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 generation tags

M2Doc has 4 generation tags right now. These are described below

* : the directive is replaced with the string representation of the query’s evaluation result
* iterated body  : 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.
* true branch … ……  : conditinonal 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).
* : 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 an height and a legend will be inserted at the specified position if one is specified (default is below).

## 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 branche’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 |

|  |  |
| --- | --- |
|  |  |

***Table 1 : 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

* :
  + **Erreur ! Signet non défini.**  :

**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).

## 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:

## Table

**Description :**

### Column

Description :

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.

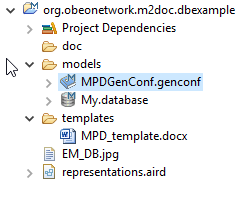
# Documentation generation

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 :

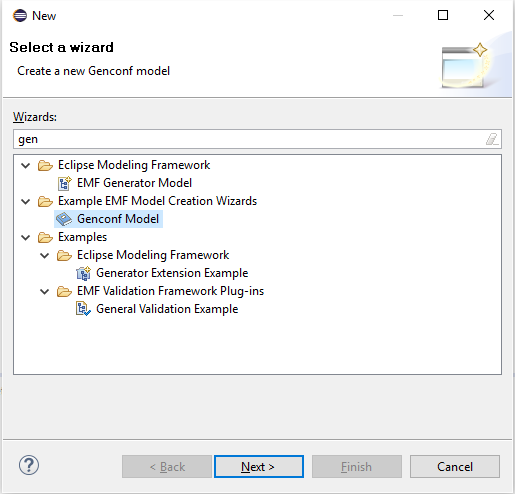
* Template authoring (in a modeling project)
* Generation configuration with the creation of a configuration model which binds values to variables
* Generation it self

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 :

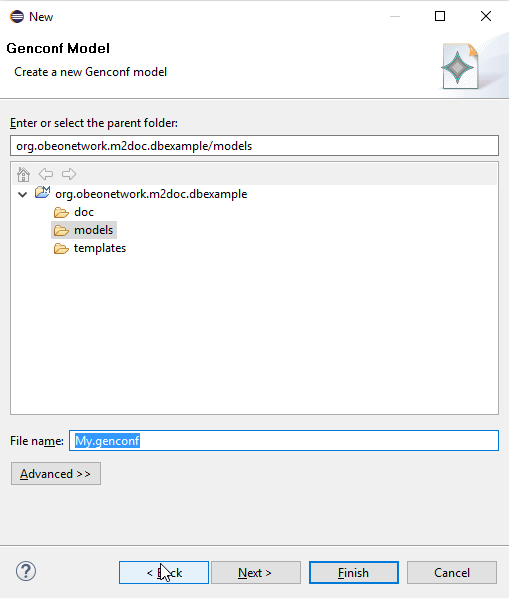


We will keep the existing genconf model and create another one to document the procedure..

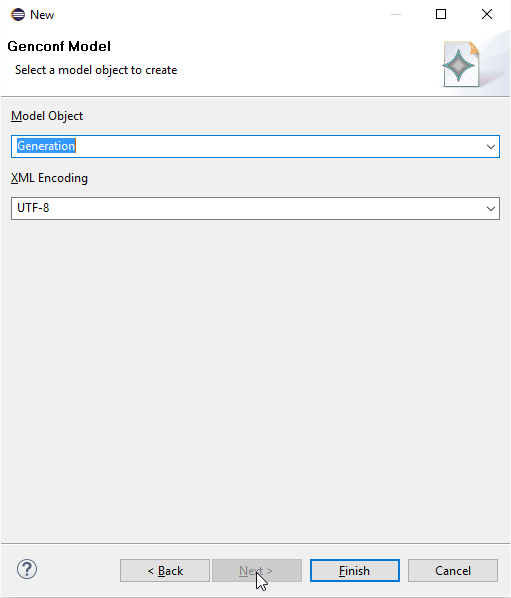
First, create the model : Open the generation configuration model wizard :



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



Select to create a Generation model object :

