M2Doc User Guide

# Template authoring

The M2Doc technology adopts an approach where the document authoring tools (Libre Office, Open Office, MS Word) are leverage as much as possible. What other tool is more adapted to style and static part authoring?

Furthermore, these tools are quite common and widely adopted so that there's no necesity to learn yet another document authoring tool. Last but not least, there's a great deal of document models legacy all over the places that should be reused as easily as possible.

Templates are made of static parts and dynamic parts. Static parts are produced in the generated document as they are in the templates while dynamic parts are replaced by some text which depends on the provided input models. Dynamic parts are provided in fields so that there's always a clear separation between static and dynamic parts.

## How to cope with field codes

The edition of template must be made in a mode where field codes are visible. In MS Word, you can toggle this mode on/off by pressing Alt-F9. When the mode is on, you should see this **Erreur ! Signet non défini.** like the next picture shows: 

To insert a new field, press Ctrl-F9. You obtain an empty field like this: That you must edit to provide the code and instructions.

The next section gives all the details that are necessary to edit M2Doc generation tags. Alternatively, if you start you’re template from the provided model (templateModel.dotx), you can use the Insert>QuickPart>AutoText to insert M2Doc generation tags. Once you know the tag, it might be faster to just edit them.

You must toggle field codes on to read this document.

## M2Doc static generation tags

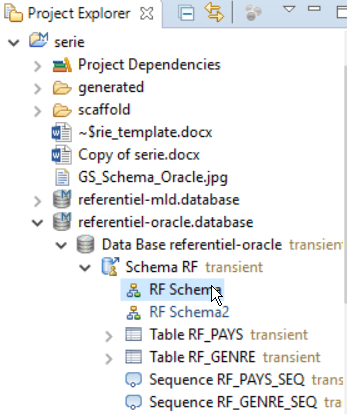
M2Doc provides right now 4 generation tags that have fixed formats. Those tags are described below:

* {m:<query>} : the directive is replaced with the string representation of the query’s evaluation result
* { m:for <var> | <query>} iterated body { m:endfor } : the directive is replaced by the iterated generation of the body over a collection of values. The specified variable is successively bound to the values of the evaluated collection so that it is accessible from queries inside the body.
* { m:if <expression1>} true branch { m:elseif <expression2> } … { m:elseif <expression\_n>} … { m:else } … {m:endif } : conditional generation. The first branch among the if and elseif directives which expression evaluates to true is processed. If no expression evaluates to true, the else branch is processed (if present).
* { m:image file:"image file path" width:"image insertion width" heigth:"image insertion heigth" legend:"image legend" legendPos:"above/below"} : inserts the image which file is specified (through a path relative to the eclipse project where the generation model is placed). The insertion will have the specified width, height and a legend will be inserted at the specified position if one is specified (default is below).

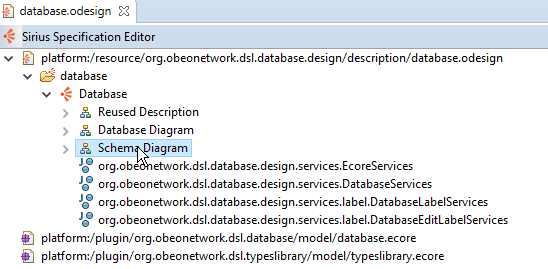
## M2Doc dynamic generation tags

M2Doc provides a way to extend its information retrieval for a tag by registering a provider by an extension point. Each provider uses its own options in addition of the one generically provided par M2Doc core. Many providers can be associated to a same kind of tag and any one of those can be choose to retrieve information in a tag.

To specify what provider to use with a tag, an option specified by M2Doc core is provided. Currently, M2Doc uses one dynamic tag "diagram" that don't have a default handling. The option for this tag used to specify the provider is "diagramProvider".   
A provider must be provided by the extension point so it can be used. M2Doc core comes with two providers handling the diagram tag in a different way:

* The diagram tag has the following format: { m:diagram width:"image insertion width" heigth:"image insertion heigth" legend:"image legend" legendPos:"above/below" providerOptionKey1:"providerOptionValue1" providerOptionKey2:"providerOptionValue2" etc…}: This tag allows insertion of images retrieved from Sirius diagrams. Width, height, legend and legend position are options handled by M2Doc. However, the image retrieval must be ensured by using provider options. Currently, M2Doc provides two ways of using this tag to get an image from a Sirius diagram with a different number of options :
  + By Sirius representation title with the provider "org.obeonetwork.m2doc.sirius.SiriusDiagramByTitleProvider": in this case, the tag would have the following format { m:diagram diagramProvider:"org.obeonetwork.m2doc.sirius.SiriusDiagramByTitleProvider" width:"200" height:"200" legend:"plan de forme du dingy herbulot" legendPos:"below" title:"'RF Schema'"}. The option used by the provider is "title". The title must correspond to an existing Sirius representation name. For example, in the previous tag, we use the title 'RF Schema' that we can see in the model of my generation project in the next picture:   
      
