SOGETI

XMLWare Forms Framework Guide

Benjamin GARRIGUES

Introduction

Warning** : this document is meant for readers already familiar with the XDFLEngine technology. If you haven't read about this technology before, then please read the "Getting started with the XDFLEngine" document first, available at* ***http://xdflengine.sourceforge.com*.*
DONT START XDFL WITH THIS DOCUMENT !

This document introduces readers to the XMLWare Forms framework**. This framework is built on top of the XDFLEngine technology, uses standard technologies (Javascript, XML, XSL, CSS), and is meant for easy development of fast and reliable web/intranet applications involving database interactions**.

The XMLWare Forms framework is already used in variety of real-world projects and applications and has been proven both stable and reliable. For information concerning companies using this technology, look at the XDFLEngine website[1].
The banjan (pronounce "banyan") open-source project[2] is a good example of what can be done with this framework, although it is also very suitable for regular form-based web interfaces reading data to and from databases.

Because the framework uses the XDFLEngine as a web-server extension to Apache or IIS, and JavaScript on the client-side , it can be used on all configurations supporting those technologies.

This document should provide the reader with enough information for him to be able to build real web applications. It is meant to be read from the start to the end, every chapter introducing the reader with more complex and powerful features. Exception to previous statement is the Architecture chapter which will certainly not be assimilated in one go but should instead be considered as a "reference" chapter.

A forum is available for all kinds of questions regarding this document (or the XDFLEngine in general) at http://www.sourceforge.com/xdflengine.

Enjoy your reading !

[1] At http://xdflengine.sourceforge.com.
[2] At http://www.sourceforge.net/banjan

Requirements

Here is a list of supported system component for running the framework :

1. Server side :
    a. OS : Windows 98/NT/2000/XP, LINUX, HP-UX, SOLARIS
    b. Database manager : Oracle 8i, 9i, PostgreSQL, MS SQLServer, generic ODBC
    c. Web server : IIS, Apache
2. Client side (minimum version):
    a. Browser : Mozilla 1.7, Firefox 0.8, Internet explorer 5.5 with MSXML 3.

Installation

Installing the framework is a very simple task and is basically a matter of copying a folder and paste it where you want. Yet, it supposes you already successfully installed your database manager and your web server.
If you have any issue regarding the framework installation, you can always ask questions on the XDFLEngine forum on the Sourceforge site.
Examples will be given for the PostgreSQL DB.

File structure

The typical file structure of an application built with the framework is as following :
-**Application**
|-**root**
|-**bin** : XDFLRun.exe and all dlls required to run the XDFLEngine.
|-**objects** (or “classes”) : XDFL scripts used to access DB and return XML data (called “class scripts”).
|-**javascript** : server-side JavaScript (optional).
|-**xsl** : server-side XSL (optional).
|-**site**
|-**xdfl** : XDFL scripts describing user screens (called “screen scripts”).
|-**css** : styles.
|-**html** : html files used only for calling javascript code.
|-**javascript** : JavaScript scripts for manipulating XML streams, creating html dynamically, and managing communications with the server. Core JavaScript framework scripts.
|-**xsl** : xsl used by the JavaScript, or directly by the browser, to display XML data.
|-**images** : images.

The root subdirectory is used to store component executed on the server side.
The site subdirectory is used to store component executed on the client side (except for files in the xdfl directory).
Initialization script (called init.xdfl) is in /site directory (and not /site/xdfl). Default.htm file is also in /site directory.

Apache configuration

TODO

IIS configuration

IIS configuration is quite simple.

1. Create a site.
2. Select the “site” directory as the site’s directory in “access path”.
3. In Properties/configuration : add “.xdfl” extension and point to the root/bin/XDFLisapi_win32.1.1.dll file. Restrict to POST and GET verbs.

That’s all !

XMLWare Forms Framework Architecture

As every client-server framework, its architecture splits into two parts : server-side and client-side. The most common scenario being : client sends query to the server (such as “get me the form for updating information ‘xxx’ in the database”), server gets the data from the database (in this case ‘xxx’), fills the appropriate template (a form) with it, then send the result back to the client that displays everything and wait for the user to do something else.
Although the XMLWare Forms framework adds important differences to this scenario, the big picture is pretty much the same.

Note : this chapter is not meant to be understood from the first reading, but should instead be considered as a “ reference” chapter. Re-reading it after having investigate new features will enable you to understand more and more the way the framework works.

Server-side

Basically, the server's only task is to receive requests and give corresponding scripts to the XDFLEngine for execution. There are three types of script that the server will deal with : initialization scripts, class scripts and screen scripts.

✍ *Initialization scripts*

The initialization of the framework for the server side is done via two scripts : the "init.xdfl" and "conf.xdfl" scripts.

✍ Init.xdfl

This scripts is used as initialization script for the XDFLEngine the same way it is used by the XDFLRun.exe for batch applications (see XDFLEngine tutorials). It is used to load libraries, execute class definition scripts, and set environment variable (most common of which is the ROOT value).

