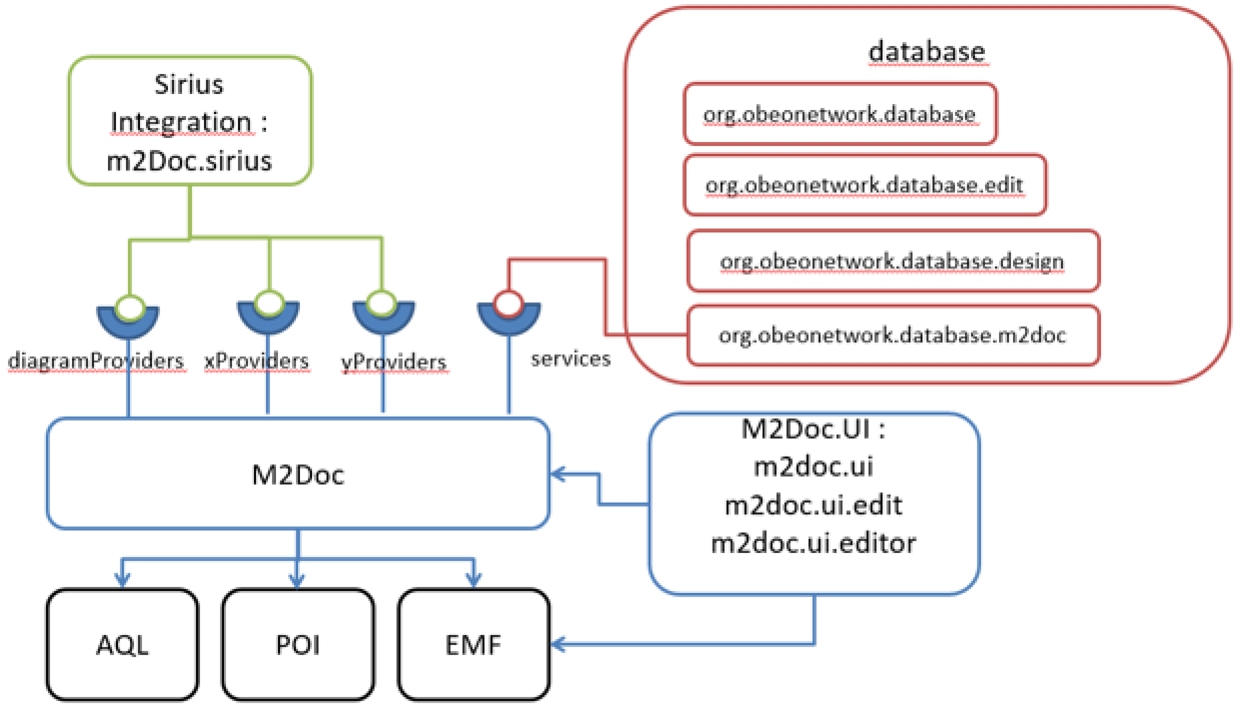
M2Doc Developer Guide

M2Doc has been designed to be an extensible template language.
It means that some tags from template language used to retrieve information from the generation environment can be used in different manner with different options. It is up to anybody to provide a way to handle a tag thanks to an extension point.

You also can provides some services to extend the AQL language.

M2Doc Architecture

To understand better how M2Doc is working and how you will have to contribute extensions, let's have a look at the architecture:


database
org.obeonetwork.database
org.obeonetwork.database.edit
org.obeonetwork.database.design
org.obeonetwork.database.m2doc
Sirius Integration : m2Doc.sirius
diagramProviders
xProviders
yProviders
services
M2Doc
M2Doc.UI : m2doc.ui m2doc.ui.edit m2doc.ui.editor
AQL
POI
EMF

The M2Doc base depends on AQL to be able to retrieve information from models or even to get a value for tag options.