    The "title" option is an AQL expression.   
    This method is the easiest one to use. But it cannot be use if the same title is used for more than one representation. In this case, the provider retrieve the first representation found with the given title. If this is a problem, then you will have to use the other method provided by M2Doc.
  + By Sirius diagram description name and a semantic model objet ("org.obeonetwork.m2doc.sirius.SiriusDiagramByRepresentationAndEObjectProvider"): in this case, the tag would have the following format { m:diagram diagramProvider:"org.obeonetwork.m2doc.sirius.SiriusDiagramByRepresentationAndEObjectProvider" width:"200" height:"200" legend:"plan de forme du dingy herbulot" legendPos:"below" rootObject:"db.schemas->first()" diagramDescriptionName:"Schema Diagram"}.   
    Provider's options are "rootObject" and "diagramDescriptionName".
    - rootObject is an AQL expression allowing to select the semantic objet that must be the root object of Sirius representations were to look for the one with the title provided by the other option.
    - diagramDescriptionName is a String corresponding to the diagram description name that the representation from which we want an image must have.

To retrieve the same representation image than the previous example, we will have to use the corresponding diagram description that we find in the design file (Schema Diagram) :

  
To select the root object we use a variable "db" specified in the generation configuration model pointing at the semantic database root model element. Then we go through it until we find the root element from which we want an image of the representation using it as a root element.  
If many representations are found for the given diagram description name and root object, then they are all shown in their retrieval order one after the other.

No other providers are provided natively by M2Doc for diagram tag. But if you have install provider created by external sources, then use their documentation to know how to use it.

## Document generation and style

The style of the fragments of generated documents is determined by the style of the templates parts.

* Iteration : there’s no text style in the tag itself. The style of the body is reproduced as is. The style of the paragraph holding the opening ***m:for*** tag, however, is reproduced throughout the iterations.
* Conditionnal : there’s no text style in the tag itself. The style of the branch's bodies are reproduced as is.
* Image : there’s no text style associated to this style. The paragraph style is reproduced thought.
* Queries : the style of the first run of the expression is used to generate the text that replaces the query’s field. For instance, {m :table.comments} will produce comments in orange while {m :table.comments} will produce comments in black.

## How to create dynamic tables

There’s no specific tag required to create a dynamic table. Here is an example of a dynamic table description to provide a database table’s details:

|  |  |
| --- | --- |
| Name | Description |

{ m:for table | db.tables }

|  |  |
| --- | --- |
| { m:**table.name** } | { m:table.comments } |

{ m:endfor }

***Table { SEQ Tableau \\* ARABIC } : tables description***

It is sufficient to enclose the dynamic part in a generation tag. Here, we have a simple iteration tag. We could have a combination of iteration and conditionnal tags or whatever other combinations is necessary.

Note : invisible characters might sneak in between the two table fragments (the header and the iterated body). Toggle on the mode where these characters are shown and remove them, if there are any, after the for tag.

## How to create dynamic lists

Creation of dynamic lists is quite similar to dynamic tables : there’s no specific tag required. As bulleted or numbered lists are style attributes the style is carried from the template to the generated doc as is :

Tables for database { m:db.name }

{ m:for table | db.tables }

* { m:table.name } :
  + { m:for col | table.columns }{ m:col.name } : { m:col.comments }
  + { m:endfor }
* { m:endfor }

**Note** : for the bullets to be correctly spaced and so that there’s no spurious carriage return introduced, the endfor tags must be on bullets at the same level as the corresponding tag (as above).

## How to create dynamic hyperlinks

Hyperlinks are WORD fields looking like the following example {HYPERLINK "http://www.obeo.fr"}.

If you want to dynamically retrieve the link with an AQL expression, it is possible. Just adds the request in a WORD field in the double quote like that:

{HYPERLINK "m:'' http://www.obeo.fr "}

**Warning** : This feature is currently an experimental one. In some context it may be not working.

## Headings and tables of contents

Headings are just treated like tables and lists are : the style makes it all.

Here’s how we would create headings that corresponds to a database tables:

## { m:for table |db.tables } Table { m:table.name }

**Description :** { m:table.comments }

### { m:for col |table.columns }Column { m:col.name }

Description : { m:col.comments }

{ m:endfor }

{ m:endfor }

Insertion of table of contents has no interactions with the templating. The table of content is just a field which is processed by Word which collects all the headings. The only requirement is to make an update of the field right after document generation.

