SOGETI

Getting started with the XDFL Engine

Benjamin GARRIGUES

1. Introduction

What is the XDFL engine ?
XDFLengine is a full-XML application server and batch processor: its XML-derived language (called XDFL for eXtensible Data Flow Language) uses generic blocks to describe processing flows.
XDFLengine is an open source project, released under GPL license.
The project is hosted by Sourceforge.net at http://sourceforge.net/projects/xdflengine/

Why this document ?
This document will help you taking the first steps with the XDFL engine. Since this technology is entirely new and is completely XML based (both for data and code), it may take some effort for a beginner to get started.
At the end of this document, reader will have quite a good understanding of the XDFL engine and should feel comfortable enough to start building her own XDFL project.
This file is not a complete reference to the XDFLEngine tags, neither a full programming guide. It shows the main features of the XDFLEngine and provides the user with examples of main usage.
Emphasis is put in explaining *why* a feature has been created and what issues it deals with rather than *how* using it. It serves as a complement to the XDFL Programming Guide found at http://xdflengine.sourceforge.net .

XDFL and XML
XDFL has been implemented with XML in mind *from day one*. The best proof is that XDFL not only uses XML as its main data format for inner communications, but XDFL actually *is* derived from XML. In order to understand how important XML is to XDFL, imagine a language like C or Java that uses XML as its only type for variables and structures, and the XML syntax for describing functions, classes and all parts of the code.
Be reassured though, because XDFL is more like a scripting language. It is very well known that XML syntax is not really convenient for typing thousands lines of code. And that's great, because it's exactly what XDFL is all about : writing as little code as possible.

XDFL main applications
XDFL shines at working with huge XML data flows. Its pipelined XML processor can handle gigabytes of data consuming incredibly little amount of memory.
Extracting data from a Database, creating an XML tree from a relational model, and manipulating this tree before sending it back to a database is made in just a few lines of code.
Because XDFL always uses the same generic blocks for manipulating data, it is both stable and *extremely* efficient.

Extensibility
Using only generic XDFL blocks enables you to do an incredibly wide variety of tasks. Nevertheless, adding your own blocks is not only possible thanks to the engine's architecture, but also highly appreciated by its developers !
Because XDFL is open-source, you have full access to its source code and full permission to customize it the way *you* want.

Portability

XDFL engine has been ported to the most common OS found in the industry (and we don't mean Windows and Linux only). It also supports the main DBMs and web servers. You can find a complete support list at http://xdflengine.sourceforge.net .

Already used in the real world !
XDFL is already being used in many real world projects of big companies, some of them being critical. When we say you can do something with the XDFL engine quickly and in a very stable and efficient way, we don't mean theory, we mean experience.

2. Setting up your environment

❖ Required steps

- Download and expand the XDFL engine binaries :
  Binaries can be found at http://xdflengine.sourceforge.net/ in the Download section. Unless you want to code new blocks to extend the engine, you don't need to download the source code version.
- Find a good editor :
  Since XDFL application are made with XDFL scripts, you don't need a powerful IDE to start having fun. Any decent text editor (such as Textpad or UltraEdit) will do the trick. Highlighting syntax files can be downloaded for common editors at http://xdflengine.sourceforge.net in the download section.
- Test if everything is working :
  Launch a command line shell and go to the xdflbinaires.win32.1.0 (depending on the version the numbers may vary) directory. Type XDFLRun.exe . You should see something like that :

```
********************************************************************************
XDFLengine V1.0
        Copyright (C) 2002_2004 Guillaume Baurand

------------------------------------------------------------------------
Xerces_C++ 2_4_0
        Copyright (C) 1999_2004 The Apache Software Foundation

SpiderMonkey Mozilla Javascript C engine 1.5 RC6
        Copyright (C) 1998-2003 The Mozilla Organization

Xalan-C++ 1.7
        Copyright (C) 1999-2003 The Apache Software Foundation


------------------------------------------------------------------------
Usage :

XDFLRun <arg1>=<val1> <arg2>=<val2> ... <argn>=<valn>

Arguments:
        script          : XDFL script to execute. (required)
        init            : Init XDFL script to execute before processing main scr
ipt.
        input           : File to read as input data of script. If not provided,
 input data is read from stdin.
        output          : File to wite as output data of script. If not provided
, output data is written to stdout.
        log             : Log file name. If not provided, log is written to stdo
ut.
        encoding                : Character encoding used.
        other args provided as argname=argvalue are available wiwthin the script
 as ARG.argvalue values.
------------------------------------------------------------------------

********************************************************************************
OK.
```

XDFLRun.exe is the application that you will use throughout this guide for launching XDFL scripts. You may want to add the path to this application to your PATH environment variable.
As you may wonder what other files are used for, most Dlls in the directory are storing code for XDFL blocks while others (ending with isapi or fcgi) are used as web server extensions (more about that in another tutorial on the XDFL web programming framework).

- Configuring for DB access : for scripts involving Database access, you need a properly configured database client. XDFL doesn't require any special configuration step for accessing database. Only thing you need to know is the name of the database and a properly configured database client (not covered by this tutorial because not specific to XDFL).

That's it !

Note : For more convenience, we suppose that all scripts and files shown in this guide are created and stored in the same directory as the XDFLRun.exe and all the dlls. Traditionally this directory is called "bin".

Following are specific configuration steps given as additional information. You can skip those steps and go directly to the next section.

❖ Optional configuration steps

- **Web server configuration steps :**
  Depending on your web server, different steps may be required. A short example of configuration is given here for extending IIS web server to support xdfl scripts:

  -In the advanced settings : add ".xdfl" as a new extension ("application mapping"). Link to XDFLisapi.dll for keyword GET and POST.

  -Select your folder containing the xdfl scripts.

  -Add a valid init.xdfl and config.xdfl scripts to the folder containing the script.

  Note : for more information on how to use the xdflengine for building complete web applications along with information on the web framework, please reefer to http://xdflengine.souceforge.net . Building web applications is *not* covered by this document.

3. XDFL basic features

Note : This chapter is meant to be read from the start until the end, without skipping any paragraphs. Don't expect to fully understand an example if you haven't read previous paragraphs. If you are looking for a reference of XDFL tags, please look for it in the documentation section at http://xdflengine.sourceforge.net .