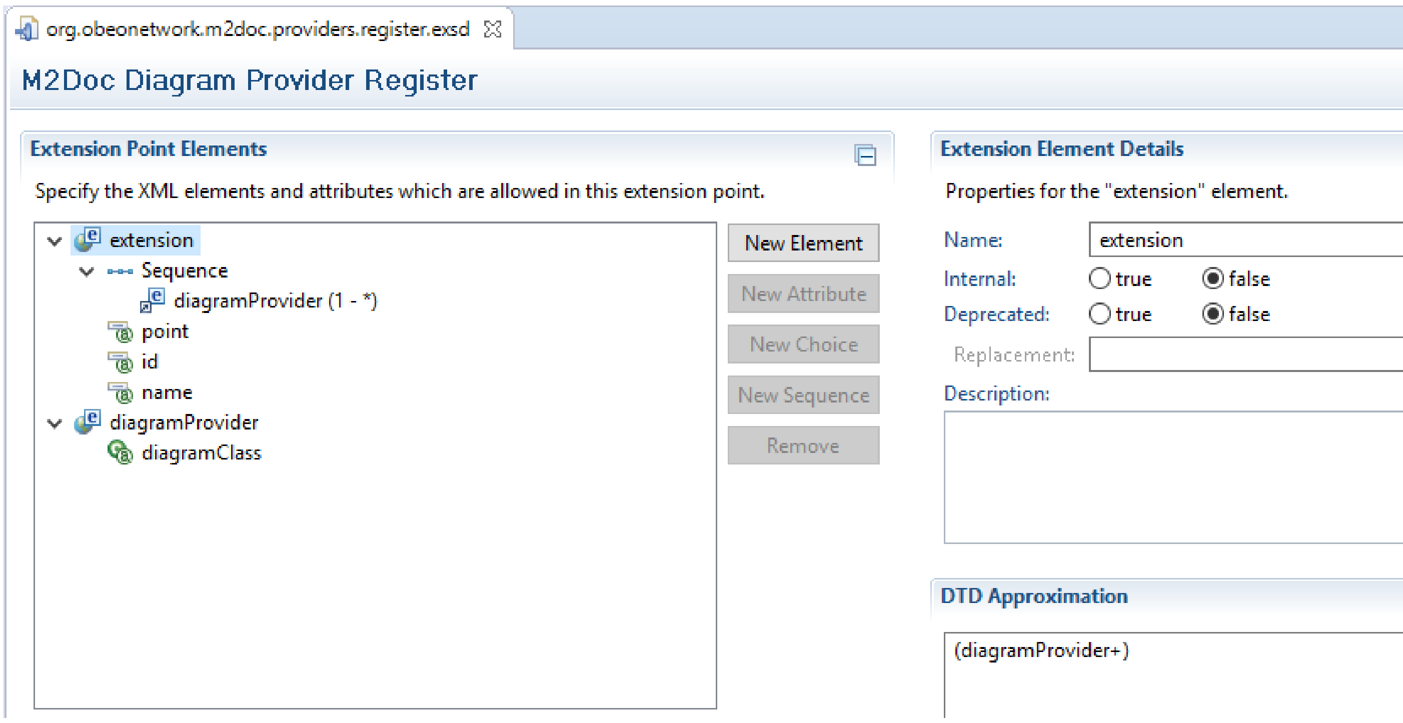
It contains a unique extension point that allows to contribute diagram providers to enhanced diagram tag capabilities.

Currently, this extension point allows to provide only diagram provider. But in the future any extensible tag with the need of a different provider kind than the diagram one could be declared by this extension point (x and yProvider).

Providing tag handling

Thanks to the extension point, you can specify how some tag considered as extensible by M2Doc will be handle to generate content and what options they will contain to achieve the handling.

To extend the way the tag is used, we provide with M2Doc an extension point (org.obeonetwork.m2doc.providers.register) you will have to use:



org.obeonetwork.m2doc.providers.register.exsd

M2Doc Diagram Provider Register

Extension Point Elements

Specify the XML elements and attributes which are allowed in this extension point.

```
extension
    Sequence
        diagramProvider (1 - *)
    point
    id
    name
diagramProvider
    diagramClass
```

Extension Element Details

Properties for the "extension" element.

Name:	extension	
Internal:	true	[✓ false]
Deprecated:	true	[✓ false]
Replacement:		
Description:		

DTD Approximation

(diagramProvider+)

You can specify your diagram providers with this extension point. For the moment, only diagram providers can be registered.
If new extensible tags are created in the future with the need of a different kind of provider than the diagram one, then the extension point will be enriched.