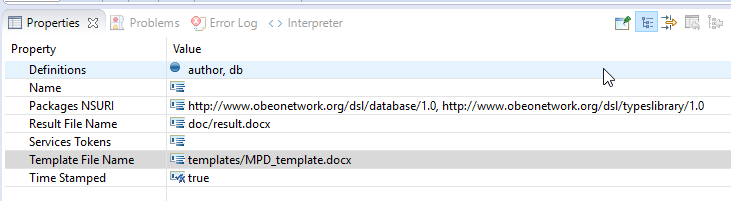


Now, press the ‘Finish’ button and open the newly created model :

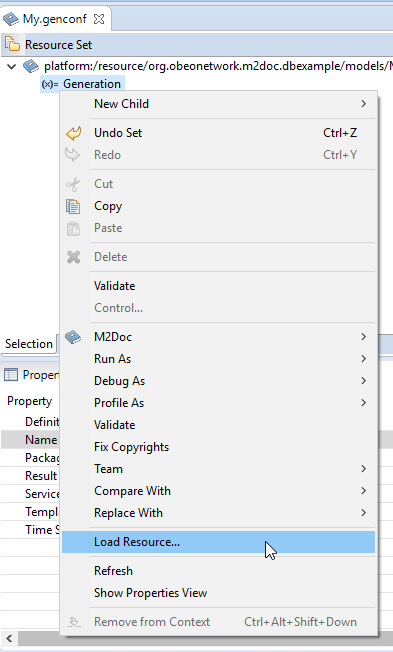
Youwill have to set those two URIs in the Package NSURI attribue :

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

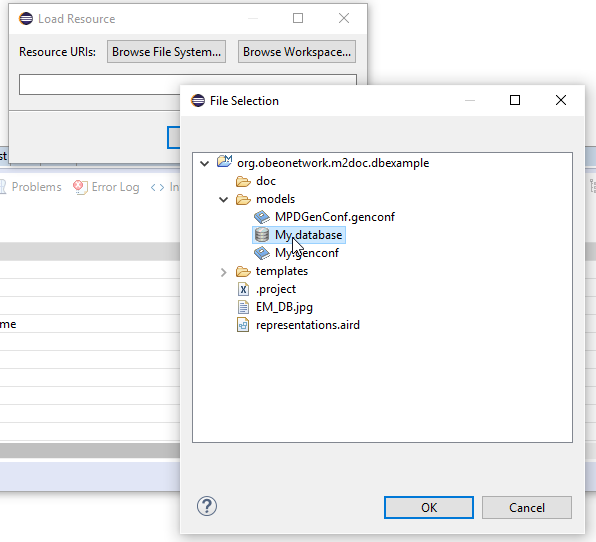
The resulting property sheet should look like this :



Now you have to load the main database resource :

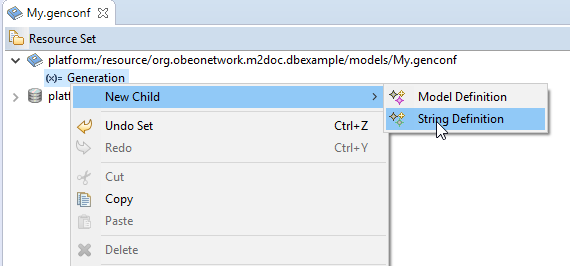


Select the database model that is in the model directory :

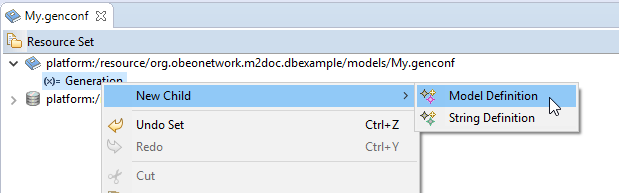


Now that the database resource is loaded, we will be able to use model element in it as variable’s values.