❖ XDFL syntax

1) Introduction

First of all, XDFL is derived from XML. Thus an XDFL script will be made with XML tags (a tag is simply a special word surrounded by "<" and ">"such as "<TAG>"). All rules that apply to an XML document apply to an XDFL document. You can browse XML documentations and tutorials to know exactly what this implies, but you will have a good idea of what correct documents are simply by browsing sample codes of this document.

2) Shortest XDFL script :

Here is the shortest XDFL script you can imagine :

Example 1 : Shortest XDFL script

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<XDFL xmlns:xdfl="xdfl">
       <xdfl:NULL/>
</XDFL>
```

The first line is required by the XML standard. It shows the version of XML used and the type of encoding used in the document.
The second line creates the "xdfl" namespace : from that point, every tag starting with xdfl: will be considered as a XDFL tag.
The third line is a xdfl tag called "NULL". This tag does nothing (we will see later what this tag still is used for). You will notice the "/" character before the end of the tag. In case you're not familiar with XML syntax, you should know that correct XML syntax tells you to close every tag you opened. Opening a tag means "<TAG>" and closing means "</TAG>". Correct writing of a tag would thus be "<TAG> </TAG>". Short-cut syntax for this is "<TAG/>".
Here we could have written "<xdfl:NULL> </xdfl:NULL>" but since we didn't want to write anything between the opening and the closing, we used the short syntax "<xdfl:NULL/>".
Last line, as you may already have guessed is the closing of the <XDFL xmlns:xdfl:"xdfl"> tag.
Here you may say "but why isn't the correct way of closing this tag done by writing </XDFL xmlns:xdfl:"xdfl"> ?". That's easy : the tag is XDFL and xmlns:xdfl:"xdfl" are *parameters* or *attributes* for the tag. If you're familiar with HTML, this should be no problem for you. If you are still confused, you may want to have a look at a full XML tutorial, or the introduction chapter of HTML tutorials.

Most important is this : your "real" XDFL code will be where the <xdfl:NULL/> tag stands, meaning your script will *always* start with those two lines and end with this last one, no matter what the XDFL script does.

You can test this script by calling the XDFLRun.exe application from the command line by typing :

```
XDFLRun.exe script="shortest.xdfl"
```

Where "shortest.xdfl" is the file containing Example 1 script.

What you should see as a result is something like this :

```
********************************************************************************
XDFLengine V1.0
        Copyright (C) 2002_2004 Guillaume Baurand
Xerces_C++ 2_4_0
        Copyright (C) 1999_2004 The Apache Software Foundation
SpiderMonkey Mozilla Javascript C engine 1.5 RC6
        Copyright (C) 1998-2003 The Mozilla Organization
Xalan-C++ 1.7
        Copyright (C) 1999-2003 The Apache Software Foundation

__| compiling |_____________________________________________________

        shortest.xdfl   :   open file.
        shortest.xdfl   :   read 93bytes.
        shortest.xdfl   :   close file.
          +--<DATA>
          +--<NULL>

_____________________________________________________________________

********************************************************************************
OK.
```

Notice that compilation process gives you indication on what the XDFLengine understands of your code and prints the tag structure of the script. This information can be quite useful for debugging.

3) Hello world

It is now time for us to write the famous "hello world" script.
First what we need is a tag that prints data on the standard output, and tell that tag to write "hello world". Well, fortunately this tag already exists, and it is called OUTPUT (original, isn't it ?).
Now how do we say what characters we want to write ?
Simply open the tag, write "hello world" and close the tag :

Example 2 : Hello world script

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<XDFL xmlns:xdfl="xdfl">
        <xdfl:OUTPUT>Hello World </xdfl:OUTPUT>
</XDFL>
```

Result is the same as for the previous script, except that you should see Hello World written right before the asterisks line.

You have just being introduced to the way XDFL uses XML to write code. It's quite easy actually : declare a namespace for XDFL "special tags" (called "active tags" or rather "active nodes") and give those tags input data by writing between the opening and the closing.

To understand a little bit better the difference between active nodes and passive nodes here is another example of Hello World, that mixes regular XML with XDFL language :

Example 3 : XML Hello world script

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<XDFL xmlns:xdfl="xdfl">
        <xdfl:OUTPUT><message>Hello World </message></xdfl:OUTPUT>
</XDFL>
```

This outputs the characters provided as input, thus :

```
<message>Hello World </message>
```

So where is the XML you may wonder ? Well, try this :

Example 4 : Bad XML Hello world script

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<XDFL xmlns:xdfl="xdfl">
```

```
        <xdfl:OUTPUT><message>Hello World </messy></xdfl:OUTPUT>
</XDFL>
```

You will get the following error :

```
ERROR #5 in XDFLSaxErrorReporter : Fatal XML Error at (3,43):
Expected end of tag 'message'
```

Expected end of tag 'message'. Indeed, the <message> tag on line 3 wasn't closed when the </messy> tag closed without being previously opened. This is bad XML syntax, and this is an error because XDFL tags take XML data as input.

In order to make the difference between XDFL tags and regular XML tags we will speak respectively of "active nodes" (meaning, understood by the XDFL engine and actually *doing* something) and "passive nodes" (regular XML nodes). You can consider active nodes as being *code* and passive nodes as *data* for this code (though it may be a little bit more complicated as we will see in the advanced features chapter, when we will use the PASSIVE XDFL tag to prevent active tags from doing anything).

Note (1) : Don't worry if things aren't clear for you yet. It takes some time to really get comfortable with mixing XML and XDFL in the same script and understand what you are doing, but you'll get used to it.