Extending the "diagram" tag

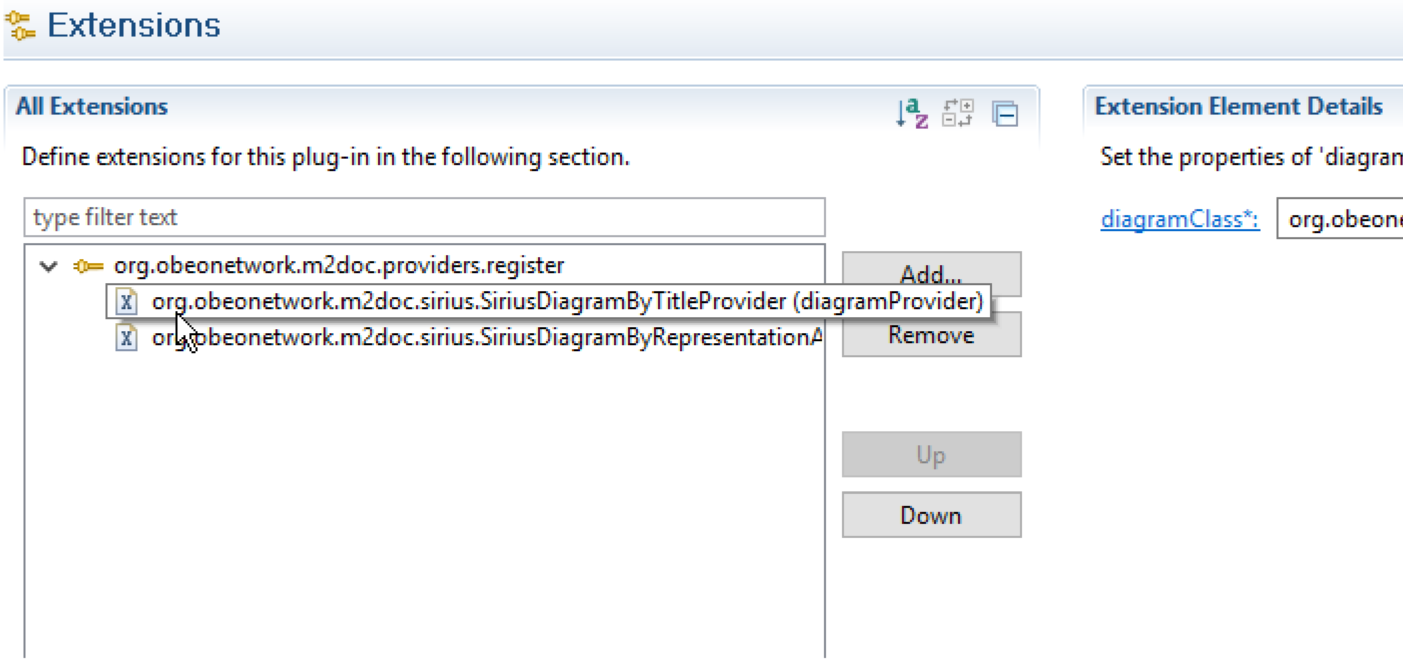
The meaning of the diagram tag is to insert an image or images one after the other from Sirius diagrams in the produced document with the use of a provider.

The extensible part of this tag is the way that images to insert are created and provided.

For example let's consider the M2Doc provider "`SiriusDiagramByTitleProvider`". This provider handles tag with the following format (activate fields if you don't see it):

{ m:diagram title:"Schema Diagram" provider:"org.obeonetwork.m2doc.sirius.SiriusDiagramByTitleProvider" }

To compute images, the provider uses the title option.
To add a provider for diagram tag, just register your provider to the extension point like in the following picture:


Extensions

All Extensions

Define extensions for this plug-in in the following section.

type filter text

org.obeonetwork.m2doc.providers.register
org.obeonetwork.m2doc.sirius.SiriusDiagramByTitleProvider (diagramProvider)
org.obeonetwork.m2doc.sirius.SiriusDiagramByRepresentationA