Sample init.xdfl :

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

<XDFL xmlns:xdfl="xdfl">

        <!-- APP CONSTANTS -->

        <xdfl:VAL context="APPLICATION"        name="ROOT" value="D:/xmlware/applications/my_application/root" />

        <xdfl:VAL context="APPLICATION"        name="DSN"  value="login1/pwd2@DB1" />

         <!-- XDFL MODULES -->

        <xdfl:_MODULE_LOAD path="@@[VAL:ROOT]@@/bin/dbModuleStreamer_ora8i_win32.1.0.dll"
connection_life_time="360000" sql_syntax="ORA"/>
        <xdfl:_ALIAS name="SQL_ORA8I" alias="SQL"/>
        <xdfl:_ALIAS name="DB_GET_ORA8I" alias="DB_GET"/>
        <xdfl:_ALIAS name="DB_SET_ORA8I" alias="DB_SET"/>
        <xdfl:_ALIAS name="DB_AUTODESC_ORA8I" alias="DB_AUTODESC"/>
        <xdfl:_ALIAS name="_DBDEF_COMPILE_ORA8I" alias="_DBDEF_COMPILE"/>

        <!-- COMPILED -->
        <xdfl:SCRIPT_EXEC file="@@[VAL:ROOT]@@/objects/xtn.dbobject.class.xdfl"/>
        <xdfl:SCRIPT_EXEC file="@@[VAL:ROOT]@@/objects/sec_functions.global.xdfl"/>
        <xdfl:SCRIPT_EXEC file="@@[VAL:ROOT]@@/objects/stuff.class.xdfl"/>

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

This example defines two values in the Application context (so that values exist as long as the server is running), imports Oracle8i modules, create 5 aliases related to DB manipulation (those aliases are compulsory), and launch 3 scripts that creates classes. Finally, it writes "OK" in the output. Notice that the full path of the application is only entered once in the script (and nowhere else) for the ROOT value. This way, moving the application only requires you to change the path at one line.

✍ Config.xdfl

Config.xdfl is used to set environment variables needed by the XDFLEngine when running along with a web server (because in that case, XDFLEngine is running in a multithreaded environment). Examples are number of threads, temp directory, name for log file, list of server variables used by the application (e.g. : "REMOTE_USER" for the login of the user).

✍ *Class scripts*

Class scripts are initially meant to get, set, or update data from the database, but they can do pretty much anything (such as sending mail, reading/writing files, etc.). A class script is called by JavaScript scripts or by other class scripts when the framework needs to make the server do something. Class scripts should be placed in the “root/class” directory or “root/object”, whatever you want (but make sure your init.xdfl file is correct).

✍ *User Screen scripts*

Screen scripts build user screens. They are either forms, views, tree-views, or custom screens. Screen scripts are stored in the “site/xdfl” directory. They are the first type of scripts accessed by the JavaScript scripts and, most of the time, they contain names of class scripts to invoke in order to get required data for the screen.

✍ *Tools scripts (XDFL scripts that will make your life easier)*

None of those scripts are strictly compulsory. Yet building an application without them is not recommended until being fully comfortable with the framework.

- **“Root/classes/xtn.dbobject.class.xdfl”** : this script gives a base class for all your classes. Its main features is to define a get/set/insert/update/delete set of methods and associate them with the dbobject defined at input of the “class_declare” tag. This way not only don’t you need to use db_set and db_get (you’ll use the get/set/insert/update methods instead), but when you call the “get” method of a class extending xtn.dbobject, you’ll automatically get the data corresponding to the class’ object (no need to dbdef_compile either).
- **“Root/classes/sec_functions.global.xdfl**” : this script defines a set of modules for using the built-in security model of the Framework. One example is the definition of the “secured_exec” module (aliased to the “secured_exec” XDFL tag). This module checks whether the user has rights for executing the corresponding action (see the "built-in security model” chapter for additional information on the security model). This set of function is required when using the “xtn.class_wrapper.xdfl” script.
- **“Site/xdfl/xtn.class_wrapper.xdfl”** : this script encapsulates all calls to a server-side module with a check for user rights. For security reasons, calling a class method should always be performed via the class wrapper. All examples given in this tutorial use the class wrapper. Class wrapper uses modules defined in “sec_functions.global.xdfl”.

Client-side

✍ *Client description and features*

The client side is built with many different technologies : JavaScript, XSL, HTML, CSS and XML. JavaScript is the core technology, its role being to manipulate all the others. Basically, when we says “client” here, we mean the set of JavaScript scripts.

In order to understand a little bit better why so many different technologies are used and how the client side works let’s get back to our previous “get me the form for updating ‘xxx’ in the database” scenario given in introduction.

Because the client-side of the framework works “dynamically”, things are a bit different from the example :
First, what the client receive is not a “ready-to-print” html document. Instead, it asks and receives two XML streams from the server : one XML stream containing the *data* and another containing the web-page *definition*. Once it has received the two streams, it dynamically builds a third set of XML data called the *description*, by merging the data and the definition. Finally, this third set of data is transformed into HTML using XSL files. The browser may then have to use CSS style sheets.
Second, since this process of building the display is done dynamically by client’s side JavaScript code, it has the possibility to dramatically modify the look of the web page at every moment, such as after a user fills in a field.
Last but not least, the *data* and *description* sets of XML data are updated automatically. Which means that modifying the web page by entering a new value for a field will result in modifying the corresponding information in the *description* and the *data* XML trees. This is to ensure consistency between data and display, but also between display and display !! [3].

