},
                        {"item":"list item Y", "conditional item":" and "}
            ],
            "table":[
                        {"item":"item A", "text":"text 3"},
                        {"item":"item B", "text":"text 4"}
            ]
}]
```

will result in duplication of section region and duplication of table/list elements



Sections

When choosing a region for the context, Templater will look for sections around the specified tags.

Continuous sections are useful for separating parts of the document without affecting layout. Sections are somewhat counterintuitive at the beginning as they are often applied with the text above.

A common pattern when showing tables is to have different table layouts depending if there is data inside or not. This is easy to implement by combining two tables and a section, such as:

Name	Description
No results found	

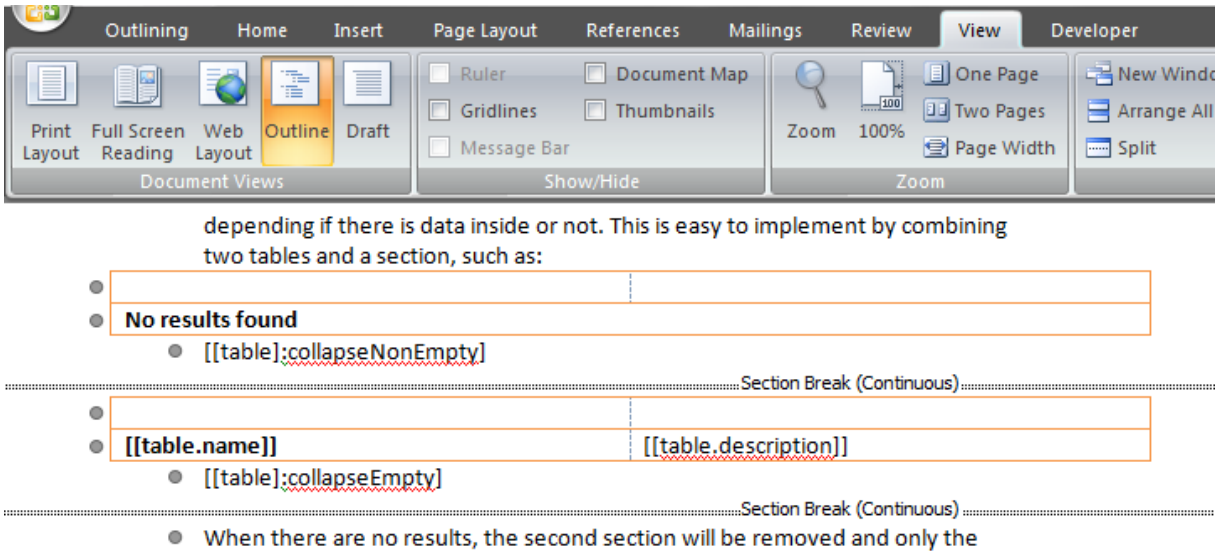
[[table]:collapseNonEmpty]

Name	Description
[[table.name]]	[[table.description]]

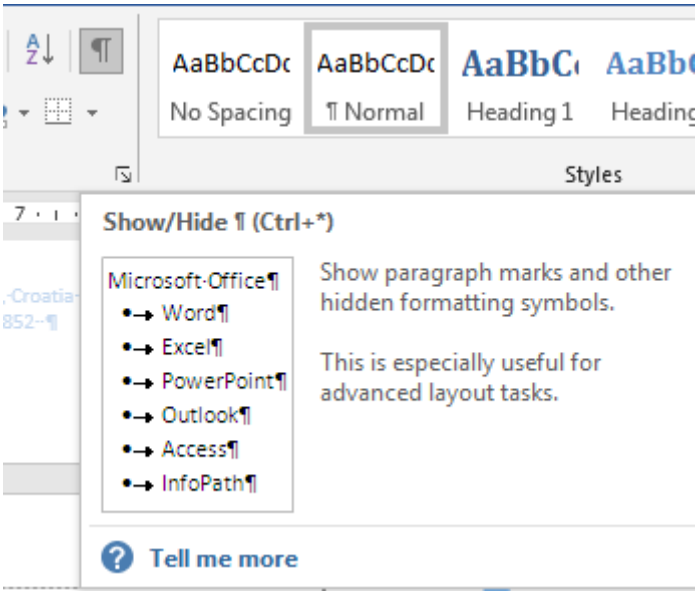
[[table]:collapseEmpty]

When there are no results, the second section will be removed and only the table with **No results found** will remain. Otherwise when there is data inside the table collection, the first section will be removed and the second table will be populated with data.

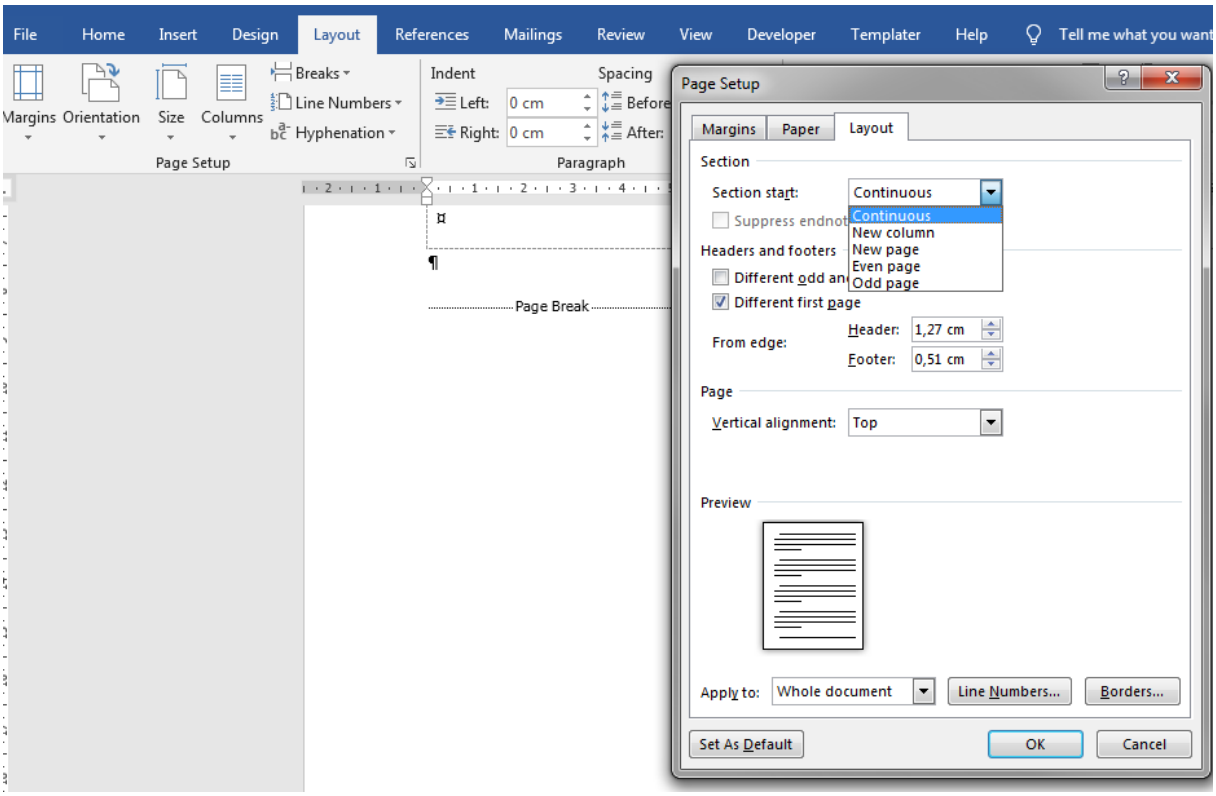
Sections are highly visible in Outline view:



In modern Word versions section will be visible when non-printable characters are displayed:



Sections can also be changed via Layout options, which is especially useful for managing ending section of the whole document:

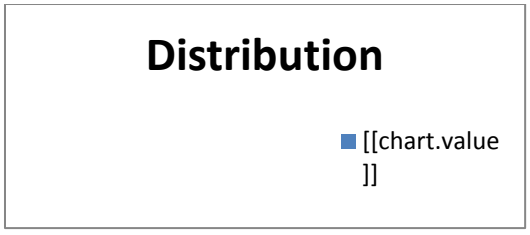


Charts

Charts are represented by embedding Excel xlsx inside Word zip docx. Depending on the chart type there is also some aggregation of values within the document XML.

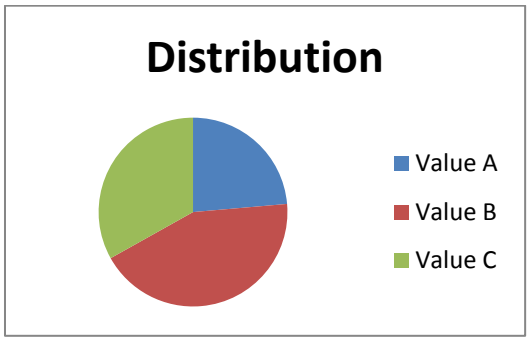
Charts are also considered resizable elements, as the underlying data source is a resizable Excel range.

Chart template is defined within Excel by adjusting original template and replacing values with tags, which results in a bit unfriendly chart template:



	Distribution
[[chart.value]]	[[chart.distribution]]

But once the underlying Excel is populated with data and range, the chart will be updated accordingly:



	Distribution
Value A	11,2
Value B	20,5
Value C	15,7

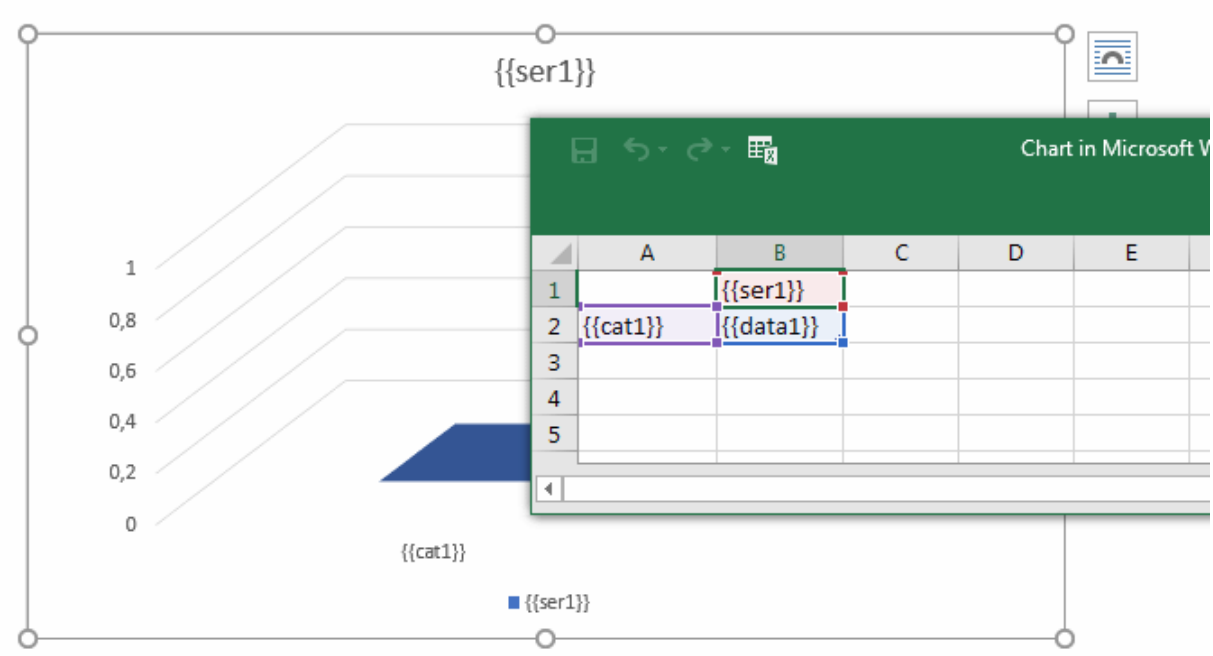
Tags defined within the Excel are visible in the **Tags** property on the *ITemplater* interface of the Word document. This makes them transparent to the application/processing. This means there is no need to unzip the docx file, process the embedded xlsx files, but rather Templater does that behind the scenes.

There are various charts in Word, such as pie charts, graphs and various others. They should all work seamlessly through Templater.

Dynamic resize

While replacing chart values works fine, sometimes there is a need for dynamic number of series on a chart. This can be implemented via at least 2 dynamic resize tags (for series, categories and values).

Example template can look like:

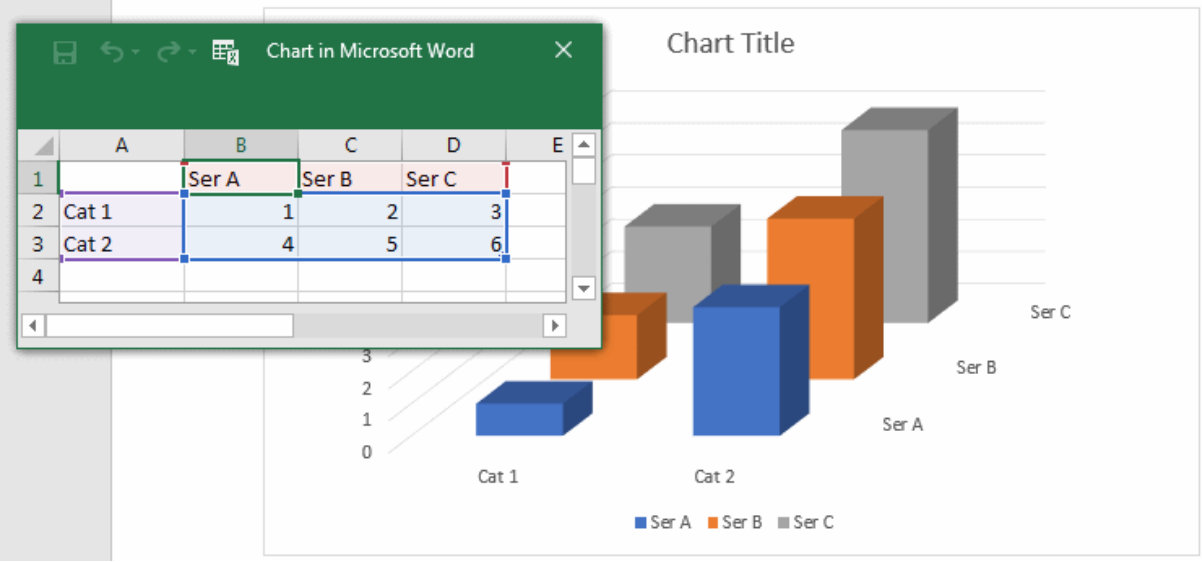


with relevant JSON, e.g.:

```
{
 "ser1": [ [ "Ser A", "Ser B", "Ser C"] ],
 "cat1": [ [ "Cat 1" ], [ "Cat 2" ] ],
```

```
 "data1": [ [ 1, 2, 3 ], [ 4, 5, 6 ] ]
}
```

will result in expected output:



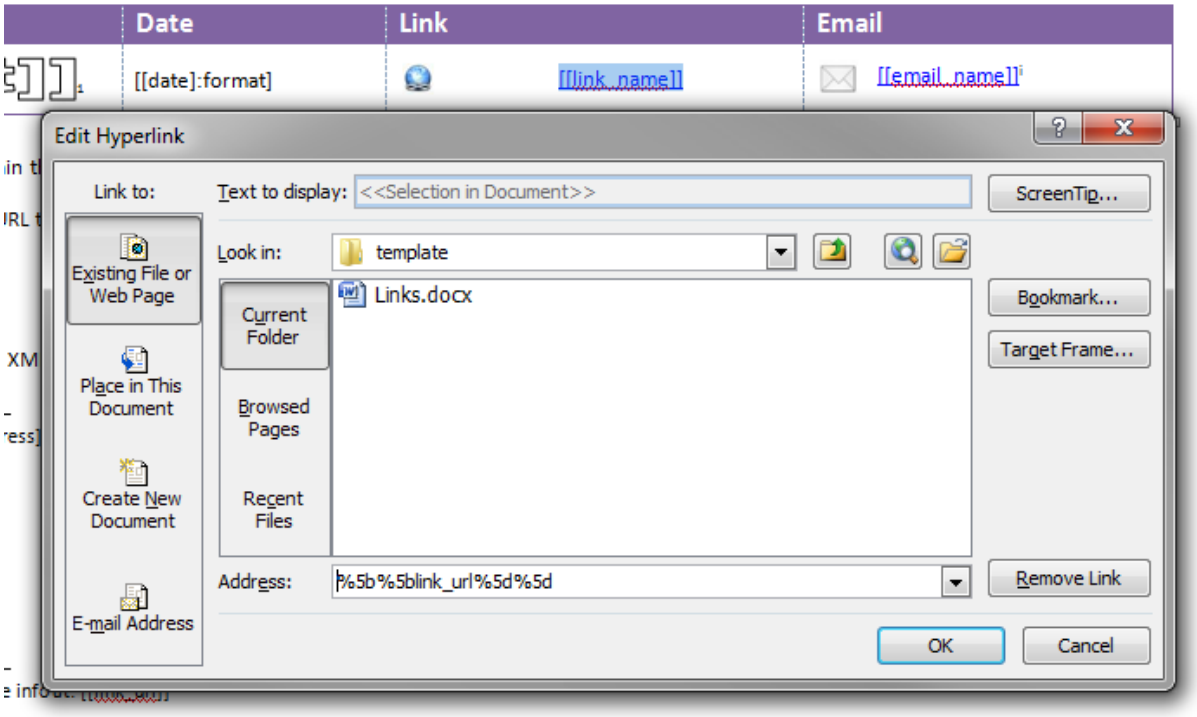
Word specific features

Links

Templater analyzes hyperlinks and thus they work as expected. Hyperlink can have multiple tags or tag can be combined with static description, e.g.:

Link to [[description]]

Address is url encoded which means that [[specific_url]] is converted into %5b%5bspecific_url%5d%5d when hyperlink is created, e.g.:



Special data types can also be used to create simple links (just a link, no custom description) when URI/URL is used as datatype.

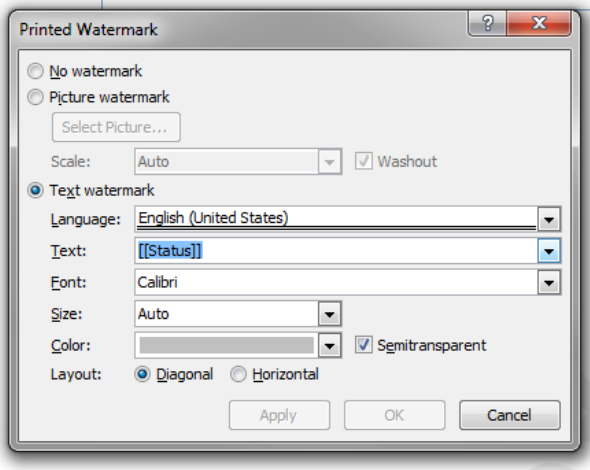
Watermark

It is common to have watermarks on documents to indicate special state[30]. Templater will detect and replace tags in watermarks.

Example of watermark on invoice would look like:

[30] Invoices often have CANCELED or PAID written over them to emphasize their state, as shown in: https://github.com/ngs-doo/TemplaterExamples/tree/master/Advanced/SalesOrderMVP%20(.NET)

Product	Quantity	Price	Line total
[[Items.Product.Name]]	[[Items.Quantity]]	[[Items.Product.Price]]	[[Items.Cost]:format(N2)]
		Total	**[[Total]:format(N2)]**



[[Status]]

Word ART/Smart ART

Tags can also be used in Word ART, Smart ART and other similar features.

There is special behavior with them on removal, as only referenced WordArt will be removed instead of surrounding region of the document. If tags are located outside of WordArt too, removal will then behave “the standard way”.

Footnotes and endnotes

While Templater supports footnotes and endnotes, it should be emphasized that their behavior is non-trivial in a sense that the tags where they are defined is bound to the footnote/endnote location, not the tag itself. This means that when footnotes are used in a table which gets resized, the footnote will also be duplicated. If tag is used multiple times, Templater will take care that all relevant tags are replaced with the expected value.

Header and footer

Tags can be used in header and/or footer along with other places in the document. Header and footer do have some special behaviors, similarly to the tags placed in the top level of the main document.

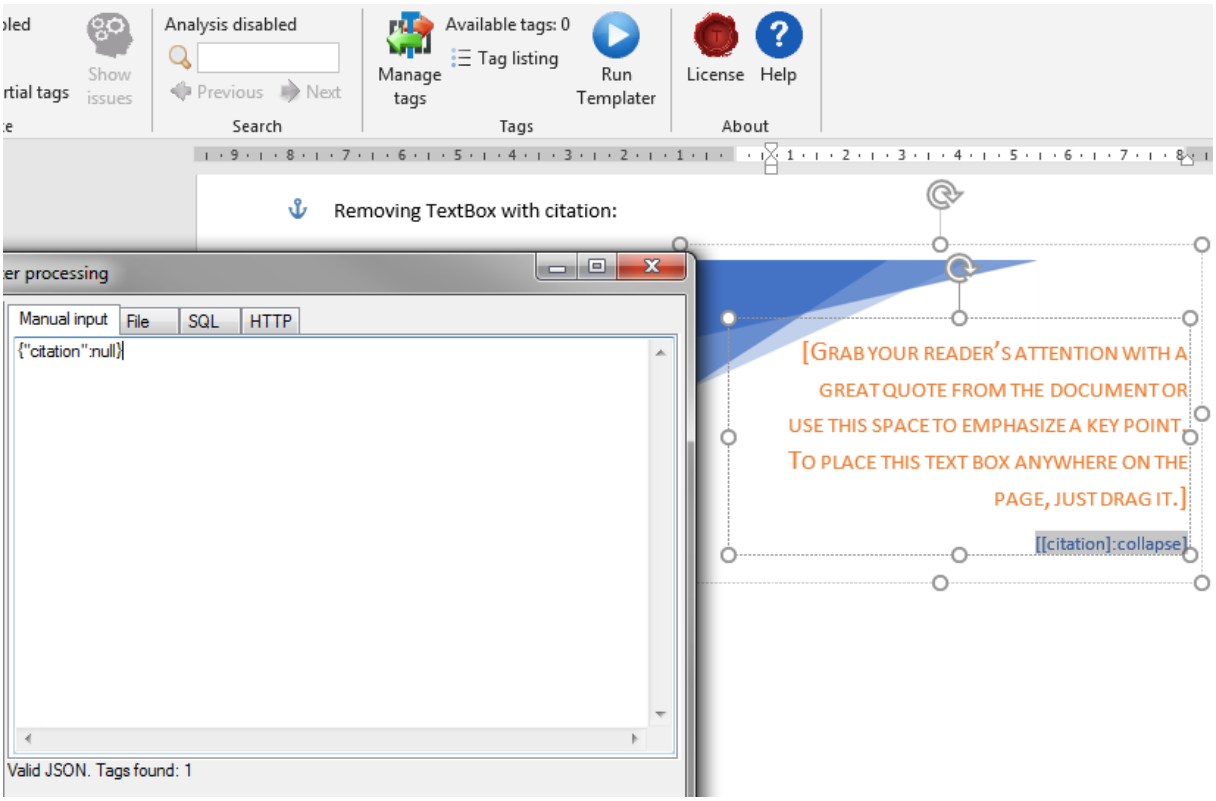
It’s common to use footer to add page numbering to the Word document (as this is a Word feature). Word will recalculate the page numbers once the document is opened.

Text Box

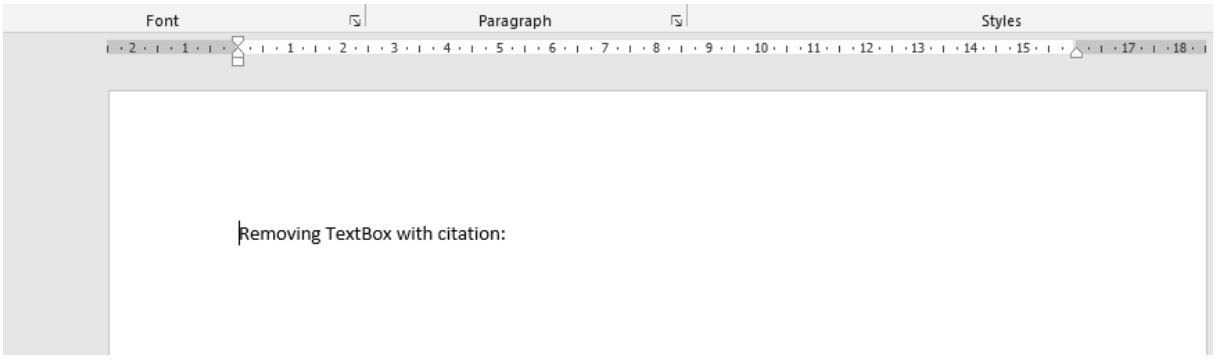
Tags are recognized within Text Box and other similar features.

There is special behavior with them on removal, as only referenced WordArt will be removed instead of surrounding region of the document. If tags are located outside of WordArt too, removal will then behave “the standard way”.

An example of TextBox removal would look like:



with the end result being:



Merge fields

Word and some other libraries only support merge fields for similar features. Templater will also recognize tags within merge fields, although it is much easier to define tags without the use of merge fields.

New images

Since duplication of context can cause image duplication, Templater will adjust the document accordingly. If new images need to be inserted into the document this can be done via Templater specific data type: ImageInfo

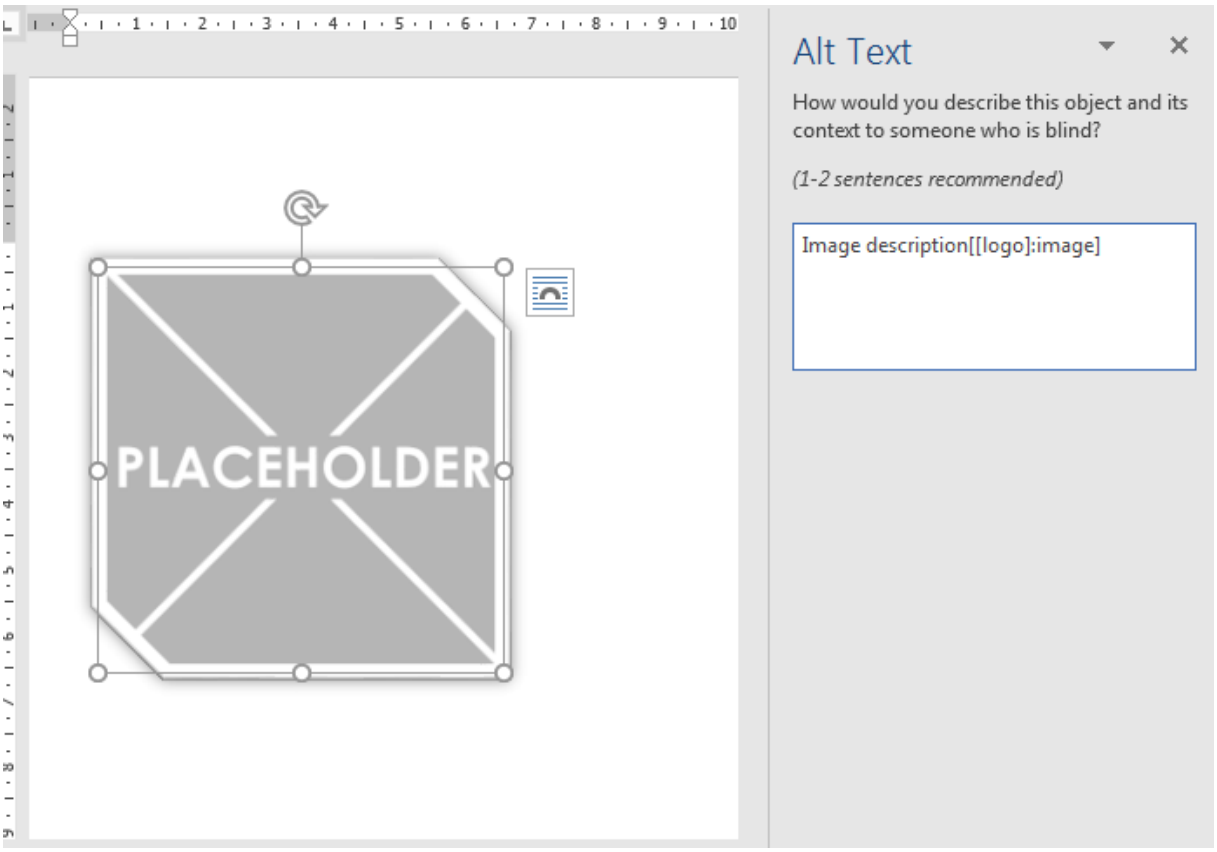
To ease image usage and support platform with custom/different image libraries, default .NET/Java image types are by default converted into ImageInfo type:

- .NET: Image and Icon
- Java: BufferedImage and ImageInputStream

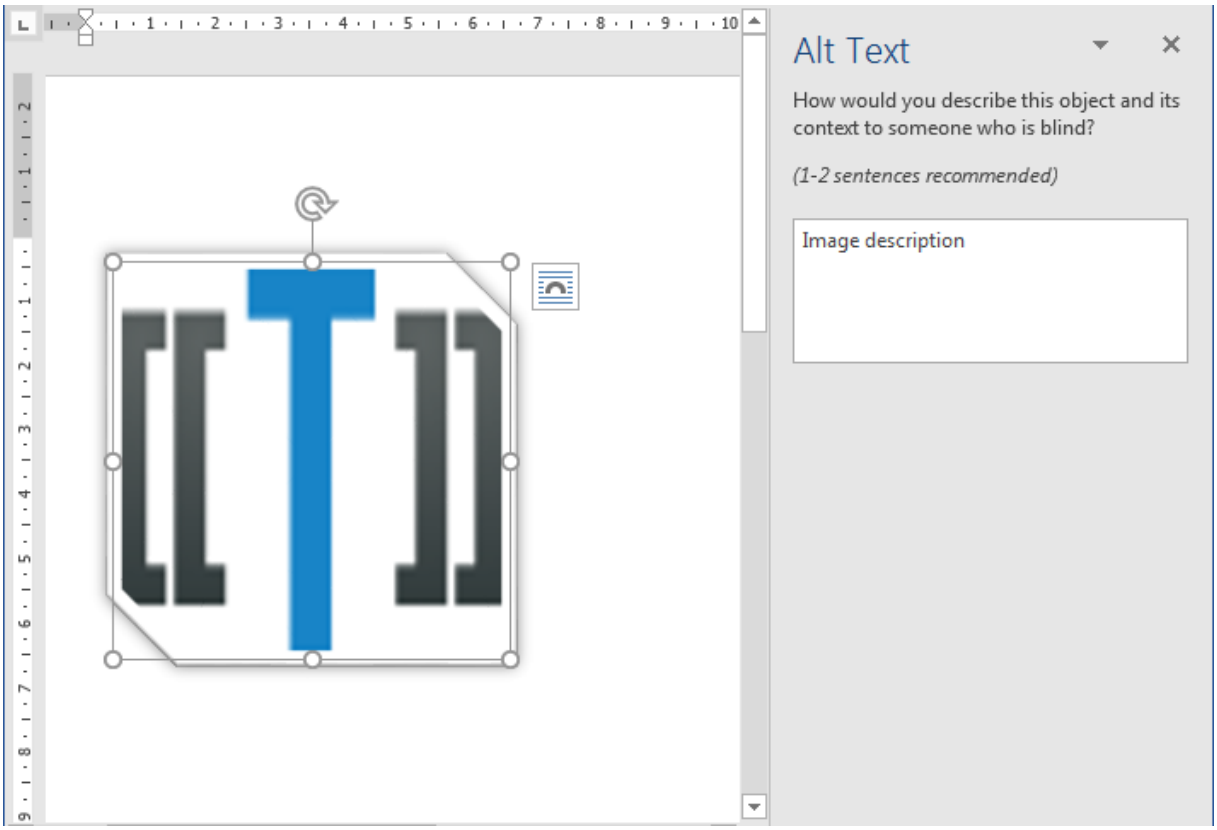
The image files will be included in the ZIP file and referenced from the relevant parts.

Existing images

While passing ImageInfo to Templater does create a new image at the tag location, not all configuration options can be accessed this way. If special image style needs to be retained, such as text wrap, 3D format or any other image specific configuration, this can be implemented by preparing such existing image in the document and adding tag into its *Alternative text* property:



When run with JSON example logo expected result is produced:



Scalable Vector Graphics images

Microsoft Word 2016 introduced support for SVG standard. This allows for vector instead of raster images which are much more print friendly. Templater will recognize SVG document as long as:

- expected type is used (XDocument in .NET and Document in Java)
- appropriate node name is used (svg with relevant namespace)

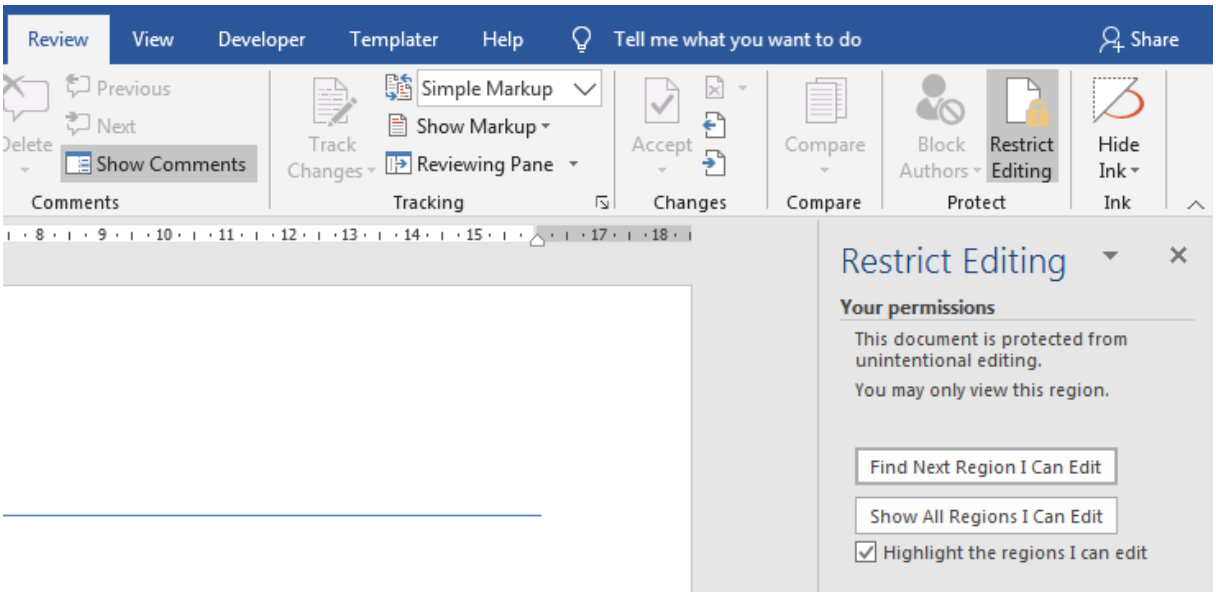
To support old Word versions, raster image can be provided as fallback. This is done by registering image conversion during library initialization in *DocumentFactoryBuilder* via *svgConverter* API.

If image fallback is not registered document will work as expected in new Word versions, but older versions will display an empty image.

SVG document can also be used on existing images through Alt Text, as long as those placeholder images are already SVG images.

Document locking

Word has several document locking features. Some of them are only UI locking which allow for underlying XML manipulation. Since document is not encrypted in that case, Templater is still able to modify the document which appears as locked to the user.



Font color and styles

Templater will use font and color from the tag definition when replacing it with a provided value. Sometimes color is dynamic and depends on other factors. In that case XML can be injected into the document which can specify color, background color or any other font attribute during the replacement.

For this to work Word format for coloring must be used, meaning appropriate XML must be passed in. By defining appropriate metadata, it is easy to define such common cases. XML which would be inserted into the document looks like:

```
<w:tc>
  <w:tcPr>
    <w:shd w:val="clear" w:color="auto" w:fill="COLOR" />
  </w:tcPr>
</w:tc>
```

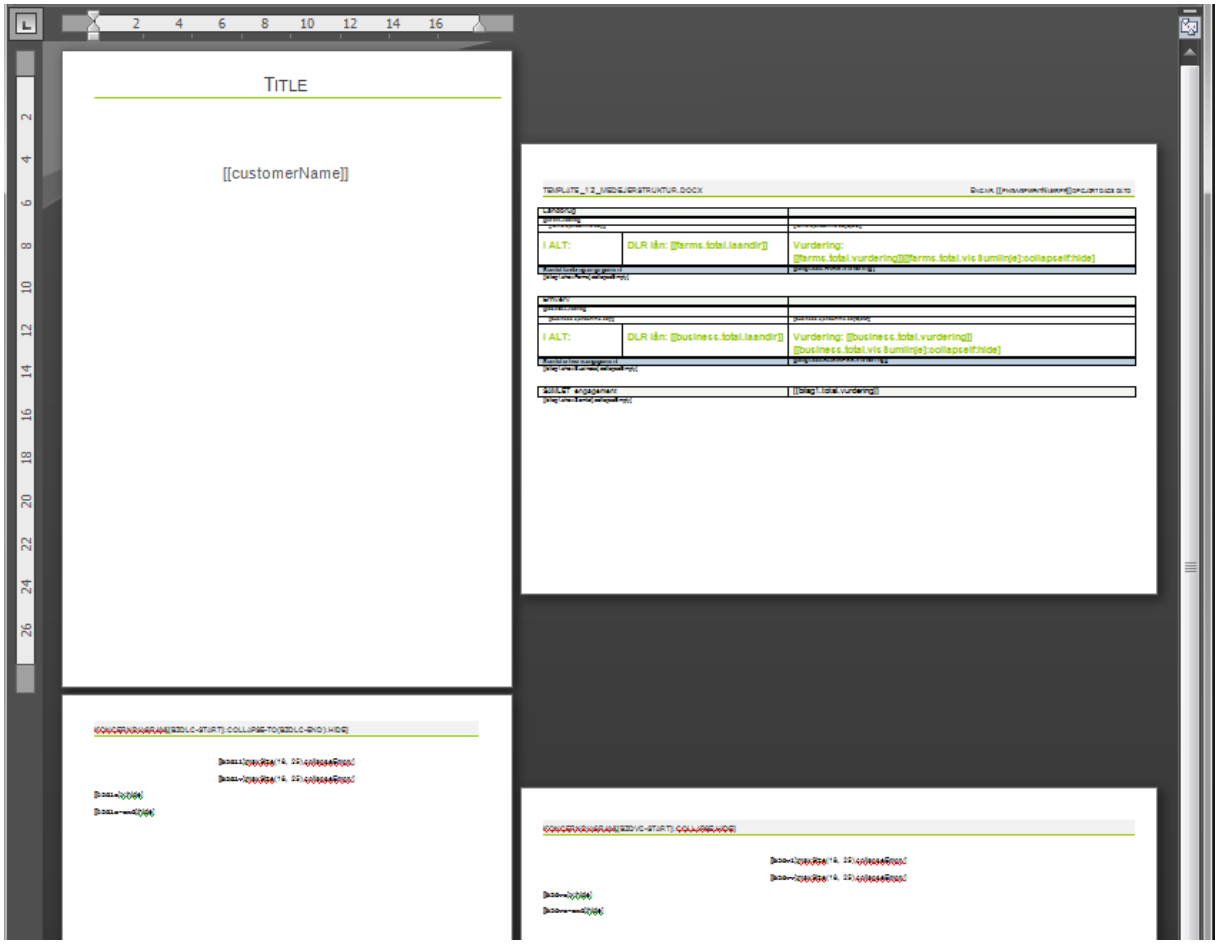
Such pattern can be encapsulated via a metadata or low-level API plugin:

```
private static object ColorConverter(object value, string tag, string[] metadata)
{
  if (value is Color == false) return value;
  var c = (Color)value;
  var fillValue = c.R.ToString("X2") + c.G.ToString("X2") + c.B.ToString("X2");
  return XElement.Parse(@"
<w:tc xmlns:w=""http://schemas.openxmlformats.org/wordprocessingml/2006/main"">
  <w:tcPr>
    <w:shd w:val=""clear"" w:color=""auto"" w:fill=""" + fillValue + @""" />
  </w:tcPr>
</w:tc>");
}
```

Page orientation

In Word page orientation can be changed from page to page (or a single orientation can be used per document). While there is nothing special in Templater to support such use cases, when pages are duplicated or removed sometimes special document adjustment is required to avoid empty pages, which Templater does behind the scenes.

Since page orientation can vary from page to page[31], by combining it with collapse (page removal) complex layouts can be defined, e.g.:



Object numbering

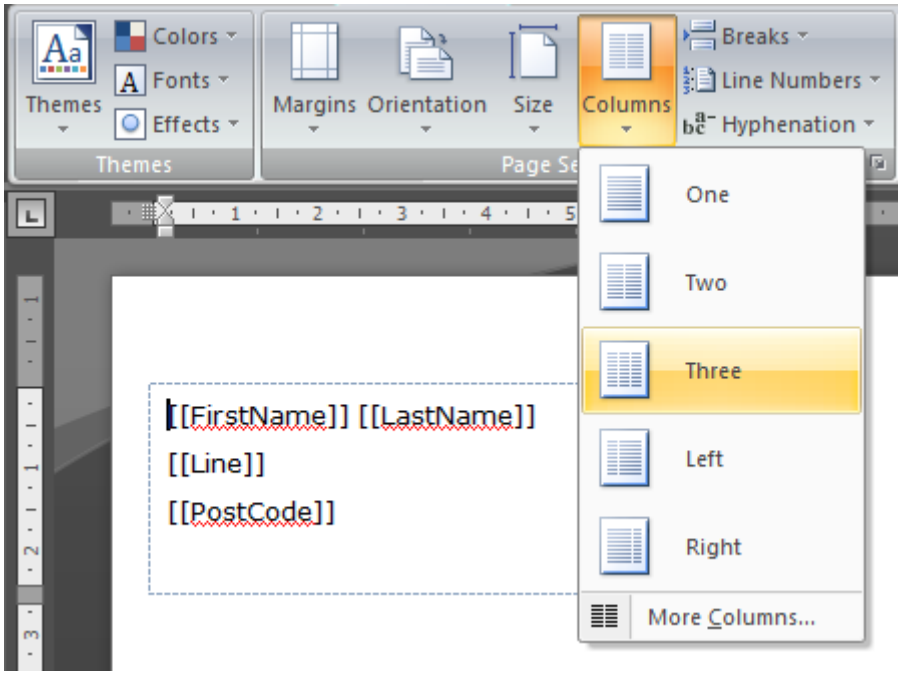
Various objects have internal ID which must be unique per document. Templater will adjust the numbering so there are no duplicates in the document.

Multi-columns

Word can have multiple columns per section of the document. This has variety of layout applications. Templater will cope with various scenarios including duplication of multi-column parts of the document.

[31] A tutorial for page orientation setup can be found here: https://www.officetooltips.com/word_2016/tips/how_to_use_different_page_orientations_inside_one_document.html

Common use case is for label printout:



Content controls

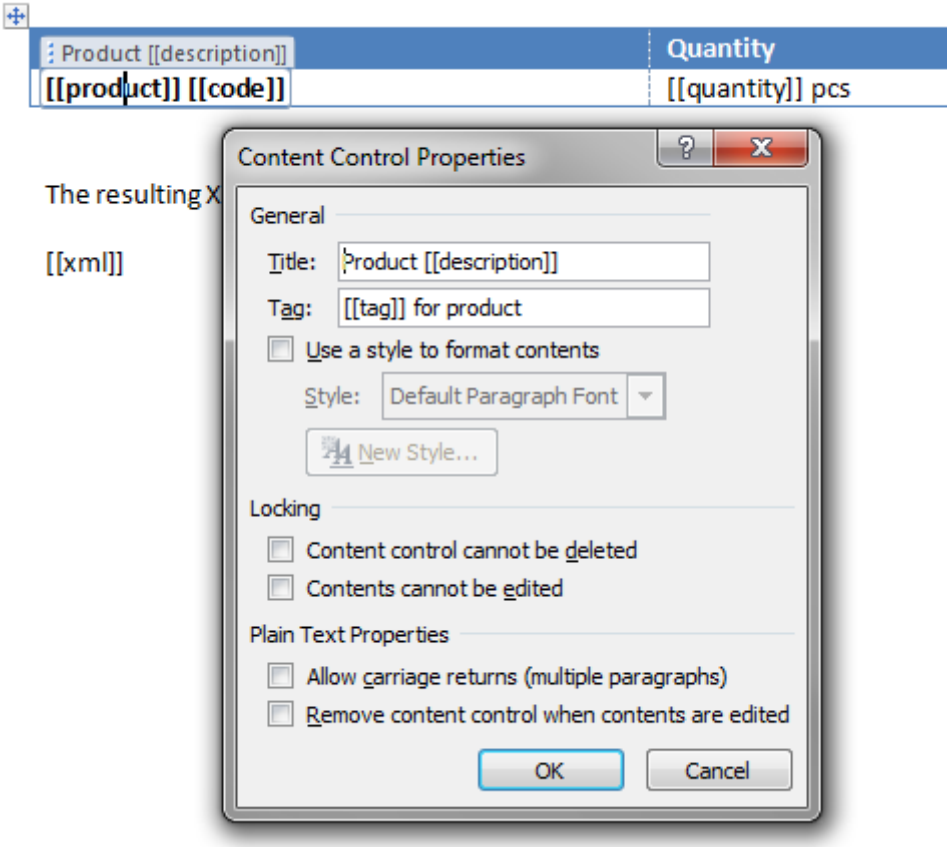
Content controls come in two basic flavors:

- unbounded content controls
- bounded content controls (aka XML binding)

Unbounded controls behave as any other special control with tags.

Bounded controls reference an embedded XML file which must be changed instead of the actual Word document. This requires some special handling and is the only exception where low level Replace can replace “multiple tags” at once - since all tags point to the same underlying value.

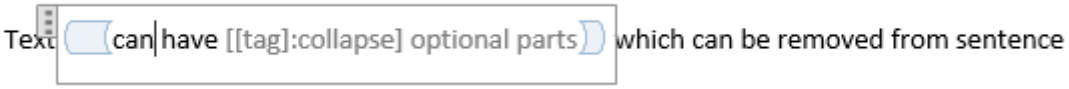
Content controls are set-up via appropriate properties window, e.g.:



Bounded controls support resizing in which case XML elements will also be duplicated.

There is special behavior with content controls on removal, as long as a single content control is referenced. Instead of looking up for surrounding region of the document, when all tags are located within the same content control, only the content control will be removed.

This allows for simple way to have surrounding text around the tag which is conditional, eg:



This behavior can also be used in lists, or tables as long as only content control is affected.

Content controls can be nested, or can contain other tables and/or lists. This makes them easy to manage alternative for conditional region of the document compared to section/page breaks.

Embedding text documents

Templater v6.1 added support for embedding documents inside Word document, which allows for various complex scenarios:

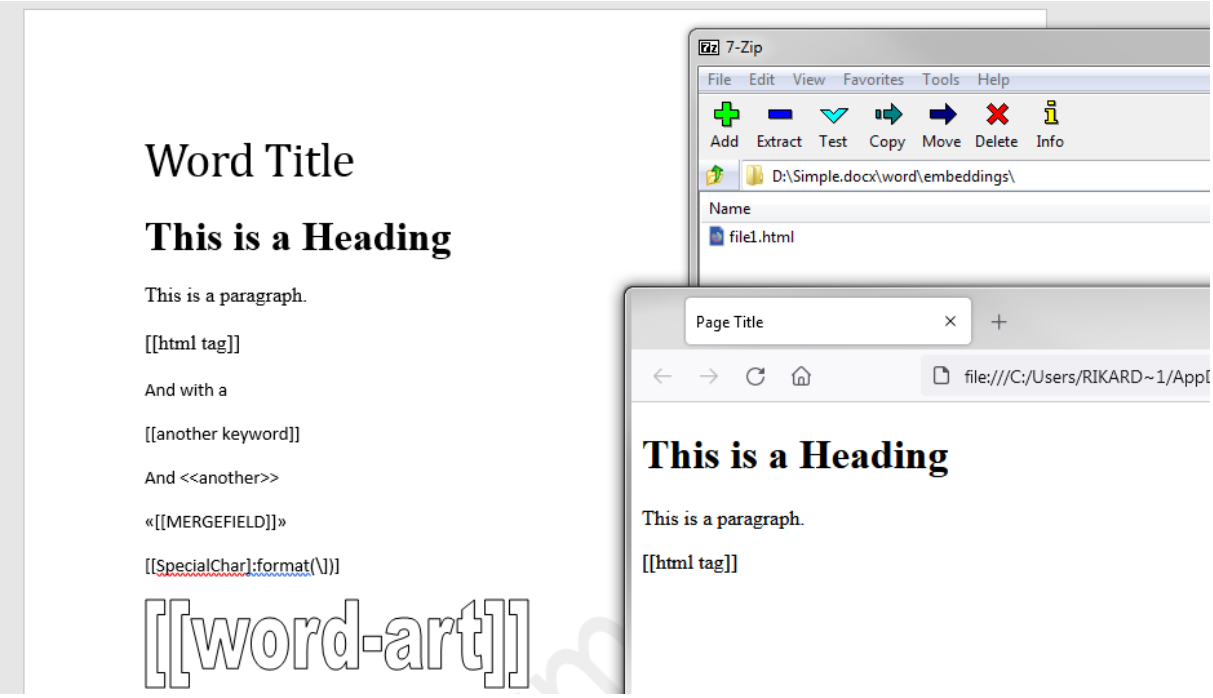
- HTML/DOC/RTF display within Word without the need for any kind of HTML -> OOXML transformations

- Conditional document embedding – when there are lots of smaller documents around, which need to be assembled into one master document, sometimes its easier to first aggregate all subdocuments into one master document, than to maintain all subdocuments as part of master document
- Merging documents – this is a common use case, where multiple documents are embedded in master document

Since tags work transparently inside embedded documents (even for HTML/XML/TXT extensions) Templater can process all combined documents through a single/standard interface.

Documents can be embedded via *System.IO.FileInfo* (.NET) or *java.io.File* (Java) types. Tags will not be available after the import, so document needs to be closed and reopened. All standard operation will continue to work (duplication of embedded document works fine – and duplicated tags are immediately available for use). Documents are embedded via Word AltChunk[32] feature, which does not work in some alternative editors (such as LibreOffice).

If one would open docx file and inspect its content it would find files embedded within /word/embeddings/ folder.



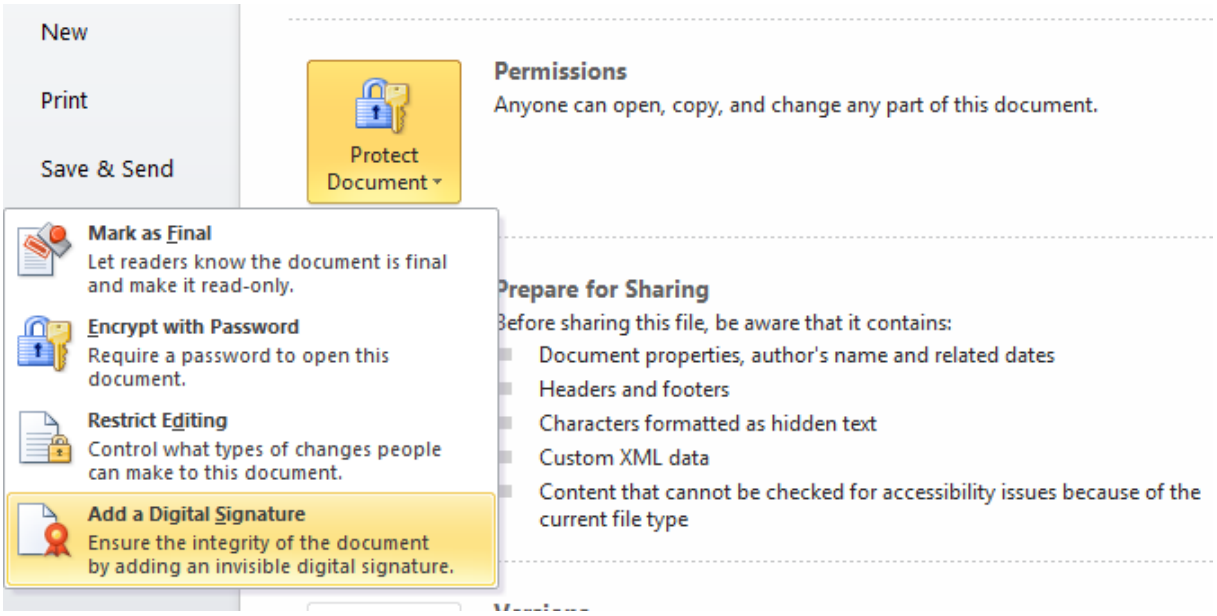
HTML embedding has few improvements compared to other extension types. They will be embedded in place whenever possible, which makes it useful to use them in tables and other resizable elements.

Digital signature

Templater can sign Word document when appropriate certificate is provided paired with a private key used for signing. Document can only be signed only if it was not already signed (even with just

[32] Word can have problems when there are too many AltChunks defined in the document

signature lines). Once signed document will be marked as final (showing the intent that it should not be further modified) with an attached signature. Signature will be valid as long as certificate is recognized by operating system/certificate authority and signed parts of the document have not been modified. This can be done manually through the UI:



Known issues

If a specific Word feature is not supported, there are few basic categories it falls into:

- feature requires Word rendering engine and thus it's not supported
  - such features include PDF export, TOC renumbering, etc...
- feature is on the roadmap, but it's not supported yet
  - an example would be support for comments or some special chart
- feature is not behaving as expected due to a bug

PDF export

A very common use case is to convert Word document into PDF. Unfortunately, this requires a Word rendering engine to work correctly.

There are several free and paid libraries which have sufficiently good PDF conversion for simple documents. But non-trivial documents quickly become non pixel-perfect during the conversion.

Whenever user can convert Word document into PDF this should be preferable. If PDF conversion needs to be done on the server and there is no access to Microsoft libraries for PDF conversion[33] the next best thing is to use proved library such as Aspose. For low-budget solutions the best alternative

[33] SharePoint allows for on demand PDF conversion: https://social.technet.microsoft.com/wiki/contents/articles/15731.sharepoint-2013-new-features-in-word-automation-services.aspx

to run LibreOffice in headless mode and use it for conversion. For such purpose there is a Dockerfile paired with Templater server. For showing document directly in the browser we suggest Silurus.

Table of Contents

As with PDF export correct updating of Table of Contents page numbers requires rendering engine, as Word does not recalculate them on load (only on print[34]). Alternative way to update TOC on load is to add macro to the document, but requires explicit consent by the user before it can be updated.

Various 3rd party solutions which are able to export to PDF can also update Table of Contents.

Embedded Excel document

While chart is also an embedded Excel within the Word, when Excel is embedded into the document, only the image of the Excel sheet is shown in the document. If underlying Excel is changed, the picture will not be updated.

[34] TOC is a field which are updated only before printing

Excel features

Templater has extensive support for many Excel features. Several advanced features are supported by Templater just updating their underlying data source and Excel refreshing them on load. Really large Excel files can be created, as long as some best practices are followed.

Complex non-streaming documents

While Excel is sometimes used just as a single sheet with lots of rows displaying some tabular data, this is often better to do in a plain CSV format which can be opened within Excel. Templater can create huge CSV documents due to support for streaming, while it will keep Excel in memory during processing.

Sometimes it's not clear what benefit does Templater provides to managing Excel, as there are various libraries for specifying cell value and thus it's rather easy to build a simple application for populating Excel with tabular data. But as Templater approach is to bind data with existing templates, instead of programming layout through code, once non-trivial features gets used and managed by Templater it quickly becomes obvious:

- rewriting the formula expressions as cells, ranges and table are copied/moved around
- propagating cell and row styles as regions are pushed around
- adjusting the tables, charts, merge cells and named range sizes during various resize operations
- handling same tag at various places, either as a simple replace or as a part of a collection
- supporting conditional formatting, comments, hyperlinks and various other simple and complex features
- duplicating sheets containing various tables, graphs, charts and other complex Excel features

Templater can be used to create really large Excel files, although they can require significant amount of memory[35]. Since Templater support processing cancellation, processing Excel files can be stopped in case of some problems, such as lack of memory.

Embedded CSV files within xlsx document can be used to process huge data sets, as PowerQuery (Get & Transform) can be set-up in a way to load such embedded CSVs and show them in Excel query tables. Templater will analyze and process such embedded CSV files in /xl/embeddings/ part of the xlsx similarly to handling of Word charts through embedded xlsx within docx.

[35] Templater will use various optimizations to keep memory down, such as XML streaming for sheets, streaming of ResultSet and similar types, but it can still take few GB of memory to create xlsx of 100MB

Resizable behavior

In Excel, where cell is the basic element, everything is considered resizable (unlike in Word where use of tables and lists is required to consider a region resizable). Still, some elements affect the decisions Templater will make, such as use of actual tables, named ranges and merge cells.

Cell range

After the initial analysis Templater has information about every tag location. When a resize is called a matching range is being detected. There are different behaviors depending on where the tags were located:

- all tags within the same sheet - result in range which encapsulates all of them
  - other elements such as named ranges and merge cells can influence the initial minimum spanning range and increase it - this is done so that when resize is performed the object influence by resize doesn't get broken
- tags are on different sheets - behavior depends on location of tags and related metadata
  - most of the time sheets will be processed separately, resulting in separate resize operations per sheet
  - if **sheet** or **page** metadata is used, resizing of relevant sheets (all sheets in between) will be performed
- tag within the sheet name - instructs that full sheet resize should be performed
  - while [[tag]] format will not work without sheet name, both {{tag}} and <<tag>> will
- tags in embedded CSV file – they have somewhat special rules which are applied only within a single embedding
  - such tags do not respect some rules, such as tag sharing/cloning since they are not part of any worksheet

Simple range

The simplest range consists from a single tag. While that works as expected, usually range is contained from multiple tags:

	A	B
1	{{col.A}}	{{col.B}}
2		

when paired with matching input:

```
{
 "col": [
  {"A": "A1", "B": "B-1"},
  {"A": "A2", "B": "B-2"}
 ]
}
```

will result in appropriate cells

	A	B
1	A1	B-1
2	A2	B-2
3		

Matching range within a sheet

Tags can be repeated multiple times and matching range must include all specified tags. This means that cells can span multiple rows and thus duplication context can be non-trivial. Templater will match tags even if they are somewhat outside of the “expected” place:

	A	B	C	D	E	F	G
1							
2		{{col.A}}	{{col.B}}	{{col.A}}			
3						{{col.C}}	
4							
5							

when paired with matching input:

```
{
 "col": [
  {"A": "A1", "B":"B-1", "C":"C-1"},
  {"A": "A2", "B":"B-2", "C":"C-2"},
  {"A": "A3", "B":"B-3", "C":"C-3"}
 ]
}
```

will result in appropriate cells

	A	B	C	D	E	F	G
1							
2		A1	B-1	A1			
3						C-1	
4		A2	B-2	A2			
5						C-2	
6		A3	B-3	A3			
7						C-3	
8							
9							

Several non-trivial features are visible in the example above:

- Templater can work with context consisting from multiple rows/columns
- cell style will be replicated (colors, font properties, etc...)
- row style will be replicated (height, etc...)
- tags can be repeated within the context

Repeating ranges across sheets

Same tag can be repeated across different sheets. It will behave accordingly to the matching input type:

- when input is a simple object, same value will be repeated across all sheets
- when input is a collection, collection within each sheet will be processed separately
- if **page** or **sheet** metadata is used, or a tag is placed within a sheet name, entire sheet will be duplicated, instead of cell range within a sheet
- if **clone** metadata is used, all current sheets will be duplicated

Finding best range match

Some Excel features influence the behavior of choosing a repeating range, such as:

- named range
- tables
- merge cells

The reason why such features influence context used for duplication is that Templater will try to avoid breaking declared group of items. Thus, table is considered a group of items, while just cells which look like a table are not.

Pushdown

When collection is resized, depending on the chosen context cells bellow the context will be adjusted. If cells are below the resizing context they will be **pushed** down. Pushdown will also adjust the relevant formulas so they are still correct after the pushdown.

	A	B
1	{{col.A}}	{{col.B}}
2		
3	Total:	{{A+B}}
4		

when paired with matching input:

```
{
 "col": [
  {"A": 1, "B": 4},
  {"A": 2, "B": 5},
  {"A": 3, "B": 6}
 ],
 "A+B":21
}
```

will result in appropriate cells

	A	B
1	1	4
2	2	5
3	3	6
4		
5	Total:	21

As shown in the images, row 3 was moved to row 5 as the cells above them pushed them down due to resize.

Pull-up

If 0 is used for resize instead of cells being pushed down, they will be pulled up if required condition is met: *the affected range includes all cells on specified rows*.

Templater will perform pull-up by hiding relevant rows so it appears the rows bellow have been pulled up.

Relevant input:

```
{
 "zero": [],
 "before":"Start",
 "after":"End"
}
```

will transform input cells

	A	B	C
1	**Before**	{{before}}	
2			
3	{{zero.Tag}}		
4			
5	**After**	{{after}}	

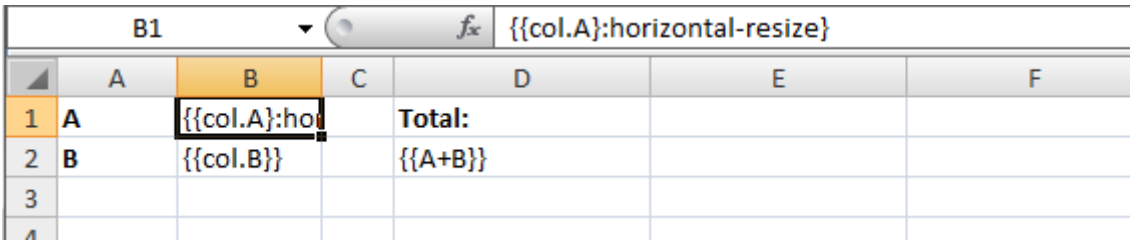
into output with 3rd row is hidden

	A	B	C
1	**Before**	Start	
2			
4			
5	**After**	End	
6			

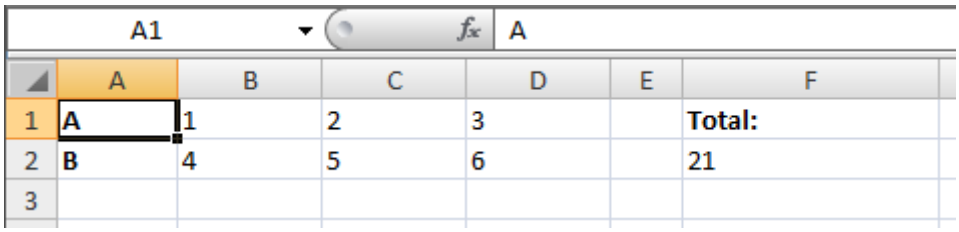
Horizontal resize

By default, calling resize on a set of tags will result in vertical duplication. Sometimes it's useful to do a horizontal duplication (to the right instead of down) with the appropriate push-right instead of pushdown.

To invoke horizontal resize, internal metadata: **horizontal-resize** must be placed at one of the tags being resized:



when paired with input as in previous example will result in expected output:



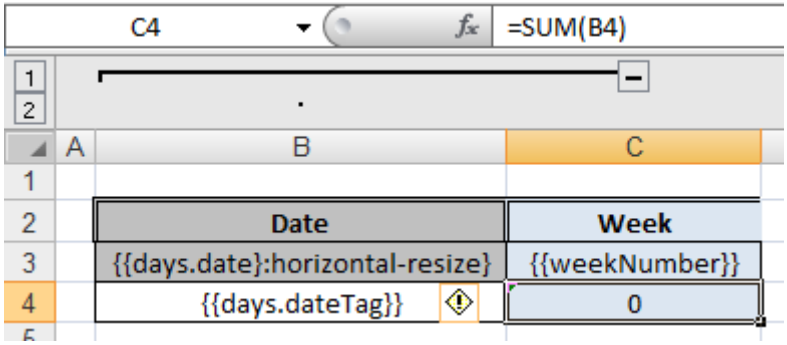
Several relevant things happened during the resize:

- cells were duplicated horizontally instead of vertically
- push to the right instead of push-down was performed on the affected cells right of the context
- column height was preserved during push-right and column duplication
- cell styles were preserved and formulas would be adjusted
- if a special internal metadata **whole-column** was used it would instruct Templater that vertical range used for resizing spans the entire column, instead of just the one matching the tags. This is useful when there is other information in different cells, as it avoids the use of helper tags just for defining appropriate region
- when outline levels are used, *whole-column* metadata is not required, as using outlines will instruct Templater to treat resize on the whole column
- when **resize** 0 is called pull to the left will be performed by hiding relevant columns

Outline levels

Outline levels will work as expected in Templater, with also the benefit on influencing resize on the whole column/row.

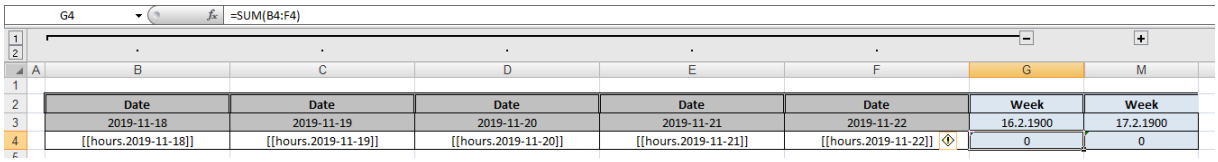
Horizontal resize often requires two passes. A template like:



When processed with the first pass using:

```
[
 {"weekNumber":47, "days":[
  {"date":"2019-11-18", "dateTag":"[[hours.2019-11-18]]"},
  {"date":"2019-11-19", "dateTag":"[[hours.2019-11-19]]"},
  {"date":"2019-11-20", "dateTag":"[[hours.2019-11-20]]"},
  {"date":"2019-11-21", "dateTag":"[[hours.2019-11-21]]"},
  {"date":"2019-11-22", "dateTag":"[[hours.2019-11-22]]"}
 ]},
 {"weekNumber":48, "days":[
  {"date":"2019-11-25", "dateTag":"[[hours.2019-11-25]]"},
  {"date":"2019-11-26", "dateTag":"[[hours.2019-11-26]]"},
  {"date":"2019-11-27", "dateTag":"[[hours.2019-11-27]]"},
  {"date":"2019-11-28", "dateTag":"[[hours.2019-11-28]]"},
  {"date":"2019-11-29", "dateTag":"[[hours.2019-11-29]]"}
 ]}
]
```

will result in intermediary step



	B	C	D	E	F	G	M
2	Date	Date	Date	Date	Date	Week	Week
3	2019-11-18	2019-11-19	2019-11-20	2019-11-21	2019-11-22	16.2.1900	17.2.1900
4	[[hours.2019-11-18]]	[[hours.2019-11-19]]	[[hours.2019-11-20]]	[[hours.2019-11-21]]	[[hours.2019-11-22]]	0	0

with second outline manually collapsed for easier visibility.

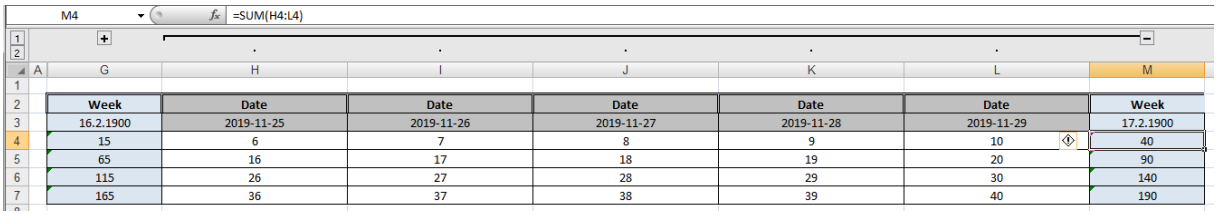
After processing such template with appropriate data structure, such as DataTable with code like

```
using (var doc = factory.Open("step1.xlsx"))
        doc.Process(new { hours = dt });
```

where data table is prepared to match expected column names (2019-11-XX) data, e.g. :

```
var dt = new DataTable();
dt.Columns.Add("2019-11-18", typeof(int));
dt.Columns.Add("2019-11-19", typeof(int));
dt.Columns.Add("2019-11-20", typeof(int));
dt.Columns.Add("2019-11-21", typeof(int));
dt.Columns.Add("2019-11-22", typeof(int));
dt.Columns.Add("2019-11-25", typeof(int));
dt.Columns.Add("2019-11-26", typeof(int));
dt.Columns.Add("2019-11-27", typeof(int));
dt.Columns.Add("2019-11-28", typeof(int));
dt.Columns.Add("2019-11-29", typeof(int));
dt.Rows.Add(new object[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 });
dt.Rows.Add(new object[] { 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 });
dt.Rows.Add(new object[] { 21, 22, 23, 24, 25, 26, 27, 28, 29, 30 });
dt.Rows.Add(new object[] { 31, 32, 33, 34, 35, 36, 37, 38, 39, 40 });
```

will be resized horizontally as expected:



	G	H	I	J	K	L	M
1							
2	Week	Date	Date	Date	Date	Date	Week
3	16.2.1900	2019-11-25	2019-11-26	2019-11-27	2019-11-28	2019-11-29	17.2.1900
4	15	6	7	8	9	10	40
5	65	16	17	18	19	20	90
6	115	26	27	28	29	30	140
7	165	36	37	38	39	40	190

Dynamic resize

Dynamic resize is a Templater specific feature which works in both Word and Excel implementations. Similarly to behavior in Word a tag can be used, which when paired with appropriate input type (jagged array, DataSet/ResultSet,..) will transform a single cell into NxM cells doing both push to the right and pushdown in the process:

	A	B	C
1	{{dr}}		Right of
2			
3	Bellow		

when paired with matching input:

```
{
 "dr": [
   ["A1", 1, "B1", 4],
   ["A2", 2, "B2", 5],
   ["A3", 3, "B3", 6]
 ]
}
```

will result in appropriate cells

	A	B	C	D	E	F
1	A1	1	B1	4		Right of
2	A2	2	B2	5		
3	A3	3	B3	6		
4						
5	Bellow					

Dynamic resize also recognizes **whole-column** metadata in which case it will duplicate all cells in a column, not just the ones defined by the minimum spanning region over the relevant tags.

Removing cell range

If **Resize**(tags, 0) is used on a cell range tags will be replaced with an empty string. Depending on the range size there will be pull down (by hiding relevant rows) if tag context spans all cells within those rows; otherwise cells will stay as is (the tags will just be removed).

Tables

Excel can display data set in various ways, either using simple cells range, or using specialized table feature. There are several additional options available when tables are used (and some restrictions) which fit naturally onto certain use cases.

Tables have certain unique properties (compared to plain cells).

- each table has a unique name (across all sheets)
- table maintain their size information
- each column inside a table must have unique name
- formula referencing same row in another column is easier to write/understand and doesn't incur additional overhead due to relative/absolute reference
- merge cells can't be used within table

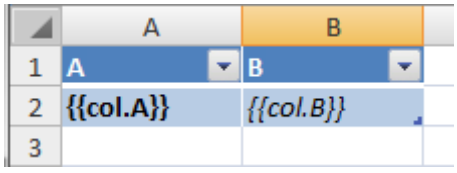
Single row context

Unlike with cell, where context is defined by minimum spanning range, context in a table is always the full table rows. Most of the time context is just a single row, although tags can be defined outside of the table in which case table will be duplicated.

Usual Templater features apply:

- styles will be replicated
- tags can be repeated multiple times
- if same tags have different metadata, different values can be shown
- major difference from tables in Word is that in Excel every cell can have its own format for displaying the value

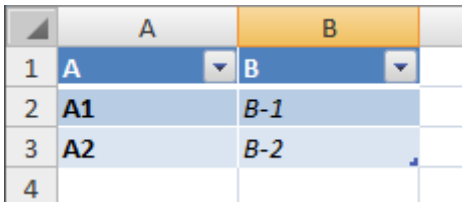
A simple example would look like:



when paired with matching input:

```
{
 "col": [
  {"A": "A1", "B": "B-1"},
  {"A": "A2", "B": "B-2"}
 ]
}
```

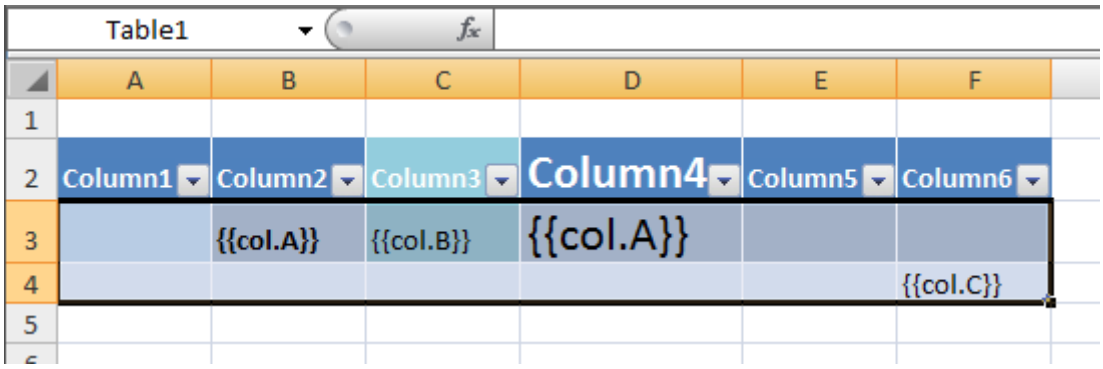
will result in resized table



	A	B
1	A	B
2	A1	B-1
3	A2	B-2
4		

Multi-row context

Table can also be used in multi-row context, although such usages are not as common. An example would look like:

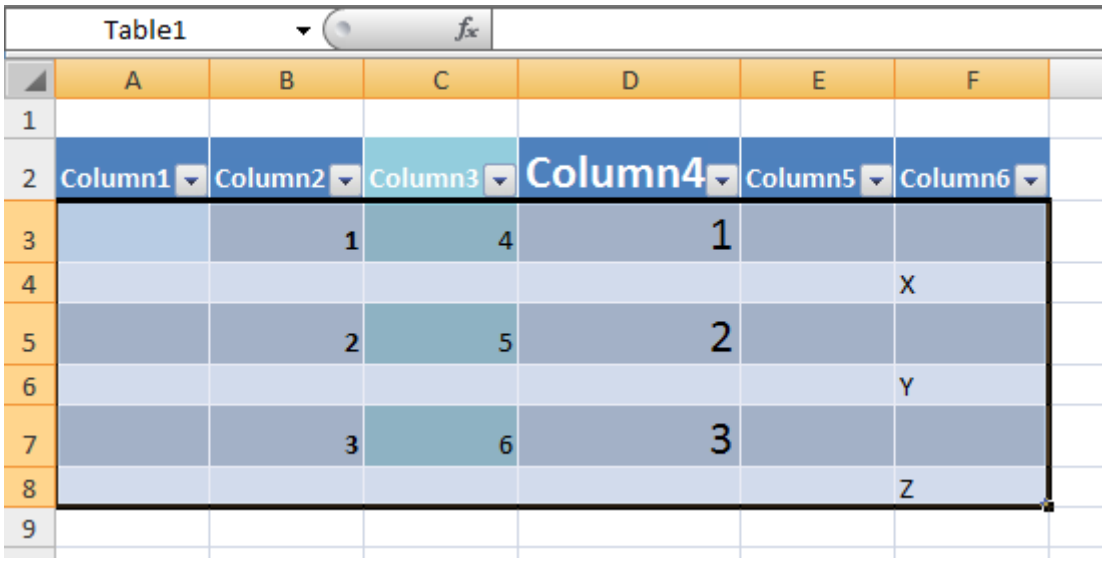


	A	B	C	D	E	F
1						
2	Column1	Column2	Column3	Column4	Column5	Column6
3		{{col.A}}	{{col.B}}	{{col.A}}		
4						{{col.C}}
5						

when paired with matching input:

```
{
 "col": [
   {"A": 1, "B": 4, "C": "X"},
   {"A": 2, "B": 5, "C": "Y"},
   {"A": 3, "B": 6, "C": "Z"}
 ]
}
```

will result in resized table



	A	B	C	D	E	F
1						
2	Column1	Column2	Column3	Column4	Column5	Column6
3		1	4	1		
4						X
5		2	5	2		
6						Y
7		3	6	3		
8						Z
9						

Table resizing behaves almost the same as cell resizing:

- multi-row context is supported
- tags can be repeated
- cell styles will be replicated
- row styles will be replicated

But there are some differences:

- whole table will be used during resize, while cells will find the minimum spanning context
- table size needs to be adjusted

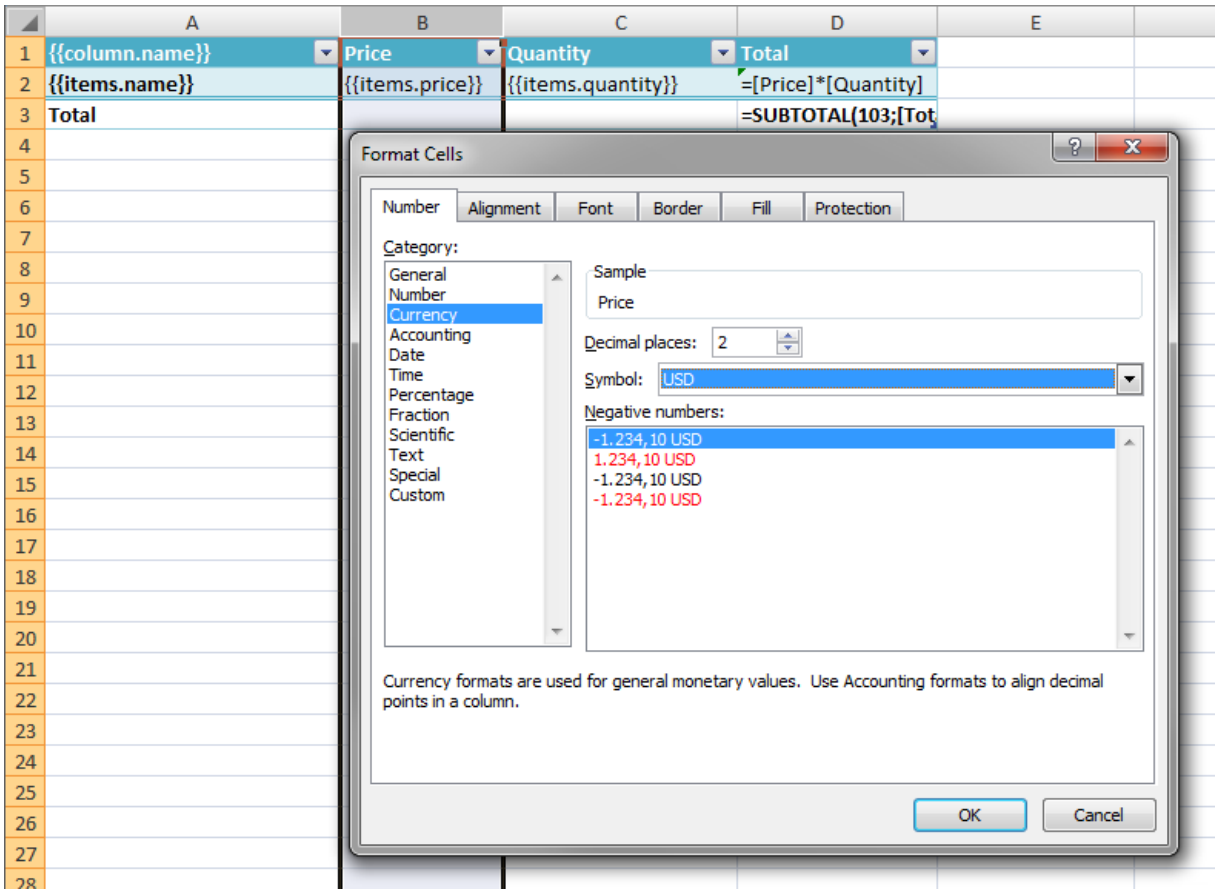
Setting up tables

While most features which are done within tables can be done with plain cells, use of tables makes them much easier to use:

- it's common to setup cell/column style for formatting purpose
- table style allows for quick/easy visual setup of banded rows/columns
- total row supports various simple formulas for aggregating table data

Tags can be used in table headers, which can also be combined with horizontal-resize (for setting up dynamic columns).

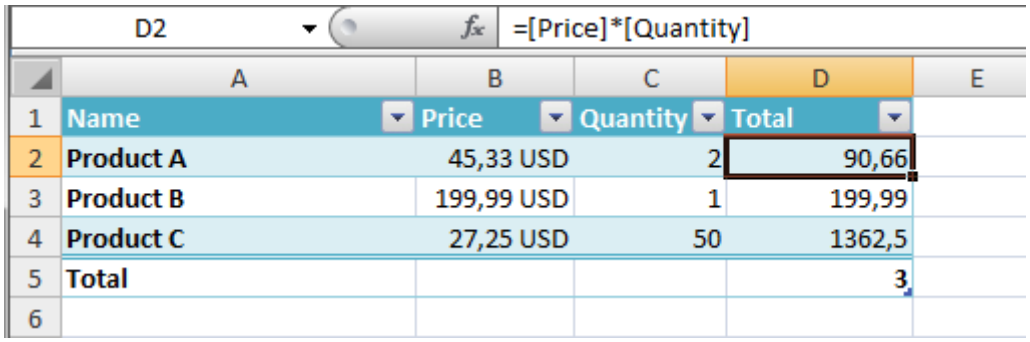
An example of non-trivial table setup could look like:



when paired with matching input:

```
{
 "column": {"name": "Name"},
 "items": [
   {"name": "Product A", "price": 45.33, "quantity": 2},
   {"name": "Product B", "price": 199.99, "quantity": 1},
   {"name": "Product C", "price": 27.25, "quantity": 50}
 ]
}
```

will result in resized table (with Show Formulas disabled under Formulas tab):



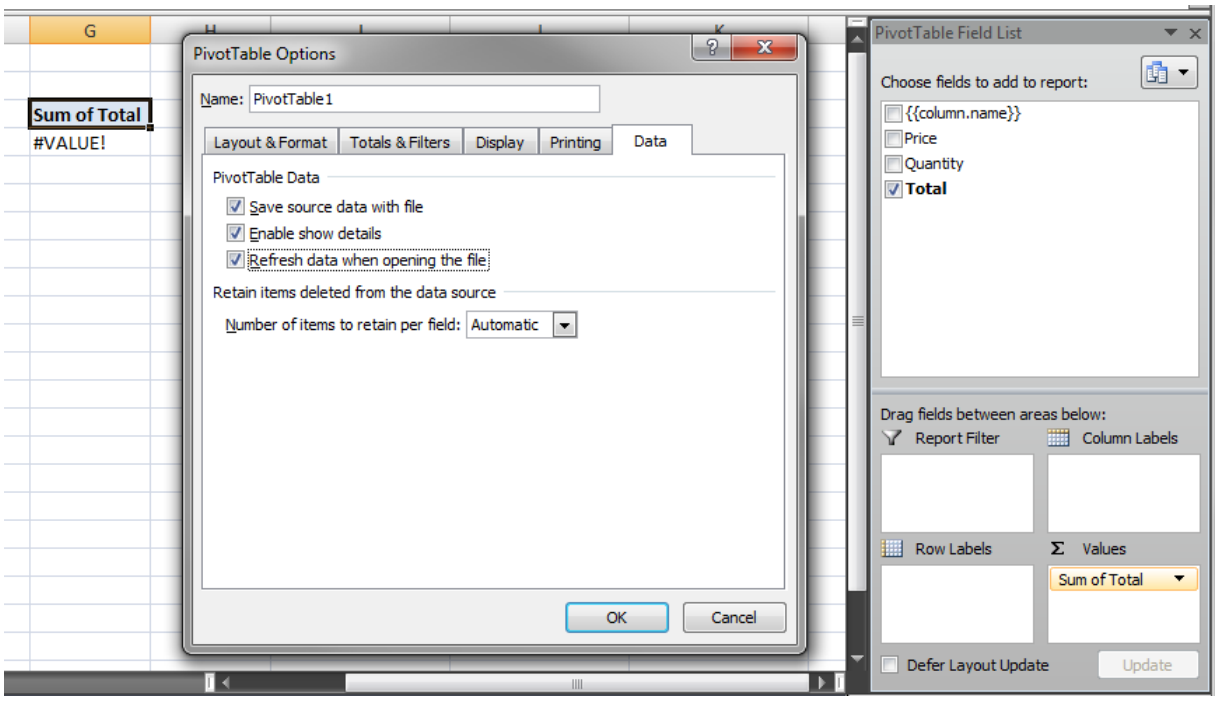
D2 | =[Price]*[Quantity]

	A	B	C	D	E
1	Name	Price	Quantity	Total	
2	Product A	45,33 USD	2	90,66	
3	Product B	199,99 USD	1	199,99	
4	Product C	27,25 USD	50	1362,5	
5	Total			3	
6					

Price column was formatted to show extra USD symbol and 2 decimals, unlike the Total column which does not have formatting and thus shows numbers in default format.

Header filters were automatically updated to list all values within the table, while various formulas are showing useful values without extra input data.

For complex reports tables are just the first step, as they are used as data source for various pivots and charts. To create a pivot data source must be defined, which can be just a table name. Pivots can be refreshed on load, which is useful to re-populate them with actual data after Templater finishes with processing. Refresh option can be set on Data tab in PivotTable Options:



Removing table content

If **Resize**(tags, 0) is called on a table, only the content of the table will be cleared up. Table will still exist with the number of rows as it did in the template. The reason for such behavior is because unlike in Word, in Excel tables must have at least one row.

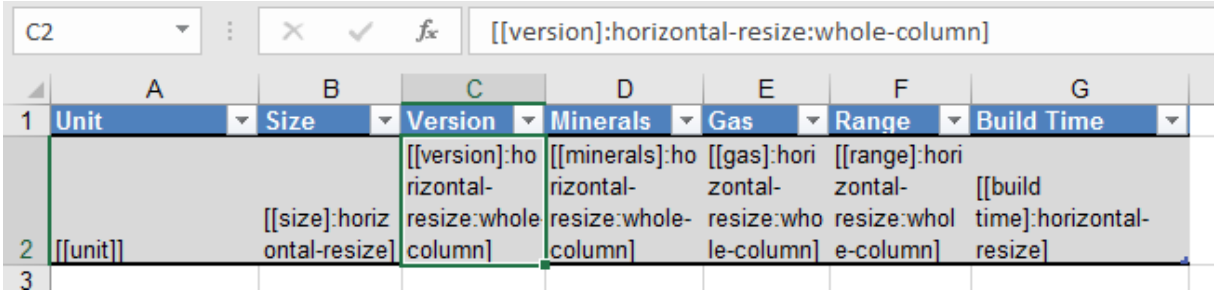
The way to completely remove a table is to put a named range around it. This way when named range is removed, the table will also be removed.

Removing/hiding specific columns

Horizontal resize can be combined with multi-step processing to perform specific column removal (they will actually be hidden). This requires extra manual code or a custom type processor since it's somewhat specific.

By calling multiple ***resize(column, 0)*** operations on relevant columns (e.g., they do not exist in input) and by stripping ***horizontal-resize*** metadata from others setup [can be configured] to process the rest of columns "normally".

Template such as:



can be processed in first pass to hide columns which are not used and in second pass to populate rows via e.g.:

```
[
 {"unit": "Battlecruiser", "size": "L", "minerals": 400, "gas": 300, "build time": 160},
 {"unit": "Firebat", "size": "S", "minerals": 50, "gas": 25, "build time": 24},
 {"unit": "Vulture", "size": "M", "minerals": 75, "gas": 0, "build time": 30}
]
```

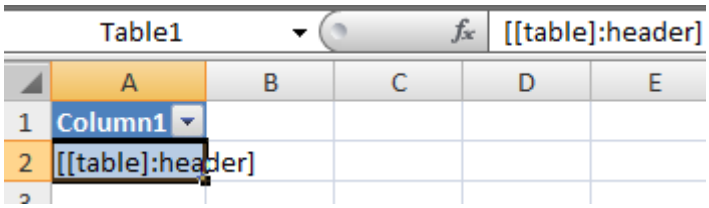
the output will have relevant columns hidden:

	A	B	D	E	G
1	Unit	Size	Minerals	Gas	Build Time
2	Battlecruiser	L	400	300	160
3	Firebat	S	50	25	24
4	Vulture	M	75	0	30
5					

Dynamic resize

Similar to dynamic resize within cells, dynamic resize can be used on a table. There are several special rules when dealing with Dynamic resize in tables:

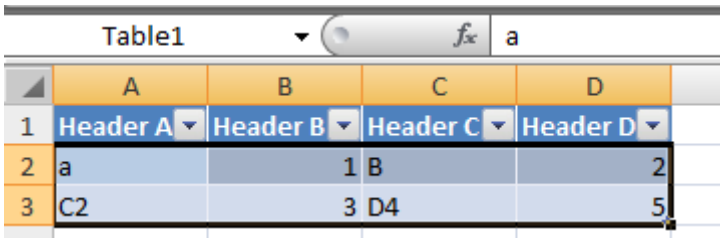
- if ResultSet/DataTable/DataReader is used, it will rename the columns to match the labels in the used data type
- if jagged arrays are used, only data will be populated, while columns will be extended with generic ColumnX names
- to use jagged arrays and replace column names, special metadata must be used: **header**
- Column names in a table must be unique. If Templater detects a column name which might be causing problems (duplicate or empty) it will use generic ColumnX name instead



when paired with matching input:

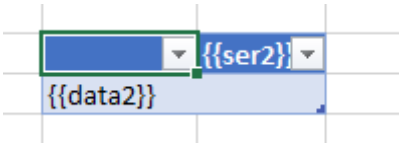
```
{
 "table": [
  ["Header A", “Header B”, "Header C", “Header D”],
  ["a", 1, "B", 2],
  ["C2", 3, "D4", 5]
 ]
}
```

will result in appropriate table



Important aspect of Dynamic resize in tables is that it will not stretch tables, unless input object boundaries go beyond the table (in which case it will stretch the table only to expand the required extended boundary). This is important when creating charts with Dynamic resize as multiple tags within the table are required to prepare the table/chart.

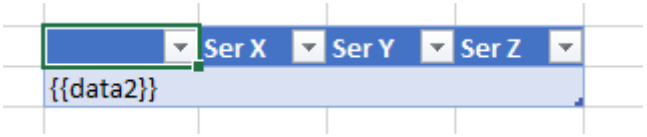
Table with two tags:



when paired with matching input:

{ "ser2": [["Ser X", “Ser Y”, "Ser Z"]] }

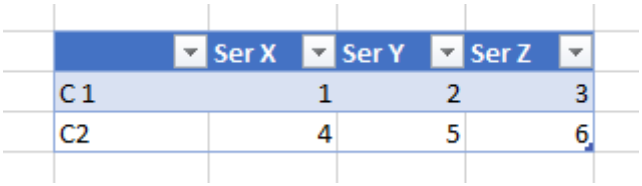
will become



and when this result is paired with relevant input:

{ "data2": [["C 1", 1, 2, 3], ["C2", 4, 5, 6]] }

It will still maintain table size:



Duplicating tables

In several scenarios table can be duplicated. When table is duplicated it will get a new name, since table names must be unique across Excel file.

Named range

Templater will respect named range and will adjust its context detection to include various Excel features during decision making, such as named range. There are various applications for a named range such as:

- defining outer range used for a resize - instead of letting Templater use minimum range spanning all tags
    - often there are cells outside of tags which ought to be included during the resize. Probably the easiest way to include them is to declare named range which will instruct Templater to use it instead of the underlying range (as long as named range has same tags as the underlying one)
- Templater will avoid breaking cells within certain features (such as tables and named ranges)
    - pushdown after the resize will move all cells within the named range, not just the ones directly below the range being resize
- removing a named range will remove all elements inside
    - this way tables and various others elements can be removed

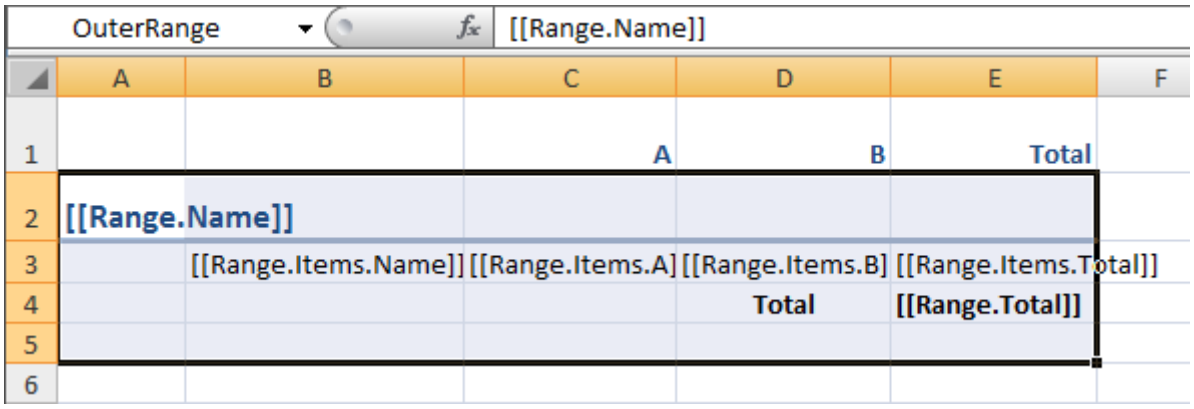
Using named ranges and tables in formula expressions is also more performant and thus recommended over using ranges.

Named ranges can also be hidden (some features such as filters on cells use hidden named ranges).

Fine tuning resize region

Most common use case for named ranges is to tweak the affected range, without introducing additional tags just for that purpose (along with :hide metadata since they are not really used).

A simple example of fine tuning would look like:

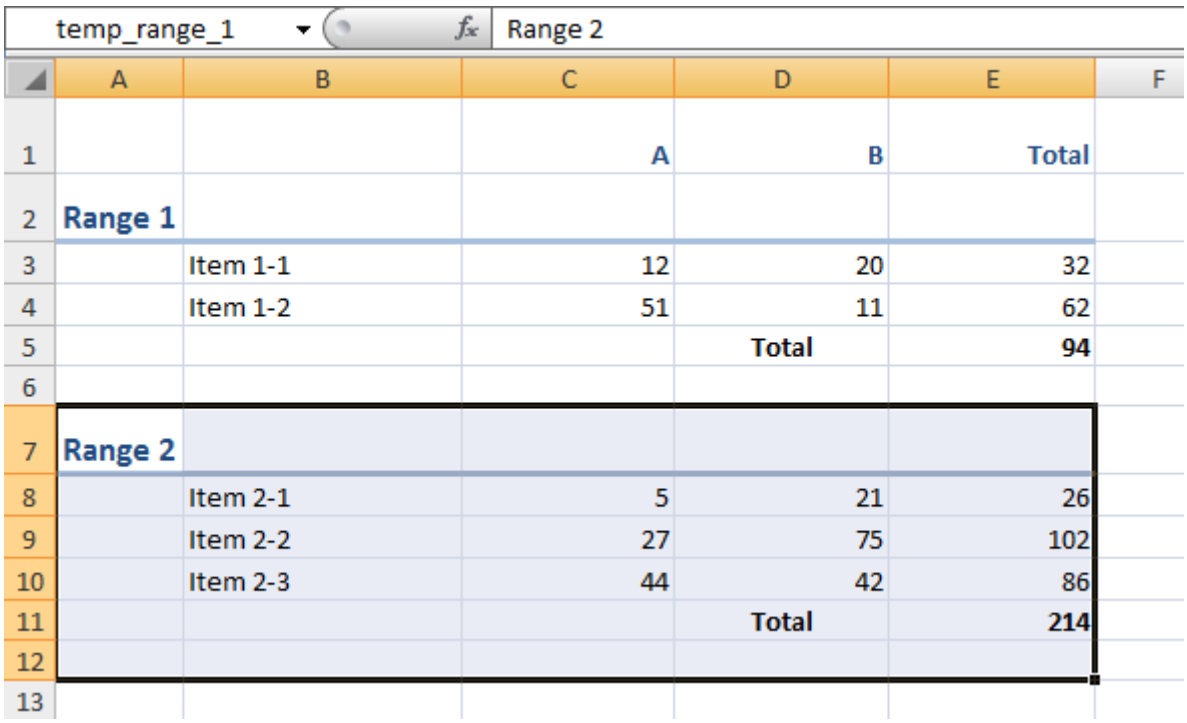


which extends the minimum range from A2:E4 to A2:E5 via an **OuterRange** named range. This example also uses a nested collection which will stretch the named range when the inner collection is resized. When paired with input such as:

```
{
 "Range": [
        { "Name": "Range 1", "Total": 94, "Items": [
                {"Name": "Item 1-1", "A": 12, "B": 20, "Total": 32},
                {"Name": "Item 1-2", "A": 51, "B": 11, "Total": 62}
        ]},
        { "Name": "Range 2", "Total": 214, "Items": [
                {"Name": "Item 2-1", "A": 5, "B": 21, "Total": 26},
                {"Name": "Item 2-2", "A": 27, "B": 75, "Total": 102},
                {"Name": "Item 2-3", "A": 44, "B": 42, "Total": 86}
```

```
        ]}
 ]
}
```

will result in output with a new named range:



	A	B	C	D	E	F
1			A	B	Total	
2	**Range 1**					
3		Item 1-1	12	20	32	
4		Item 1-2	51	11	62	
5				**Total**	**94**	
6						
7	**Range 2**					
8		Item 2-1	5	21	26	
9		Item 2-2	27	75	102	
10		Item 2-3	44	42	86	
11				**Total**	**214**	
12						
13						

Named range must have a unique name within the Excel file and thus Templater will give it name which starts with **temp_range_**. While Templater could remove such named ranges, they are left within the document for cases when documents are processed multiple times.

Since the named range is larger than minimum range, there will be extra row after each resize, as it was specified by named range.

Both ranges were stretched to accommodate the resized collections within them.

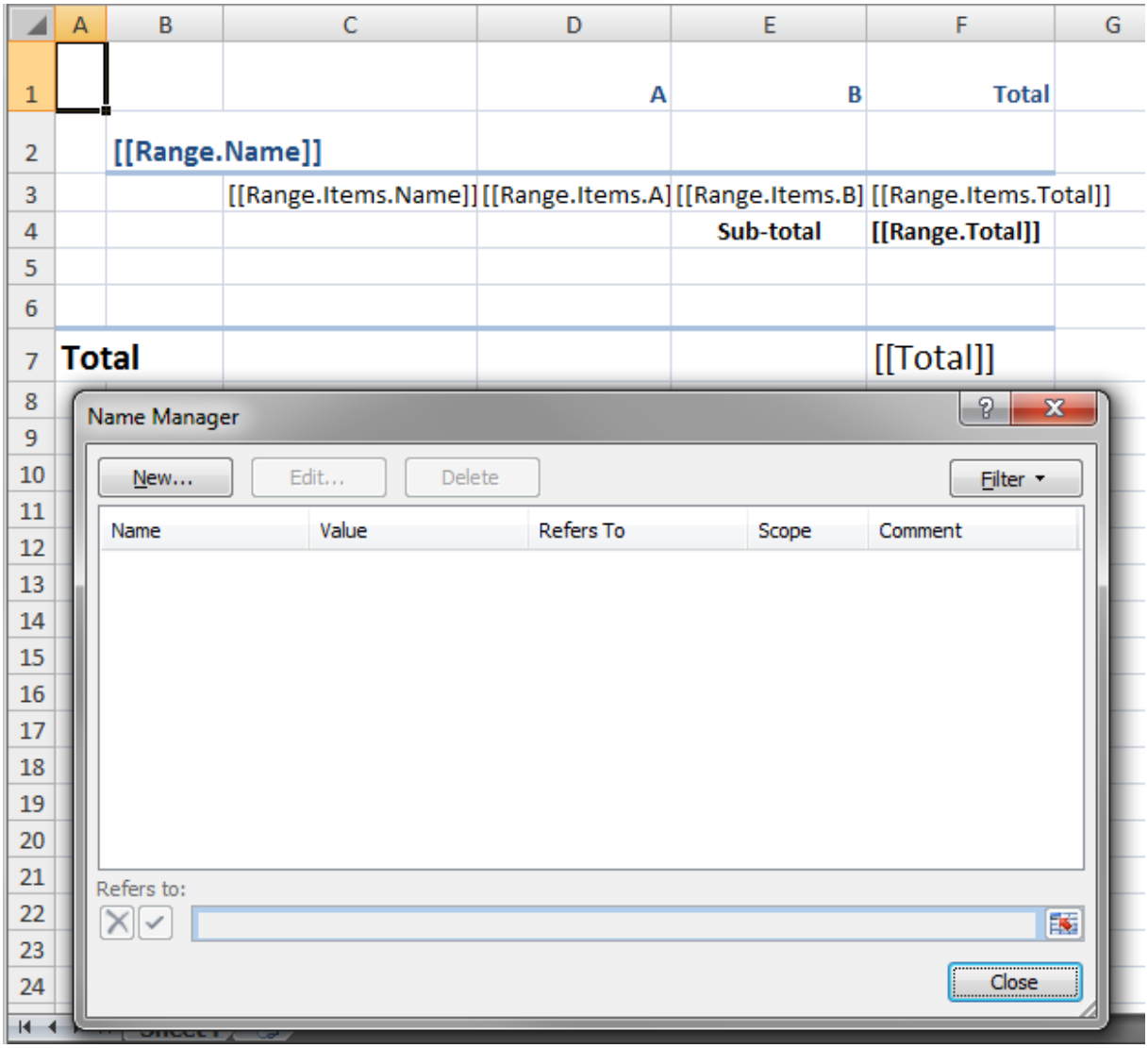
Preventing range splitting

Another common use case for named ranges is to prevent “unexpected” cell pushdown. Often after the resize a pushdown can break-up totals which were put at the end. There are several ways to prevent such split:

- use table for totals
- named range around the totals
- merge cells at/above the totals

In all of those cases, the underlying theme is that pushdown will try to push only cells directly bellow itself. To prevent this named range can be put at the place where we want range to behave as a unit and thus by giving it a name we will prevent “unexpected” behavior.

This is quite common in nested ranges which have inner collections and thus partial resize can break the document layout:



In the above example there are no named ranges (or tables or merge cells) in the document. Thus when **Range.*** is resized, only B2:F4 will be used as a range. This will result in moving of B7:F7, while A7 will remain at its original position. The resized document when paired with appropriate input:

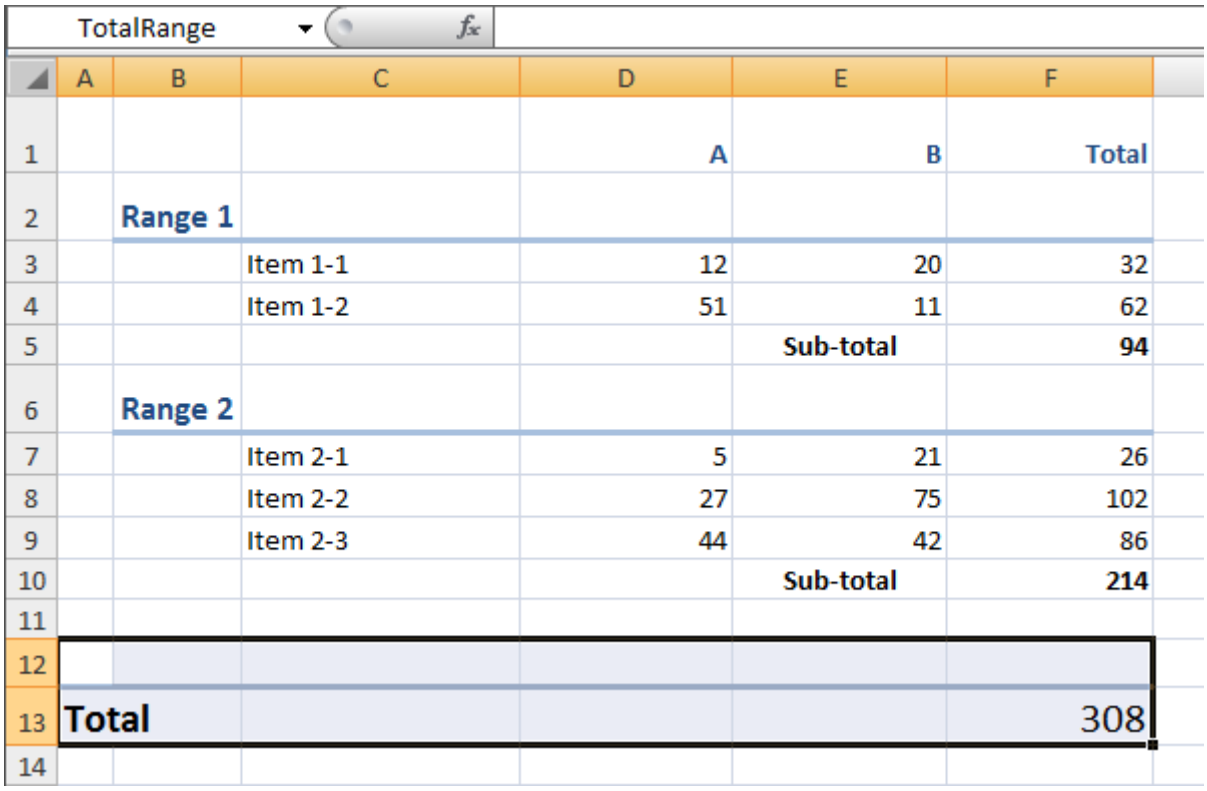
```
{
 "Range": [
        { "Name": "Range 1", "Total": 94, "Items": [
                {"Name": "Item 1-1", "A": 12, "B": 20, "Total": 32},
                {"Name": "Item 1-2", "A": 51, "B": 11, "Total": 62}
        ]},
        { "Name": "Range 2", "Total": 214, "Items": [
                {"Name": "Item 2-1", "A": 5, "B": 21, "Total": 26},
                {"Name": "Item 2-2", "A": 27, "B": 75, "Total": 102},
                {"Name": "Item 2-3", "A": 44, "B": 42, "Total": 86}
        ]}
 ],
```

"Total": 308
}

will look broken:

	A	B	C	D	E	F
1				A	B	Total
2		Range 1				
3			Item 1-1	12	20	32
4			Item 1-2	51	11	62
5					Sub-total	94
6		Range 2				
7	Total		Item 2-1	5	21	26
8			Item 2-2	27	75	102
9			Item 2-3	44	42	86
10					Sub-total	214
11						
12						
13						308
14						

To fix this output it's sufficient to define named range as A6:F7 (A6 is used instead of A7 since border was defined in row 6) which will prevent only subset of region to be pushed down. End document will look as expected:



	A	B	C	D	E	F
1				A	B	Total
2		**Range 1**				
3			Item 1-1	12	20	32
4			Item 1-2	51	11	62
5					**Sub-total**	**94**
6		**Range 2**				
7			Item 2-1	5	21	26
8			Item 2-2	27	75	102
9			Item 2-3	44	42	86
10					**Sub-total**	**214**
11						
12						
13	**Total**					308
14						

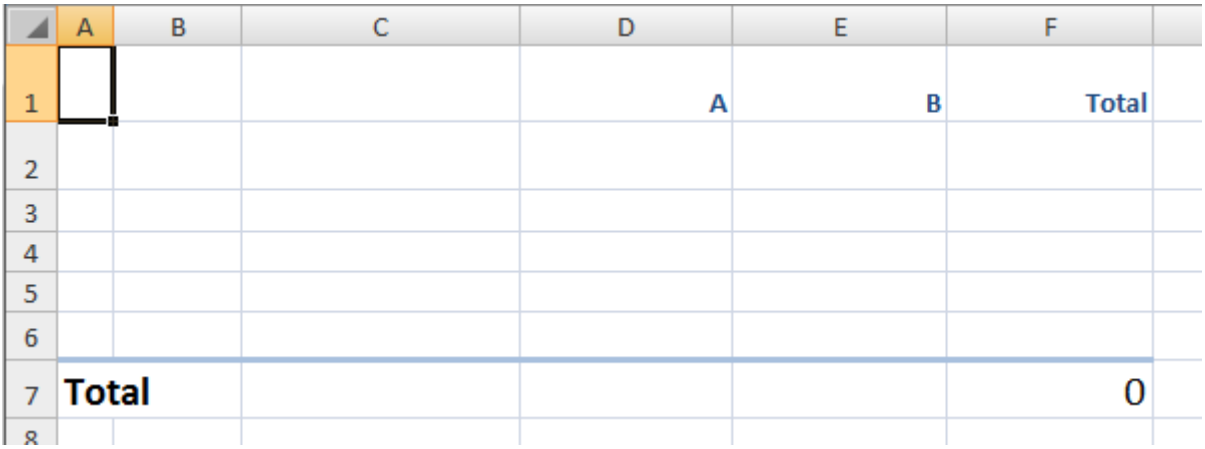
Removing a named range

Named range will only be removed if all tags are specified during **Resize**(tags, 0) operation. When named range is removed pull-up can also be performed when affected range includes all cells in relevant rows. If pull-up is performed, rows will become hidden.

If previous example was used with a different input:

```
{
 "Range": [],
 "Total": 0
}
```

will have part of rows cleared:



	A	B	C	D	E	F
1				A	B	Total
2						
3						
4						
5						
6						
7	**Total**					0
8						

and with pull-up performed hidden:

	A	B	C	D	E	F
1				A	B	Total
5						
6						
7	**Total**					**0**
8						

Excel specific features

There are various other Excel features Templater recognizes and can manage. Some of them are simple, while others can be quite complex.

Formulas

Cell in Excel can display a fixed value, or have a formula which in the end display value based on the evaluated formula expression. Formula expression can be quite complicated and contain:

- reference to other cells, ranges, tables or named ranges
- fixed values
- mathematical expressions
- call into functions

Excel will put cached value in the same cell where the formula is defined. Some Excel viewers know how to recalculate/evaluate formulas, but most of them do not and require Excel for evaluating the formulas again.

Templater also does not evaluate formulas[36] but it will adjust them accordingly to the changes being done in the document. Those changes can be from very simple, to highly complex:

- pushdown will adjust the relevant cells which were pushed
- resize can create new formulas or change the existing ones
- new formulas sometime use new tables and/or new named ranges

An example of formula adjustment would look like:

	A	B	C	D	E	F
1	Conversion factor:	{{exchange.rate}}				
2						
3	Name	Price	Price (local)	Quantity	Total	Total (local)
4	{{items.name}}	{{items.price}}	=B4*B7	{{items.quantity}}	=[Price]*[Quantity]	=E4*B1
5	Total				=SUBTOTAL(103;[Total	=SUBTOTAL(105;[Total (I
6						
7	Conversion factor:	{{exchange.rate}}			=SUM(Table2[Total])	=SUM(F4)
8						

[36] This feature might be introduced in some future version

when paired when relevant info:

```
{
 "exchange": {"rate": 2.7},
 "items": [
   {"name": "Product A", "price": 45.33, "quantity": 2},
   {"name": "Product B", "price": 199.99, "quantity": 1},
   {"name": "Product C", "price": 27.25, "quantity": 50}
 ]
}
```

will result in formulas which were correctly adjusted:

	A	B	C	D	E	F
1	Conversion factor:	2,7				
2						
3	Name	Price	Price (local)	Quantity	Total	Total (local)
4	**Product A**	45,33	=B4*B9	2	=[Price]*[Quantity]	=E4*B1
5	**Product B**	199,99	=B5*B9	1	=[Price]*[Quantity]	=E5*B1
6	**Product C**	27,25	=B6*B9	50	=[Price]*[Quantity]	=E6*B1
7	**Total**				**=SUBTOTAL(103;[Total**	**=SUBTOTAL(105;[Total (l**
8						
9	Conversion factor:	2,7			=SUM(Table2[Total])	=SUM(F4:F6)

This example has several different cases of formula adjustment:

- new rows copied the formula and pointed to the relevant row[37]
- formulas which referenced cell above the table did not modify that reference
- formulas which referenced cell below the table (which got pushed down) had to modify the reference
- formula which referenced range within a table which stretched due to resize changed their original range from a single cell to a range stretching over all the new cells

As visible in the example above, some formulas did not change at all; like the **[Price]*[Quantity]** and **SUM(Table2[Total])**; while some had to be changed (at least the visual representation which is saved into the resulting document). Whenever possible formulas which do not change during Templater operations are preferable since they will be processed much faster (this is highly noticeable on large Excel files).

Formula reference syntax

Templater recognizes the special cell reference syntax (relative and absolute) and will adjust the formulas accordingly:

- D5 - syntax means that cell can move both horizontally and vertically
- $D5 - syntax means that cell can only move vertically, but column is fixed and will not change
- D$5 - syntax means that cell can only move horizontally, but row is fixed and will not change
- D5 - syntax means that cell is fixed and will not move (neither horizontally or vertically)

[37] Internally Excel uses non-changing representation of such formulas and thus they are not really changed, but rather just point to a different location

Still, this syntax is only for matching the Excel rules with regards to references. Sometimes Templater still needs to adjust the formula even when it uses a fixed syntax

Formulas are only allowed to reference elements within a single Excel file. References to another file are not supported.

New images

If new images need to be inserted into the document this can be done via Templater specific data type: ImageInfo

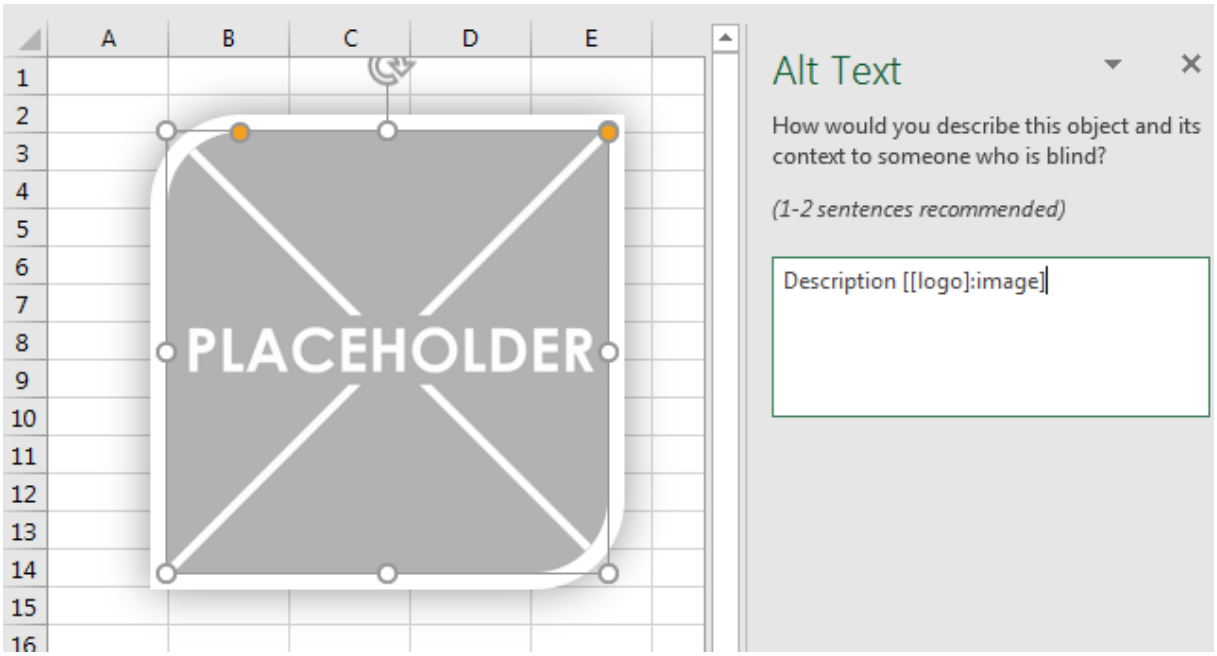
To ease image usage and support platform with custom/different image libraries, default .NET/Java image types are by default converted into ImageInfo type:

- .NET: Image and Icon
- Java: BufferedImage and ImageInputStream

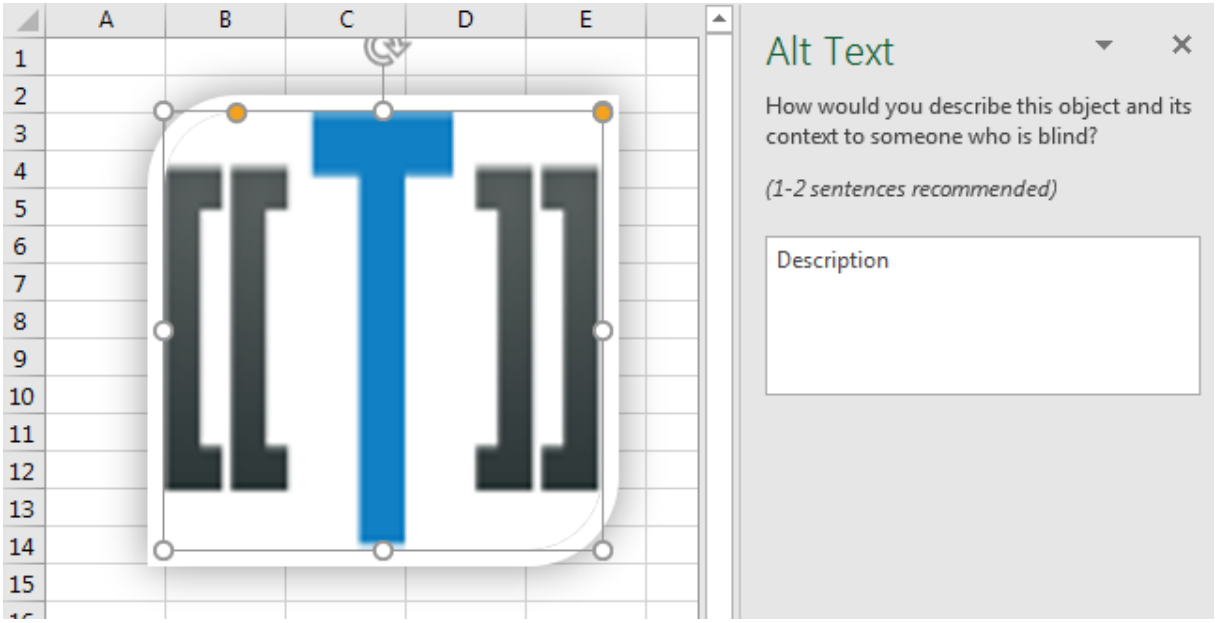
The image files will be included in the ZIP file and referenced from the relevant parts.

Existing images

While passing ImageInfo to Templater does create a new image at the tag location, not all configuration options can be accessed this way. If special image style needs to be retained, such as text wrap, 3D format or any other image specific configuration, this can be implemented by preparing such existing image in the document and adding tag into its *Alternative text* property:



When run with JSON example logo expected result is produced:



Scalable Vector Graphics images

Microsoft Excel 2016 introduced support for SVG standard. This allows for vector instead of raster images which are much more print friendly. Templater will recognize SVG document as long as:

- expected type is used (XDocument in .NET and Document in Java)
- appropriate node name is used (svg with relevant namespace)

To support old Excel versions, raster image can be provided as fallback. This is done by registering image conversion during library initialization in *IDocumentFactoryBuilder* via *svgConverter* API.

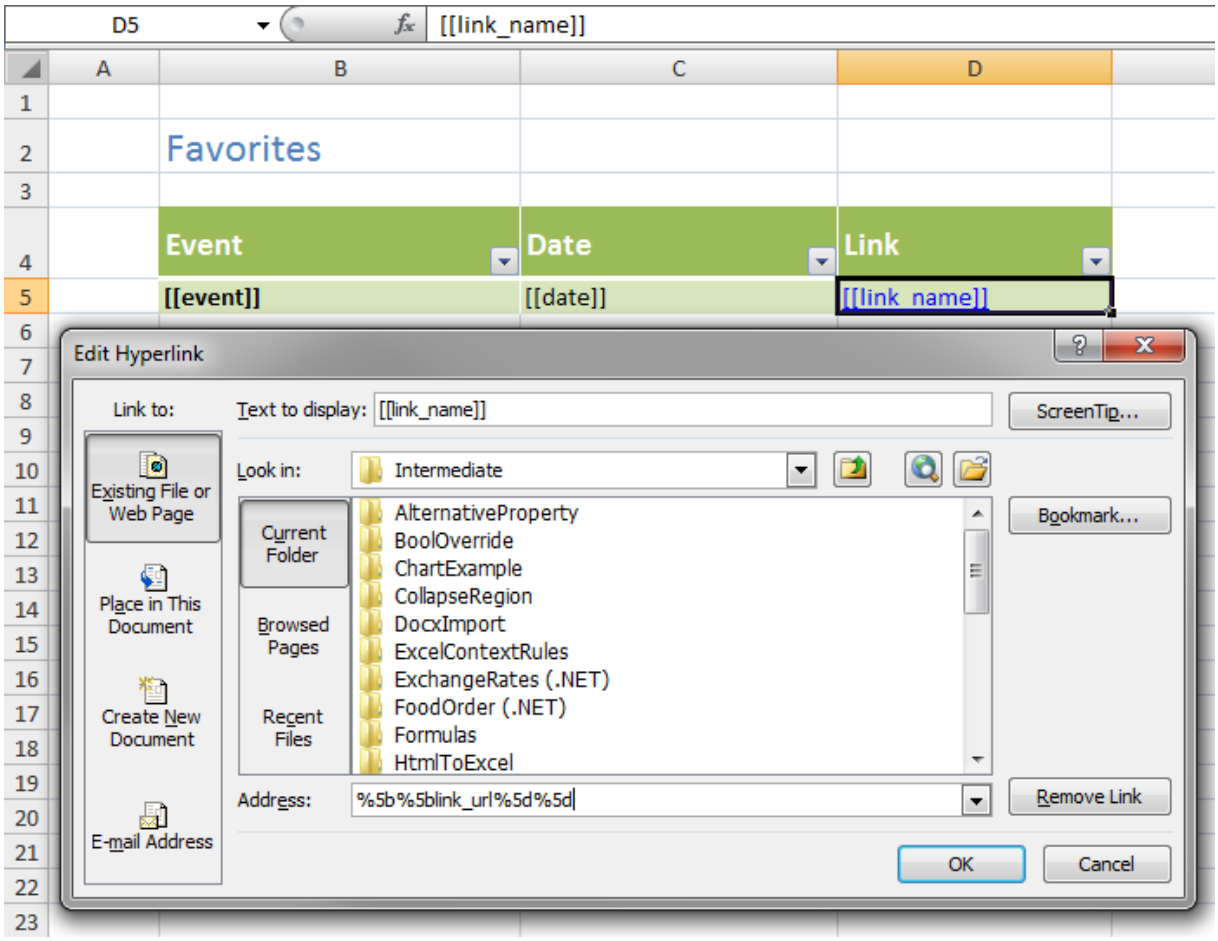
If image fallback is not registered document will work as expected in new Excel versions, but older versions will display an empty image.

SVG document can also be used on existing images through Alt Text, as long as those placeholder images are already SVG images.

Links

Hyperlinks are supported in Excel in the same way as they are supported in Word.

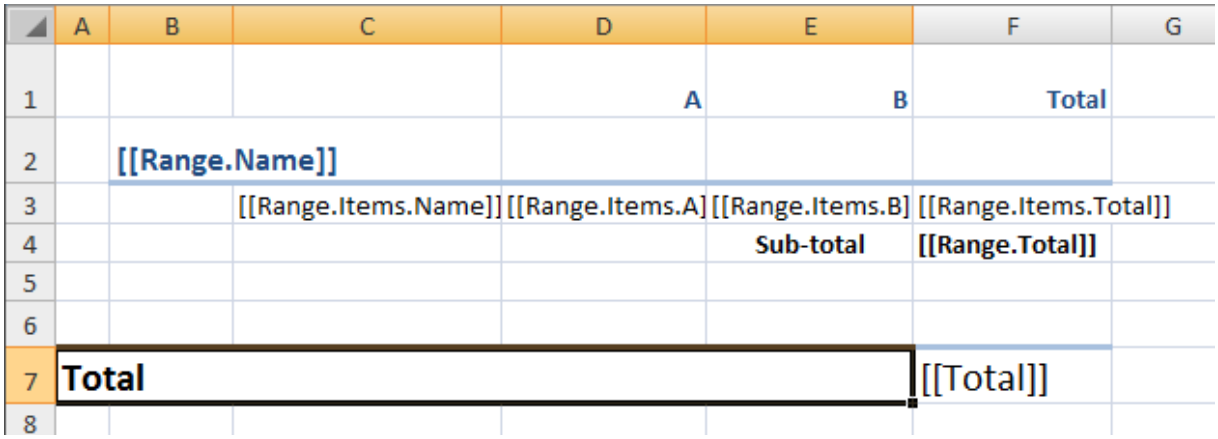
Links have multiple parts, as visible in the example:



Merged cells

Cell merging has various applications, usually to fine tune the cell display, as cells can have text wrapped. With text wrapping using multiple columns defines how wide a text can look. Templater will duplicate, stretch and move merge cells around when document is changed. But it will also influence regions of document which are pushed down.

Alternative way to void region splitting is to use merge cell instead of named range or a table. Previous example with merge cells:



when resized using same data will look as expected:

	A	B	C	D	E	F
1				A	B	Total
2		**Range 1**				
3			Item 1-1	12	20	32
4			Item 1-2	51	11	62
5					**Sub-total**	**94**
6		**Range 2**				
7			Item 2-1	5	21	26
8			Item 2-2	27	75	102
9			Item 2-3	44	42	86
10					**Sub-total**	**214**
11						
12						
13	**Total**					308
14						

This works because Templater will avoid breaking up merge cells and thus extending the region which is affected by the pushdown.

Merged cells also after the behavior of formulas (to some extent):

- formulas can stretch due to merge cell stretching
- formulas intersecting merge cells will behave differently that the ones which do not

Similar to tables and named ranges, during resize merge cells can be:

- duplicated
- stretched
- removed

Merge cells requires at least two columns to create a horizontal merge. If merge stretching is required, the easiest way to implement such a feature is to add additional column and setup document accordingly, such as in this example:

	A	B	C
3			
4	**ASSETS**		
5	**[[groups.name]]**		
6	*[[groups.description]]*		[[groups.items.name]]
7	**[[total.name]]**		
8	**TOTAL ACCOUNT**		

which get's converted into a visible merge cell once rows are duplicate:

	A	B	C
3			
4	**ASSETS**		
5	**Group 1**		
6	*Description 1*		group 1 index 1
7	**Group 2**		
8	*Description 2*		group 2 index 1
9			group 2 index 2
10	**Group 3**		
11	*Description 3*		group 3 index 1
12			group 3 index 2
13			group 3 index 3
14	**total 0**		
15	**total 1**		
16	**TOTAL ACCOUNT**		

Cell styles

Excel has extensive support for cell styling:

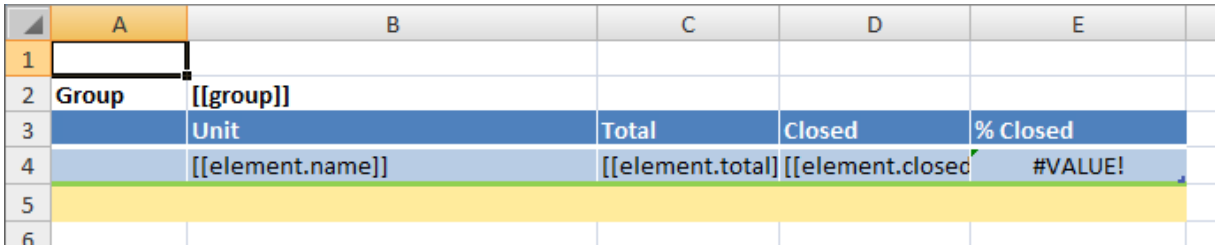
- alignment
- text direction
- font/text properties
- colors
- text wrapping
- formatting
- borders
- and few others...

Templater will maintain cell styles during duplication, pushdowns and similar changes. This way style can be easily set-up in Excel, while Templater will maintain those styles across changes.

Conditional formatting

While cell styles allow for static style setup, sometimes cell style depends on the actual cell value. In those cases it's useful to apply conditional formatting on the relevant cells so they can have custom rules based on their values.

A template example with conditional formatting:

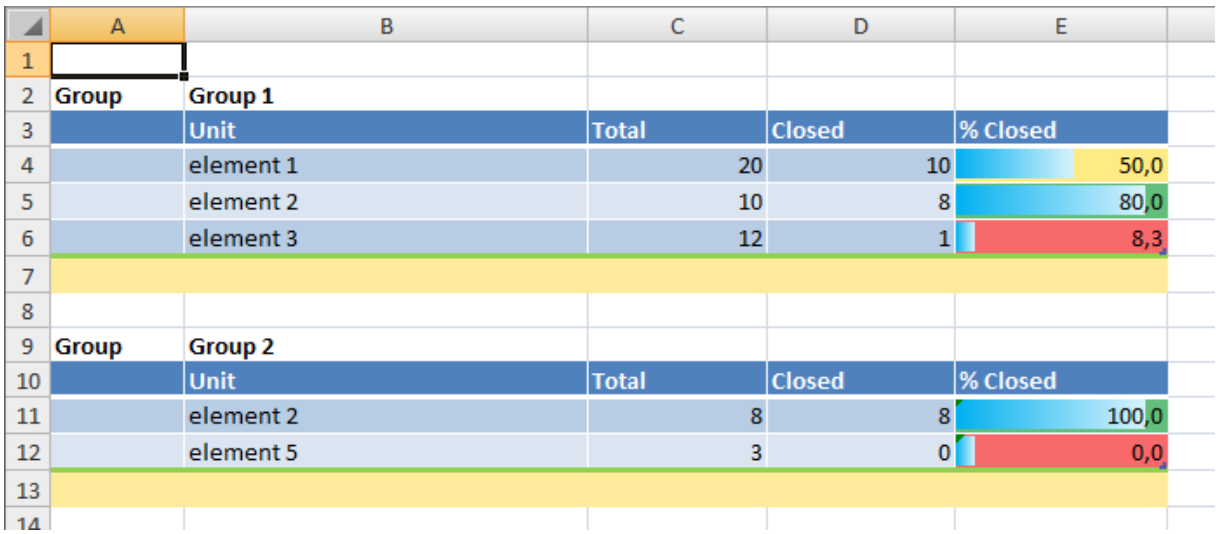


	A	B	C	D	E
1					
2	Group	[[group]]			
3		Unit	Total	Closed	% Closed
4		[[element.name]]	[[element.total]	[[element.closed	#VALUE!
5					
6					

when paired when relevant data:

```
[
 { "group": "Group 1", "description": "first description", "element": [
   {"name":"element 1","total":20, "closed": 10},
   {"name":"element 2","total":10, "closed": 8},
   {"name":"element 3","total":12, "closed": 1}
 ]},
 { "group": "Group 2", "description": "second description", "element": [
   {"name":"element 2","total":8, "closed": 8},
   {"name":"element 5","total":3, "closed": 0}
 ]}
]
```

will result in two tables with conditional formatting at column E:

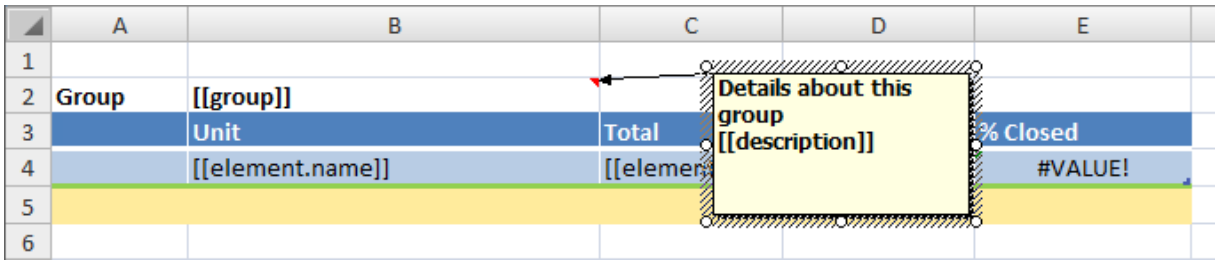


	A	B	C	D	E
1					
2	Group	Group 1			
3		Unit	Total	Closed	% Closed
4		element 1	20	10	50,0
5		element 2	10	8	80,0
6		element 3	12	1	8,3
7					
8					
9	Group	Group 2			
10		Unit	Total	Closed	% Closed
11		element 2	8	8	100,0
12		element 5	3	0	0,0
13					
14					

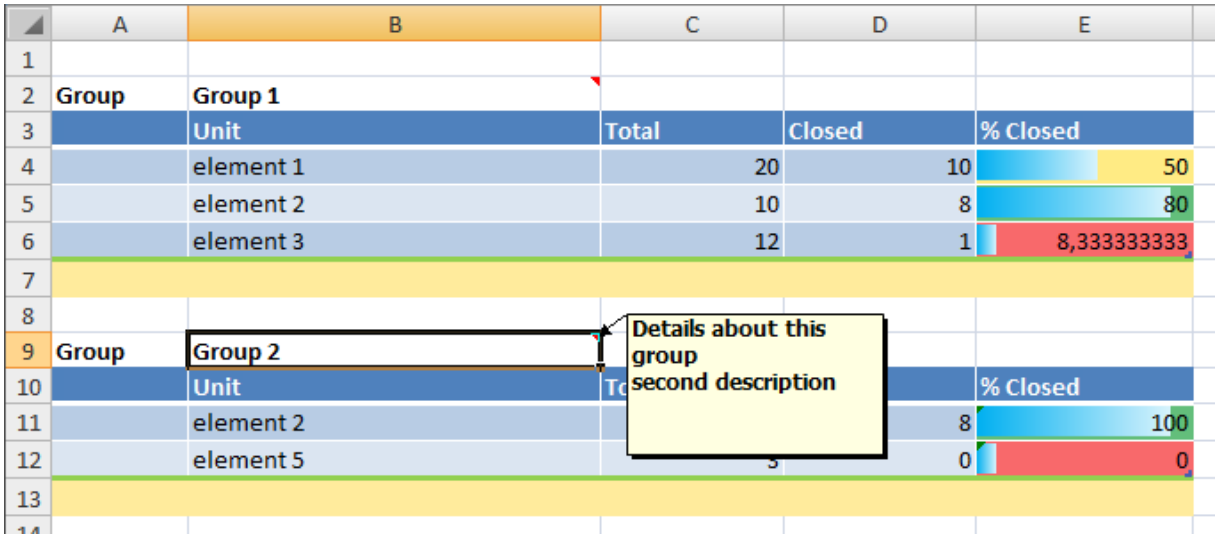
Comments

Tags can be used even in comments, in which case they will be bound to the cell comment is referencing. Even when there are no tags in comments, when comments reference a cell which gets duplicated, a comment will also be duplicated.

A template with comments could look like:



when paired with previous example produces:



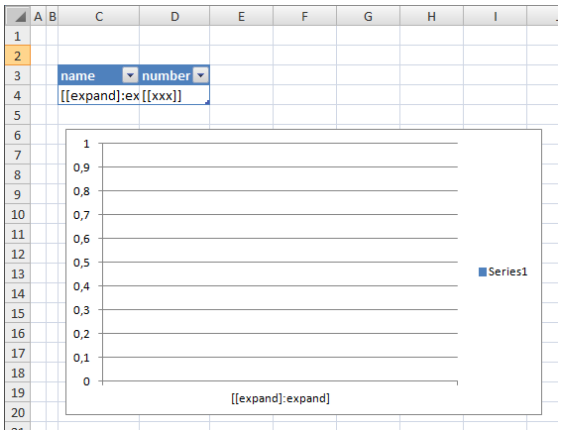
If **Resize**(tags, 0) is invoked on the range which has comments, those specific comments will be removed.

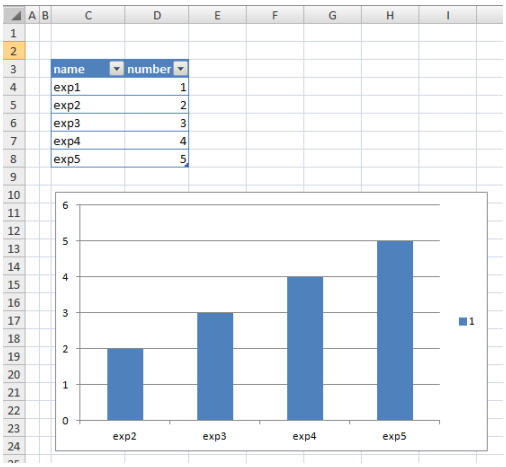
Drawings

Templater will manage drawings on changes to the documents. Drawings come in various types:

- images
- WordArt
- TextBox
- Charts

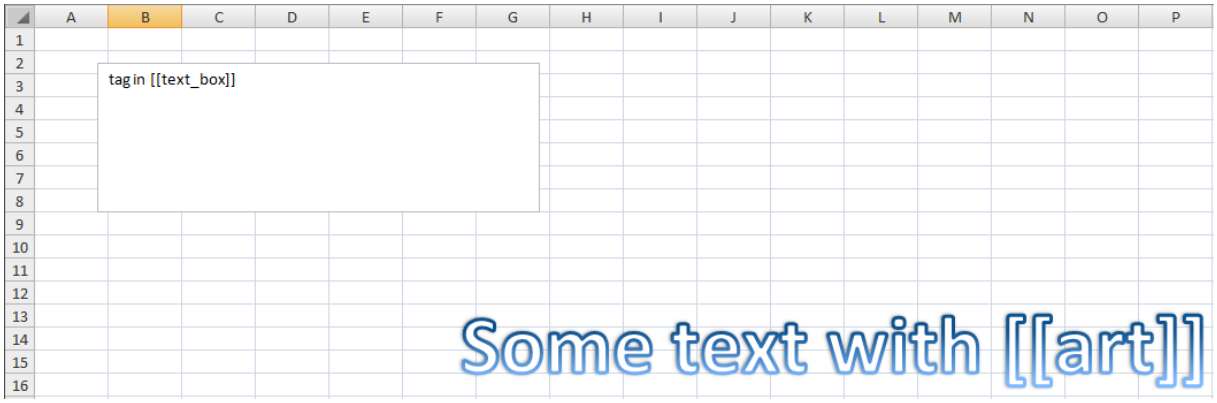
and several others. On resize, push-down/push-right they will be moved accordingly.





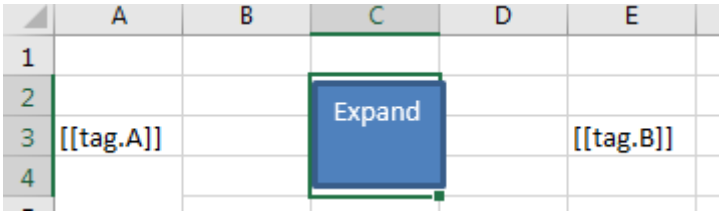
TextBox and WordArt

Tags can be used even in TextBox and WordArt, in which case they will be bound to the cell drawing is referencing. As with comments text can be a combination of tags and normal text. A template with could look like:



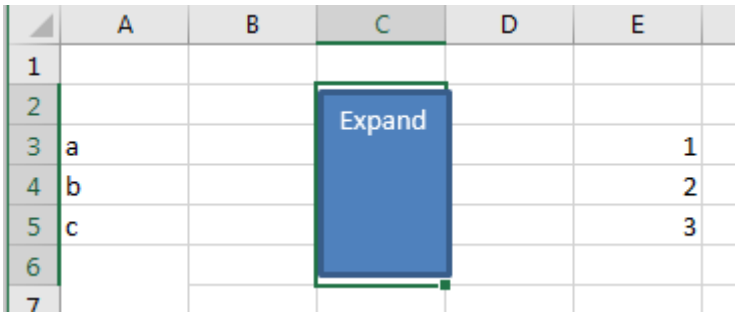
TextBox and similar drawings have special behavior when it comes to stretching. If a context is enlarged due to resize, by default such special element will not be stretched, but rather will remain in place. To get stretching behavior we need to put it in a merge cell region so that when merge cell gets stretched, TextBox will be stretched too.

For example, template such as, when paired with relevant data

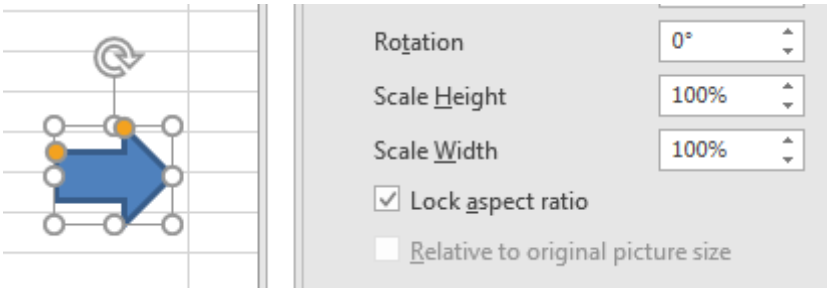