Note (2) : If you really wanted to write the character string "<message>Hello World</messy>" then you should have written it the way the XML standard wants you to do it, using the <![CDATA[…]]> special tags, that is :

Example 5 : Bad XML Hello world script (corrected with <![CDATA[...]]>)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<XDFL xmlns:xdfl="xdfl">
        <xdfl:OUTPUT><![CDATA[<message>Hello World </messy>]]></xdfl:OUTPUT>
</XDFL>
```

Special characters included between "<![CDATA[" and "]]>" will not be affected by XML syntax check and thus will not throw any syntax error.
Still, output for this script may not be what you expect : "<" special character will be written in its XML code "<" and ">" in ">". Thus output for this script is :

```
&lt;message&gt;Hello World &lt;/messy&gt;
```

This conversion of special characters into their standard representation can be bypassed using values (presented below).

❖ Files

The second step after writing to the output is working with files. Although XDFL makes it a pretty straightforward step. We simply will be using two tags called "FS_SET" and "FS_GET" in order to respectively write and read to a file.
Here is a short example of writing a character string to a file, reading it, and printing the output to the standard output :

Example 6 : Working with files

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<XDFL xmlns:xdfl="xdfl">
        <xdfl:FS_SET target="C:/test.txt" action="append">Hello World</xdfl:FS_SET>
        <xdfl:OUTPUT> Result is :
                <xdfl:FS_GET target="C:/test.txt"/>
</xdfl:OUTPUT>
</XDFL>
```

You may have noticed the "action" attribute to the FS_SET tag. "action" attribute may have 3 different values : "flush" in order to flush the file (default value), "delete" in order to delete the file or "append" to append input data to the file.

You may see debug information between “Result is” and “Hello world” and around. That shows you how the script is executed :

```
FS_SET is executed ;
FS_SET writes Hello World to the file test.txt ;
FS_SET stops being executed ;
OUTPUT tag is executed ;
OUTPUT tag encounters a characters string (valid XML text information, passive) and prints it;
OUTPUT tag encounters a tag belonging to the xdfl namespace ;
FS_GET is executed;
FS_GET read file “test.txt”;
FS_GET returns content of the file “test.txt” as a valid XML stream;
OUTPUT receives XML stream from FS_GET node and prints it;
FS_GET stops;
```

You should retrieve all those steps in that order when reading the output (some may not log anything, though). I will let you retrieve those steps in the output as an exercise.

This example was a huge step towards building XDFL scripts for two reasons :

1. It shows how tags can be written one *after* another to produce separate processing streams flowing data and being processed one after another (stream starting with FS_SET and stream starting with OUTPUT)
2. It shows how tags can be written one *inside* another to produce a single data stream made of chained tags (OUTPUT and FS_GET, with data flowing from FS_GET to OUTPUT).

Those are the only two ways of chaining XDFL tags to produce data streams.

Notice that the result of tags combination will always be a tree structure (and it better be, since XDFL is XML, and XML-valid documents are trees), with the <XDFL/> tag as the root of the tree.
Thus, not only is the code organized as a tree, *but so are the data streams with data flowing from leafs to root*.
This will have to be taken into account when dealing with scripts design. Especially because a tree structure is not as convenient to use as, for example, a graph structure.

❖ Buffers and Values

Values and buffers provide you with a way to store a data stream at some point in the script and re-use this stream elsewhere. This is a very convenient way to bypass issues raised by the tree structure of scripts. Buffers are accessed using BUF_SET for writing and BUF_GET for reading to and from a buffer. Values are a little bit different, so let’s start with buffers.

1) Buffers

First let’s see one example of how buffers can make things easier :

Example 7 : Buffers

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<XDFL xmlns:xdfl="xdfl">

	<xdfl:FS_SET target="C:/Header.txt">A very nice header</xdfl:FS_SET>

	<xdfl:BUF_SET name="header" create="1"><xdfl:FS_GET
target="C:/Header.txt"/></xdfl:BUF_SET >

	<xdfl:FS_SET target="C:/mail1.txt">
		<xdfl:BUF_GET name="header"/>
		Content for mail 1
	</xdfl:FS_SET >
	<xdfl:FS_SET target="C:/mail2.txt"><xdfl:BUF_GET name="header"/>
		Content for mail 2
	</xdfl:FS_SET >
	<xdfl:FS_SET target="C:/mail3.txt"><xdfl:BUF_GET name="header"/>
Content for mail 3</xdfl:FS_SET >

</XDFL>
```

Note : indentation and space characters within tags DO matter as they will be discussed in this paragraph.

Here we use BUF_SET to store the content of a file in a buffer called "header". We then use the content of this buffer as a header for different files. Using buffer enables you to call FS_GET only once. It wouldn't have been possible otherwise.

About spaces and indentation :
If you copied/paste the code, you could see by looking at mail1.txt, mail2.txt and mail3.txt files (with a text editor capable of reading Unicode) that the result is slightly different. That is because every single character between opening and closing of a tag are read by the tag and transmitted. That includes blanks, "Enter" and "Tab" keypads. This is particularly important for pure text XML nodes[1] meant to be used "as is", such as in our example.

2) Values

Values are also a way to store an XML stream at some point in the script. Let's see right away how to use a value :

Example 8 : Values

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<XDFL xmlns:xdfl="xdfl">
	<xdfl:VAL name="val1" value="content for val1"/>
	<xdfl:VAL>
		<val2>
		content for val2
		</val2>
	</xdfl:VAL>

<xdfl:OUTPUT>
	<xdfl:GET val="val1"/>
	@@[VAL:val2]@@
	</xdfl:OUTPUT>
</XDFL>
```

Output :

```
Content for val1
Content for val2
```

This code sample shows two ways of creating values and two ways of reading values.
First way of writing value is pretty straightforward : simply use the VAL tag , give the name and the value, and that's it.
Second way is a bit more sophisticated : you use the same VAL tag, but instead of giving a name and value as attributes of the tag, you provide an XML stream as input. XDFLengine then automatically creates Values for the stream, with XML path of nodes containing text being Value's names and text being corresponding value. In our example, val2 is a root node having only text for child, thus XDFLengine created a value "val2" equals to "content for val2".
Suppose we had this :

```
<xdfl:VAL>
<val2>
	<a> content for a </a>
	<b> content for b </b>
</val2>
</xdfl:VAL>
```

Then two values would have been created : one named "val2.a" containing text "content for a", a second named "val2.b" containing "content for b".

Now let's get back to example 8 :
The first way of using values is easy : simply use the GET tag with a name and you'll get the corresponding value.

[1] XML text nodes contain only text and no tag

The second way uses *parsed expression*. Parsed expressions are a way to dynamically insert special values a little bit everywhere in a XDFL script. XDFLengine provides several predefined parsed expression ready to be used, such as @@[DATE:file]@@ or @@[DATE:sql]@@ which are replaced by the current date in two different format. Parsed expression for using Values starts with "VAL:" followed by the path to the value[2].
Reading values using parsed expression allows you to do things that would be impossible otherwise, such as using values in attributes[3]:

```
        <xdfl:VAL name="val1" value="val2"/>
        <xdfl:VAL>
                <val2>
                content for val2
                </val2>
        </xdfl:VAL>

<xdfl:OUTPUT>
        <xdfl:GET val="@@[VAL:val1]@@"/>
        </xdfl:OUTPUT>
</XDFL>
```

Output :

```
Content for val2
```

Also note that you can read values in both ways no matter how you've created them.
In case you wondered, you can use parsed expression to read buffers, using @@[BUF:name]@@.