To sum it all up, the JavaScript code has the main task of building and updating up to a maximum of FOUR sets of data[4] :

1. The DEFINITION : sent by the server after the client requested it (either by calling nav_openView() or nav_openForm() functions), and a direct result of “user screen scripts” execution (see Server-side)
2. The DATA : send by the server after the client requested it by calling a “class script”. Name of the class script to call is given in DEFITINION.
3. The DESCRIPTION (forms only) : built by the client by merging the two previous sets of data.
4. The DISPLAY : built by the client by applying XSL on the DESCRIPTION (or the DATA for views), and inserted into the “window.document” browser object.

There are many benefits to this architecture, some of which are :

- ✍ Client always has access to the original data stream in a readable format (XML).

[3] Suppose that a single data is displayed in two different fields in the web page. If the user updates a field somewhere, then the new value will automatically be updated in *both* fields.
[4] The PAGE set of data (view template only) which is a DOM Tree containing all the nodes of the DATA that should be displayed in the current page of a view could also have been included in the list, but wasn’t for simplicity purpose.

- ✍ Formatting the data is done on the client side, thus resulting in less server CPU usage.
- ✍ Because the XSL are applied to data on the *client* side, heavy graphic modifications of the page can be done dynamically, directly on the client, without any communication between the client and the server (less network intensive).
- ✍ Code is relatively clean : data presentation is kept separated from code and data definition.

Of course, the downside is that the client needs to have both a not-to-old browser supporting recent JavaScript/XML code and a decent CPU (yet nothing outrageous).

✍ *JavaScript files*

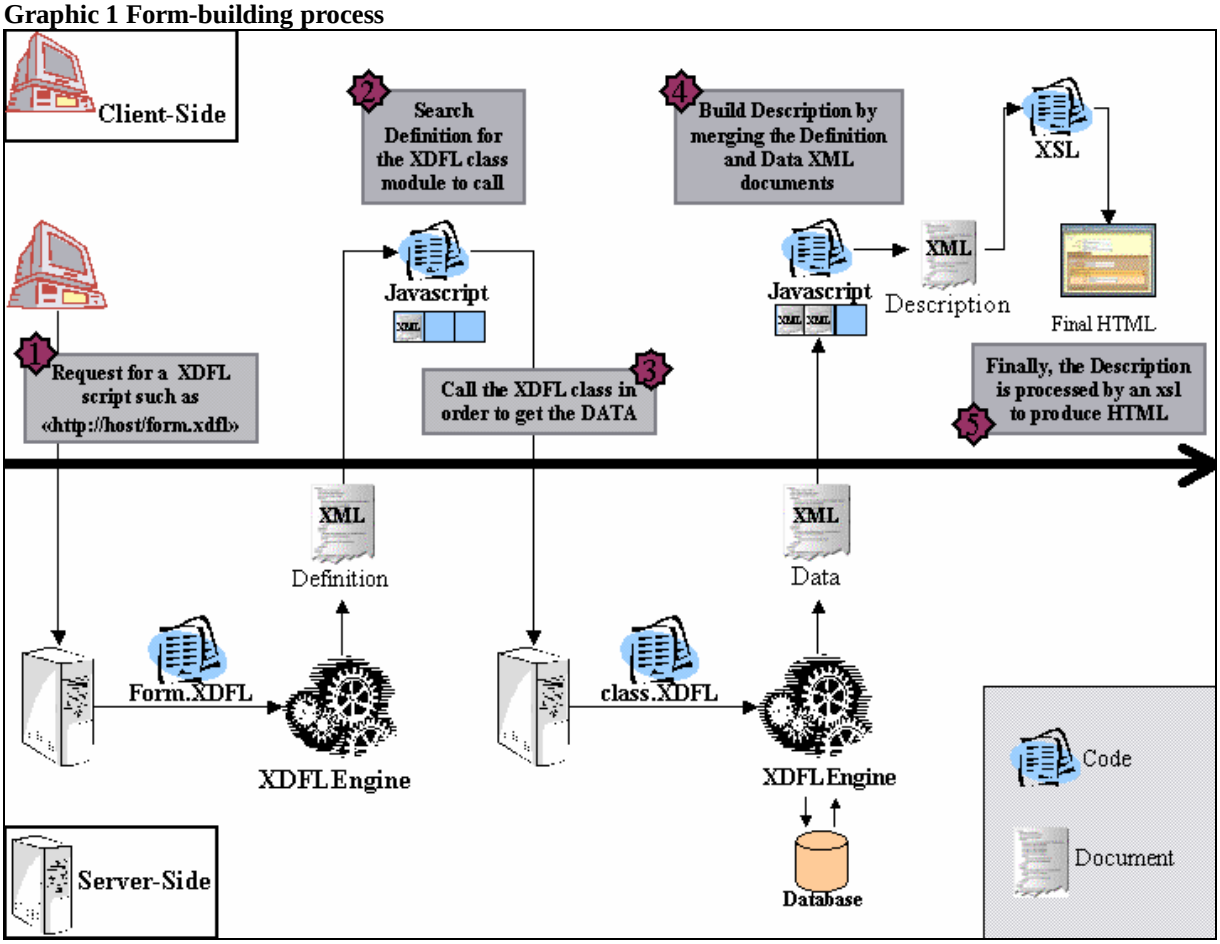
Those files are all in the /site/JavaScript directory.