```
{"tag":[
  {"A":"a","B":1},
  {"A":"b","B":2},
  {"A":"c","B":3}
]}
```

will become:

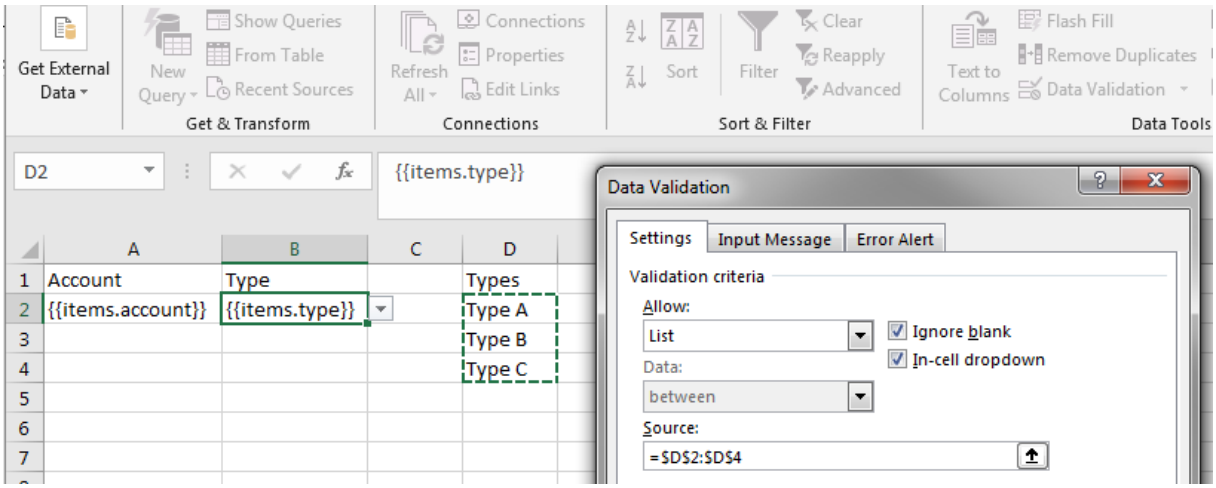


If there is a need for drawing not to be stretched, this can be specified by checking the Lock aspect ratio checkbox

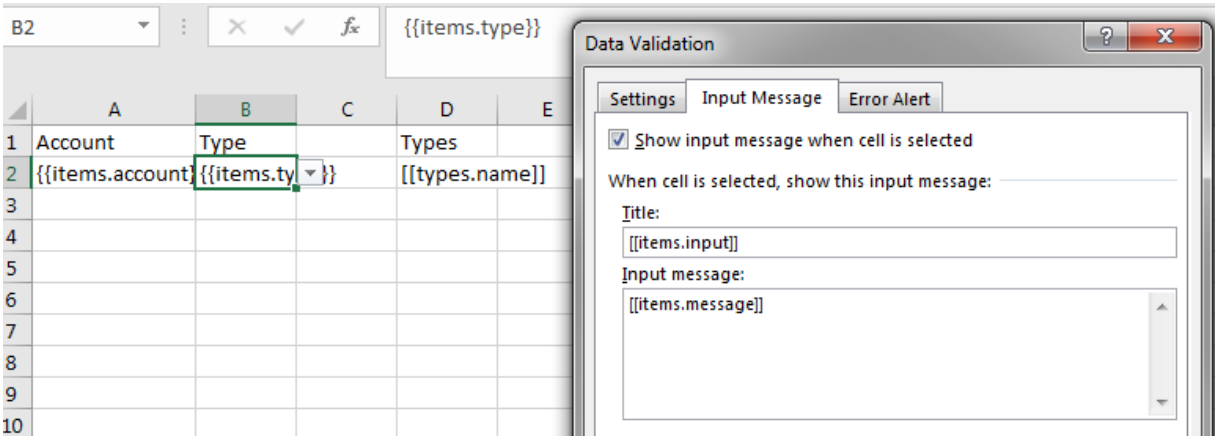


Data validations

With v8.1 Templater supports Excel data validation feature. It will adjust ranges, replace tags and perform usual activities (copy, adjustment). An example of such a feature would be defining allowed values for a dropdown



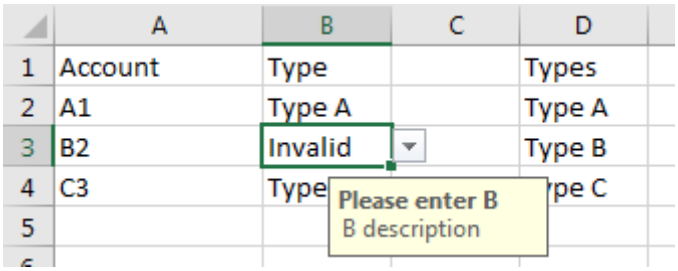
Tags are also supported on popups and standard resize features can be used for source, which allows examples such as:



when populated with:

```
{"items":[
 {"account":"A1","type":"Type A","input":"Please enter A","message":"Value A description"},
 {"account":"B2","type":"Invalid","input":"Please enter B","message":"B description"},
 {"account":"C3","type":"Type C","input":"Enter C","message":"Value C"}
], "types":[
 {"name": "Type A"},
 {"name": "Type B"},
 {"name": "Type C"}
]}
```

will result in output such as:



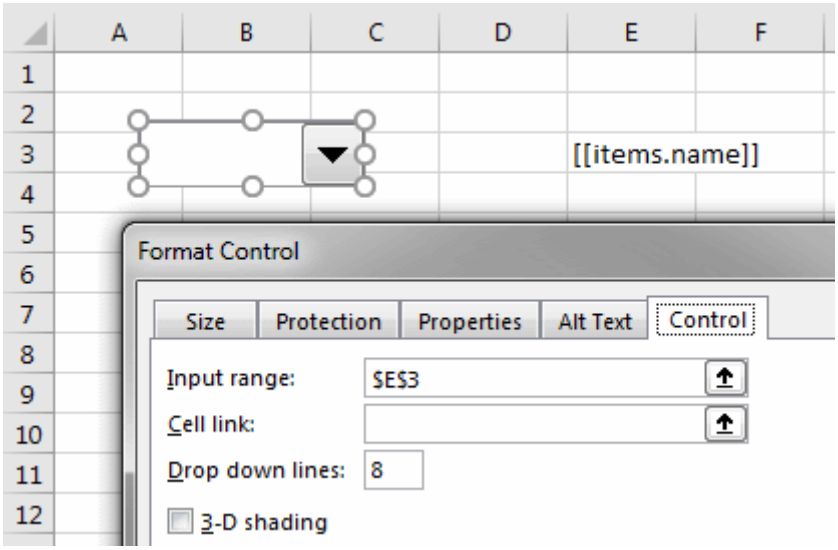
Form Controls

Excel has support for various controls. They usually refer to cells or cell ranges. Examples of such controls:

- Checkbox
- Combo Box
- List Box
- …

When Control references a range and this range is resized, Templater will perform required adjustment so that Control is updated and points to the new range.

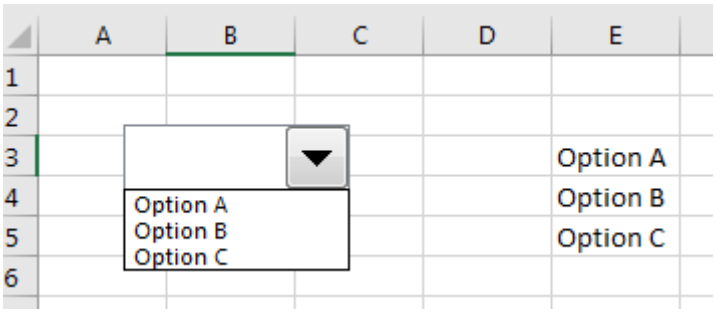
In practice this looks like:



when paired when relevant data:

```
{ "items": [
   {"name":"Option A"},
   {"name":"Option B"},
   {"name":"Option C"}
]}
```

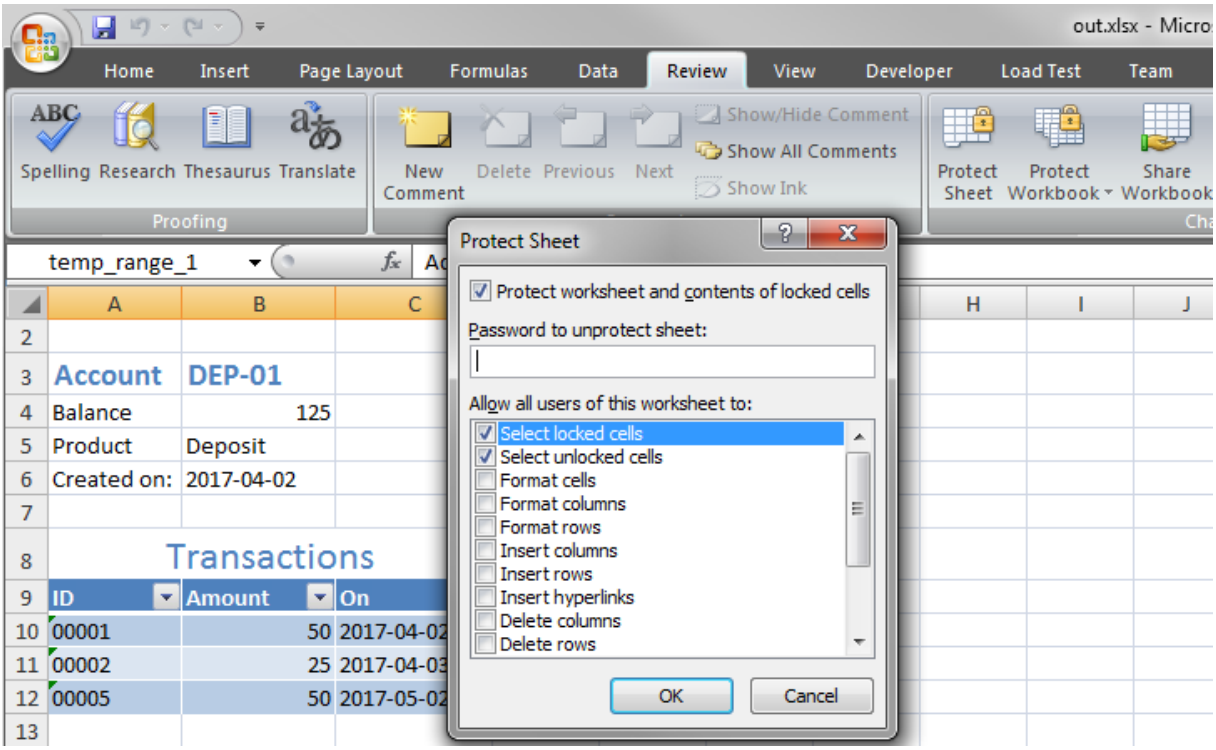
will adjust control to correct input range



Sheet locking

Similar to locking in Word, a sheet in Excel can be locked. This is a UI feature which doesn't really prevent data manipulation, which means that Templater can still modify the document as usual, but to the user document appears locked (or some parts of it).

It's available through Review -> Protect Sheet menu:



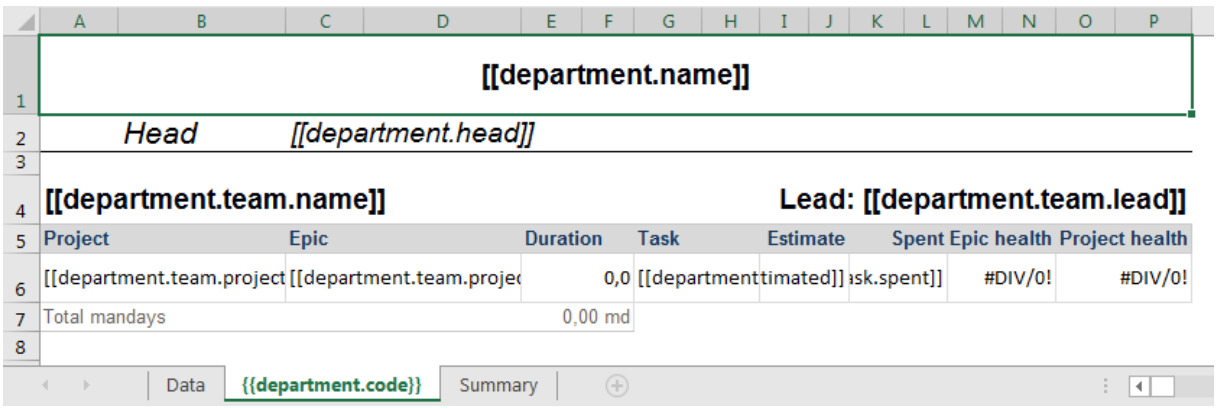
Sheet duplication/removal

Templater can duplicate entire sheet ranges in certain conditions:

- tag is used in sheet name
- tag has relevant metadata: ***page*** or ***sheet***

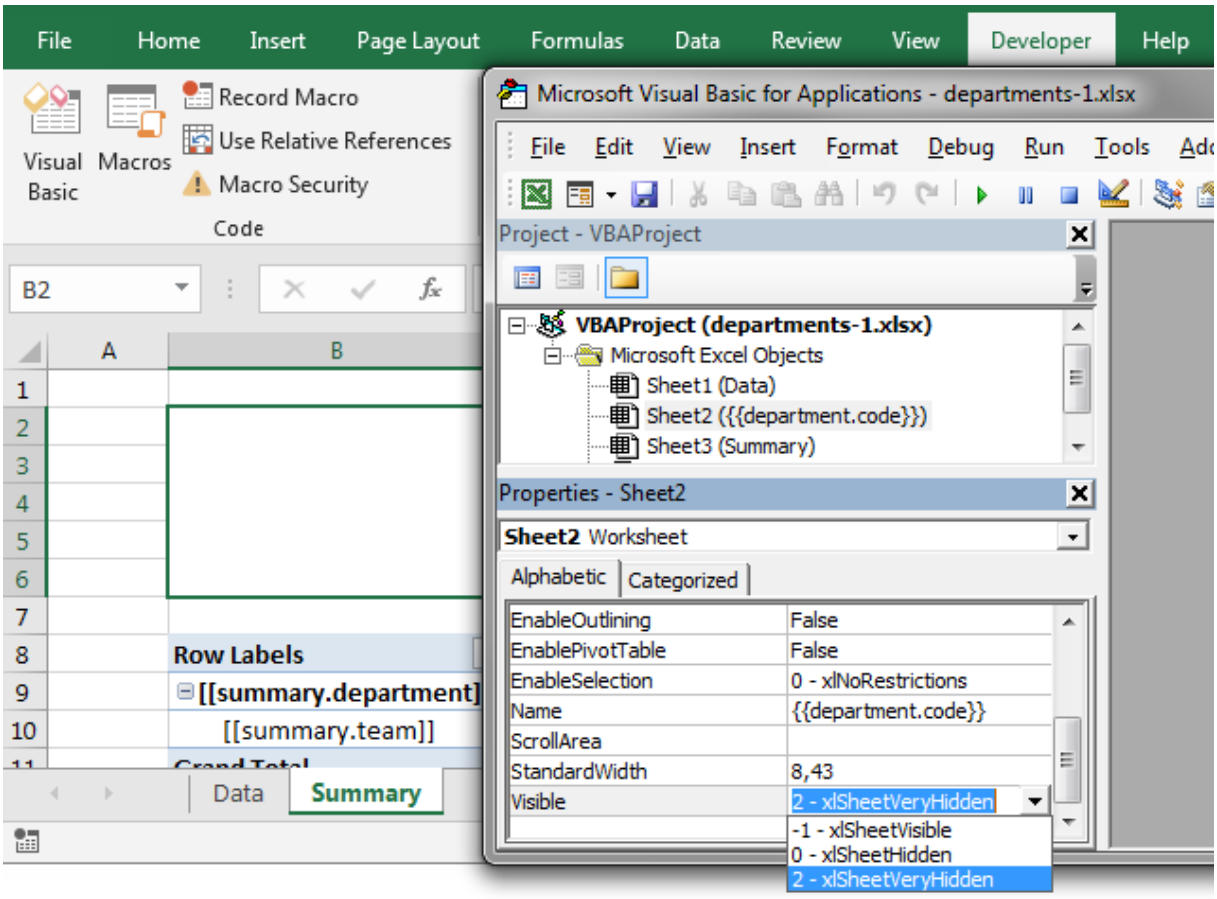
All objects within the sheet will be duplicated so it's possible to define tables, pivots, charts and other advanced Excel objects and have them duplicated with relevant sheets.

Excel has certain restrictions on sheet names, so metadata can't be used and result can't be longer than 30 characters.



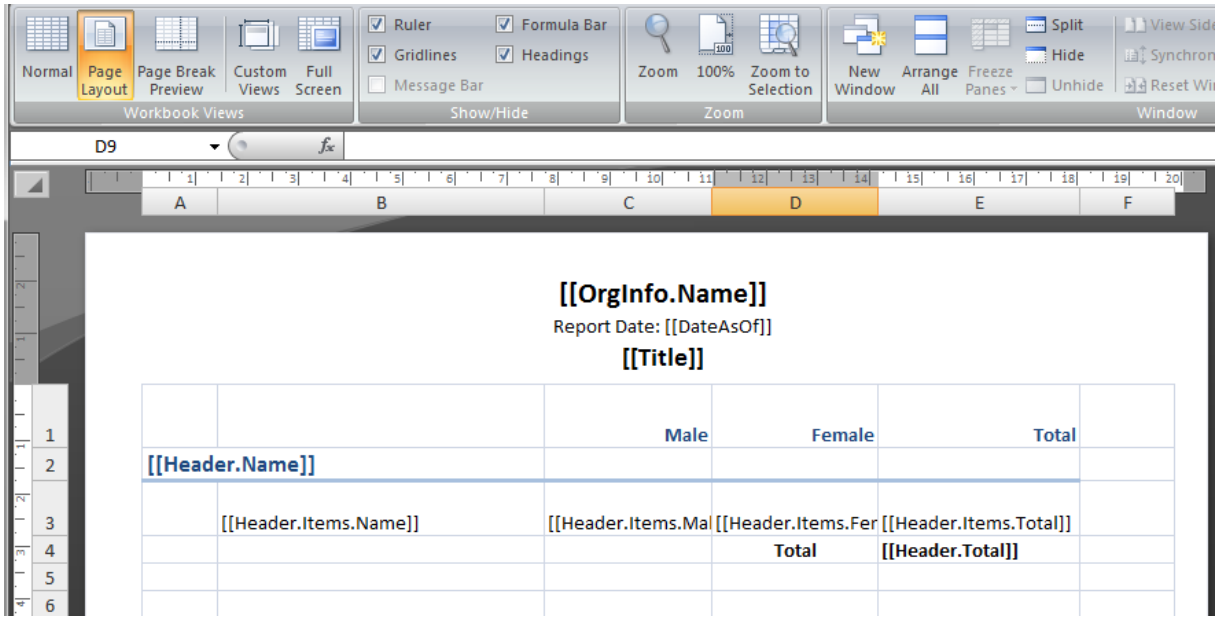
When **resize** 0 is used to remove the sheet, Templater will use **very hidden** attribute on the sheet to leave it in xlsx, but hide it from list of sheets and disallow listing of those sheets in Unhide popup.

Sheet can be reenabled via VBA properties:



Headers and footers

Tags can be used in headers and footers, although only a subset of Excel features is available at those places. Headers can be defined in Page layout mode, e.g.:



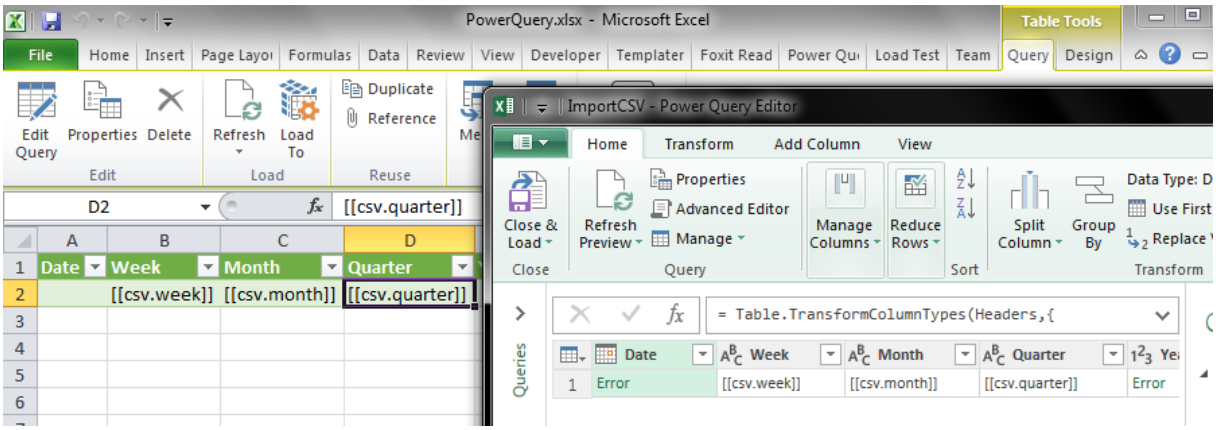
Power Query / Get & Transform

There is basic support for Power Query, as in when underlying data source uses tags, they need a special treatment. This allows usage of complex Power Query use cases. More complicated use case such as use of embedded CSV file within XLSX file which gets processed by Templater and then consumed by Power Query allows for sending complex reports over the network as a single file.

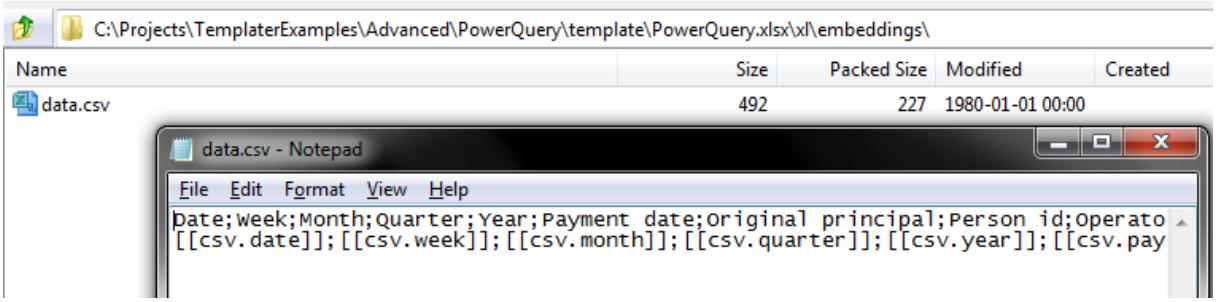
Templater will analyze and process tags in embedded CSV files[38] similarly as it processes embedded xlsx files within docx (for charts). This allows for displaying huge amount of data, as CSV is much smaller and faster to process. Embedding CSV file within xlsx is somewhat custom process, consisting from:

- setting up CSV dependencies so that Excel does not remove embedded CSV file on next save
- putting CSV file within xlsx zip so that it can be recognized/processed during processing
- setting up somewhat complicated process of unzipping xlsx and passing embedded CSV into PowerQuery transformations
    - there are various restrictions with common unzipping, as streaming zip (natively produced by Java version) is much harder to support
    - since CSV is embedded within the same file, some tricks must be used, such as having formulas pointing to the file, building table from those parameters and using them in expressions to locate the expected file

Excel setup can look like:

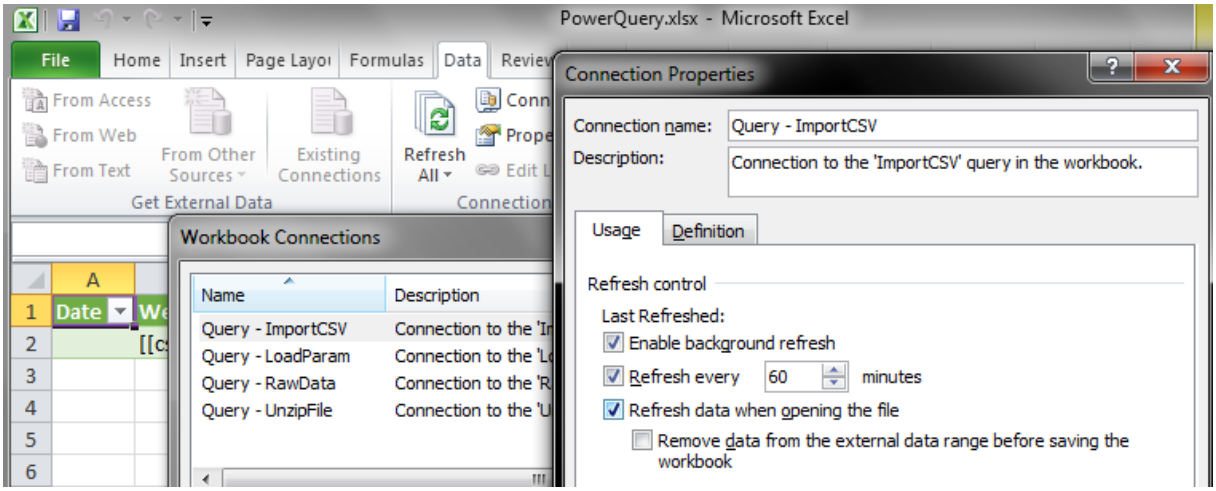


with CSV file looking like:



[38] There are some minor differences to regular xlsx processing within docx

It is recommended to turn on Refresh on load feature accessible through the connection properties:



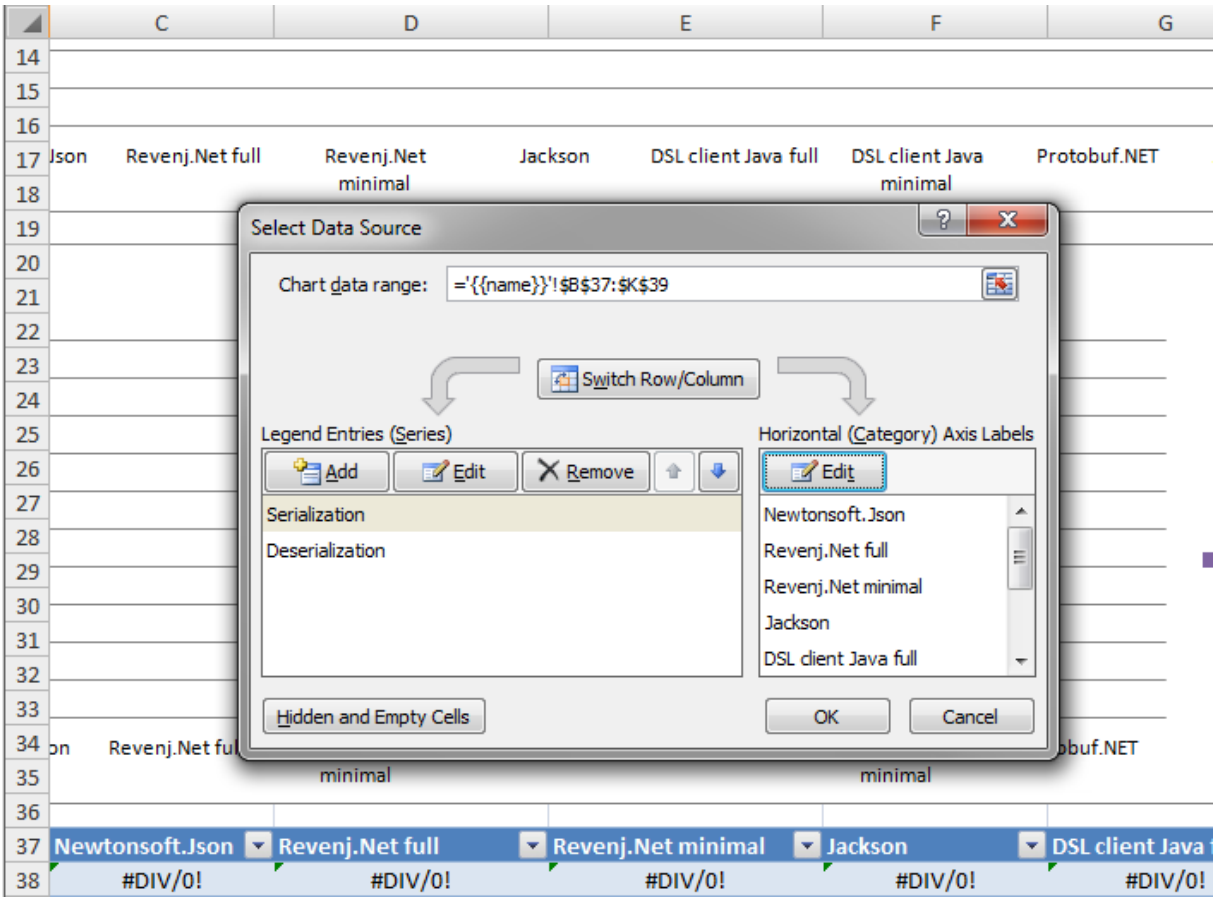
While such usage is not as easy to manage as regular Templater tags, due to requirement for column transformation, when appropriate, the cost of such manual transformation should be minimal compared to the value extracted by the report.

Pivots and charts

Advanced Excel features usually consume data source(s) (specific range, table or a whole sheet) to present data in various advanced analytics friendly way. As with many other features, Templater can duplicate such pivots and charts which allows for very complex reports.

Most charts (column, line, pie, bar, area...) work the same way and are supported by Templater.

There are usually multiple data source definitions for each specific feature, e.g.:

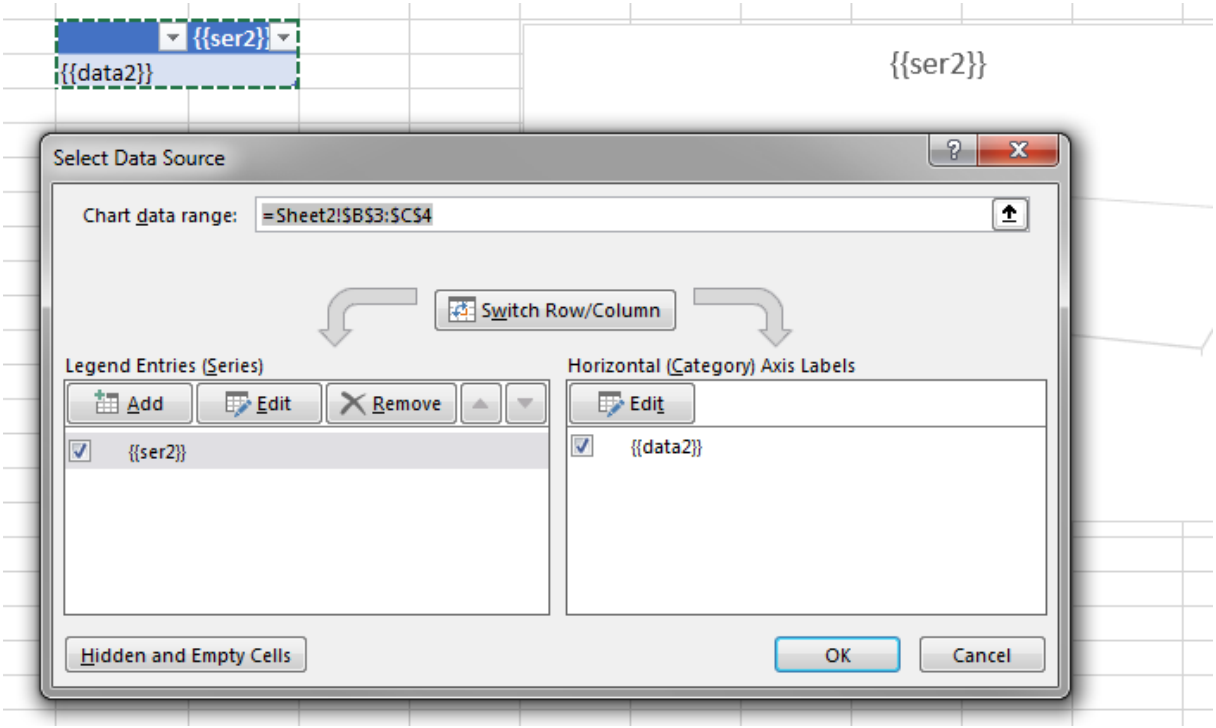


When charts and pivots are duplicated, data ranges will be adjusted accordingly; even if the sheet name changes (as in the above example).

Charts with dynamic number of series

Special behavior is required for charts where number of series is not known upfront. Templater supports both *horizontal-resize* and Dynamic resize to implement such behavior.

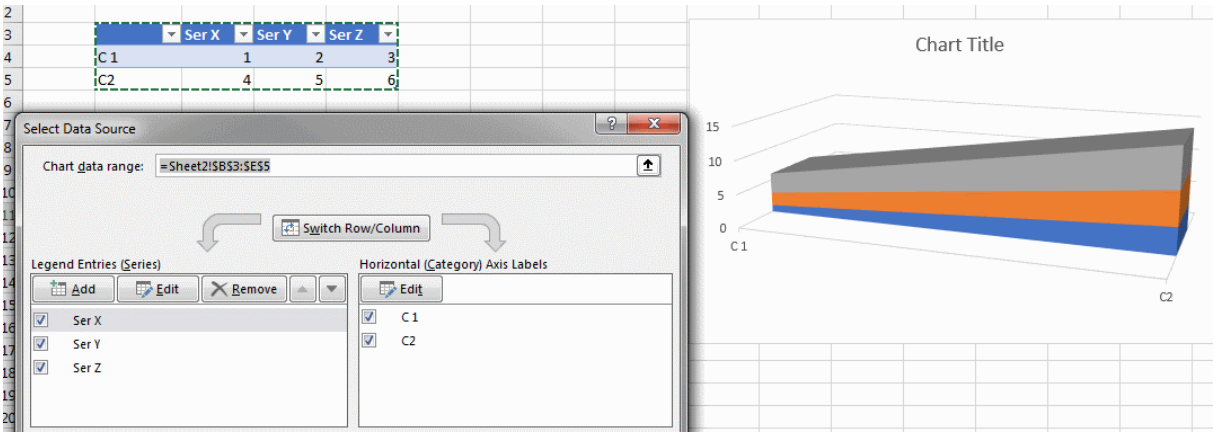
Example template:



paired with relevant input:

```
{
 "ser2": [ [ "Ser X", "Ser Y", "Ser Z"] ],
 "data2": [ [ "C 1", 1, 2, 3 ], [ "C2", 4, 5, 6 ] ]
}
```

will result in expected output:



Print Area

Templater supports hidden named ranges and will adjust them on document manipulation. This means print area should be correct after processing by Templater.

Formula conversion

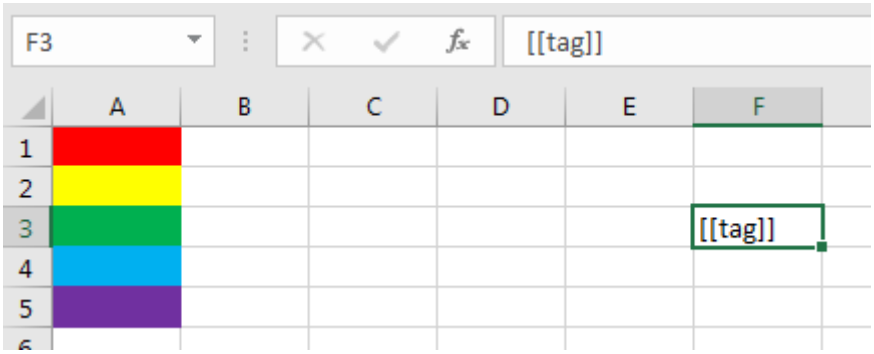
Tags can't be used within formulas. There is a special way to convert cell values into formulas at the end of processing. If a cell starts with [[equals]] tag it will be converted into formula representation.

Font color and rich text

Similar to Word XML can be used to inject values into xlsx directly. But unlike in Word, only subset of features can be specified this way. Prior to v6 background could only be set via Conditional formatting. With v7 internal Excel attributes can be provided via XML value:

- cell and row styles can be copied from another cell/row by using appropriate attributes
  - templater-cell-style=CELL
  - templater-row-style=ROW
- other row attributes can be specified
  - templater-row-height
  - templater-row-custom-height
  - templater-row-collapsed
  - templater-row-custom-format
  - templater-row-hidden
  - templater-row-thick-bottom
  - templater-row-thick-top

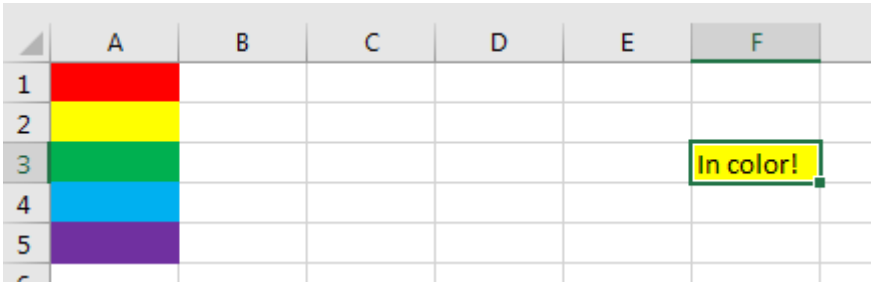
In practice this means if there is a sheet with several cells with predefined styles, e.g.



By passing in relevant XML for tag:

```
var xml = XElement.Parse("<t templater-cell-style=\"A2\"
xmlns=\"http://schemas.openxmlformats.org/spreadsheetml/2006/main\">In
color!</t>");
templater.Replace(“tag”, xml);
```

After processing Sheet will look like



Simple rich text can be entered as XML as long as the underlying Excel OOXML format is understood.

Cell merging via metadata

Internal metadata **merge-nulls** works in Excel also[39].

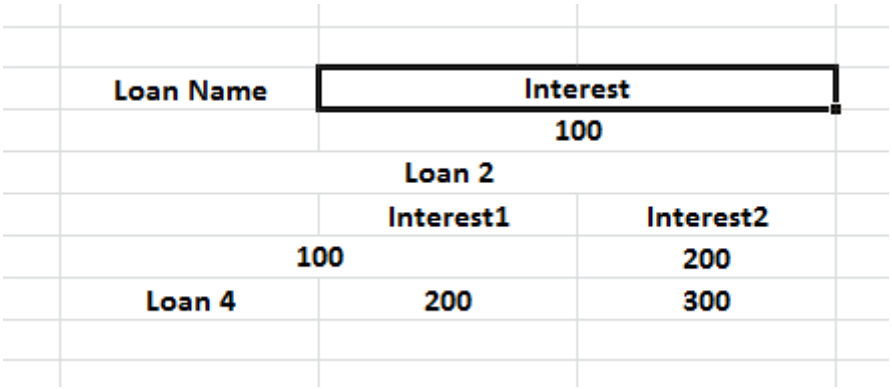
When simple template such as



Is populated with two-dimensional array

```
var table = new object[,]
{
        {"Loan Name", "Interest", null},
        {null, 100, null},
        {"Loan 2", null, null},
        {"", "Interest1", "Interest2"},
        {100, null, 200},
        {"Loan 4", 200, 300}
};
```

It will result in merged cells



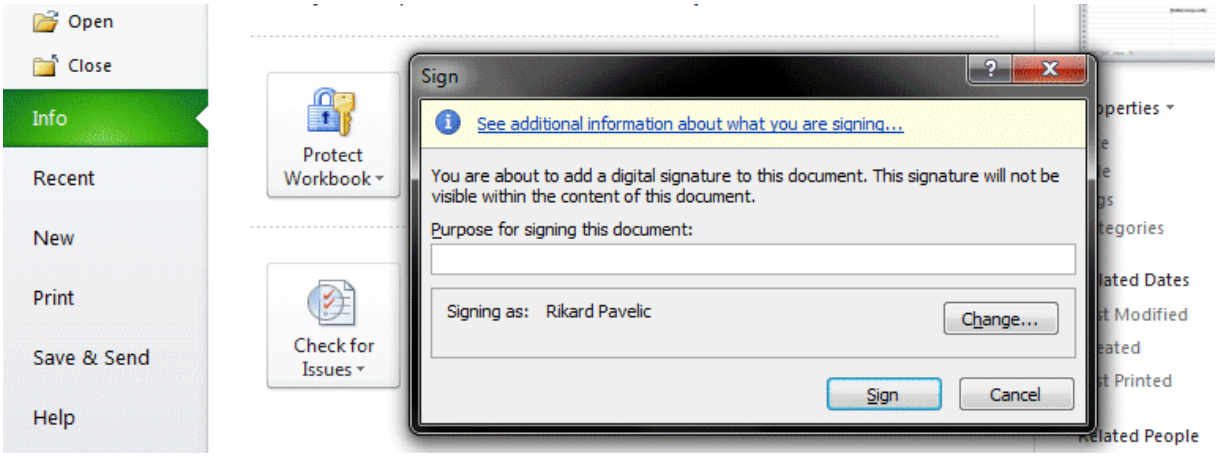
Digital signature

For the purpose of providing authenticity of the data in the document, Templater supports document signing which marks spreadsheet as final and prevents changes to signed parts of the spreadsheet.

Main downside of signing Excel files is that Templater leaves formula recalculation to Excel, which invalidates document. For this reason, signing document will also change formula calculation mode to manual which will result in cell values for formulas empty.

Signing can be done manually via Excel interface:

[39] span-nulls is not currently supported in Excel



Known issues

If a specific Excel feature is not supported, there are few basic categories it falls into:

- feature requires Excel rendering engine and thus it's not supported
  - such features include PDF export, etc...
- feature is on the roadmap, but it's not supported yet
- feature is not behaving as expected due to a bug

PDF export

A common use case is to convert Excel document into PDF. Unfortunately, this requires an Excel rendering engine to work correctly.

There are several free and paid libraries which have sufficiently good PDF conversion for simple documents. But non-trivial documents quickly become non pixel-perfect during the conversion.

Whenever user can convert Excel document into PDF this should be preferable. Even running LibreOffice in headless mode supports only limited feature set of Excel. For simple documents there is a Dockerfile paired with Templater server which can be used to ease the PDF conversion via LibreOffice.

Power Query cloning

Templater currently does not support cloning of sheets with Power Query. This restriction might be lifted in some future version.

Formula values

Templater will remove cached values from formula cells. Also, it will not evaluate formulas at the end of processing. This means that when a document is opened it must be displayed via an application which knows how to evaluate the formulas on load (such as Excel). When spreadsheets are signed, formula recalculation is disabled to prevent signature invalidation, which causes formula values not to be recalculated.

PowerPoint features

Templater has high coverage of various PowerPoint features, although many of them use embedded Excel files within the pptx file. Various features are supported out-of-the box without any special code, while some require special handling and will be introduced over time.

Ready-to-use presentations

Prior to v4.0 Templater did not support PowerPoint format due to lack of useful use cases. But due to deeper integration of Templater into various applications it became obvious that creating presentations from various in-depth analyses has become a common use case. Therefore, most of Templater features work also on PowerPoint format, although common use cases consist from only a few features:

- resizing/populating table to show raw numbers
- populating charts to display numbers in visually informative ways

While both of those can be done in Word or Excel, by automating export into PowerPoint manual step of copy-pasting numbers is avoided.

Resizable behavior

Processing PowerPoint has its own special rules for detecting how resize behaves. Unlike Word which has a single main document and Excel where the basic element is a cell even if there can be many sheets, in PowerPoint the basic element is a slide. To understand resizing behavior of Templater few rules have to be understood. When Resize(tags, count) is called Templater will

- find the best matching region on the slide or across the slides which encapsulates all specified tags
    - regions will be limited to the rows in a table (matched for starting and ending row)
        - table region can span multiple rows
    - relevant list levels will be matched
        - list levels can match the hierarchical structure of the model
    - when tags are neither in table, embedded Excel (chart) or a list, whole slide will be used
- if all tags are inside tables/lists/chart, instead of duplicating the slides, tables, lists and/or charts will be resized instead
    - this means when a same collection is repeated both in a table and in a chart, that those tables and charts will get new rows instead of slides being duplicated
- when count = 0 indicating removal of the content part of the presentation, slides will be removed
    - all slides can be removed from the presentation
    - this is useful for conditionally presenting only a relevant part of the presentation

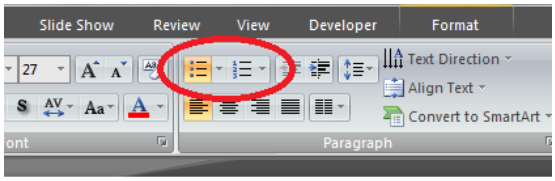
Lists

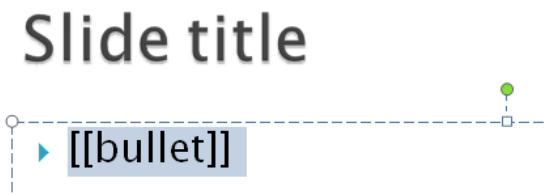
Not every element in a slide is considered resizable (like in Excel where each cell can be considered in such a way). Unless text is marked as list within the slide of the presentation the whole slide will be used as context instead of only that list element.

Like in the Word lists can be:

- bullets
- numbered
- multi-level

List can be located even in notes, but they must have special list marker attached to them. Marker is visible in the menu when caret is located on the relevant element:



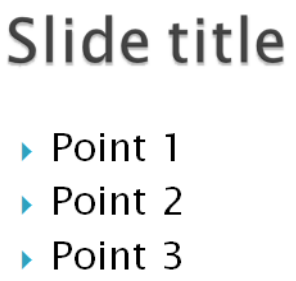


Duplicating list elements will retain all their properties (font style, nesting level, colors, etc...)
Matching above template with appropriate input:

```
[
 {"bullet": "Point 1"},
 {"bullet": "Point 2"},
 {"bullet": "Point 3"}
]
```

will result in multiple list elements:





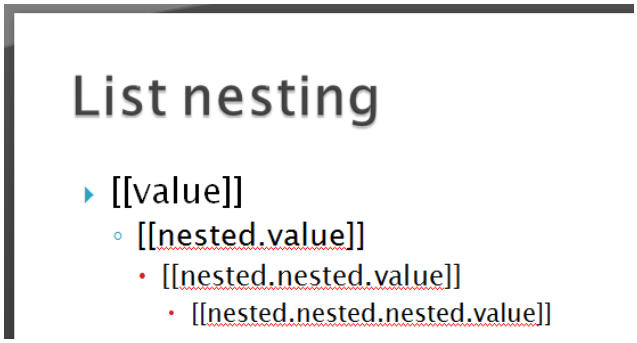
Nesting

Common use case for lists is pairing it with deep nesting or even with recursive structures.

When specialized data structure is used, such as:

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

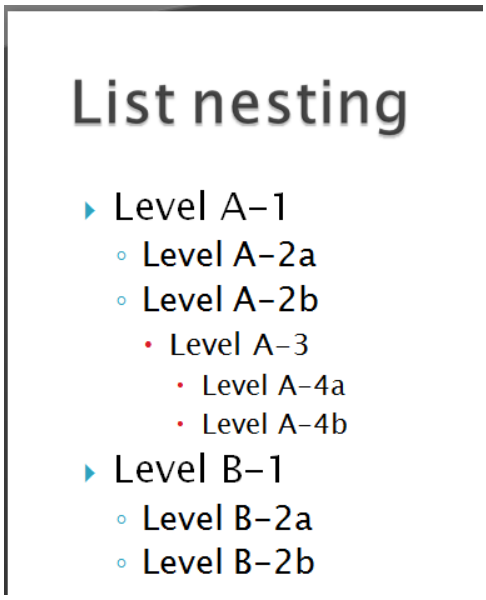
it is rather easy to pair it with nested list by predefining maximum nesting level, e.g.:



Based on the input, resulting list will match the nesting levels and values, e.g. for input as:

```
[
  { "value":"Level A-1", "nested":[
    { "value":"Level A-2a", "nested":[] },
    { "value":"Level A-2b", "nested":[
      { "value":"Level A-3", "nested":[
        { "value":"Level A-4a", "nested":[]},
        { "value":"Level A-4b", "nested":[]}
      ]}
    ]}
  ]},
  { "value":"Level B-1", "nested":[
    { "value":"Level B-2a", "nested":[] },
    { "value":"Level B-2b", "nested":[] }
  ]}
]
```

a matching list will be created:



Since style is defined on the list, while Templater only binds the data with the list, complex list representations can be easily constructed.

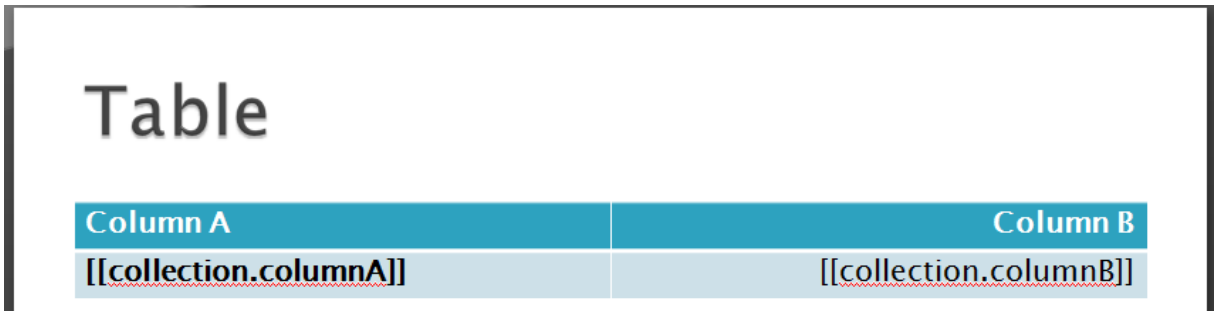
Tables

While tables in PowerPoint are not as feature rich as the ones in Word, they are still quite useful and used often. Table can have various options attached to it, such as:

- styles
- spacing
- alignments
- borders
- cell merging
- text direction

which allows easy setup of complex layout.

Resizing a table is quite intuitive in Templater. When a table like:

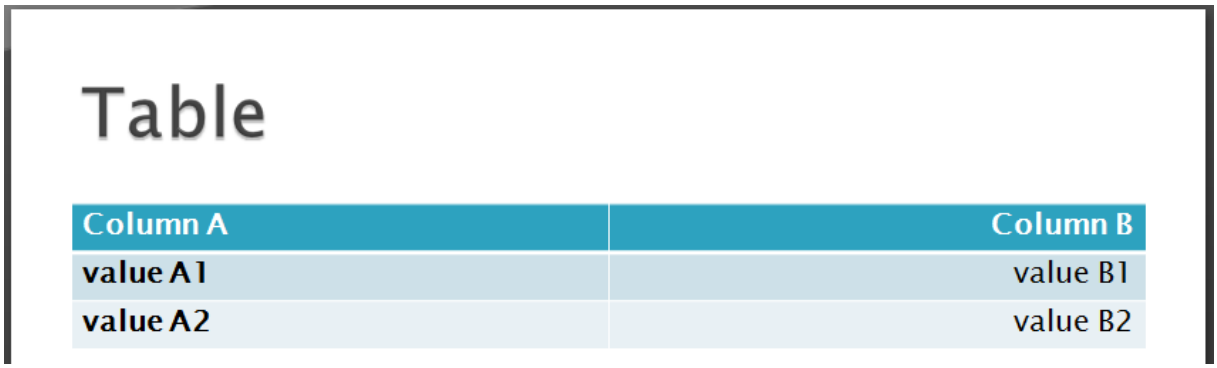


is matched with an appropriate input, e.g.:

```
{
  "collection": [
```

```
  {"columnA": "value A1", "columnB": "value B1"},
  {"columnA": "value A2", "columnB": "value B2"}
 ]
}
```

The result will look as expected:



Table

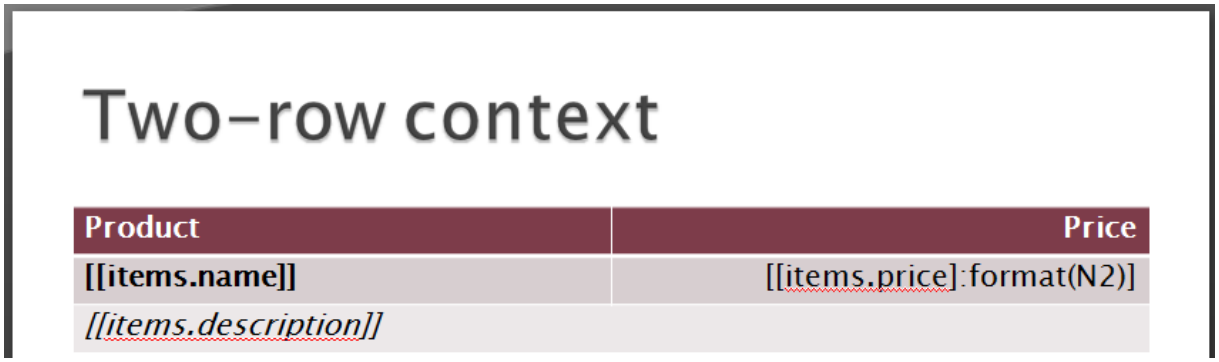
Column A	Column B
value A1	value B1
value A2	value B2

A really important aspect of such transformation is:

- it is implied by the document structure
- there are no loop or start/end constructs in the document
- it matches against the input “intuitively” by using dot (.) for navigation

Multi-row context

Templater supports context use over multiple rows, such as:



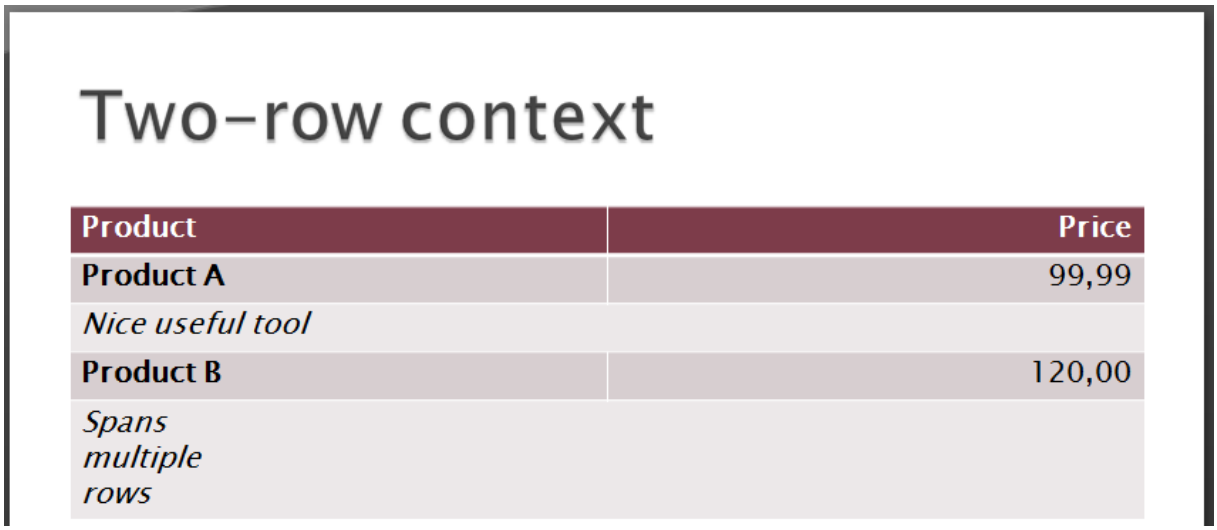
Two-row context

Product	Price
[[items.name]]	[[items.price]:format(N2)]
[[items.description]]	

when matched with an appropriate input, e.g.:

```
{
 "items": [
  {"name": "Product A", "price": 99.99,"description":"Nice useful tool"},
  {"name": "Product B", "price": 120,"description":"Spans\nmultiple\nrows"}
 ]
}
```

Produces an expected table which looks like:



and has several non-trivial features:

1. context is no longer a single row, but two rows, since tags were defined across several rows
2. simple number formatting can be used to tweak the output into expected format
3. bolding, italics and other text features were preserved
4. newlines in text input resulted in newlines in cell values

Dynamic resize

A special feature of Templater is processing specific input types (two dimensional collections and DataReader/ResultSet) in a specialized way.

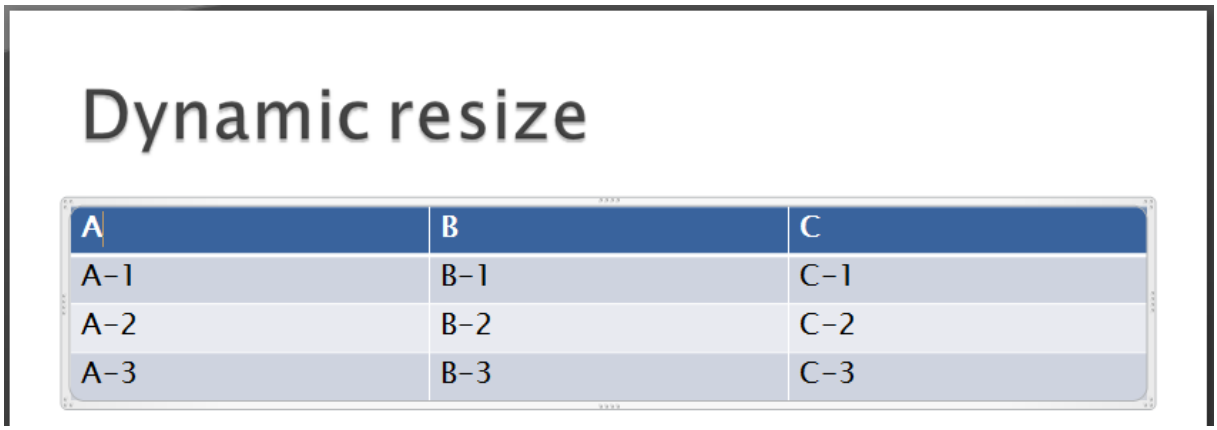
A basic use case for Dynamic resize would be to transform table template into a final output, e.g.:



when matched with an appropriate input, e.g.:

```
{
  "table": [
    ["A", "B", "C"],
    ["A-1", "B-1", "C-1"],
    ["A-2", "B-2", "C-2"],
    ["A-3", "B-3", "C-3"]
  ]
}
```

it will be transformed into a table with 3 equal columns and 4 rows:



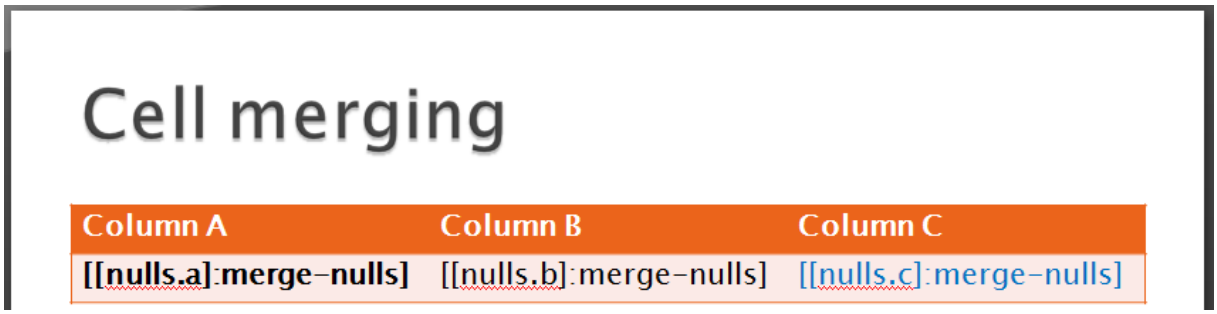
While this is useful for some scenarios, usually explicitly defined table templates are used since they allow for more fine-grained tuning.

Cell merging

While cells can be merged in the template, there are use cases when they need to be merged during table generation/population. For this reason, there are two built-in metadata plugins:

- merge-nulls - invokes horizontal cell merging when cell value is null
- span-nulls - invokes vertical cell merging when cell value is null

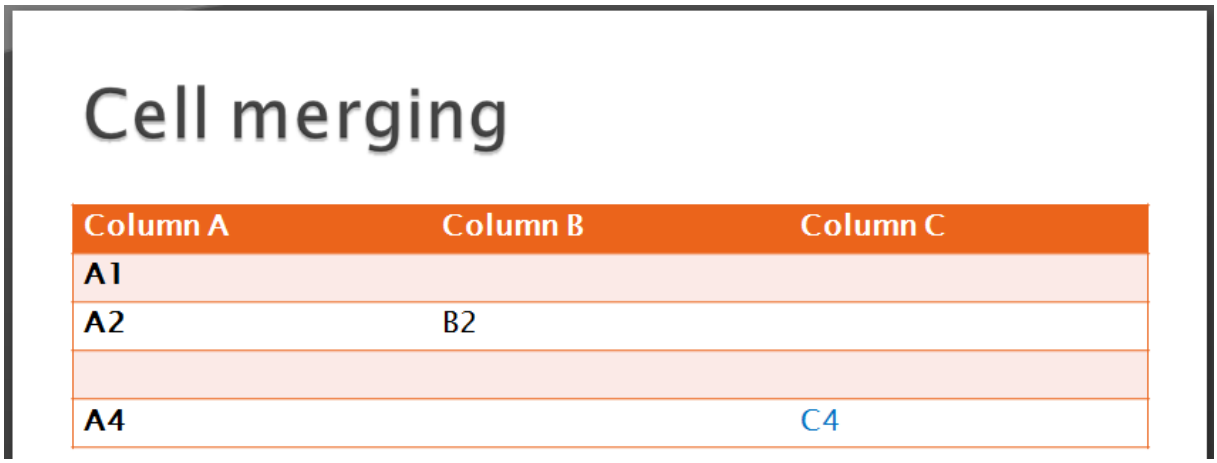
Cell merging works both in Dynamic resize and standard table resize. Table such as:



when paired with input such as:

```
{
 "nulls": [
  {"a":"A1", "b":null, "c":null},
  {"a":"A2", "b":"B2", "c":null},
  {"a":null, "b":null, "c":null},
  {"a":"A4", "b":null, "c":"C4"}
 ]
}
```

will result in table with merged cells:



Cell merging

Column A	Column B	Column C
A1		
A2	B2	
A4		C4

Existing merge cells

If there are existing merge cells in the table, Templater has specialized behavior for dealing with them:

- if tag range does not touch start of a merge cell or goes beyond end of merge cells, merge cell will be stretched
- otherwise context will be expanded to include merge cell(s)
- if tag is contained within the merge cell, context will include merge cells in minimum spanning range

Stretching merge cells in a table looking like:

Merge cell	Col A	Col B
	[[txt]]	[[num]]

When matched with an appropriate input, e.g.:

```
[
 { "num": 1, "txt":"A" },
 { "num": 2, "txt": "B" }
]
```

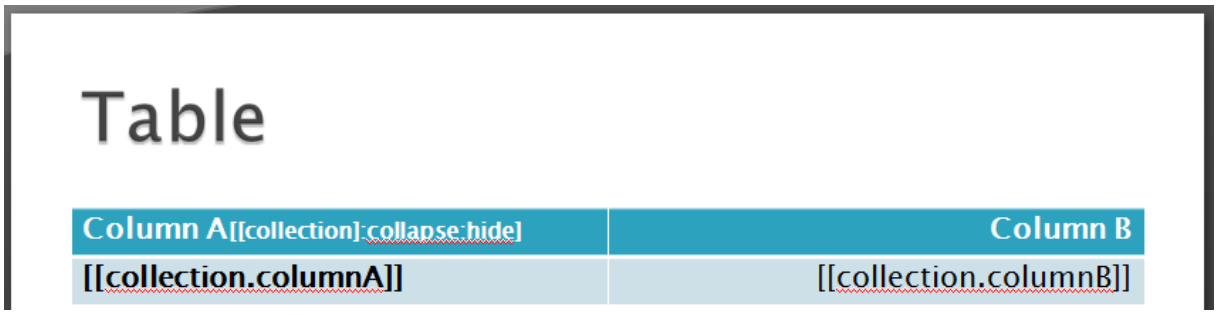
Will result in table with merge cell stretched:

Merge cell	Col A	Col B
	A	1
	B	2

To change from stretching to duplication in this use case, additional tag can be added to first row, e.g. [[num]:hide] which will be removed after processing, but will influence the behavior.

Removing a table

When Resize(tags, 0) is called on a table, relevant rows will be removed. Sometimes this means that entire table will be removed, but often for the table with headers which don't have any tags the header remains at the end of the resizing. In the case when there is a separate header without tags a common workaround is to add special tag on the header with collapse and hide metadata:



This way when collection is empty a separate resize 0 will be called just for the header row. When collection is not empty hide metadata will take care of not showing any text in place of the tag.

To remove the whole slide, instead of just table, this helper collapse tag can be added to slide title in which case Templater will conclude that resize("collection", 0) should result in removal of entire slide along with all tags within the slide.

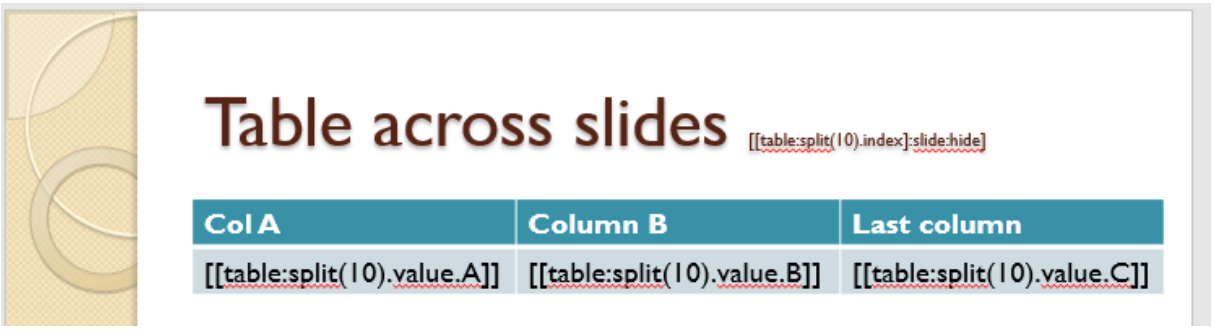
Splitting a table across slides

If a table has many rows its common that the content of the table will not fit in a single slide. To approach such a problem, we need to utilize 2 different operations:

- slide duplication
- splitting data across slides

The easiest way to approach that problem is to create a plugin for converting input list data structure into nested sublist data structure where each sublist will correspond to one slide. Such a problem can be resolved via a plugin like in the example.

The slide duplication can be implemented via special :slide metadata accompanied with :hide metadata so that we don't need to show any value, or by placing tag in a location where duplication will mean duplication of slides, not rows/elements (such as a slide title).

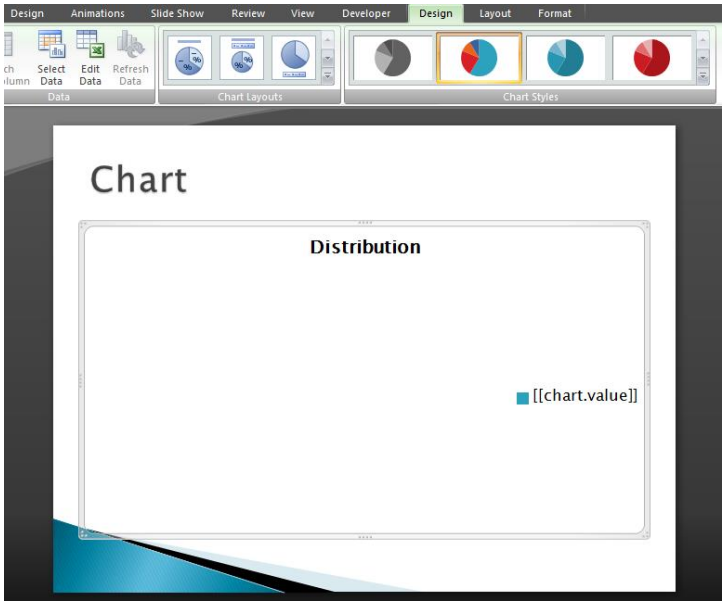


Charts

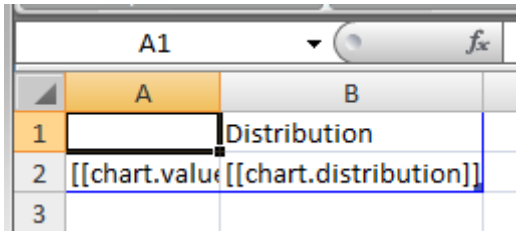
Charts are represented by embedding Excel xlsx inside PowerPoint zip pptx. Depending on the chart type there is also some aggregation of values within the slide XML.

Charts are also considered resizable elements, as the underlying data source is a resizable Excel range.

Chart template is defined within Excel by adjusting original template and replacing values with tags, which results in a bit unfriendly chart template:



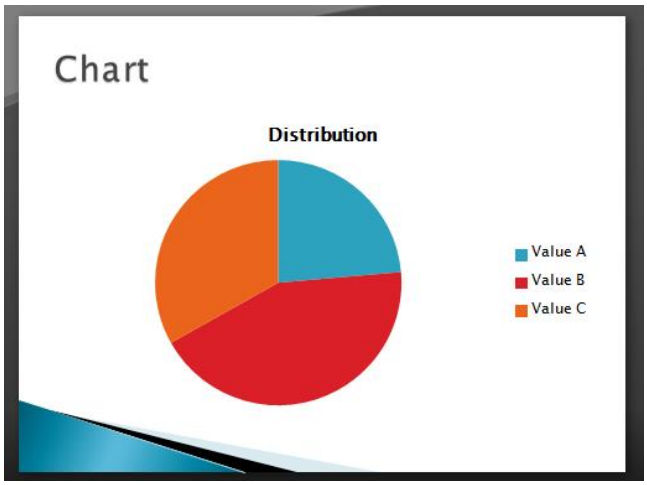
based from the Excel template:

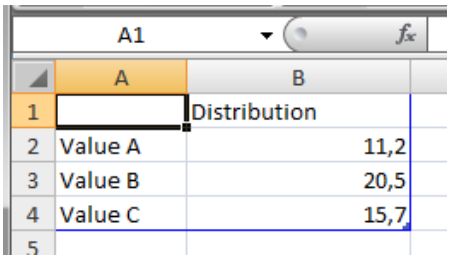


But once the underlying Excel is populated with data, e.g.:

```
{
 "chart": [
  {"value":"Value A","distribution":11.2},
  {"value":"Value B","distribution":20.5},
  {"value":"Value C","distribution":15.7}
 ]
}
```

the chart will be updated accordingly:





Tags defined within the Excel are visible in the **Tags** property on the *ITemplater* interface of the PowerPoint document. This makes them transparent to the application/processing. This means there is no need to unzip the pptx file, process the embedded xlsx files, but rather Templater does that behind the scenes.

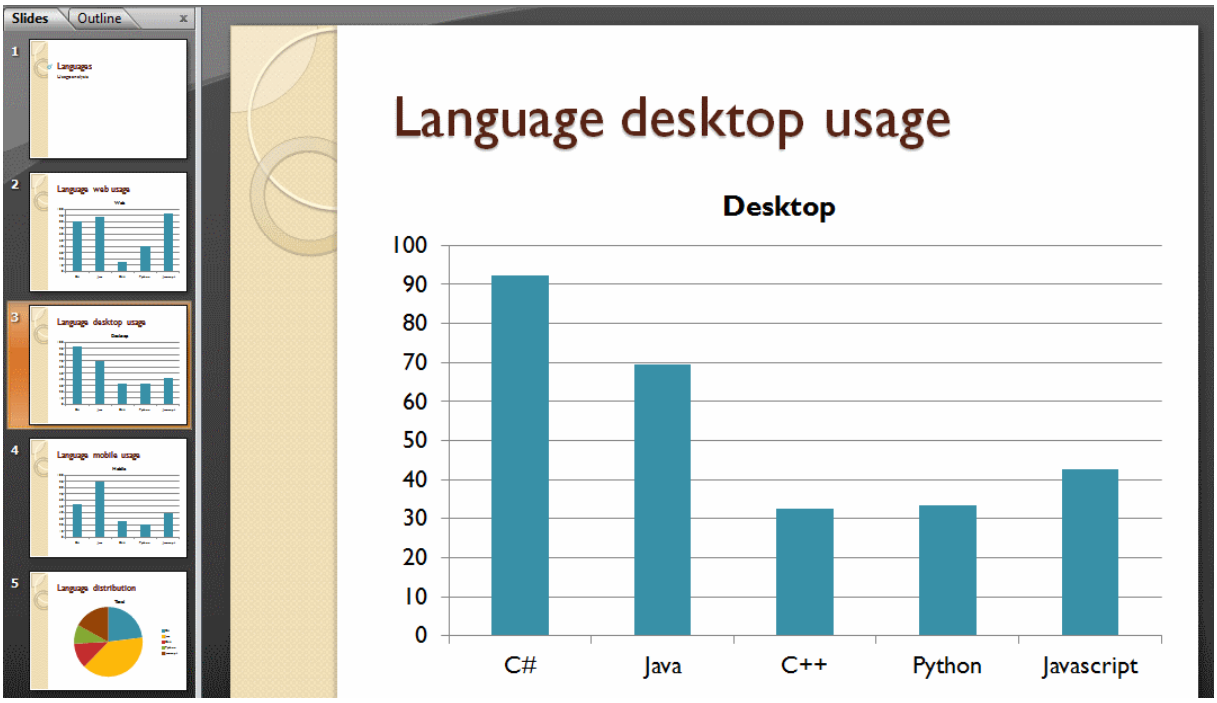
There are various charts in PowerPoint, such as pie charts, graphs and various others. They should all work seamlessly through Templater. Same collection can be used in multiple charts/tables which allows for different representation of the same data.

PowerPoint specific features

Slides

Resize behavior of slides is slightly different from the single Word document and from Excel sheets. When a tag is detected in a slide, but not inside list/table or chart the whole slide duplication will be invoked on resize. Resize(tags in a slide, 0) will remove the relevant slide(s). This is similar to the sheet duplication, but will happen more often/easily. Some common use cases for slide duplication/removal are:

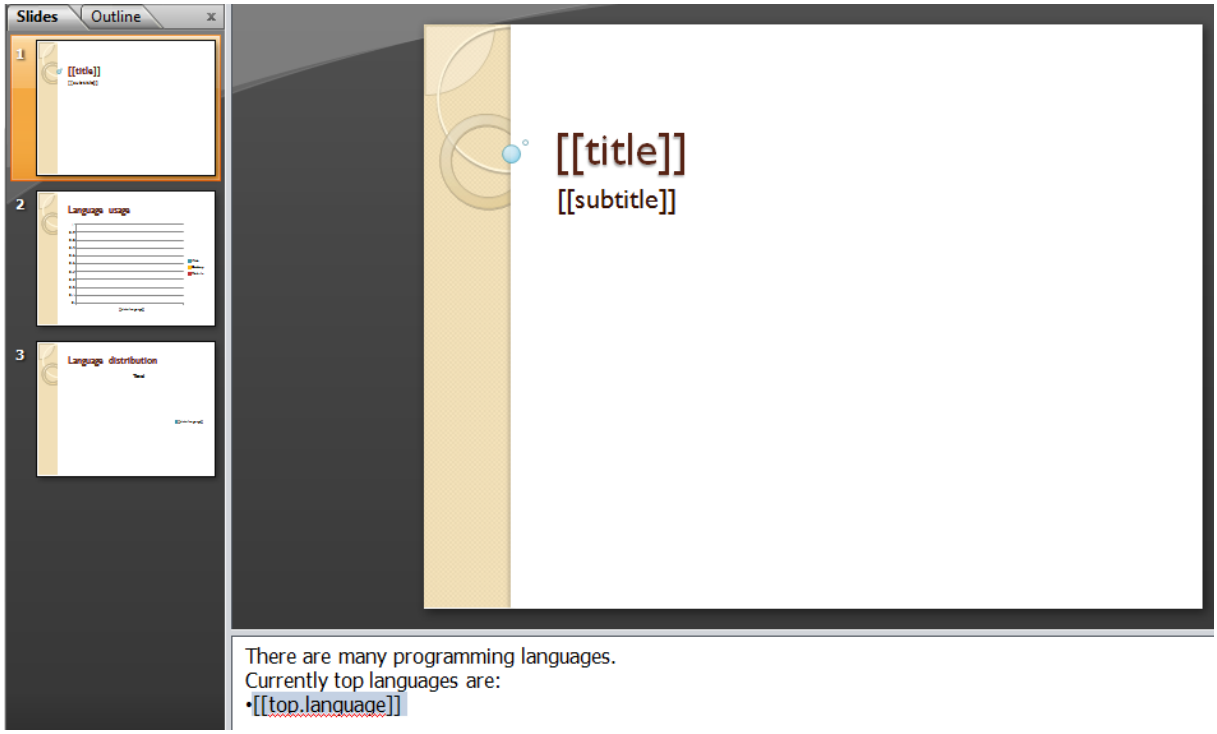
- conditional adjustment of presentation
  - by having many possible slide templates and removing non-relevant ones presentation can be adjusted to fit a specific role from a more general template
- repeating of same visualization for different data sources
  - a generic slide can be used to display graph of a relevant specific information; same slide template can be used to display different information in a same way



Notes

Each slide can have notes attached to it. Notes are used to provide contextual information for a slide and have much reduced feature set. If tag is detected within a list inside notes, resizing will only affect the specific list, while otherwise resize will affect the entire slide.

An example of a tag in a list inside notes looks like:



Images

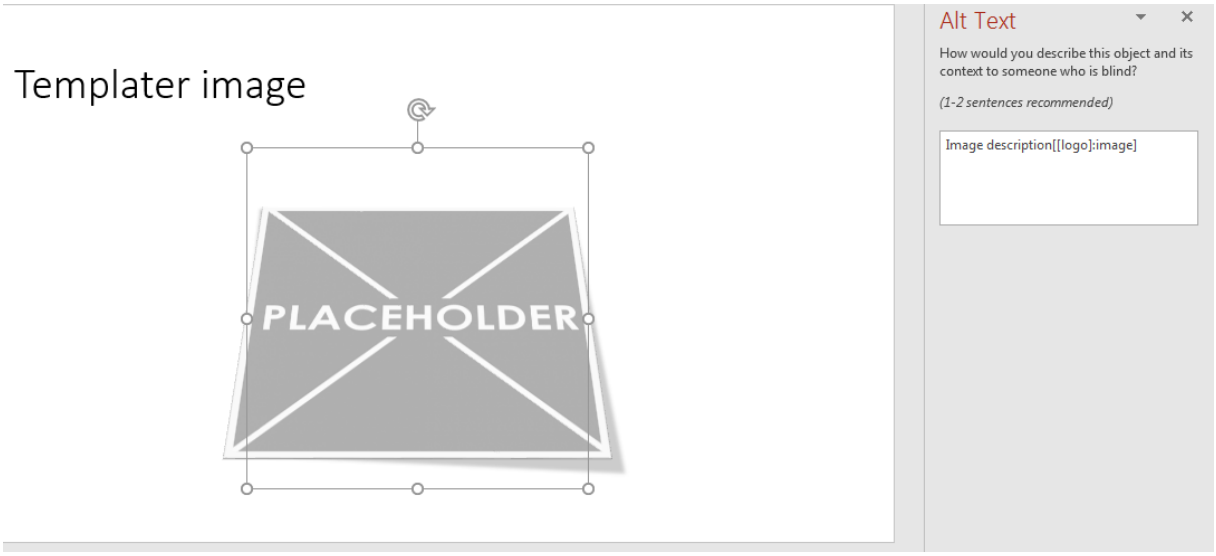
Unlike in Word and Excel, Templater only supports replacing predefined images in PowerPoint. This is done the same way via Templater specific data type: ImageInfo

To ease image usage and support platform with custom/different image libraries, default .NET/Java image types are by default converted into ImageInfo type:

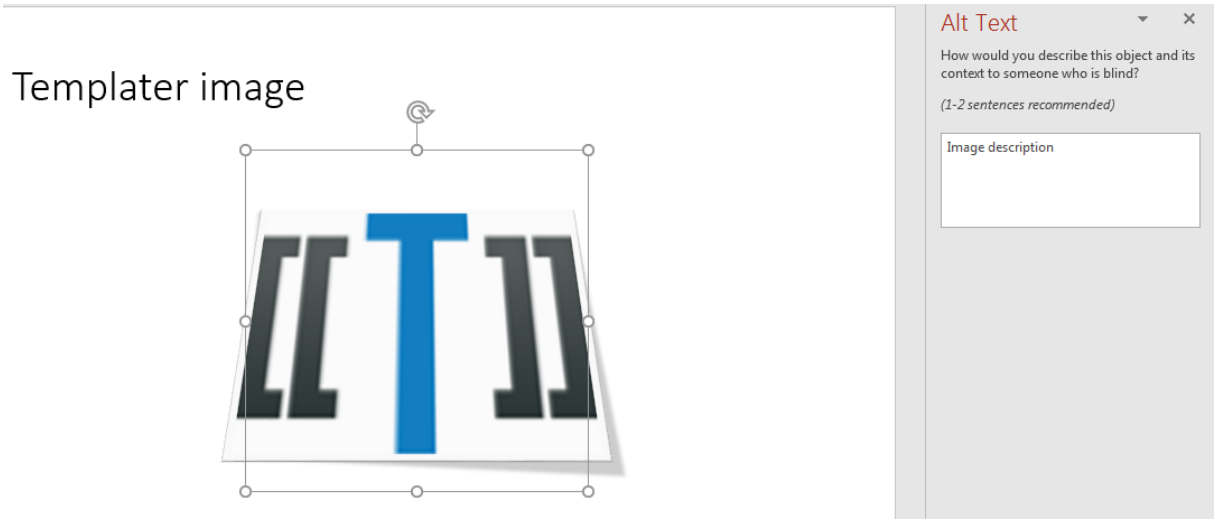
- .NET: Image and Icon
- Java: BufferedImage and ImageInputStream

The image files will be replaced in the ZIP file and referenced from the relevant parts.

This way image template can be preconfigured in advance. Special image style, such as text wrap, 3D format or any other image specific configuration, should be set-up, while Templater will recognize this image as long as it detected tag in *Alternative text* property:

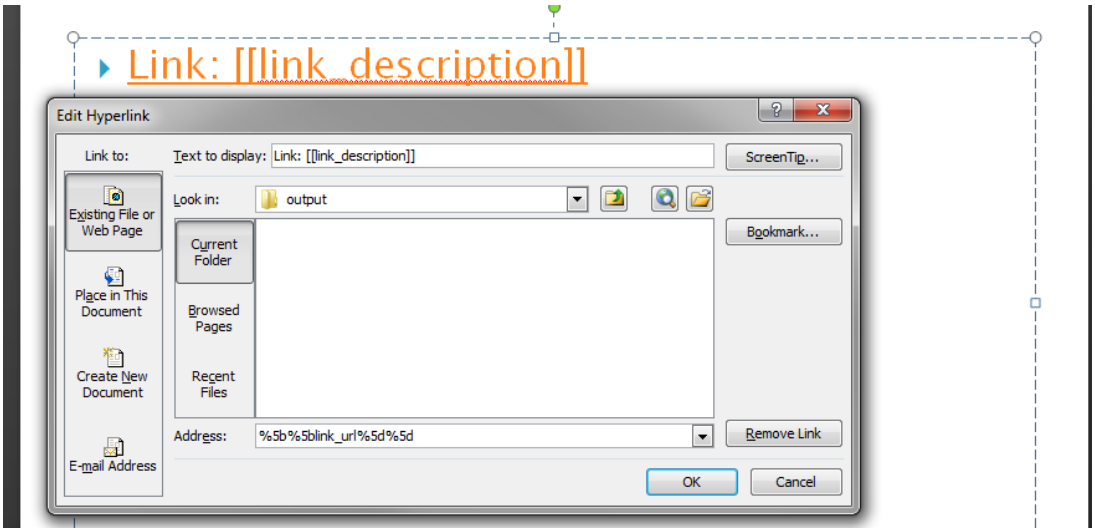


When run with JSON example logo expected result is produced:



Links

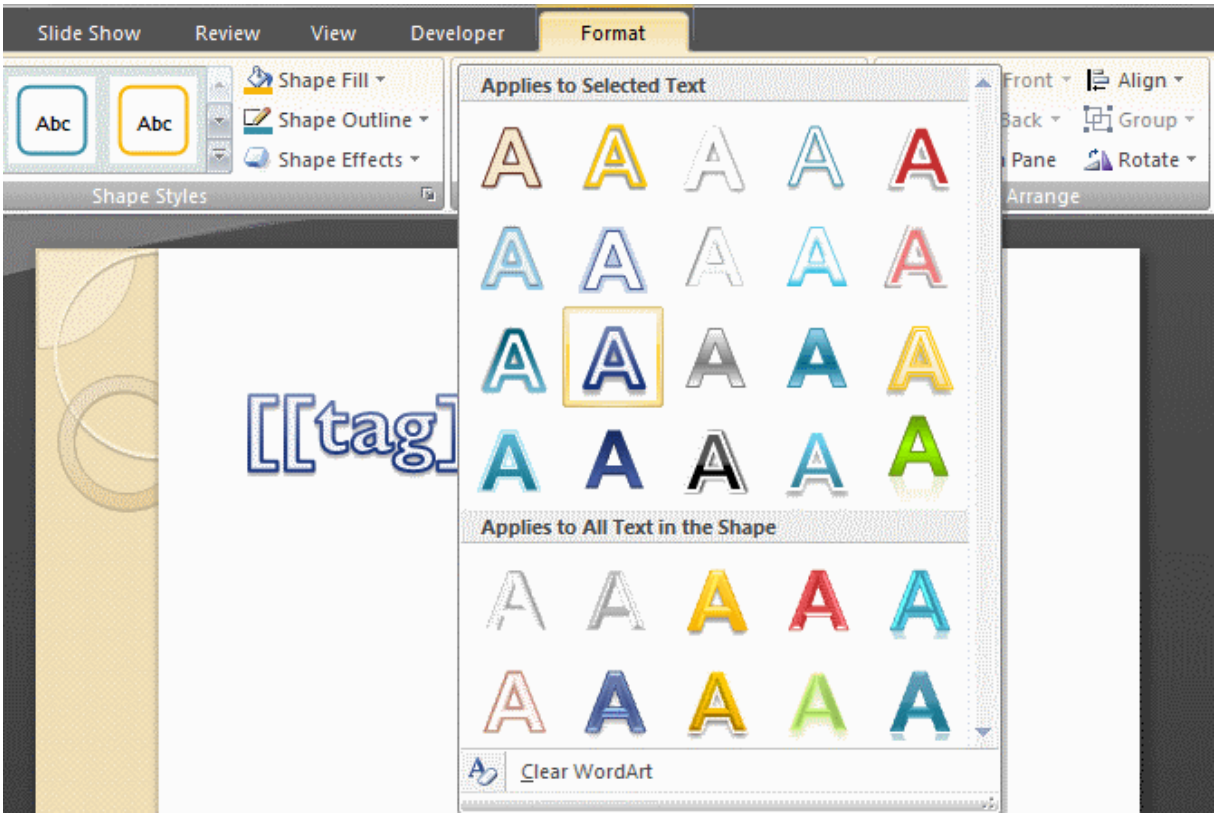
Templater analyzes hyperlinks and thus they work as expected. Hyperlink can have multiple tags or tag can be combined with static description. Address is url encoded which means that [[specific_url]] is converted into %5b%5bspecific_url%5d%5d when hyperlink is created, e.g.:



Special data types can also be used to create simple links (just a link, no custom description) when URI/URL is used as datatype.

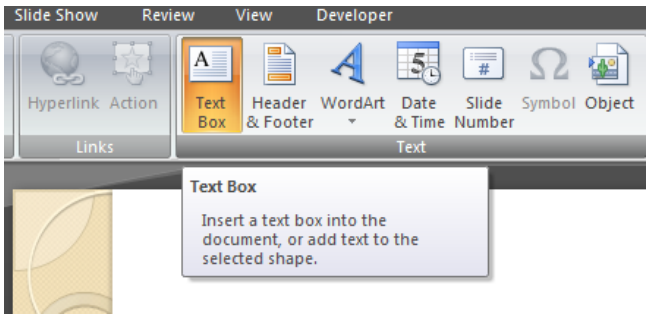
Word ART

Tags can also be used in Word ART and other similar features.



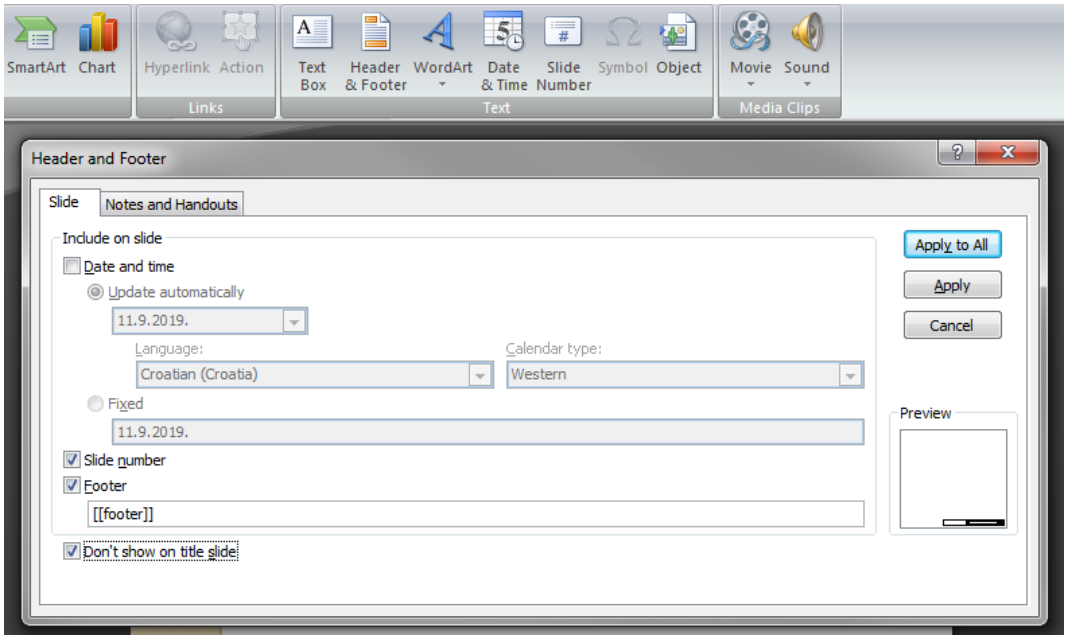
Text Box

Text box behave like any other region within a slide.



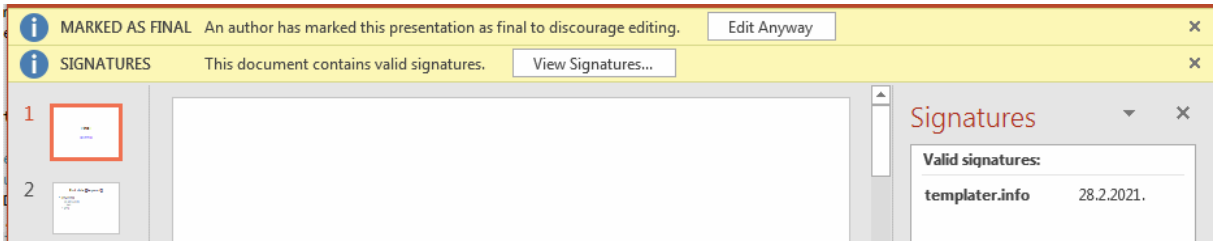
Header/footer

Slide numbers can be easily injected via Header & Footer functionality. If tags are used in footer they will be repeated automatically on all slides:



Digital signature

When creating presentations, it is useful to be able to trust numbers in the document. For this purpose, presentation can be signed to prove origination of the document.



Known issues

PowerPoint is the latest addition to the Templater as of v4.0 so some features are not yet supported. Most of them will be supported over time, unless they require rendering engine.

PDF export

A very common use case is to convert PowerPoint document into PDF. Unfortunately, this requires a PowerPoint rendering engine to work correctly.

There are several free and paid libraries which have sufficiently good PDF conversion such as Aspose or Silurus. But non-trivial documents quickly become non pixel-perfect during the conversion.

CSV/text features

Templater API spans various document formats. While there are various solutions to build text/html/CSV plain text output, there are some advantages if Templater is used for such purpose:

- CSV/text can be user configured (in the same way as Word/Excel/PowerPoint files can)
- same processing can be done on different formats (data structures can be reused to create export for xlsx, docx, pptx or csv without any code changes)
- Templater is heavily optimized and can output text files at high speed
- streaming can be utilized to create huge documents
    - while streaming can be utilized on xlsx (and docx/pptx), not all streaming features can be used as document is optimized for keeping all data structures in memory before the end of processing
- CSV/text will not add watermark message into the document
    - free version can be used without buying a license
- CSV can be used as data source for Power Query embedded within the same xlsx file

Simple documents

There are various use cases for simple text format usage:

- CSV (comma separated values) export
    - similar to xlsx (can be opened by Excel)
- fixed-width text format export
    - various legacy formats are exchanged between system as a specialized fixed-with format
- simple html/email messages
    - signup email and similar simple messages
- sms/chat messages
    - notification messages (upcoming events, late payments, ...)

Templater is able to process large number of documents, so if user facing customization is required, it's an appropriate choice for such a problem.

Usual Templater features work in text format, although some data types (such as Image or XML which inject XML as-is into the OOXML formats) don't have such meaning in text formats.

When CSV is used, additional low-level plugins for quoting string values should be registered, to simplify the conversion without extra metadata information.

Multi-line context

Templater supports multi-line context even for text processing. This is useful for non-trivial CSV exports which shown single row on two lines.

When **Resize**(tags, 0) is called, rows will be removed (and thus a pull-down will be performed).

Streaming documents

Unlike xlsx, pptx and docx, a text processing will stream to output if certain conditions are met:

- if there are more than streaming size[40] of processed rows
  - streaming will only be invoked on large number of rows
- if there are no tags left before the first tag which is currently resized
  - streaming will only be available if the context which is being processed has special tag setup
- when resized is called multiple times during processing
  - output will be flushed during the resize
  - this can be done manually, or by using a streaming data type such as Iterator/Enumerator

From a pseudocode streaming processing would look like:

1. open document
2. process headers, filters and other non-streaming data
3. repeat until end of data stream
   a. resize to accommodate for the current streaming chunk
      i. naive implementation would call resize(tags, 2) to create extra row
      ii. then call resize(tags, chunk size) for processing the current chunk
   b. process the current chunk
4. remove the extra row if necessary (it's not necessary only in some edge cases for non-trivial implementations)
5. process extra remaining tags (if any) which were dependent on the streaming data (and were located at the end of the template)

A streaming template can look like:

```
Date:;[[filter.date]];;;
User:;[[filter.user]];;;
;;;;
ID;Amount;Date;User;Timestamp
[[data.id]];[[data.amount]];[[data.date]:format];[[data.createdBy]];[[data.createdOn]]
```

when opened in Excel would look like a regular cell (without any styles)

[40] Default streaming size is 16k. This can be configured via streaming method in the configuration API

	A	B	C	D	E	F
1	Date:	[[filter.date]]				
2	User:	[[filter.user]]				
3						
4	ID	Amount	Date	User	Timestamp	
5	[[data.id]]	[[data.amount]]	[[data.date]:forma	[[data.createdBy]]	[[data.createdOn]]	
6						

To open CSV within Excel some culture specific requirements must be met:

- number decimal sign should match
  - dot (.) is decimal sign in US countries, while comma (,) is in most EU countries
- comma separator should match
  - comma (,) is separator in US countries, while semicolon (;) is in most EU countries

It's common to stream into a ZIP stream directly (instead of file) and thus further reduce the amount of used memory. Templater will reuse data structures and thus will only consume constant amount of memory, which means if data is iterated over in a streaming fashion (instead of being loaded all into memory) huge documents can be created.

XML features

Templater v7 adds support for XML as extension. This way, similarly to CSV/text, but within the rules of XML formats, Templater can be used to create XML/HTML files. While there are numerous existing solutions to this problem already, Templater unifies all this formats behind a same API, with same processing methods and plugins.

XML processing is also high performance and supports streaming, so Templater can be used to create rather complex/large XML documents if it needs be. Therefore, there are some advantages if Templater is used for such purpose:

- XML can be user configured (in the same way as Word/Excel/PowerPoint/CSV/txt files can)
- same processing can be done on different formats (data structures can be reused to create export for xlsx, docx, pptx, csv or xml without any code changes)
- Templater is heavily optimized and can output XML files at high speed
- streaming can be utilized to create huge documents with low memory usage
    - while streaming can be utilized on xlsx (and docx/pptx), not all streaming features can be used as document is optimized for keeping all data structures in memory before the end of processing
- Comment watermark message will be added into the document if unlicensed version is used
    - but since comments are ignored by most tools and should not create problems. Licensed version removes the watermark message

Simple documents

There are various use cases for simple XML format usage:

- allow runtime configuration of integration messages
- create expected XML document based on some external application format
- simple html messages which follow the XML as xhtml
    - signup email and similar simple messages

Templater is able to process large number of documents, so if user facing customization is required, it's an appropriate choice for such a problem.

Usual Templater features work in XML format, although some data types (such as Image) don't have any meaning in XML format.

Templater will respect the XML rules, meaning it will escape expected characters, so no additional configuration is required.

Streaming documents

Unlike xlsx, pptx and docx, a text and XML processing will stream to output if certain conditions are met:

- if there are more than streaming size[41] of processed rows
  - streaming will only be invoked on large number of rows
- if there are no tags left before the first tag which is currently resized
  - streaming will only be available if the context which is being processed has special tag setup
- when resized is called multiple times during processing
  - output will be flushed during the resize
  - this can be done manually, or by using a streaming data type such as Iterator/Enumerator

From a pseudocode streaming processing would look like:

1. open document
2. process headers, filters and other non-streaming data
3. repeat until end of data stream
   a. resize to accommodate for the current streaming chunk
   b. process the current chunk
4. process extra remaining tags (if any) which were dependent on the streaming data (and were located at the end of the template)

A streaming template can look like:

```
<data>
	<filter after="[[filter.after]]" before="[[filter.before]]" />
	<items>
		<item attribute="[[items.attribute]]">
			<name>[[items.name]]</name>
			<number>[[items.number]]</number>
		</item>
	</items>
</data>
```

It's common to stream into a gzip (instead of file) and thus further reduce the amount of used memory. Templater will reuse data structures and thus will only consume constant amount of memory, which means if data is iterated over in a streaming fashion (instead of being loaded all into memory) huge documents can be created.

While processing attributes, if null value is provided, Templater will remove the relevant attribute from output.

[41] Default streaming size is 16k. This can be configured via streaming method in the configuration API

Best practices

While Templater API is a minimal one, the feature set is quite big and therefore there are various best practices which can be followed when using Templater. Some of them are only applicable for enterprise applications, but it's good to be familiar with them since they will provide deeper understanding of how Templater works and how it should be used.

Complex documents

On the surface Templater looks just like a mail merge library. But once you scratch the surface all kinds of complex patterns emerge from deceptively simple operations:

- duplicate or remove tag
- replace tag with a value

Some of those emerging patterns are:

- document can consist for parts which are only conditionally shown
    - this way complex document can cover all use cases and then adjusted to fit only the relevant use case for the specific processing
- external code can be used to integrate complex behavior during the replacement
    - using plugins to load image on demand or convert it from argument
    - consuming third party libraries for complex conversions such as verbalizing numbers into text
    - enriching common use cases with appropriate metadata over time
- code can be reused between reporting and other parts of the system
- dynamic types allow for maximum ease of use due to natural matching with tags
- template defined for a single object can work for a collection of objects without any changes
    - this allows for easy bulk export instead of having a separate bulk only export

Still, the most important aspect of complex documents is that they are constructed in rich editor such as Microsoft Office. This provides all kinds of benefits:

- document layout can be prepared/defined much faster and will produce much nicer looking results - in contrast to defining layout logic in code
    - most of the time, it's not even feasible to create complex layouts which can be done in MS Office through code
- non-developers can take ownership of defining/managing the documents which provides better separation of work
- existing documents can be used as a starting point when some legal document needs to be created
- minor changes to the document are done in fraction of time and minimize the number of involved parties

Multi-step processing

Complex processing sometimes requires several passes through the template:

- first prepare the tags for second pass
  - using horizontal-resize, dynamic resize or just regular processing which creates new tags
- in the second pass process all tags with the expected values

Common use case for double processing is when multiple columns need to be used to display row information per some subset. While there are multiple solutions to this problem (such as dynamic resize per row) sometimes it's much faster to prepare the layout in first processing and then process the data in second, especially when large number of rows are exported.

A common pattern for such exports is to use a dictionary/map for dynamic part of the schema[42], while reusing existing classes/fields for static part of the schema.

An example of such template

	A	B	C	D	E	F	G
5	**GL Code**	**Account Name**	**Change in Balance**	**Closing Balance**	**[[organization.name]:horizontal-resize:whole-column]**		
6			**Total**	**Total**	**[[organization.description]]**		
7	**[[accs.acco**	**[[accs.accountName]**	**[[accs.totalBalance]**	**[[accs.closingBala**	[[organisation.tag]]		
8							

would be converted into:

	A	B	C	D	E	F
5	**GL Code**	**Account Name**	**Change in Balance**	**Closing Balance**	**Region A**	**Region B**
6			**Total**	**Total**	**Sub-total**	**Sub-total**
7	**[[accs.acco**	**[[accs.accountName]**	**[[accs.totalBalance]**	**[[accs.closingBala**	[[accs.orgA.subtotal]]	[[accs.orgB.subtotal]]
8						

Another alternative (for this specific example) would be to instead show the data in raw tabular format and then use the pivot to transform it into another format, but sometimes preparing data before passing it to Excel allows for more user-friendly design of various charts and other pivot tables.

When processing templates in such a way, in memory streams should be used between steps; which means even multi-step processing should be fast.

Hierarchical structures

Templater supports arbitrary deep hierarchies[43]. A common pattern is that once the document is setup for exporting a single instance (such as a single invoice), Templater can support export of multiple instances by just passing in collection instead of a single object instance for processing.

[42] When allowing the use of maps for navigation it's important to prepopulate maps with all possible values for keys, not just the ones which are present in specific rows. This will reduce problems with tags which were left in the document since there was no data in the map.
[43] There is a configurable limit of 8 to prevent bad context detection. This can be changed during initialization

But data structure hierarchy should closely match document hierarchy. That way it will be easy to reason about which regions of the document need to be duplicated and how data translates from the model to the document.

If part of documents needs to be duplicated this means data structure should have an appropriate matching collection.

Sometimes it's useful to transform collection into map, especially when collection has a specific key as identifier. This way only relevant part of the collection can be shown as a column, instead of region of the document being duplicated.

Another common pattern is to have first/last property instead of a collection, as this is the only relevant information most of the time.

Ideally domain model can combine all of the above and thus allow for customization of documents in various ways. An example could look like:

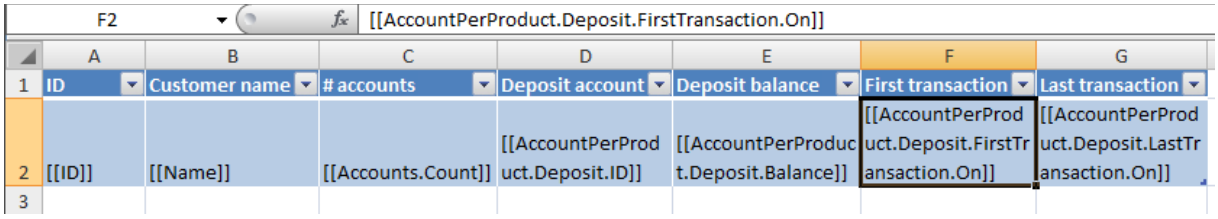
```
public class Customer
{
        public string ID;
        public string Name;
        public List<Account> Accounts;
        public Dictionary<string, Account> AccountPerProduct;
        public Account FirstAccount;
        public Account LastAccount;
}
public class Account
{
        public string ID;
        public decimal Balance;
        public string Product;
        public DateTime CreatedOn;
        public List<Transaction> Transactions;
        public Transaction FirstTransaction;
        public Transaction LastTransaction;
}
public class Transaction
{
        public string ID;
        public decimal Amount;
        public DateTime On;
}
```

Common patterns can be extracted into appropriate properties:

- list of all accounts/transactions is available on customer/account
- accounts are grouped per product into a map
    - this allows use of product id as a key over the dictionary
    - dictionary should be populated for all possible products, with nulls for accounts which do not exists for a product - this way generic template can be designed which will work in all cases, not just when there is a particular product on a client

- first/last account/transaction is available on customer/account
    - depending on business logic and use cases there could be rules which are perfect fit for such a model, as it could allow only a single active account
    - quick info about when the last/first transaction happened can be shown along the customer

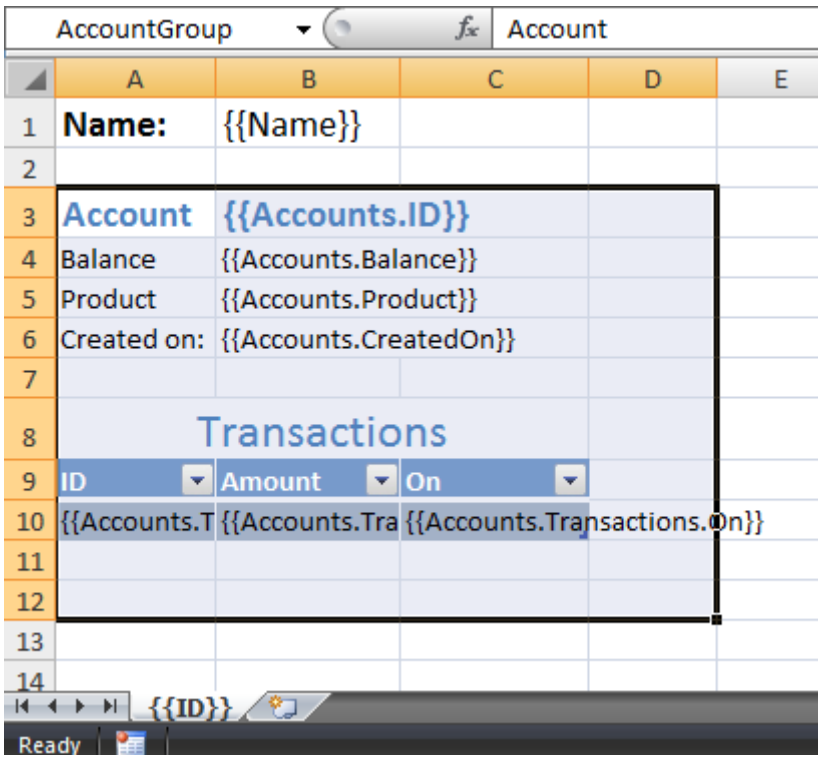
By using this model, a rich client row can be displayed, which does not contain any nesting, even though the actual model has 2 level nesting (Customer -> Account -> Transaction). An example:



Common use case for hierarchical structures is to display them in hierarchical way, which for the same model could mean:

- sheet per customer
- accounts repeated in a sheet
- transactions repeated for an account

Such a template could look like:

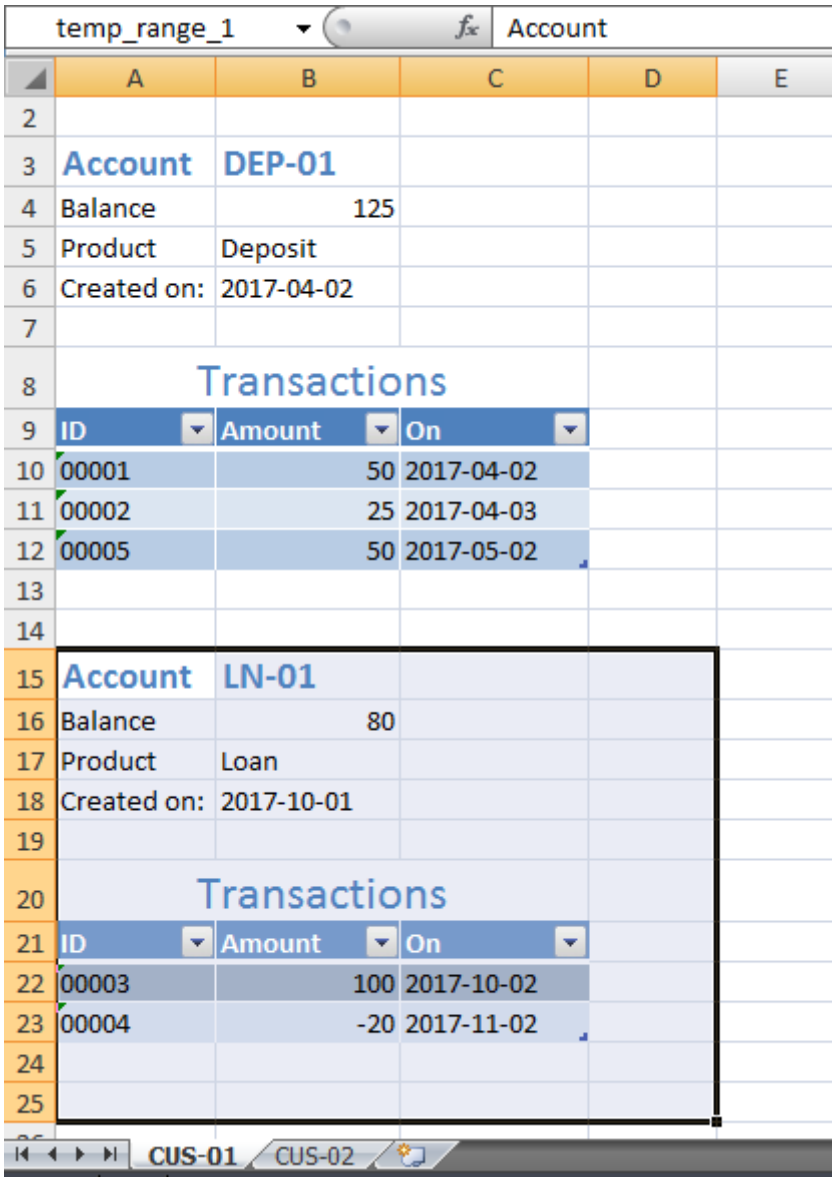


when paired when hierarchical data:

[