❖ Init script

Initialization script is necessary when you need to load modules, new tags, or to set application variables, or anything that will be used during all the application lifetime. Here is an example of init script :

Example 9 : Init script

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<XDFL xmlns:xdfl="xdfl">
        <xdfl:_MODULE_LOAD path="../bin/dbModuleStreamer_ora8i_win32.1.0.dll"
                                        connection_life_time="360000" sql_syntax="ORA"/>

        <xdfl:_ALIAS name="SQL_ORA8I" alias="SQL"/>

        <xdfl:OUTPUT ><ok/></xdfl:OUTPUT >
</XDFL>
```

What you see in this script is the loading of a dll that contains Oracle DB access tags (the _MODULE_LOAD tag). The second line sets an alias for SQL_ORA8I tag called "SQL". Aliases are a convenient way to define your own names for tags, or in this case to let your code being independent from the type of database you want to use.

Note : calling _MODULE_LOAD from anywhere else than an init script will not work. That is often true with tags starting with "_".

In order to call this script as an init script you need to pass the name of the file to the "init" parameter of the XDFLRun command :

```
XDFLRun.exe script=your_script.xdfl init=your_init_script.xdfl
```

[2] Note that paths are different from usual paths, since special character is "."(dot) and not "/" or "\".
[3] This way you can create a value for the root path of your application, set its value once for all in the initialization script, and use it every time you need path !

❖ Break !

Let's have a break : you are now familiar with basic elements of the XDFL language. You have seen how to combine tags either by nesting one inside another, or by using one after another. You also are familiar with the XDFL equivalents of variables : Buffers and Values.You also had a first taste of parsed expressions.
I suggest you to start having fun reading the Programming Guide, and looking at all the different nodes the XDFL language provides and to try them out.

Next two sub-sections will introduce more advanced features , the feature you will probably use for building simple real world applications.

❖ Simple Database Query

One of the first purpose of the XDFLengine was to interact with databases. This chapter introduce the reader to the simple way of accessing databases using the XDFLengine, that is using the "SQL" tag. It does not give all the details for this tag. For a complete review of database query, readers should look at the Programming Guide.
Note : We will use the following "Items" table in examples.

Figure 1 : Table "Items"

ID	Name	Price	Year
1	Aqua Table.	200	2003
2	Green chair.	30	2003
3	Electronic Keyboard	2500	2004

1) Reading database

SQL command enables you to execute an SQL request. The SQL command takes the connection string to the database (eg : login/pwd@db for Oracle) and the SQL string for parameters. For reading a table, simply use the SELECT request. Result for this request will be an XML stream. For a SELECT * FROM ITEMS request, the stream will look like this by default :

```
<ID>1</ID>
<NAME>Aqua Table </NAME>
<PRICE>200</PRICE>
<YEAR>2003</YEAR>
<ID>2</ID>
<NAME>Green chair </NAME>
<PRICE>30</PRICE>
<YEAR>2003</YEAR>
<ID>3</ID>
<NAME>Electronic Keyboard</NAME>
<PRICE>2500</PRICE>
<YEAR>2004</YEAR>
```

Note(1) : you can affect the structure of the result in two ways :

- *By using the "enclose_record" attribute of the SQL tag in order to enclose every record by a given tag.*
- *By using special SQL aliases ("as" clause) in the SQL request, that will be interpreted by the XDFLengine when building the output stream. Those special aliases are XML_ATTR, XML_OPEN_*, XML_CLOSE_*, XML_TEXT and XML_CDATA.*

You can find more information about this in the Programming Guide.

Note(2) : handling DB errors is done the same way as traditional errors. See "Handling and Raising Errors" chapter in the Programming guide for more information.

Note(3) : remember to use an initialization script to load the Database tags and alias them !!

2) Writing to the database

Writing to the database is done the same way, using the SQL tag with INSERT or UPDATE commands instead of SELECT. In order to use the input stream of the SQL tag as data for the SQL request you will have to use linked variables. When doing this, be very careful to use the variables *in the same order* as they appear in the input stream. More information on this in the Programming Guide.

❖ Calling scripts

When designing big applications, it is often necessary to split the work into many scripts. In order to do that you need be able to call a script from one another. This is done using the SCRIPT_EXEC tag.
Syntax is as follow :
<xdfl:SCRIPT_EXEC file=”path_to_file.xdfl”/>
SCRIPT_EXEC is often use in initialization scripts for calling module compilation scripts (we’ll talk about modules at the end of this document).

Note : for the moment, it is not possible to pass arguments to the script being called. This tag really is meant for calling initialization scripts from within the main init script…

4. Leveraging XML technologies : XSLT, XPath.

XML would be nothing without XSLT and XPath. That's why XDFLEngine integrates those two technologies for convenient XML stream manipulation.

❖ Applying XSL to XML streams

Using XSL is the most adapted way of transforming XML. Here is how you can apply an XSL to an XML stream very easily with the XDFLEngine :

Example 9 : Apply_xsl.xdfl

```
<?xml version="1.0" encoding="iso8859-1"?>

<XDFL xmlns:xdfl="xdfl">

        <!-- Read XSL file and compile it -->
<xdfl:XSL_COMPILE name="xsl_sheet"><xdfl:FS_GET
target="@@[VAL:ARG.xsl]@@"/></xdfl:XSL_COMPILE>

        <!- Transform the input, send result to a file -->

        <!-Write in the file -->
<xdfl:FS_SET target="output.html" >
        <!-Apply the xsl transformation -->
                <xdfl:XSL_TRANSFORM name="xsl_sheet">
                <xdfl:INPUT/>
        </xdfl:XSL_TRANSFORM >
        </xdfl:FS_SET>
</XDFL>
```

First step is to compile the xsl code. In this example the xsl is contained in a file (and the name of the file is the value of the "xsl" argument in the script's call).
Next step is to use the compiled xsl to transform the XML stream. Here, the stream is the input. You can use the stream redirection command ("<") from the command line to provide input to the script from a file or simply replace the xdfl:INPUT tag with XML in the code.

Now let's see a real example of using this script :
First, we will use this structure as input (we're going to store this data in an XML file for more convenience):

Sample XML input Data : "Input_data.xml"