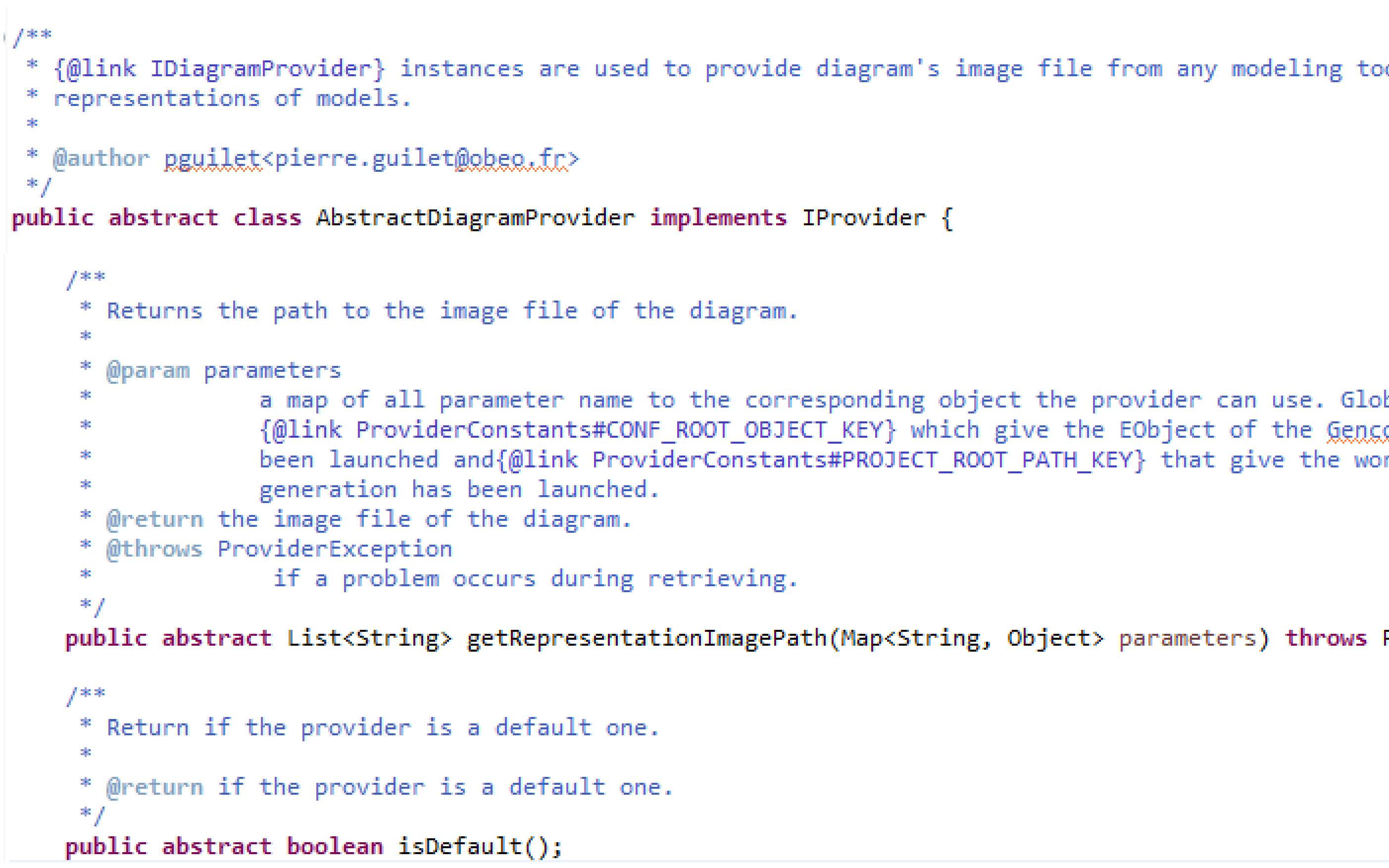
Extension Element Details

Set the properties of 'diagran

diagramClass*: org.obeon

```
   <extension
         point="org.obeonetwork.m2doc.providers.register">
      <diagramProvider

diagramClass="org.obeonetwork.m2doc.sirius.providers.SiriusDiagramByTitleProvider"
>
      </diagramProvider>
   </extension>
```

Your provider to be fully functional must be an implementation of org.obeonetwork.m2doc.provider.DiagramProvider class:



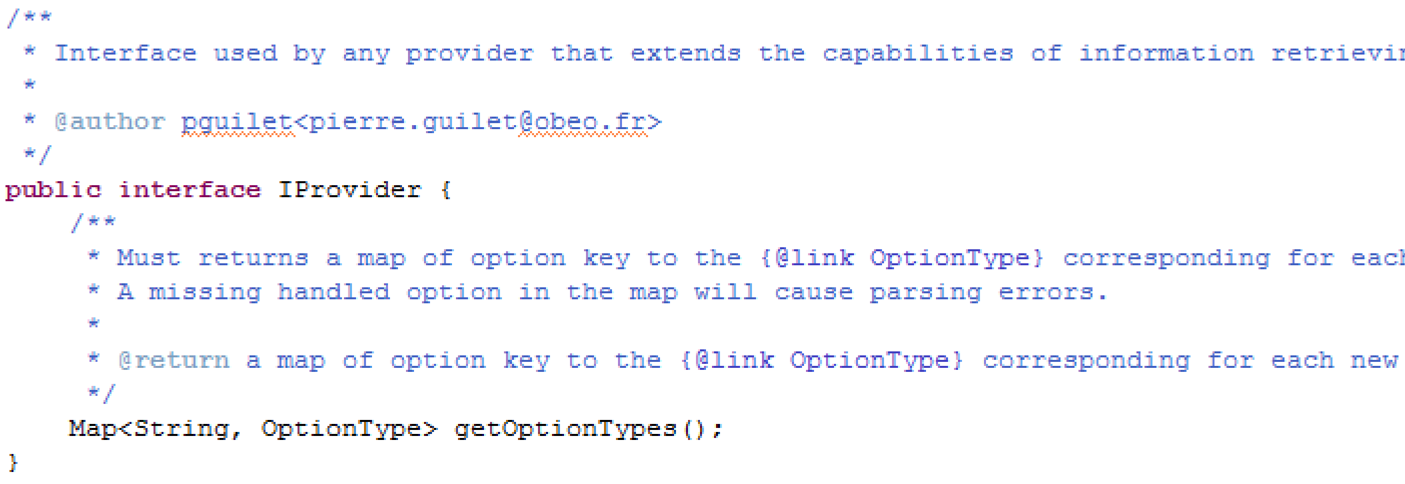
```
/**
 * {@link IDiagramProvider} instances are used to provide diagram's image file from any modeling too
 * representations of models.
 *
 * @author pguilet<pierre.guilet@obeo.fr>
 */
public abstract class AbstractDiagramProvider implements IProvider {

    /**
     * Returns the path to the image file of the diagram.
     *
     * @param parameters
     *            a map of all parameter name to the corresponding object the provider can use. Glob
     *            {@link ProviderConstants#CONF_ROOT_OBJECT_KEY} which give the EObject of the Genco
     *            been launched and{@link ProviderConstants#PROJECT_ROOT_PATH_KEY} that give the wor
     *            generation has been launched.
     * @return the image file of the diagram.
     * @throws ProviderException
     *             if a problem occurs during retrieving.
     */
    public abstract List<String> getRepresentationImagePath(Map<String, Object> parameters) throws P

    /**
     * Return if the provider is a default one.
     *
     * @return if the provider is a default one.
     */
    public abstract boolean isDefault();
```

This class contains two methods

- "getRepresentationImagePath" that takes as entry a map of options parameters and returns the list of all images path to insert as replacement of the tag in the produced Word document.
  The image behind the path must exists on the file system when returned or no image will be inserted.

The options map will contains all options evaluated that were present in the diagram tag except the one handled by M2Doc that are "width", "height", "legend", "legendPos" and "provider".

Options will be evaluated regarding their specified kind that must be provided with the method coming from the interface IProvider of all provider diagram or not:



```
/**
 * Interface used by any provider that extends the capabilities of information retrievi
 *
 * @author pguilet<pierre.guilet@obeo.fr>
 */
public interface IProvider {
    /**
     * Must returns a map of option key to the {@link OptionType} corresponding for each
     * A missing handled option in the map will cause parsing errors.
     *
     * @return a map of option key to the {@link OptionType} corresponding for each new
     */
    Map<String, OptionType> getOptionTypes();
}
```

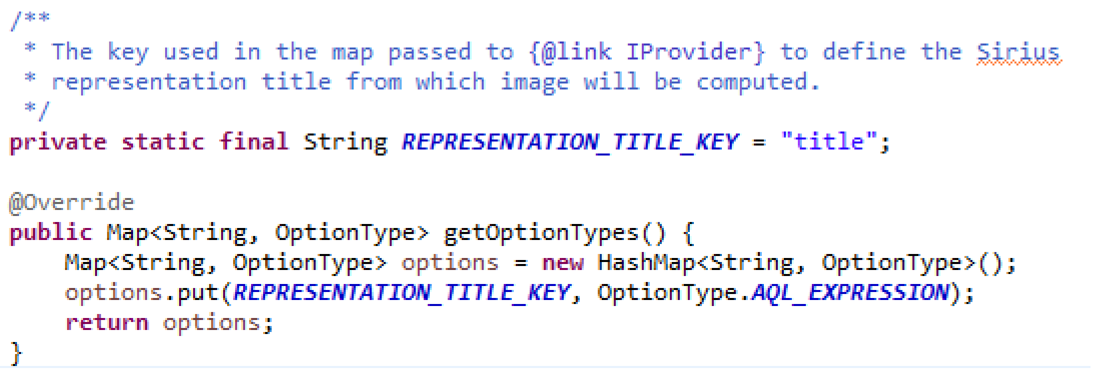
This method allows to specified how each option will be parsed and provided by associating the option’s key and an evaluation kind.