## Miscelaneous points

* Adding dynamic content in headers and footers is simply done by inserting generation tags in the header and footer.
* Dynamic content in text area isn’t supported yet. Any generation tag that appears in a text area will just be reproduced as is and won’t be processed as a generation tag.

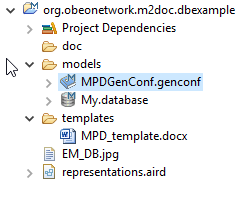
# Documentation generation

As of now, custom properties are not implemented and a generation configuration model must be used to generate a document from a template.

Document generation is done in three steps :

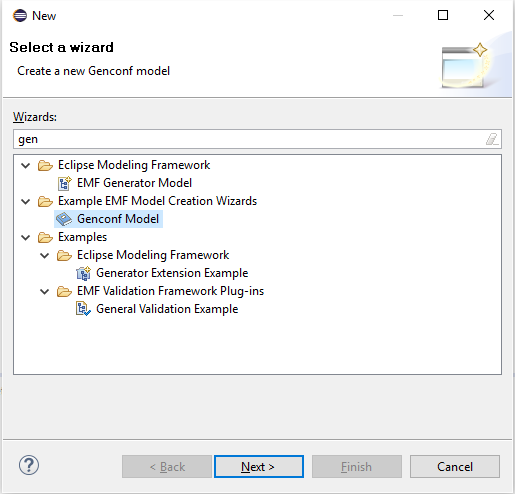
* Template authoring (in a modeling project)
* Generation configuration with the creation of a configuration model which binds values to variables
* Generation it self

We provide a step by step tutorial that explains how to generate a document from a database model. To start with, M2Doc and the database DSL must be installed in your bundle. Then, import the example modeling project (org.obeonetwork.m2doc.dbexample) in your workspace. (File>Import>General>Existing Project into Workspace). Open the project. You should have this folder layout :

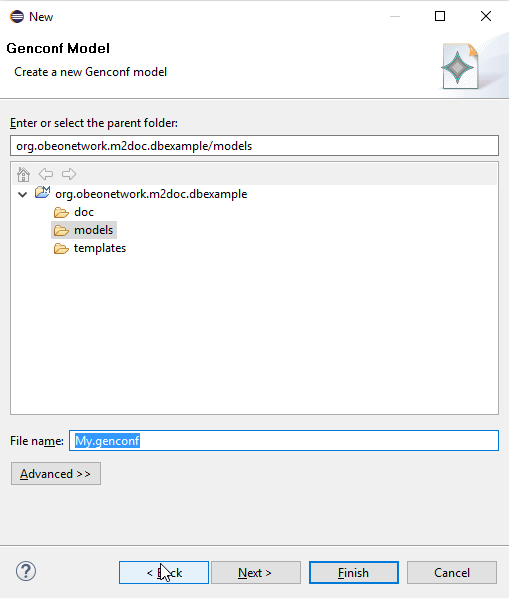


We will keep the existing genconf model and create another one to document the procedure..

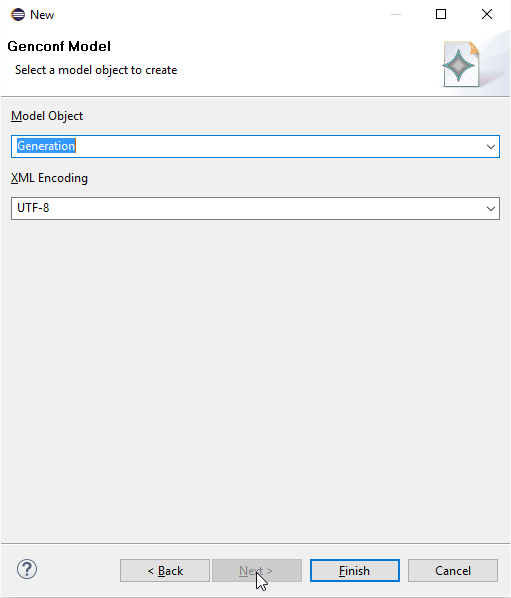
First, create the model : Open the generation configuration model wizard :



Select the Genconf Model wizard and fill the fields as follows :



Select to create a Generation model object :

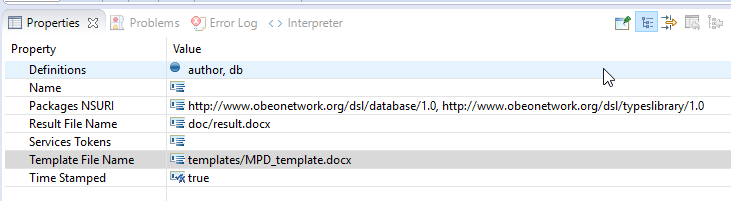


Now, press the ‘Finish’ button and open the newly created model:

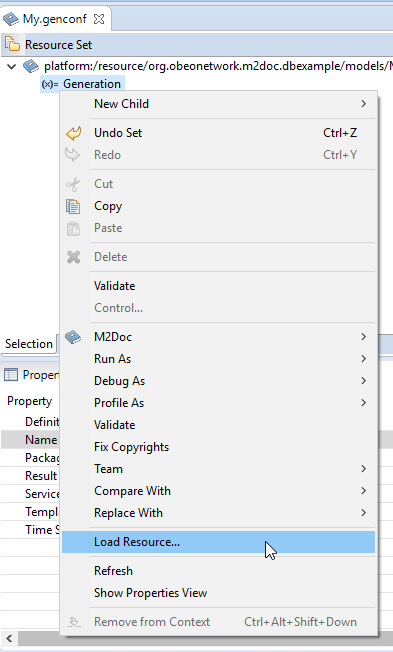
You will have to set those two URIs in the Package NSURI attribute:

* http://www.obeonetwork.org/dsl/database/1.0
  + 0

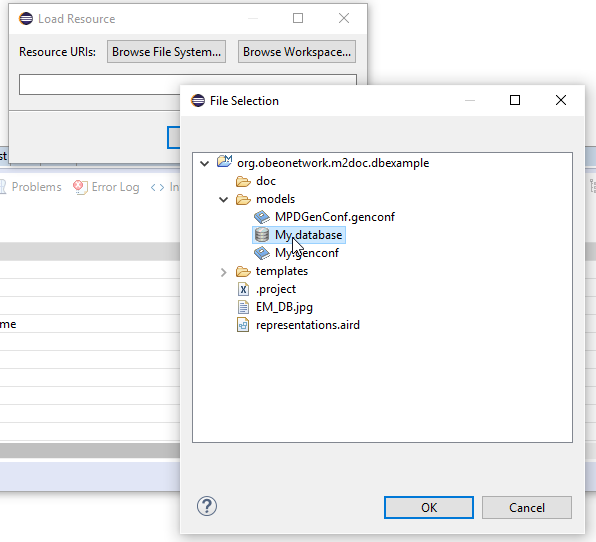
The resulting property sheet should look like this:



Now you have to load the main database resource :

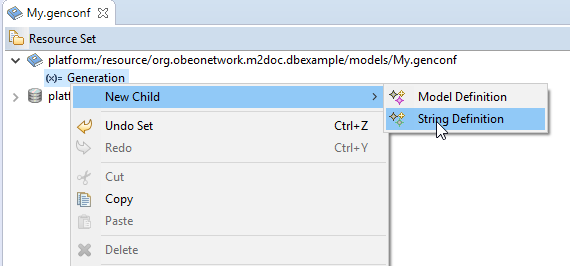


Select the database model that is in the model directory :

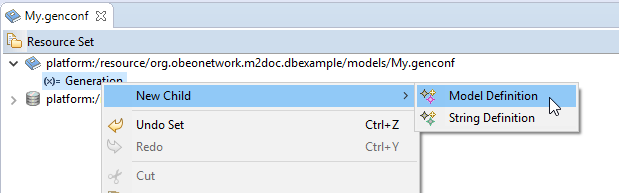


Now that the database resource is loaded, we will be able to use model element in it as variable’s values.

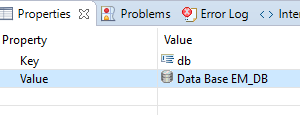
We first create a string variable :



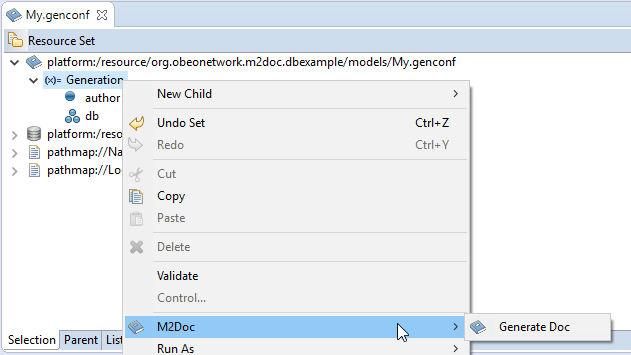
And a model variable :



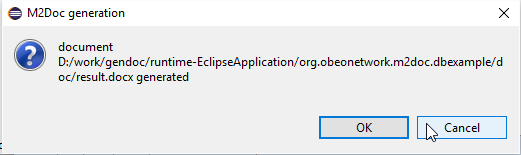
Set the values of the variables in the property view. The model variable must be defined like follows :



Now, you just have to invoke the generation by right clicking on the generation model element :



A message indicates everything went well :



Refresh the the doc directory. The result should be in it.