```
<DATA>
<ROW>
        <ID>1</ID>
        <NAME><![CDATA[Aqua Table]]></NAME>
        <YEAR>2003</YEAR>
        <PRICE>200</PRICE>
</ROW>
<ROW>
        <ID>2</ID>
        <NAME><![CDATA[Green Chair]]></NAME>
        <YEAR>2003</YEAR>
        <PRICE>30</PRICE>
</ROW>
<ROW>
        <ID>3</ID>
        <NAME><![CDATA[Electronic Keyboard]]></NAME>
        <YEAR>2004</YEAR>
        <PRICE>2500</PRICE>
</ROW>
<ROW>
        <ID>4</ID>
        <NAME><![CDATA[test]]></NAME>
        <YEAR>2004</YEAR>
        <PRICE>1</PRICE>
</ROW>
<ROW>
        <ID>7</ID>
        <NAME><![CDATA[test]]></NAME>
        <YEAR>2004</YEAR>
        <PRICE>1</PRICE>
</ROW>
```

```
</DATA>
```

Does that remind you of anything ? Of course, since it is the result of our SQL SELECT statement we used in the Simple Database Query chapter, only we used enclose-record=”ROW” and we surrounded the overall result with a <DATA></DATA> in order to have a valid XML stream.

Next, let’s see what kind of XSL we’re going to use :

Sample XSL : “sample.xsl”

```
<?xml version="1.0" encoding="iso8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

	<xsl:template match="DATA">
		<html>
		<body>
			<table>
			<tr>
				<td>ID </td>
				<td>Year</td>
				<td>Price</td>
				<td>Name</td>
			</tr>
			<xsl:apply-templates select="ROW"/>
			</table>
		</body>
		</html>
	</xsl:template >

	<xsl:template match="ROW">

	<tr>
		<td>
			<xsl:value-of select="ID"/>
		</td>
		<td>
			<xsl:value-of select="YEAR"/>
		</td>
		<td>
			<xsl:value-of select="PRICE"/>
		</td>
		<td>
			<xsl:value-of select="NAME"/>
		</td>
	</tr>

	</xsl:template >
</xsl:stylesheet>
```

For those who are not familiar with XSLT language[4]., this code format the result of our query into an html table

Last but not least, let’s see how we’re going to invoke our “apply_xsl.xdfl” script shown earlier :

```
XLDFRun.exe script=apply_xsl.xdfl xsl=sample.xsl
```

Now open output.html file, Et voila !
There you have it ! You successfully applied a XSL to a XML stream ! Easy wasn’t it ?
As a bonus, supposing that you followed this tutorial from the beginning, you should now be able to extract data from a database, turn it to an XML stream, format this stream as an html table using XSL and print the result in an HTML file. You definitely *are* getting familiar with XML technologies now, aren’t you ?

❖ Xpath

Xpath is just a technology to access data in an XML stream. XDFLEngine provides a very simple way of doing this, using the XPATH_SELECT tag.

[4] Then you definitely should play and modify the sample xsl !

What this tag basically do is return all nodes verifying the XPATH request given in the “path” attribute.
This is particularly useful for applying filters to records of data extracted from a database (using the node[attribute=value] syntax).
Since this tutorial is not an Xpath tutorial, I will not dig into any more details.

Note : Just as a reminder, remember that in case you only have one set of different nodes, another very powerful way of accessing specific data in an XML stream is to use the VAL tag and its input auto-parsing ability.

5. Your first real XDFL application : Offline Catalogue Creator

It is now time for you to use all that you have learned in what could very well be a real-life application. We will call this application “Offline Catalog Creator” (OCC)

❖ Purpose of the application

The purpose of our application is to create an offline html catalog from a database containing items.
The catalog will have two directories : the “main” directory storing html files corresponding to products being sold this year along with an index file, and the “archive” directory containing zipped file of older items summaries.
The database will be kept as simple as possible because :

1- We don’t want to pollute our code with obscure SQL requests (this document is not a SQL tutorial).

2- XDFL has a very elegant and simple way of dealing with complex database structures manipulation, but this way will only be shown in the advanced features section.

❖ Example of usage

The application should be invoked this way :
XDFLRun.exe init=init.xdfl target=occ.xdfl year=2004 archivexsl=archive.xsl catalogxsl=catalog.xsl archivename=catalog_old.html.gz catalogname=catalog_2004.html

Where init.xdfl contains all configuration steps (loading dll, setting path,…) , occ.xdfl is our script, and year is the year of the catalog.
Archivexsl and catalogxsl are the name of the xsl that will be used to convert xml to html and Archivename and catalogname are the name of the two files created by the program.

The result will be a file called catalog_2004.html containing the description of all our items released in year 2004 and their price, plus a catalog_old.html.gz file containing the short description of all the items released in year before 2004.

❖ Database

We will use the same table we used before, only with two new fields called “description” and “short description”. Thus the table is as follow :

Figure 2 : Table “Catalog”

ID	Name	ShortDescription	Description	Price	Year
1	Aqua Table.	Aqua table for your living-room.	A wonderful Aqua table that will impress your neighbors very much.	200	2003
2	Green chair.	Chair. Italian design.	A simple yet classy chair designed in Italy. Also available in blue.	30	2003
3	Electronic Keyboard	Keyboard with foot pedal.	Play the music you’ve always wanted with this brand-new electronic keyboard. Big screen and foot pedal included.	2500	2004
4	Pacific Bed	King-size water bed	The biggest water bed ever sold. Very safe, 100 years guaranteed !	2400	2004

❖ Scripts

1) Initialization script : "init.xdfl"

- Task :
  - load database library and create aliases.
  - create DSN and ROOT values

Now let's write the init script :

Init.xdfl

```
<XDFL xmlns:xdfl="xdfl">

        <xdfl:VAL context="APPLICATION" name="DSN" value="xdflengine/xdflengine@XDFLENGINE"/>
        <xdfl:VAL context="APPLICATION" name="ROOT" value="D:\xdflengine\bin" />
        <xdfl:_MODULE_LOAD path="@@[VAL:ROOT]@@/zipModuleStreamer_win32.1.0.dll"/>