1. xml.js : set of basic XML/DOM manipulation functions. This document ensure that all XML manipulation functions are available whether the browser is Internet Explorer or Mozilla. Typical function is “xml_testXMLParser()” which tests what XML parser the framework should use (MSXML or other).
2. Common.js : set of functions for tasks common to all templates. This range from environment initialization to server communication.
3. Config.js : configuration variables such as CT_MODE_DEBUG for setting debug mode to true or false.
4. View.js : All functions related to the view template (reading view definition, getting the data, displaying the data using view’s XSL)
5. Form.js : All functions related to the form template (reading definition, getting the data, building the description, displaying the description using form’s XSL).
6. Treeview.js : All functions related to the tree-view template (reading definition, displaying the tree-view using the treeview.xsl file).

The big picture

Here is a graphic summing up the whole process of building a page (in this case, the page is a form). Step 4 doesn't exist for views (on step 5, XSL is applied directly on data). Steps 3 and 4 doesn't exist for tree-views (on step 5, XSL is applied directly to definition).

Graphic 1 Form-building process



Building Your first screen

This chapter will teach you the first steps to build a simple screen. The screen will use the view template and its purpose is to list the content of a database table called “book”.

✍ Data model

Here is the data table we’re going to list :
Table name : BOOK
Table fields : ID (integer), NAME (string[255])

✍ The class script

The class script is the script defining the object corresponding to the table previously described (a book) as well as the method to manipulate it.

Here is the definition of the object :

/root/classes/book.class.xdfl

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

<XDFL xmlns:xdfl="xdfl">

<!-- CLASS : book -->

	<xdfl:_CLASS_DECLARE class="book" extends="xtn.dbobject" init_class="init_class">
		<xdfl:PASSIVE >
			<node name="book" DBtable="BOOK" pkey="@id" >
				<field name="@id" DBfield="id" />
				<field name="name" DBfield="name"/>
			</node >
		</xdfl:PASSIVE >
	</xdfl:_CLASS_DECLARE >
</XDFL>
```

Note that the object extends xtn.dbobject which is a base class provided by the framework to make database object creation easier. For more on Database Objects, look at the XDFL engine documentation.

Now that we defined the class and the object, we can start displaying all the books from the BOOK table in a view.

✍ The view script

/site/xdfl/book.tabview.xdfl

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

	<xdfl:OUTPUT xml-header="1" >

		<VIEW_DEFINITION>

			<CONFIG>
				<title>Books</title>
				<xsl>../xsl/book.xsl</xsl>
				<pagesize>50</pagesize>
			</CONFIG>

			<MENU >
				<title-bar icon="../images/book.gif" label="Books"/>
				<menu-bar>
					<menu-navpage/>
				</menu-bar>
			</MENU>
```

```
                        <SEARCH>
                                <label>Search</label>
<search-field name="@id"    label="id" op="eq" />
<search-field name="name" label="name" op="~*" /> <!-- use op="find" for ORACLE -->

                        </SEARCH>

<DATA>
        <input-request url="../xdfl/xtn.class_wrapper.xdfl" classname="book" action="get" flat="1">
                <request>
                        <order field="name" op="ASC"/>
                </request>
        </input-request>
</DATA>
                </VIEW_DEFINITION>
        </xdfl:OUTPUT >
</XDFL>
```

EXPLANATION

Although it may seem a little confusing at first sight, this views actually simply displays all the lines of the table. The CONFIG section deals with configuration options, the MENU section define what the menu tab will contains, the SEARCH shows search tools and the DATA section tells the framework how to get the view's data.

This definition will be used by the XDFLEngine and the client side of the framework to build the final screen.

The DATA section is where information for data's retrieval is found. What the "input-request" line says is that the "get" method of the "book" class will be called. A filter on the data is added within a "request" tag for ordering the data by the "name" field, in an ascending manner. But then, once the data is retrieved in the form of a XML stream and appended to the DATA node, how to display it ?

Well, as you may have noticed, a XSL file is referenced in this definition in the CONFIG part of the file : book.xsl.

This file is the view's style sheet : it says how the view should deal with the XML Data sent to the page, how to display it.

✍ Simplified xtn.class_wrapper.xdfl

Traditional xtn.class_wrapper.xdfl file uses the security model to check whether the user has the rights to call the class' method or not. This simplified version does not check anything and simply return the method's result. Any real-world application should obviously not use this class wrapper.

In order to use this class wrapper, simply replace the content of xtn.class_wrapper.xdfl by the following (backup the file first) :

/site/xdfl/xtn.class_wrapper.xdfl (simplified, no security check)

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

<XDFL xmlns:xdfl="xdfl">

        <xdfl:OUTPUT xml-header="@@[VAL:ARG.xml_header]@@">
                <xdfl:MODULE_EXEC class="@@[VAL:ARG.classname]@@" name="@@[VAL:ARG.action]@@" >
                        <xdfl:INPUT filter="@@[VAL:ARG.filter]@@"/>
                </xdfl:MODULE_EXEC >
        </xdfl:OUTPUT >
</XDFL >
```

As you see, this script simply executes the module whose name's given in parameters in the correct class.

✍ The XSL script

The XSL script is used to parse view's data (in this case, books) and transform the XML data into displayable HTML that will be inserted into the page.
In this XSL, we want books to be displayed as lines in a table. Table has two columns, one called "BOOK ID" and the other called "TITLE".
"Class" attributes reefers to CSS looks. Those CSS looks are defined in the "custom.css" file.

In order to understand this code, you have to be familiar with the XSL/T technology (good tutorials can be found on the web).
Note that the structure of what's under "DATA" is given by the definition of the "book" object in the book.class.xdfl file.

/site/xsl/book.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">

		<table class="book_container">
			<tr class="book_line">
				<td class="book_column" style="width:20%;">BOOK ID </td>
				<td class="book_column" style="width:80%;" >TITLE </td>
			</tr>
			<xsl:apply-templates select="book"/>

		</table>


	</xsl:template >

	<xsl:template match="book">

		<tr class="book_line">
			<td class="book_cell" style="width:20%;">
				<xsl:value-of select="@id"/>
			</td>

			<td class="book_cell" style="width:80%;">
				<xsl:value-of select="name"/>
			</td>

		</tr>

	</xsl:template >


</xsl:stylesheet>
```