```
 {"ID":"CUS-01","Name":"Customer 1","Accounts":[
  {"ID":"DEP-01","Balance":125,"Product":"Deposit","CreatedOn":"2017-04-02","Transactions":[
    {"ID":"00001","Amount":50,"On":"2017-04-02"},
    {"ID":"00002","Amount":25,"On":"2017-04-03"},
    {"ID":"00005","Amount":50,"On":"2017-05-02"}
  ]},
  {"ID":"LN-01","Balance":80,"Product":"Loan","CreatedOn":"2017-10-01","Transactions":[
    {"ID":"00003","Amount":100,"On":"2017-10-02"},
    {"ID":"00004","Amount":-20,"On":"2017-11-02"}
  ]}
 ]},
 {"ID":"CUS-02","Name":"Customer 2","Accounts":[
  {"ID":"DEP-02","Balance":300,"Product":"Deposit","CreatedOn":"2018-01-15","Transactions":[
    {"ID":"00010","Amount":100,"On":"2018-02-12"},
    {"ID":"00011","Amount":200,"On":"2018-03-12"}
  ]}
 ]}
]
```


would be transformed into nested representations:



Collapse

Most of the time collapse metadata should not be used. Instead data models should indicate what region of the document should be removed during processing. But on more complex documents, especially when there are different display variants for the same part of the data, built-in or custom collapse plugins are required.

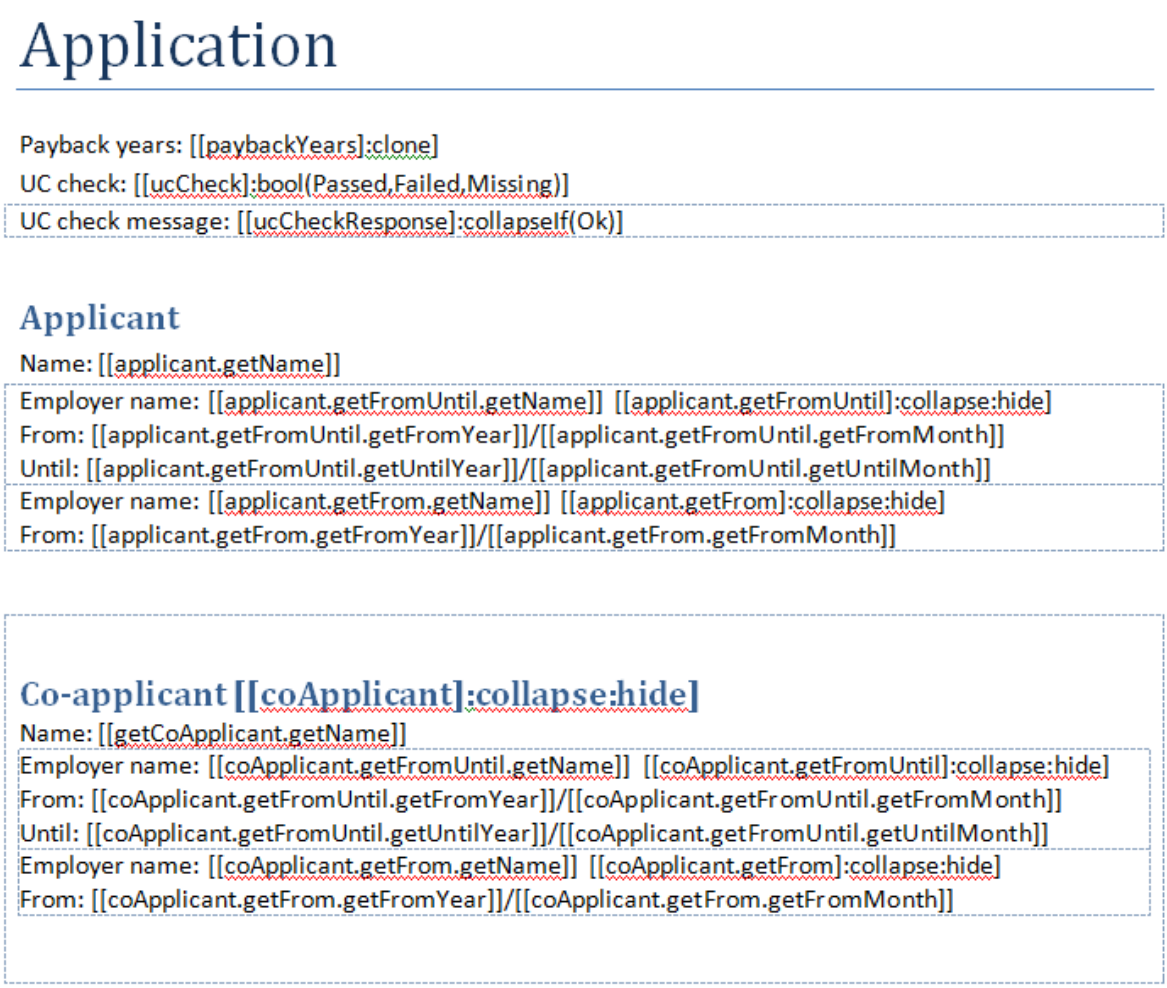
Calling **Resize**(tags, 0) on specific parts of the document can have many applications. Often, it is sufficiently good just to remove a single row, instead of providing whole alternative layout for some special input.

For highly complex documents it's also useful to combine collapse and multi-document processing since it will be much easier to reason about how Templater will behave.

Common use case for removal part of the document is when that document part has no values and thus should not be displayed. An example would be a loan application which has an optional co-applicant and thus we want to include relevant part of the document only when co-applicant is used:

```
public class Application
{
        public int paybackYears;
        public bool? ucCheck;
        public string ucCheckResponse;
        public Applicant applicant;
        public Applicant coApplicant;
}
```

paired with template:

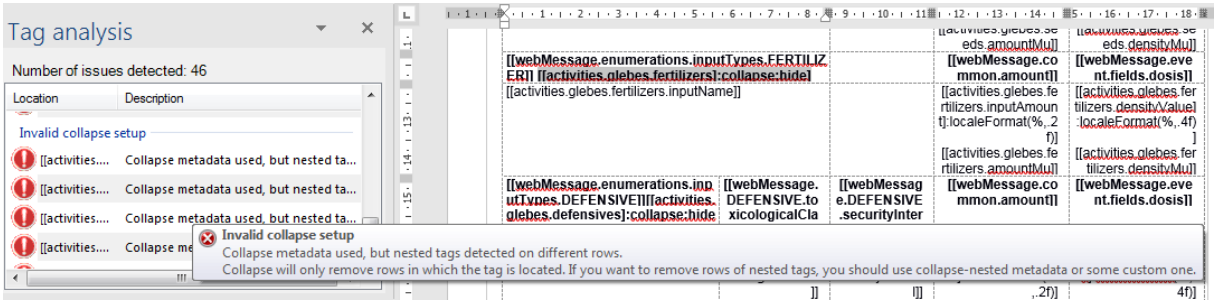


will manage the visibility of Co-applicant part of the document through the null value of **coApplicant** property. When collapse is paired with hide metadata this means that when the value is present, instead of being displayed as ToString representation of an instance, it will be hidden instead.

It is sufficient to just specify a single tag if that will remove entire chunk of the document since during removal Templater will inspect which other tags will be removed and remove them also in the process.

Another common use case of collapse is when paired with sections to have two different layouts for same data (or portion of the data). Sections are used to indicate start/stop boundary which will be removed during collapse.

Templater Editor will check for common setup problems, such as using collapse instead of collapse-nested when appropriate:



Performance/memory optimizations

While Templater is quite optimized and high performance for most documents processing it's good to be aware of various minor details which can improve the performance, sometimes quite significantly.

Tag sharing across sheets

When a same collection is used across different Excel sheets, Templater will process it in a specialized way. This incurs some memory and performance overhead. While this is not important for simple documents (with only few thousand rows) it could be noticeable on really large documents. A quick fix which will improve the performance of the processing is to use a different tag for different sheets. An example of such fix would be to expose multiple properties as aliases, e.g.:

```
public class Report
{
        public HeaderInfo Header = ...;
        public List<Item> Items = ...;
        public List<Item> Items1 { get { return Items; } }
        public List<Item> Items2 { get { return Items; } }
        public List<Item> Items3 { get { return Items; } }
}
```

This way Sheet1 can use the **[[Items1.**.. tags, Sheet2 can use the **[[Items2.** tags, etc...

Sometimes even better workaround would be not to use a single report, since it might be better to create multiple variants of a report instead of trying to put multiple variants inside a single Excel file.

Since v5 it is discouraged to add additional properties to the model in favor of using custom navigation paths for such purpose. Templater will share tags as long as they have the same path, but this can be changed by introducing specific metadata into navigation, e.g.:

[[Items:id(1).Property]]

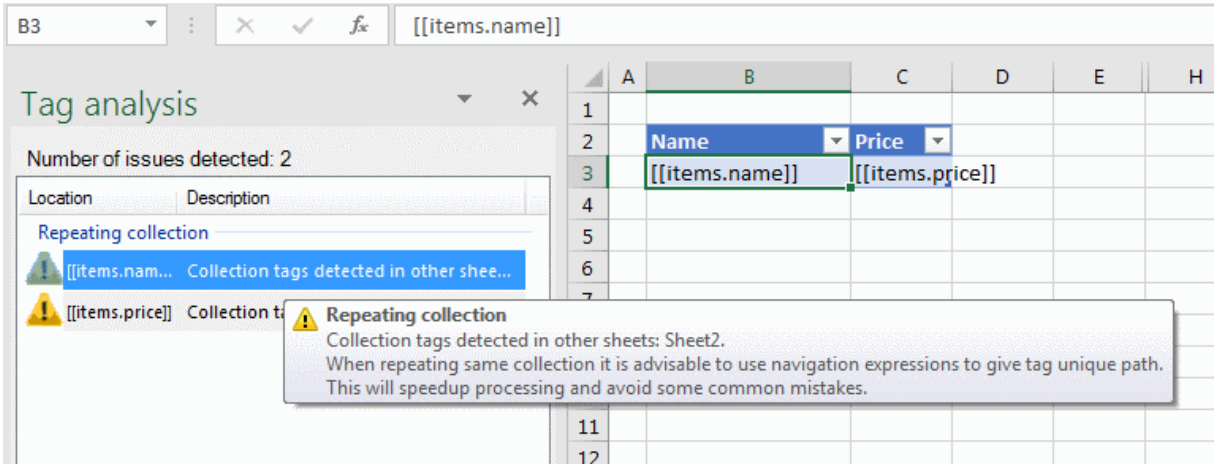
[[Items:id(2).Property]]

…

Where semicolon is defined as navigation separator.

These workarounds will only work on collections which can be processed multiple times. If processing a collection consumes it (a streaming collection) then only the sharing will work as expected.

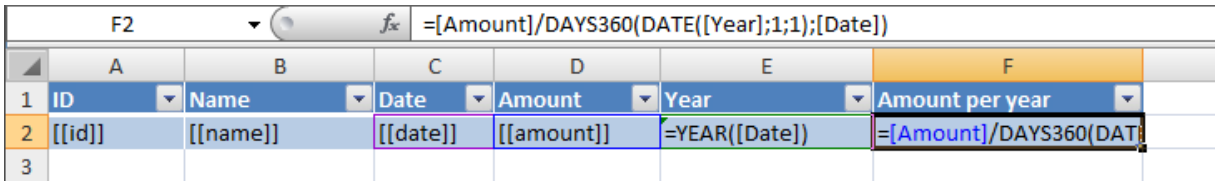
Templater Editor will report such setup with warnings:



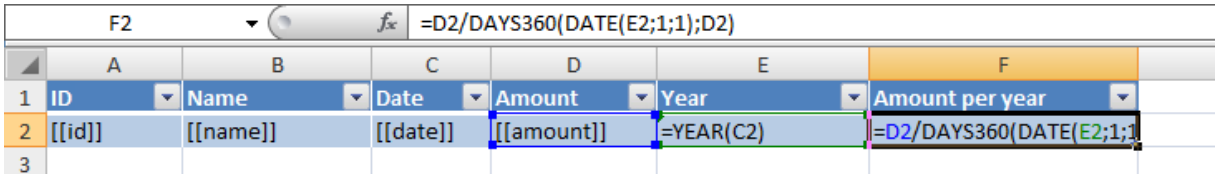
Using formulas without cell references

During row duplication Templater needs to parse, rewrite formula so it can be used in a new row. If formulas are defined in such a way that resize will not change their expression Templater will process it much faster.

Optimal formula example:



Suboptimal formula example:

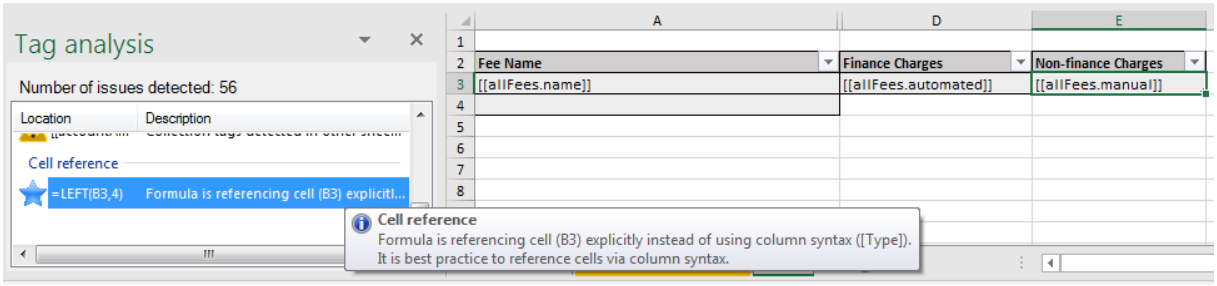


The main difference between an optimal and suboptimal formula is that optimal formula will have the same display expression even when pushed around or duplicated.

There are few other basic rules:

- operations can reference named ranges instead of ranges which change
  - SUM([named_range]) vs SUM(E2:T2)
- it's better to put in explicit value than to use formula
  - sometimes formula expressions can be expressed in the domain model
  - this can make complex Excel file much smaller and thus faster to process and open after processing
- property navigation can be used instead of formula expressions
  - [[date.Year]] can produce same value as evaluating =YEAR([date_cell]) in Excel
    - as a bonus it doesn't require evaluation after document is opened
- often tags can be combined instead of complex formulas
  - [[date.Year]] / [[date.Month]] can produce the same result as =YEAR(XX) & "/" & MONTH(XX)

Templater Editor will suggest formula improvements:



Unnesting hierarchical models

While Templater encourages deep hierarchies, on large complex documents there are few performance tricks which can sometimes yield a significant performance improvement.

One trick to "unnest" a hierarchy is to build a specialized model which looks like a hierarchy, but it's actually flat (at least one level smaller).

E.g., for hierarchy such as:

Client collection -> Account collection

can be flattened into

Account collection (with Client info)

Model such as:

```
public class Client
{
        public string Name;
        public string ID;
        public List<Account> Accounts;
}
public class Account
{
        public string ID;
```

```
        public decimal Balance;
        public DateTime OpenedOn;
}
```

can be written as[44]

```
public class Account
{
        public string ID;
        public decimal Balance;
        public DateTime OpenedOn;

        public Client Client; //use for Client on every row
        public Client ClientFirst; //use for Client only on first row
}
public class Client
{
        public string Name;
        public string ID;
}
```

In both cases report can have model such as:

```
public class Report
{
        public List<Client> Clients;
        public List<Account> Accounts;
}
```

For reports which need to list all clients and their accounts instead of having report such as:

	A	B	C	D	E	F
1	Client ID	Client name	Account ID	Account balance	Account created	
2	[[Clients.ID]]	[[Clients.Name]]	[[Clients.Accounts.ID]	[[Clients.Accounts.Balance]]	[[Clients.Accounts.OpenedOn]]	
3	Total			0		
4						

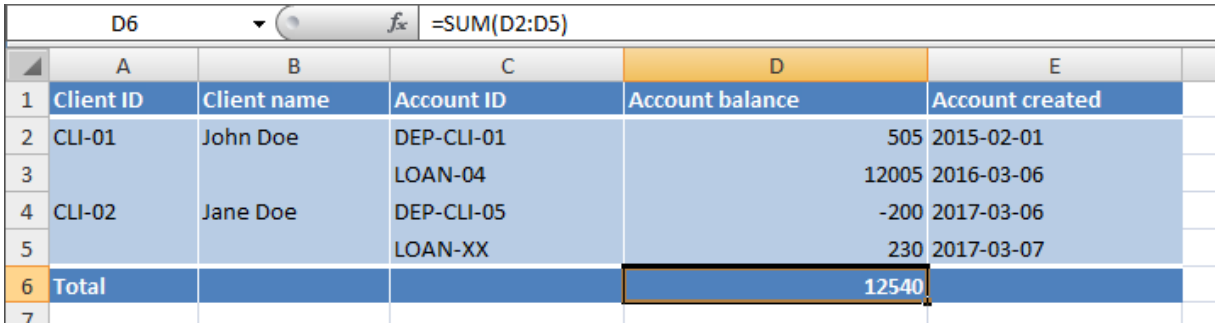
when paired with input such as:

```
{
 "Clients":[
  {"ID":"CLI-01","Name":"John Doe","Accounts":[
   {"ID":"DEP-CLI-01","Balance":505,"OpenedOn":"2015-02-01"},
   {"ID":"LOAN-04","Balance":12005,"OpenedOn":"2016-03-06"}
  ]},
  {"ID":"CLI-02","Name":"Jane Doe","Accounts":[
   {"ID":"DEP-CLI-05","Balance":-200,"OpenedOn":"2017-03-06"},
   {"ID":"LOAN-XX","Balance":230,"OpenedOn":"2017-03-07"}
  ]}
 ]
```

[44] If model is written for reporting purpose only, its fine to even have both options in a model, which is what happens anyway on models which are evolved over time for transition purposes

}

will result in output:

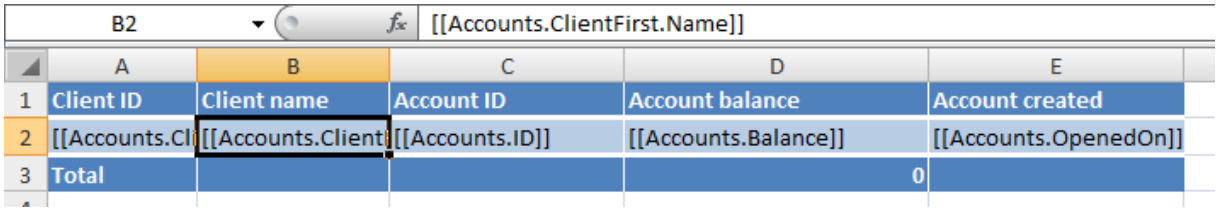


	A	B	C	D	E
1	Client ID	Client name	Account ID	Account balance	Account created
2	CLI-01	John Doe	DEP-CLI-01	505	2015-02-01
3			LOAN-04	12005	2016-03-06
4	CLI-02	Jane Doe	DEP-CLI-05	-200	2017-03-06
5			LOAN-XX	230	2017-03-07
6	Total			12540	

Same result can be created with only a single nesting level if the second model is used where ClientFirst is populated only on the first account for a client[45], e.g.:

```
{
 "Accounts":[
  {"ID":"DEP-CLI-01","Balance":505,"OpenedOn":"2015-02-01",
   "ClientFirst":{"ID":"CLI-01","Name":"John Doe"}},
  {"ID":"LOAN-04","Balance":12005,"OpenedOn":"2016-03-06","ClientFirst":null},
  {"ID":"DEP-CLI-05","Balance":-200,"OpenedOn":"2017-03-06",
   "ClientFirst":{"ID":"CLI-01","Name":"John Doe"}},
  {"ID":"LOAN-XX","Balance":230,"OpenedOn":"2017-03-07","ClientFirst":null}
 ]
}
```

by using a different template:



	A	B	C	D	E
1	Client ID	Client name	Account ID	Account balance	Account created
2	[[Accounts.Cl	[[Accounts.Client	[[Accounts.ID]]	[[Accounts.Balance]]	[[Accounts.OpenedOn]]
3	Total			0	

Java XML memory usage

Prior to v7 Templater relied heavily on underlying XML libraries. Thus, it was recommended to setup specific xml libraries during initialization:

- xmlBuilder
- xmlTransformer

With v7 there is no more need for this tweaking, as Templater uses own internal XML libraries.

[45] This is much easier to implement when actual data structures are used, rather than JSON which is passed around

Java XML streaming

OutputStream for XML should be memory based without compression or backed by buffer. By default, Java will frequently flush content of XMLStreamWriter to output stream. Significant performance improvements can be gained by wrapping output file with a BufferedOutputStream. Code would then look like:

```
InputStream is = ...
FileOutputStream fs = ...
try (BufferedOutputStream bos = new BufferedOutputStream(fs, 65536);
     TemplateDocument doc = Configuration.factory().open(is, "xml", bos)) {
    doc.process(...);
}
```

Factory reuse

When Templater is used to create/process large number of small documents, reusing factory can yield significant gains. Templater will not switch threads during processing, so thread locals and other tricks can be used to further optimize interaction with plugins configured during library initialization.

Class visibility in Java

While Templater will only work with public methods and fields, non-public classes can be sent for processing in which case Templater will have to change visibility during reflection invocation. Since it restores visibility after the operation to previous state this operation has high overhead. To avoid it used classes should be public which will not require visibility change and thus they will be very performant.

Execution monitoring

While most parts of Templater are highly optimized, when large documents are being processed, application can run out of memory and go into special Garbage Collection loop which slows down processing significantly, often resulting in OutOfMemory exception.

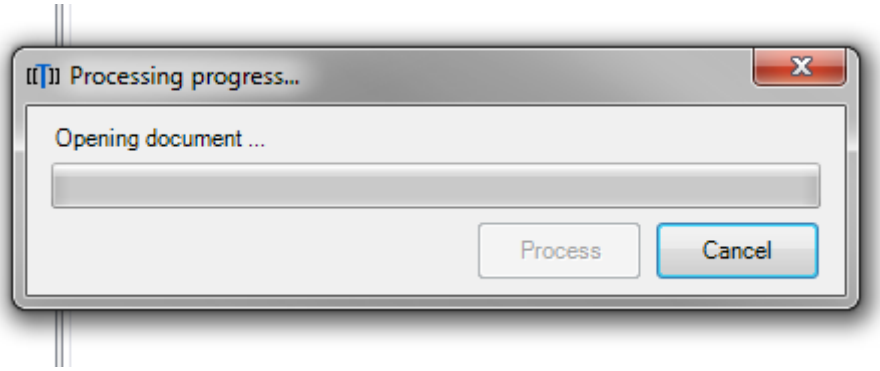
To combat such problems in a generic way, Templater processing allows for a cancellation token argument, so execution can be canceled. If processing fails to finish within the specified timeframe, processing can be notified of cancellation request, which means Templater will stop processing quickly after that[46].

An example of processing with timeout of 30 seconds would look like:

```
var cts = new CancellationTokenSource();
cts.CancelAfter(30000);
var factory = Configuration.Factory;
using (var doc = factory.Open(input, extension, output, cts.Token))
{
```

Same feature is available from Templater Editor to cancel processing:

[46] Templater checks the state of CancellationToken on critical places, and will throw relevant exception in case of such signal to stop further processing

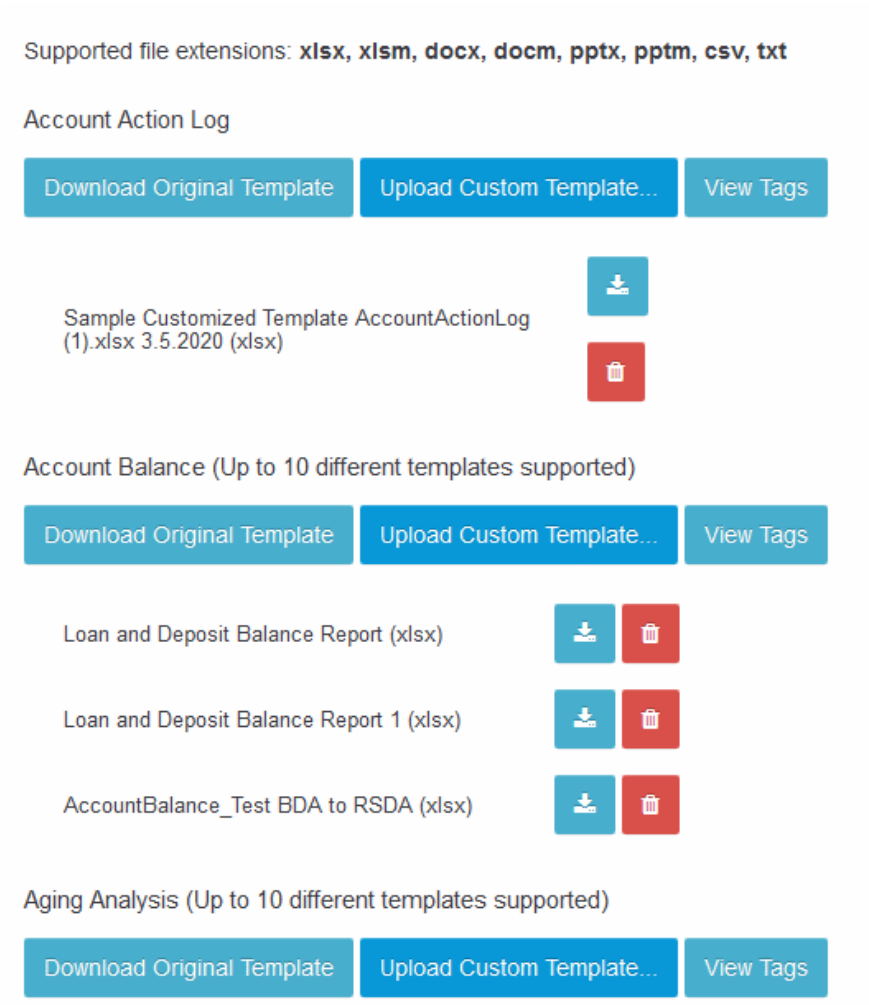


Tag management

Due to the way tags are defined within the document it's prudent to have some specialized logic around the tag management:

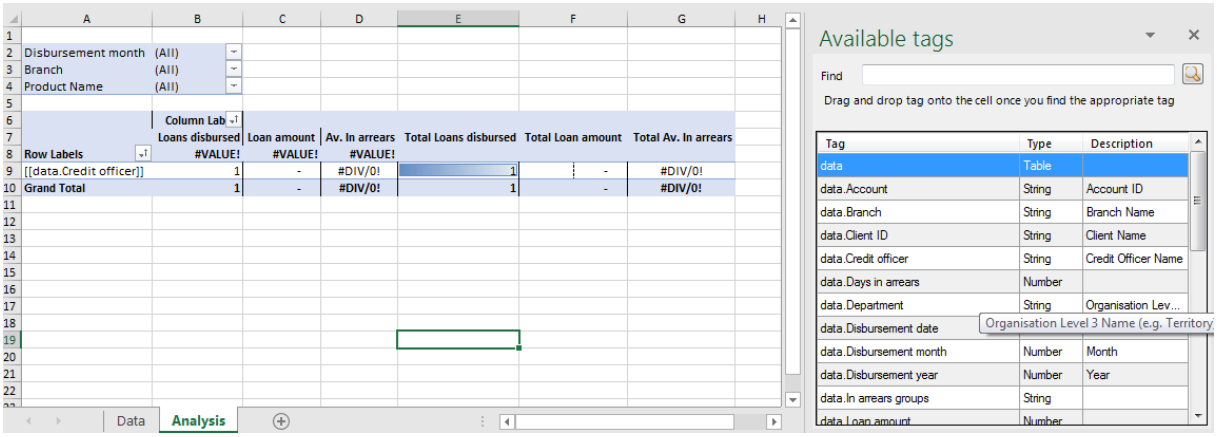
- tag discovery - it should be easily discoverable which tags exists
  - sometimes tag is bound to the actual data on the systems (especially when exposed through maps)
  - this might be more complicated when there is no actual schema in the system
  - ideally templates should be bound to schema which Templater Editor picks up
- document validation - before the document is "accepted" by the system, tag validation should be performed on it
  - unless the system restricts how the tags can be bound with the document, it will be quite common to have tag typos
  - in a living systems tag can change from version to version - and thus tags valid in an earlier version can be invalid in a new version
  - Templater Editor can pick up most of the heavy lifting here, as long as its integrated with the system via schema definition
- additional documentation – while tags should be named clearly which should explain their purpose, it is very useful to provide additional information for each tag via metadata integration
  - tag description – explaining the purpose and usage of the tag
  - tag status – to deprecate problematic tags over time
  - example – showing actual expected value for easier understanding
  - category – grouping tags into same buckets

An example of tag management looks like:



where each report has several variations, its own specialized validations/tags/schema and example data set.

It is highly recommended to embed schema information into templates before returning file to the user on download, as this will have the best possible experience for tag management.



User defined plugins

If metadata or low-level plugins are used, Templater will call them quite often on large documents. This means plugins should avoid allocations whenever possible, either by caching, thread local variables or similar means.

Normally, it's often not possible to avoid allocations completely, which can be fine most of the time. It's recommended to use a profiler for checking if user defined plugins are causing problems if there is significant memory usage or processing takes a while.

F.A.Q.

Q: **I'd like to test Templater before buying. How can I get a demo/trial license?**
A: Templater can be tested without buying a license, as all features are available even without a license. In case when license is not provided a watermark message will be added to the document.

Q: **Can I export files to PDF?**
A: Templater does not have a PDF export. All libraries we know of suffer from pixel perfect issues. We suggest using a MS Office for PDF conversion and when this is not possible use 3rd party libraries such as Aspose, Spire or Silurus. For low budget solutions, LibreOffice in headless mode can be used, although there will most likely be issues on complex documents.

Q: **Can I insert image into the document?**
A: Yes. Use of special data type is required to insert an image: ImageInfo. Builtin conversion exists for language specific types (.NET: System.Drawing.Image/Icon, Java: BufferedImage/ImageInputStream).

Q: **How can I pass JSON for processing?**
A: Dictionaries/maps and lists/arrays can be used out-of-the-box. Once JSON is converted into maps and lists, it can be passed to Templater for processing.

Q: **I need some help, but my support period has expired. What can I do?**
A: We suggest asking question at Github: https://github.com/ngs-doo/TemplaterExamples/issues. Even when support period expires, license will be valid for new minor version releases. Often just using the latest version might resolve some problem.

Q: **Which license I need to buy for a SaaS product?**
A: Templater license allows for integration into third party apps which add significant other functionality. This means that if your product is a business application you need to buy only one license. If your product is exposing Templater API to others over the Internet you need to sign additional Metering agreement.

Q: **Once I try to open document Office complains about corrupted document. How can I fix this?**
A: Most of the time reason for document corruption is use of non MS Office tools to prepare the document. "Corruption" usually means that MS Office tools are unable to recognize some specific feature when loading the file. In those cases, just saving the template in MS Office tools can resolve the issue. Sometimes document must be recreated from scratch in MS Office tools. If the error still persists, please send a reproducible for documents which were created in MS Office.

Q: **Document does not look as I expect it to after processing.**
A: If you are within your support period, please send us an email with a reproducible and the expected output. An easy way to create a reproducible is to use JSON builds: https://github.com/ngs-doo/TemplaterExamples/releases/latest.
If your support period has expired, please use the public channels such as Github (https://github.com/ngs-doo/TemplaterExamples/issues) for community help