        <xdfl:_MODULE_LOAD path="@@[VAL:ROOT]@@/dbModuleStreamer_ora8i_win32.1.0.dll"
                                connection_life_time="360000" sql_syntax="ORA"/>
        <xdfl:_ALIAS name="SQL_ORA8I" alias="SQL"/>

        <xdfl:OUTPUT ><ok/></xdfl:OUTPUT >

</XDFL>
```

There really isn't much to say. We created a ROOT value to store the path where the XDFLRun exe and all the libraries are stored, and we decided to store this value in the APPLICATION context (to make it available from everywhere)[5]. Then we load the libraries, and create the alias for the SQL tag.

To tell the truth, the initialization script is often not complete until the end of the coding process. Still, knowing what you will need in this script before you started to code anything is a sign of good designing.

2) Catalog creation script : "occ.xdfl"

- Tasks :
  - create two files, one containing the catalog for the given year, the other containing short descriptions for all previous items.
- Argument :
  - "year" : the year for the catalog
  - "catalogname" : the name of the catalog
  - "archivename" : the name of the archive
  - "catalogxsl" : name of the XSL for the catalog
  - "archivexsl" : name of the XSL for the archive
- Output :
  - "catalogname"(.html) : the catalog
  - "archivename"(.gz) : the archive

Let's write a short pseudo-algorithm describing the script :

Pseudo algorithm for OCC.xdfl

```
-Read content from database with year="ARG.year"
-Appy xsl "ARG.catalogxsl" to this content
-write the result in "ARG.catalogname".html
THEN
-Read content from database with year<"ARG.year"
-Apply xsl "ARG.archivexsl" to this content
-Zip the content and write the zip in "ARG.archivename"
```

[5] You may want to look at the programming guide for more information about contexts

We can see right now that there will be two streams of data in the script.

Next thing to do is to find a tag doing what we want (presumably one tag for each line in the pseudo-code).
Line 1 : reading content from database will be done via the SQL tag.
Line 2 : applying an XSL is done via the XSL_COMPILE / FS_GET and XSL_TRANSFORM tags (first read and compile the XSL, then apply it to the stream).
Line 3 : writing the stream to a file is done using FS_SET.

The THEN line indicates that what's coming next is a separate data stream.

Let's write the first stream :

"OCC.xdfl" (first stream, catalog part)

```
  <xdfl:XSL_COMPILE name="catalog_xsl_sheet"><xdfl:FS_GET
target="@@[VAL:ARG.catalogxsl]@@"/></xdfl:XSL_COMPILE>

  <xdfl:FS_SET target="@@[VAL:ARG.catalogname]@@">


       <xdfl:XSL_TRANSFORM name="catalog_xsl_sheet">
              <DATA>
              <xdfl:SQL statement="SELECT * FROM ITEMS WHERE YEAR='@@[VAL:ARG.year]@@'"
                     connection="@@[VAL:DSN]@@" enclose_record="ITEM" />
              </DATA>
       </xdfl:XSL_TRANSFORM>

  </xdfl:FS_SET>
```

Notice that we surrounded the result of the SQL command with <DATA></DATA>. That's because XSL_TRANSFORM wants a valid XML stream as input. That means a tree with a single root. If we didn't add the DATA tag, we would have had a list of <ITEM></ITEM> as input, but no single root.

Note : Adding <DATA> as a root will have to be taken into account when creating the XSL. Also notice that for clarity reason we decided to compile the XSL at the beginning of the script rather than in the middle of it.

Here comes the catalog.xsl :

Catalog.xsl

```
<?xml version="1.0" encoding="iso8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

       <xsl:template match="/DATA">
              <html>
                     <head>
                     <title>Catalog for the year <xsl:value-of select="ITEM/YEAR"/></title>
                     </head>
              <body>
                     <table>
                            <tr>
                                   <td>NAME </td>
                                   <td>DESCRIPTION </td>
                                   <td>PRICE </td>
                                   <td>ID </td>
                            </tr>

                            <xsl:apply-templates select="ITEM"/>

                     </table>
              </body>
              </html>
       </xsl:template >

       <xsl:template match="ITEM">
       <tr>
              <td><xsl:value-of select="NAME"/> </td>
              <td><xsl:value-of select="DESCRIPTION"/> </td>
```

```
                <td><xsl:value-of select="PRICE"/> </td>
                <td><xsl:value-of select="ID"/> </td>
        </tr>

        </xsl:template >


</xsl:stylesheet>
```

Now you can run this script :