✍ The "custom" CSS file

Custom.css file stores custom CSS looks for your views. All custom styles for all pages are stored in this file. In this example, the custom.css file contains "book_column", "book_line", "book_cell" and "book_container" looks.

/site/css/custom.css

```
.book_column
{
	background-color:#809090;
	font-weight:bolder;
	COLOR : black;
	PADDING:2px;
	height:10px;
	BORDER:inset 2px;
```

```
        BORDER-COLOR :#809090;
        FONT-SiZE : 12;
        FONT-FAMILY:verdana;
}

.book_line
{
        background-color:#808096;
        BORDER:outset 1px;
}

.book_cell
{
        background-color:#CFCFCF;
        BORDER:inset 1px;
}

.book_container
{
        background-color:#D0D0D9;
        width:100%;
}
```

✍ The initialization script

Initialization script loads libraries and creates values used all the time, such as aliases.
First value that should be defined is the “ROOT” value indicating the root directory of the application (make sure to set it to the path you chose for the application).

/site/init.xdfl

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

<XDFL xmlns:xdfl="xdfl">
    <!-- APP CONSTANTS -->

        <xdfl:VAL context="APPLICATION"        name="ROOT" value="F:/XMLware/Applications/books/root" />


         <!-- XDFL MODULES            -->
        <xdfl:_MODULE_LOAD path="@@[VAL:ROOT]@@/bin/xsltModuleStreamer_win32.1.1.dll" />
        <xdfl:_MODULE_LOAD path="@@[VAL:ROOT]@@/bin/javascriptModuleStreamer_win32.1.1.dll" />

        <!-- POSTGRESQL-->
        <xdfl:VAL context="APPLICATION"        name="DSN"                              value="dsn=test_pg;" />
        <xdfl:_MODULE_LOAD path="@@[VAL:ROOT]@@/bin/dbModuleStreamer_odbc_win32.1.1.dll"
connection_life_time="360000" sql_syntax="PGSQL"/>
        <xdfl:_ALIAS name="SQL_ODBC" alias="SQL"/>
        <xdfl:_ALIAS name="DB_GET_ODBC" alias="DB_GET"/>
        <xdfl:_ALIAS name="DB_SET_ODBC" alias="DB_SET"/>
        <xdfl:_ALIAS name="DB_AUTODESC_ODBC" alias="DB_AUTODESC"/>
        <xdfl:_ALIAS name="_DBDEF_COMPILE_ODBC" alias="_DBDEF_COMPILE"/>

        <!-- COMPILED -->
        <xdfl:SCRIPT_EXEC file="@@[VAL:ROOT]@@/classes/xtn.dbobject.class.xdfl"/>
        <xdfl:SCRIPT_EXEC file="@@[VAL:ROOT]@@/classes/book.class.xdfl"/>

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

</XDFL>
```

This initialization script is for windows. For UNIX, you would have to load corresponding modules (same names, but not containing “win32”, nor ending with “.dll”, and starting with “lib”).
In case you don’t use POSTGRESQL but ORACLE, then load the corresponding DB library.

✍ Default.htm

Here is the default.htm page to which your web site should point. This code simply opens the view.html template with correct parameters (such as path to the definition script).

/site/default.htm

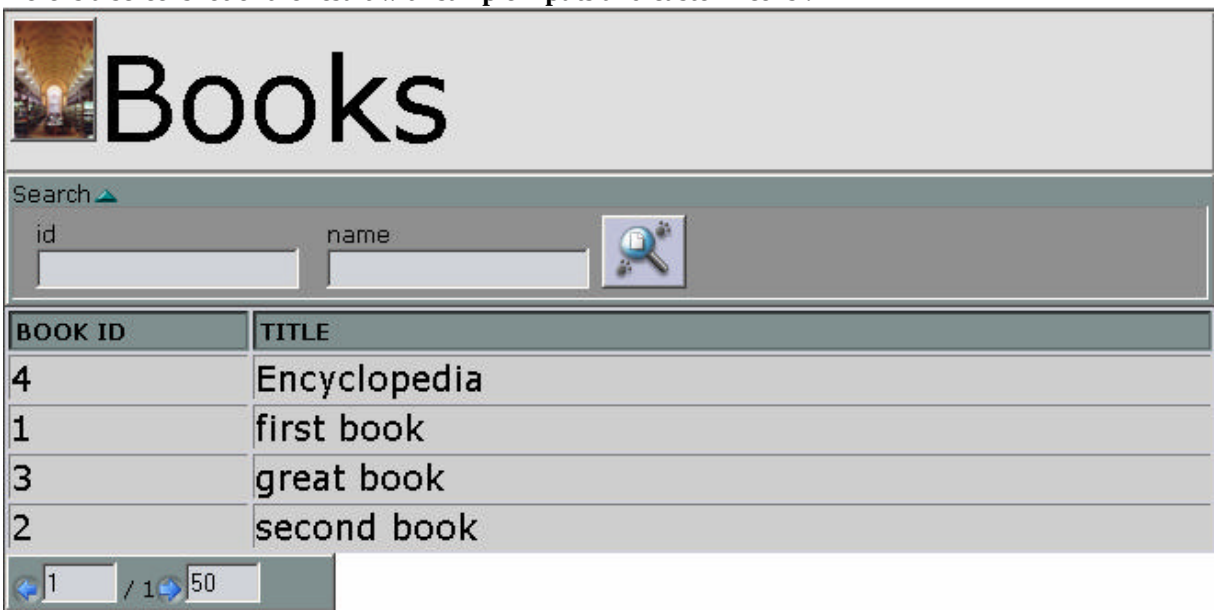
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>

   <head>
      <title>Books</title>
   </head>

         <frameset name="app" longdesc=0 framespacing=0  border=0 frameborder=0 >
                  <frame src="./html/view.html?def=../xdfl/books.tabview.xdfl " width="350px;"  name="view"
marginheight=0 marginwidth=0 framespacing=0  border=0 frameborder=0 scrolling="yes">
         </frameset>
</html>
```