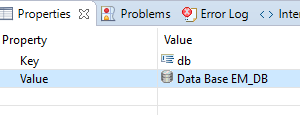
We first create a string variable :



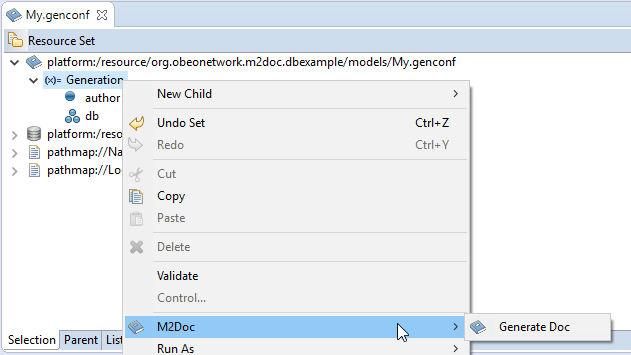
And a model variable :



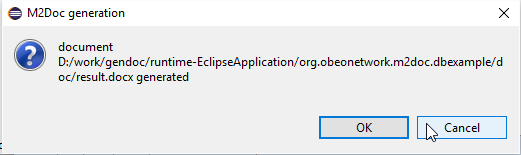
Set the values of the variables in the property view. The model variable must be defined like follows :



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.

# AQL Service registry

One of the ncie feature of the AQL engine is that the set of services is extensible at will. In the field of document generation, it means that we may provide, for a given application,a set of services that are taylored to ease the development of templates against a given meta model.

As the technology matures, we will also probably provide a set of services that are of general interest for model to document generation.

Yet, how do we register AQL services ? Here’s a step by step guide based on the example provided in the M2Doc repository.

## Writing AQL Services

AQL Services are provided through simple Java classes with a no-argument constructor. Services are the method of the class. The values of the method parameters are provided through the arguments of the service call in the AQL queries (for that matter, the target of the service call is considered as the first argument of the service call). For instance, the next AQL expression

db.allTables()

makes a call to the corresponding method in the Database service’s class :

public List<Table> allTables(DataBase db)

In that example, there’s a single parameter which is filled with the service’s call target.

Nothing prevents to pass on other arguments.

The service class looks like this :

public class DatabaseServices {

/\*\*

\* No arg constructor required by the AQL runtime.

\*/

public DatabaseServices() {

}

/\*\*

\* Returns the content of a table cell that is checked when the column is in

\* a foreign key.

\*

\* @param col the column.

\* @return the character ‘X’ when the column is a foreign key.

\*/

public String checkForeignKey(Column col) {

return col.isInForeignKey() ? "X" : "";

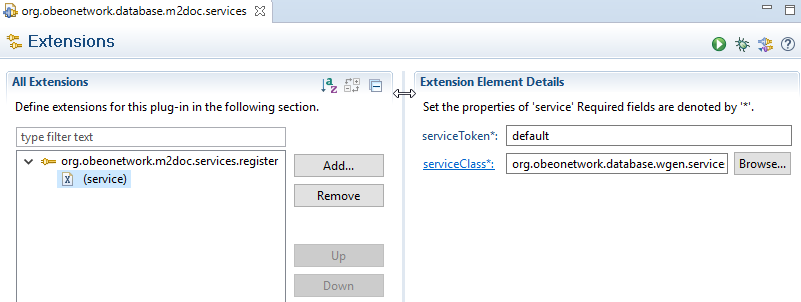
}

}

We only left the first method to illustrate the way of writing services.

## AQL Service registration

For the M2Doc engine to be able to call services from AQL queries, it is necessary that those are registered in some way. To allow this, the M2Doc runtime provides an extension point. A plug-in that contribute services to the M2Doc runtime must then declare an extension to do so :



The extension has two attributes :

* The service class itsefl
* A service token. This attribute is intended to isolate services dedicated to a given M2Doc application in a bundle. This feature is not complete yet and any registration must be made through the default token. When the feature is complete, it will be required to provide the set of service’s token required through the template’s custom properties. This will allow to have several services with the same name for distinct applications without interference.

Alternatively, a given M2Doc integration is free to provide a set of standard services that will be registered in the integration code. In such a scenario, documentation generation will probably be made through integration’s specific actions that aren’t documented here.