For the moment, two kinds of evaluation can be done by M2Doc with option:

- The STRING one that parse the option as a string and give it to the provider as the same exact string.
- The AQL_EXPRESSION one that parse an option as a string and interpret it as an AQL expression and give to the provider the evaluated expression result that can be any JAVA object.

A JAVA enum contains all the kind of evaluation that M2Doc provides (org.obeonetwork.m2doc.provider.OptionType).

In our example, “title” is an AQL expression. The method look like this:



```
/**
 * The key used in the map passed to {@link IProvider} to define the Sirius
 * representation title from which image will be computed.
 */
private static final String REPRESENTATION_TITLE_KEY = "title";

@Override
public Map<String, OptionType> getOptionTypes() {
    Map<String, OptionType> options = new HashMap<String, OptionType>();
    options.put(REPRESENTATION_TITLE_KEY, OptionType.AQL_EXPRESSION);
    return options;
}
```

If no option type is provided for an option, then the default STRING one will be use.

- “isDefault”: first, let’s explain how a provider is chosen.
  If we have a provider tag, we try to get this provider. If it is not found, we get all the registered providers via the m2doc extension point, we iterate on them and the first provider witch have its mandatory option tags matching is chosen. If a provider is defined by default, it will be added at the beginning of this list, so it will have priority on the others.

The provider tag is optional, if your provider is by default, it will be tested before the other ones. That allows to write <m:diagram title=”myDiagram”> instead of <m:diagram title=”myDiagram” provider:"org.obeonetwork.m2doc.sirius.SiriusDiagramByTitleProvider">

Error handling

If some parameters are missing or are incorrect, you can warn M2Doc by throwing a ProviderException with the wanted message.
When M2Doc catch this exception it will insert your message at the tag position.

Extending other tags

Currently, only the tag "diagram" can be extend. But any new tag that will be added and which will be considered as extensible could have providers attached to it.

Providing AQL services by services registry

One of the nice 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 tailored 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 :

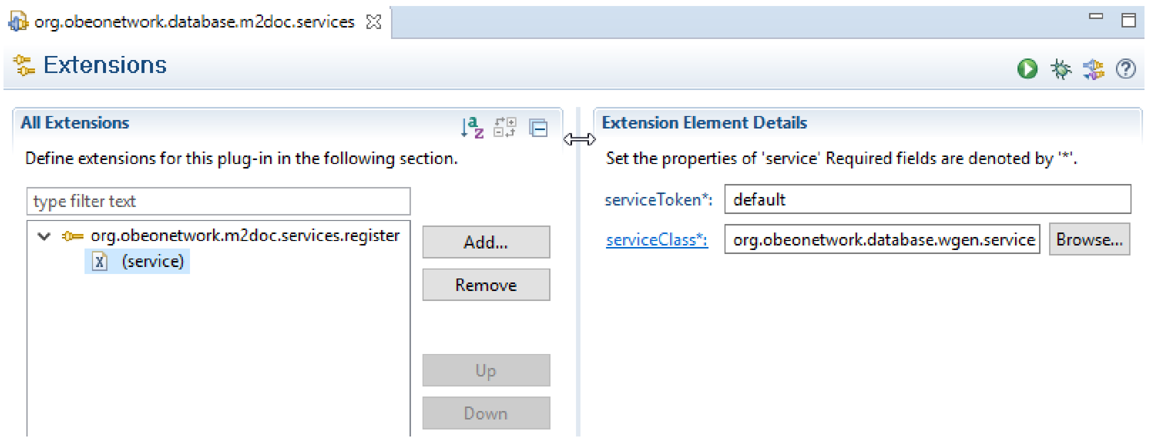
```
public class DatabaseServices {

        /**
```

```
         * 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 :


org.obeonetwork.database.m2doc.services
Extensions
All Extensions
Define extensions for this plug-in in the following section.
type filter text
org.obeonetwork.m2doc.services.register
(service)
Add...
Remove
Up
Down
Extension Element Details
Set the properties of 'service' Required fields are denoted by '*'.
Browse...

serviceToken*: default
serviceClass*: org.obeonetwork.database.wgen.service

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.