✍ Others :

Modified CSS files : view.css, common.css.
Those files handle colors, font sizes and spaces for templates elements. Feel free to modify those files (but be sure to keep a backup of the original ones).

✍ Result :

Here is a screenshot of the result with sample inputs and custom icons :



You can see the four pre-mentioned elements of the view : the title, the search panel, the list of entries, and the navigation panel (from top to bottom).

✍ What you may wonder :

✍ How come my screen don't look exactly the same, even when modifying the view and common CSS files ? How come your navigation panel is at the bottom of the screen ?

The answer is simple : I cheated. I modified the view template. You will find how to customize the framework to get this result in the sub-chapter "hack them" of the "Views"

chapter. In your result, the navigation bar should be on top of the search pane, and there should be spaces between panes.

- ✍ **How did we actually *get* the data ? where is the SQL ?**

That's the beauty of database objects (see the getting started with XDFLEngine guide and Programming Guide for more on it) : once the object is defined, the DB_SET and DB_GET tags can automatically generate SQL requests. Those DB_SET and DB_GET tags are in the xtn.dbobject.class.xdfl file. Extending this class makes it even simpler, since all you have to do is to call the (inherited) get and set modules of your class.

XMLWare Forms templates

This chapter shows you how to use templates and how templates work in order to be able to customize them to suit your needs.
Templates are web pages models that use a XDFL script as definition.
Using a template is done by calling a special html page (for example “form.html” or “view.html”) with the correct parameters (most important of which is the name of the XDFL script that defines template’s parameters).
The XMLWARE FORMS framework mainly consists of JavaScript scripts and two templates: views and forms. Basically, views are meant to list data and choose elements to edit, and forms are meant to read and update data. Another template is the tree-view, which provides access to main screens and is meant to stand in a side frame.
Views are simpler than forms, and tree-view is the simplest template, so let’s start with this one.

Note :
The list of template tags and elements may (slightly) change with new releases. In order to make sure you can use a tag, you are encouraged to search for template’s elements matching in the .xsl files. The “common.xsl” file defines elements that are common to all templates (such as menu bars).

Tree view

Tree-view is the simplest template. Its main role is to let the user access application features. It is meant to be used the same way as the left frame of the Windows explorer, with nodes and leafs (folders and files). Nodes being group of features, and leafs being user screens.

✍ *Sample analyzed*

Here is a sample tree view from a real application (names have been changed) :

sample.treeview.xdfl

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

<XDFL xmlns:xdfl="xdfl">


<xdfl:OUTPUT xml-header="1" >
        <TREE_DEFINITION>

                <JAVASCRIPT>

                </JAVASCRIPT>

                <TREE_LAYOUT>

                <node name="Services" opened="1">
                        <leaf name="Real-estate Managers" icon="../images/sec.group.16.gif"
url="../html/view.html?def=../xdfl/sample.module1.search_building.tabview.xdfl" session_id='@@[VAL:ARG.session_id]@@'
target="view" />
                        <leaf name="Services details " icon="../images/ properties.16.gif"
url="../html/view.html?def=../xdfl/sample.module2.search_building.tabview.xdfl" session_id="@@[VAL:ARG.session_id]@@"
target="view"/>
                        <leaf name="Exit..." icon="../images/back.16.gif" url="../html/endsession.asp?end=1"
session_id="@@[VAL:ARG.session_id]@@"/>
                        </node>
                </TREE_LAYOUT>

        </TREE_DEFINITION>
</xdfl:OUTPUT >

</XDFL>
```

and here is the output for this screen :

Sample treeview output

The “services” folder folds and unfold when you click on it. Clicking on leafs (“building managers”, “services details” and “exit”) launch the corresponding screen/action.

Understanding the definition is now quite obvious. “Nodes” in the definition correspond to items that folds and unfolds when you click on it, and “leafs” link to screen and actions. You can see by reading the definition, that both “Building Managers” and “Services details” link to views, and that clicking on “Exit” launch the endsession.asp script (the application only had to run on windows clients, so it was perfectly ok to use asp scripts to handle sessions).

- ***Use them***

CALLING TREE VIEWS

In most case you will use tree views in frames either on the right or the left of your application screen. Here is a line sampled from a default.htm file of a real application :

Calling a tree view from an html :