```
XDFLRUN init=init.xdfl script=occ.xdfl catalogname=catalog2004.html  catalogxsl=catalog.xsl
year=2004
```

Ok, this may not be the fanciest HTML page you've ever seen, but I think you got the idea (not to mention the fact that I wanted to keep the XSL as simple as possible for the sake of clarity).
Feel free to customize your catalog the way you want, by adding colors, font properties, etc !
By adding a "TYPE" column in your SQL table and using <xsl:if test="TYPE=xxx"> you can even add pictures or icons showing that the item is a table, a chair, etc.

Second stream :
The second stream is very similar to the first one except that :

- instead of FS_SET you will have GZ_SET (same attributes)
- instead of <xsl:value-of select="DESCRIPTION"> you will display the SHORTDESCRIPTION column.
- Instead of SQL:SELECT * FROM ITEMS WHERE YEAR= you will find WHERE YEAR < (remember you can not use < inside an XML attribute, so you will have to use < instead).
- In the archive.xsl, change the title (you can't use item/year since all items are not from the same year), and creates a new column for the item's year.

And that's it !
I will let you finish this program as an exercise. Remember you can always use the XDFL:LOG tag everywhere in the script to display the data stream in the console. This is always useful for debugging. We may give the complete application later in the future at http://xdflengine.sourceforge.net.

6. Presentation of advanced features

This section shows JavaScript microDOM, Classes Objects and Modules, and an elegant way of dealing with complex database manipulation using DB_SET and DB_GET tags. For a full and complete documentation of those features please reefer to the Programming Guide.
The last section of this chapter shows how to encapsulate database operations in classes and modules in order to build a strong and reusable layer for accessing database. This last section is the achievement of this document and shows the most advanced parts of the XDFLEngine.

❖ Using JavaScript within XDFL

For some data manipulation, it is sometimes useful to use JavaScript code within an XDFL script. Common examples are arithmetic operations and string manipulation in the data stream but it can be useful in *many* cases.
Using JavaScript is done via two tags :

- JS_COMPILE : compile the JavaScript functions you want to use.
- JS_EXEC : execute the JavaScript function.

Passing data from the XDFL stream to the JavaScript function is done the same way as with other's tag, by nesting the stream in the JS_EXEC opening and closing tags. You can then access this stream within the JavaScript function using XDFL predefined object and its "input()" method.
Because parsing a XML stream is not a thing you want to code every time you need to use JavaScript in a script, the XDFL object also has the useful "select(Xpath)" "selectSingleNode(node,Xpath)" and "selectNodes(node,Xpath)" methods that retrieve values and set of nodes in an XML tree[6]. Beware though that the Xpath parameter isn't really a full featured Xpath path the way the standard defines it (that's why it is called jsMicroXpath). Only very basic syntax is supported. For example, you can't use [condition] in the path, but you should instead use selectNodes() and select the nodes you want among the returned set of nodes using custom JavaScript code.

I strongly emphasis the point that using JavaScript for doing things that could be done using regular XDFL tags is *not recommended at all*. It is often a sign of bad design and if you come to the point where your XDFL scripts contains as much XDFL code as JavaScript functions, it is time to seriously reconsider your design.
If you don't have other choice, and if it really can't be done using XDFL tags, then maybe you should consider the option of writing your code into a custom XDFL tag (this time using C++).

For a complete list of functions available within JavaScript functions, reefer to the Programming Guide.

Example :
Let's suppose we want to convert prices in every Item to a different currency (by applying a multiplier). Note that we don't want to simply display the price differently but really modify it in the XML stream. If what we wanted was only to display the price differently, then we would have done it in the XSL (see previous chapters).

Jscript example : convert prices

```
<?xml version="1.0" encoding="iso8859-1"?>
```

[6] Actually, an XML DOM document representing the input XML stream. The parsing of the input into a searchable DOM tree is done automatically when the parse_dom attribute of the JS_EXEC tag is set to 1.

```
<XDFL xmlns:xdfl="xdfl">

<xdfl:JS_COMPILE>
	<![CDATA[
var multiplier=0.65

function convert_price(){
	try{
		//first let's select the items
		var input= XDFL.XMLinput()
		var xml_Nodes = XDFL.selectNodes( input,"ITEM")

		//if there is any
		var length=xml_Nodes.length
		if (length){
			//then apply the modifier to the price
			for (i=0 ; i!=length ; i++){
				xml_selectSingleNode(xml_Nodes[i],"PRICE").nodeValue *=
multiplier
			}
		}
		XDFL.outputXML(input)

	} catch(e){XDFL.log(e)}

}

]]>
</xdfl:JS_COMPILE>

<xdfl:OUTPUT>
	<xdfl:JS_EXEC call="convert_price()" parse_dom="1">
		<DATA>
		<xdfl:SQL statement="SELECT * FROM ITEMS "
			connection="@@[VAL:DSN]@@" enclose_record="ITEM" />
		</DATA>
	</xdfl:JS_EXEC>
 </xdfl:OUTPUT>

</XDFL>
```

Once again, I strongly encourage you to have a look at the programming guide if you want to know all the options you have for using JavaScript in your code.
Note that using try / catch is compulsory even for simply displaying errors thrown by the JavaScript code.

❖ Classes, Objects and Modules

XDFL provides you with a object-oriented-programming-like approach to manage your code. XDFL is by no means an object oriented language, so don't expect to be fully comfortable with those notions right away even if you are already familiar with Java or C++ (although it may certainly help).
Note : If you are not yet familiar with execution contexts, please read the corresponding chapter in the Programming Guide before reading this.

1. **MODULES** : modules are the XDFL equivalents to functions in that they are meant to store and reuse bits of code. You can define a module with the xdfl:_MODULE_COMPILE tag (the underscore at the start of the tag usually means that it may only be called in the initialization script) and call this module with the xdfl:MODULE_EXEC tag. Passing parameters to the call is done in the traditional XDFL way by adding attributes to the tag. It is often a good thing to add a commentary saying what are the parameters for the module.

   For examples of MODULE_EXEC and _MODULE_COMPILE please look in the programming guide.

Note : don't forget the PASSIVE tag in the module definition, since what you want is to compile the tags themselves and not their results. Also don't forget to define the xdfl

namespace before using it in the module code, for example in a NULL tag (as shown in the Programming Guide example for modules).

Power tips : the "tag" attribute for MODULE_COMPILE creates a XDFL tag that calls the module.

2. **CLASSES** : now that we have defined modules, we can easily define classes by saying that classes are names given to groups of modules (or namespace). You usually start by defining a class, then modules with the "class" attribute of the MODULE_COMPILE tag equals to the name of the defined class. Once a module belongs to a class, it can not be called by a MODULE_EXEC unless the "class" attribute is set to the name of the class, or unless you already are in the correct class (for example when you are calling a module from within a module already belonging to the class).
   Defining a class is done by using the _CLASS_DECLARE tag. This tag means "let's create a name that I will use later in my modules definitions". Once you have defined a class, you can then use its name in module definitions by using the "class" attribute of the _MODULE_COMPILE tag.
   Extending a class is done at class definition by using the "extend" attribute of the _CLASS_DECLARE tag. If class1 extends class2, then class2 will have access to all modules of class1 ("class" attribute of the MODULE_EXEC can be set to class1 or class2). Once you have extended a class, you can redefine modules simply by defining a module with the same name. Then you can use the "class" or the "ancestor" attribute of the MODULE_EXEC to specify which module you want to call (see the Programming Guide for code examples).

3. **Objects** : XDFL's objects are closer to traditional OOP's objects than what you may think at first sight : an object of a class can be defined as a context used to store data and having access to the class modules.
   Creating an object of a class is done via the CLASS_CREATEOBJECT XDFL tag. What this tag does is create the context for the object's data and call the "init" method of the class (which can be seen as the class' constructor) in its newly created context. *You actually have to fill the init method yourself.*
   Here is a typical class definition for being able to create objects from:

Typical Init and Get_Data methods for using objects :