```
<frame src="./html/treeview.html?def=../xdfl/sample.treeview.xdfl" name="tree" marginheight=0 marginwidth=0
framespacing=0  border=0 frameborder=0 scrolling="no">
```

As you see, calling a tree view is done by calling the treeview.html file with the "def" URL parameter set to the name of the XDFL script defining the tree view. The treeview.html file contains JavaScript scripts that will read the definition and create the corresponding html code. Note that it is not really the XDFL script itself that is sent back to the client, but rather its result (more on that in the next section). That means you can decide, on the server side, whether a user should see a leaf based on some conditions (using <xdfl:IF> tags).

1. Treeviews tags / elements

? **<TREE_DEFINITION>** : this tag contains the tree definition.
- **<JAVASCRIPT>** : this tag contains custom JavaScript definitions for scripts used in the tree view.
- **<CONFIG>** : contains all configuration tags. Parameters and sub-nodes :
  - **<debug>** : if set to 1, display debug buttons for displaying DOM Trees for the page.
  - All other tags are added to the array of configuration parameters. Tree views use none (views and forms do make use of some).
- **<MENU>** : in case you want to add menu to the tree view. Parameters and subnodes :
  - **"@class"** : class attribute set the class of the div section containing menu elements.
  - **"@style"** : style attribute set the style of the div section containing menu elements.
  - **<title-bar>** : a title bar element. Parameters and sub-nodes :
    - **"@class"** : class attribute set the class of the table containing title elements.
    - **"@style"** : style attribute set the style of the table containing title elements.
    - **"@icon-class"** : set the class of the <img> element for the title-bar.
    - **"@icon-style"** : set the style of the <img> element for the title-bar.
    - **"@icon"** : url of the icon for the title-bar.
    - **"@label-class"** : class of the title-bar's label.
    - **"@label-style"** : style of the title-bar's label.
    - **"@label"** : label for the title-bar.
    - **Other** : a title-bar can contains MENU and all of its sub-nodes (recursive structure).
  - **<title-bar-icon>** : icon inside the title bar. Parameters and sub-nodes :
    - **"@src"**: icon (URL).
  - **<menu-bar>**: a menu bar. Parameters and sub-nodes :

- **"@class"** : class for the div section containing menu-bar elements.
- **"@style"** : style for the div section containing menu-bar elements.
- **Other** : a menu-bar can contains MENU and all of its sub-nodes (recursive structure).

? **<menu-button>** : a menu button. Parameters and sub-nodes :
- **"@action"** : JavaScript function to call when user clicks on the menu.
- **"@class"** : class for the button.
- **"@style"** : style for the button.
- **"@icon"** : icon for the menu (URL).
- **"@icon-class"** : class for the menu's icon.
- **"@icon-style"** : style for the menu's icon.
- **"@label"** : label for the button.
- **"@label-class"** : class for the button's label.
- **"@label-style"** : style for the button's label.

? **<menu-label>** : a menu label. Parameters and sub-nodes :
- **"@class"** : class for the label
- **"@style"** : style for the label
- **Other :** text for the label has to be put within the tag's opening and closing (text node).

? **<menu-select>** : a dropdown list menu element. Parameters and sub-nodes :
- **"@action"** : action for when an item of the list is selected.
- **"@searchfield-class"**: class for the dropdown list.
- **"@searchfield-style"**: style for the dropdown ilst.
- **<option>** : different values in the dropdown list. Parameters and sub-nodes :
  - **"@value"** : option's value.
  - **"@selected"** : if present, pre-select the option.
  - **other** : text node for the option set the text displayed for the option (or the option's name If you prefer).

Note : the <MENU>, <JAVASCRIPT> and <CONFIG> sections are common to all templates. Their definitions are in the "common.xsl" file.

- **<TREE_LAYOUT>** : contains the definition of the graphical elements for the tree view. Parameters and sub-nodes :

? **<node> or <NODE>** : a node contains nodes or leafs. Its main feature is to open and close itself upon user clicks. Parameters and sub-nodes :
- **<node> or <NODE>** : recursive structure.
- **"@name"** : name of the node being displayed in the tree.
- **"@opened"** : tells whether the node starts as opened (attribute set to 1) or closed (by default).

? **<leaf> or <LEAF>** : leaf of the tree that usually links to an URL. Parameters and sub-nodes :
- **"@url"** : url to call upon user click. Can be a link to a web page or a JavaScript.
- **"@target"** : name of destination frame.
- **"@icon"** : icon for the leaf (URL).

- ✍ **"@name"** : name for the leaf (text that will be displayed).
- ✍ **other :** all other attributes will be append to the call's URL.

✍ *Hack them*

BUILDING PROCESS :

This section details the whole building process from the treeview xdfl definition to its html result.

client-side

1. URL entered by the client : src="./html/treeview.html?def=../xdfl/treeview.xdfl"
2. loading treeview.html.
3. tree_init() is called after treeview.html page has loaded *(treeview.js)*
4. check client parameters, such as browser version. *(common.js)*

✍ *loading definition*

5. get the name of the definition for the treeview (in this case "../xdfl/treeview.xdfl") *(common.js)*
6. tells the server to execute the definition and send the result back. *(common.js)*

server-side

7. Executes treeview.xdfl and send the result back to the client. Basically, result is everything between the two <xdfl:OUTPUT> tags. This part is pure XDFLengine. For anything related to it, please reefer to XDFLEngine documentation.