```
<!-- initialization -->
<xdfl:_MODULE_COMPILE class="class1" name="init">
       <xdfl:PASSIVE ><xdfl:NULL xmlns:xdfl="xdfl">

       <!-- create the buffer containing data specific to the object -->
       <xdfl:BUF_SET context="OBJECT" create="1" name="object_buffer">
       <xdfl:INPUT />
       </xdfl:BUF_SET >

       </xdfl:NULL ></xdfl:PASSIVE >
</xdfl:_MODULE_COMPILE >

<!-- add objet's data -->
<xdfl:_MODULE_COMPILE class="class1" name="add_data">
<xdfl:PASSIVE ><xdfl:NULL xmlns:xdfl="xdfl">

       <xdfl:BUF_SET context="OBJECT" name="object_buffer">
       <xdfl:INPUT/>
       </xdfl:BUF_SET>

</xdfl:NULL ></xdfl:PASSIVE >
</xdfl:_MODULE_COMPILE >
```

```
<!-- get object's data -->
<xdfl:_MODULE_COMPILE class="class1" name="get_data">
        <xdfl:PASSIVE ><xdfl:NULL xmlns:xdfl="xdfl">

                <xdfl:BUF_GET name="object_buffer"/>

        </xdfl:NULL ></xdfl:PASSIVE >
</xdfl:_MODULE_COMPILE >
```

This sample defines the object's data structure as a buffer, but it can also contains values or anything that a context can store.
One of the main difference with traditional OOP, when defining object that way, is that you don't restrain the object's data structure at class definition. In fact you never define the object data structure at all but constantly modify the object's context the way you want (yet only from within the object's modules). If you want to set the data structure of an object once for all, then you will have to build your own "safe" class, and derive all classes from it.
Yet remember that XDFLEngine is *not* an Object Oriented Language, but is instead a scripting language with a basic and raw (yet convenient) object layer.
One last thing about object and modules : please not that modules of an object are executed inside the object context and not the contrary. Otherwise you wouldn't be able to access object's data from its modules. That's what the "object" argument of the MODULE_EXEC stands for : specifying the name of the object's context in which you want to execute the module. That mechanism of indicating the name of the object in the method's call is usually hidden in modern object oriented languages, but it works the same way.

❖ Database access in object mode

You have already seen how to execute SQL commands using the SQL XDFL tag. But the XDFLEngine has another way of dealing with database using database objects. Note that the term "object" has nothing to do with class' objects of the previous chapter.
A XDFLEngine database object is the definition of a compact extract of the database's relational model you're working with. For example a set of tables along with their keys. With this approach, every database interaction is done through the use of a get/set mechanism. Both Get and Set use objects definitions to automatically creates the corresponding INSERT/UPDATE/DELETE/SELECT SQL commands, depending on tag's parameters.

This mechanism works with three tags called _DB_DEF_COMPILE to create database object definition, DB_GET and DB_SET to map the object on the database for reading and writing respectively.
One additional tag often used with those tag is the DB_AUTODESC tag. This tag automatically completes an object definition by looking into the database. This is *extremely* convenient, since that means you don't need to write all fields of a table in the object definition but only the most important ones. That also means that using this feature efficiently makes your code able to manipulate fields of a table that did not exist at the time you coded the application.

The great benefit of this approach is to provide a structural approach for database manipulating applications. It is very common indeed that relational databases are used to store information about entities, and split those information among different tables. In this case, manipulating data from the database as objects proves to be not only easier, but more natural than using SQL directly. As a matter of fact, this process of decomposing a relational model into objects have already been proven successful in running real world applications.

This document won't give additional information on how to create database objects since the Programming Guide already gives all the details in a very explicit manner.

❖ Conclusion

When things start to get serious with Database manipulation, you can't really rely solely on the SQL tag anymore. Using classes, objects, modules , DB_SET and DB_GET tags, and the supernatural Autodesc feature, you will have the tools to build strong, reliable, database-intensive XDFL applications.
As a final example, here is the init method of the xtn.dbobject class. This class is meant to be used a the root class for classes that use database objects. This way you can define the database object directly as input of the _CLASS_DECLARE tag (input of this tag is passed as input to the init module).

Final example : xtn.dbobject sample

```
<xdfl:_CLASS_DECLARE class="xtn.dbobject" />

	<!-- xtn.dbobject.init_class -->
	<xdfl:_MODULE_COMPILE class="xtn.dbobject" name="init_class" >
		<xdfl:PASSIVE ><xdfl:NULL xmlns:xdfl="xdfl" >

			<xdfl:IF val1="@@[VAL:ARG.init_class_as]@@" val2="" op="strne">
				<xdfl:VAL name="DBOJECT_NAME"
value="@@[VAL:ARG.init_class_as]@@.dbobject" />
			</xdfl:IF >

			<xdfl:IF val1="@@[VAL:ARG.init_class_as]@@" val2="" op="streq">
				<xdfl:VAL name="DBOJECT_NAME"
value="@@[VAL:ARG.class]@@.dbobject" />

				<xdfl:_DBDEF_COMPILE name="@@[VAL:DBOJECT_NAME]@@"  >
					<xdfl:DB_AUTODESC connection="@@[VAL:DSN]@@"
expand_field="1">
						<xdfl:INPUT />
					</xdfl:DB_AUTODESC >
				</xdfl:_DBDEF_COMPILE >
			</xdfl:IF >

			<xdfl:VAL context="APPLICATION" name="@@[VAL:ARG.class]@@.dbobject_name"
value="@@[VAL:DBOJECT_NAME]@@" />

		</xdfl:NULL ></xdfl:PASSIVE >
	</xdfl:_MODULE_COMPILE >
```

7. Common mistakes

Here is a list of common mistakes when starting with XDFL programming. Send us yours[7] and we will add them to this document !

❖ General mistakes

❖ Jscript common mistakes

- Using "<" or ">" for comparing two values :
  Using '<' or '>' in a JavaScript is a source of error. Even when enclosing JavaScript code with CDATA. Remember that < is converted to "<" and ">" to ">". Most common mistake is to use it in a "for" loop.
- Not enclosing code within a try{}catch{XDFL.log(e)} : this will prevent you from catching error thrown by JavaScript at execution.

[7] at benjamin.garrigues@sourceforge.net