client-side

8. Parse result as a DOM tree and store the tree in the g_XMLDEF variable. *(common.js)*

✍ *initializing display*

9. Get the name of the xsl that we will use for displaying in the "style" URL parameter or use "treeview.xsl" if none is provided. In our case none is provided since we want to use treeview.xsl. *(common.js)*
10. Get the xsl from the server. *(common.js)*

server-side

11. Return asked xsl file

client-side

12. Parse returned xsl file as a DOM tree and store this definition in the g_XMLSTYLE variable. *(common.js)*
13. Apply the XSL to the definition and load the result into the document. *(common.js)*

That all !

CUSTOMIZATION :

Now let's try to customize this template. Since there isn't much to do we'll modify the look of the tree to show how it's build. Tree is made with tables, so displaying tables border will help us seeing the structure of the tree.
This information could be in two places : either in a CSS file, or directly within the treeview.xsl file. Let's have a look at the xsl.
What we see is two different templates matching : one for nodes and another one for leafs. We look right under the "match="node|NODE"" line and find a table with a border attribute set to "0". Let's change that to "1". Victory ! we now see the underlying structure for tree nodes. By taking a closer look we can see that "lines" are actually table cells with gray/black background color and 1 pixel width.

But let’s make sure that’s the way it works : let’s open the common.css file. We see a .treeline CSS class with a single attribute : background-color. Let’s change this attribute to #FF0000. We changed the color of the lines to a bright red !

That was only minor customization, but the tree view template is too small to be really customizable.

Views

View is the simplest of the two remaining templates. Yet it is a lot more complex than tree views. It uses more JavaScript, more DOM trees, and has more user interactions. This chapter supposes you've already read the tree view chapter and understood it.

✍ *Sample analyzed*

Here is a sample view to analyze (once again, sampled from a real world application, with names changed). This sample is purposefully much more complex than the view of our "first screen" chapter. It shows how a template can be used along with regular XDFL code, on the server side, to achieve complex results.

Complex view example :

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

<XDFL xmlns:xdfl="xdfl">

<xdfl:MODULE_EXEC class="GLOBAL" name="get_user"/>

<xdfl:VAL name="is_authorized" value="0"/>

<xdfl:IF val1=",REAL_ESTATE_MNGT," val2=",@@[VAL:auth_user.categories]@@," op="in">
        <xdfl:VAL name="is_authorized" value="1" />
</xdfl:IF>

<xdfl:BUF_SET create="1" name="perimeters">
        <xdfl:SQL statement="SELECT CODE as XML_ATTR_value, NAME as XML_TEXT FROM O_PARAM WHERE
DOMAINE='PERIMETER'" enclose_record="option" connection="@@[VAL:DSN]@@"/>
</xdfl:BUF_SET>

<xdfl:IF val1="@@[VAL:is_authorized]@@" val2="0" op="eq">
<xdfl:OUTPUT xml-header="1">
        <VIEW_DEFINITION>
        <CONFIG>
                <title>ACCESS DENIED</title>
                <xsl>../xsl/optimmo.operation.view.xsl</xsl>
                <pagesize>50</pagesize>
                <width>800</width>
                <height>500</height>
                <debug>0</debug>
                <begin-with-data>0</begin-with-data>
        </CONFIG>
        <MENU >
                        <title-bar icon="../images/error.gif" label="You don't have the rights to use this application.
Please contact the system administrator."/>

        </MENU>
        <DATA>

                <input-request url="../xdfl/xtn.class_wrapper.xdfl" classname="optimmo.search_operation"
action="get_extended" >
                        <request>
                                <filter field="fk_charge">-1</filter>
                        </request>
                </input-request>
        </DATA>
        </VIEW_DEFINITION>
</xdfl:OUTPUT>
</xdfl:IF>

<xdfl:IF val1="@@[VAL:is_authorized]@@" val2="1" op="eq">
<xdfl:OUTPUT xml-header="1" >
        <VIEW_DEFINITION>

                <CONFIG>
```

```
                        <title>Building</title>
                        <xsl>../xsl/operation.view.xsl</xsl>
                        <pagesize>50</pagesize>
                        <width>800</width>
                        <height>500</height>
                        <debug>0</debug>
                        <begin-with-data>0</begin-with-data>
                </CONFIG>

                <JAVASCRIPT>
                <![CDATA[
                        function on_click_operation(id_operation,id_budget_operation)
                        {
        nav_openForm("../xdfl/optimmo.budget_operation.form.xdfl",id_operation,"update","id_budget_operation="+id_bu
dget_operation+"&year=@@[VAL:ARG.year]@@")
                        }

                        function on_click_help()
                        {
                                window.open("../html/help.html","help","height=370,width=520,top=200,left=300")
                        }
                ]]>

                </JAVASCRIPT>


                <MENU >
                        <title-bar icon="../images/search.16.gif" label="List of operations on this building"/>
                        <menu-bar>
                                <menu-button icon="../images/msg.gif" label="Help" action="on_click_help()" />

                                <menu-navpage/>
                        </menu-bar>

                </MENU>

                <SEARCH icon="../images/test.gif" >
                        <label>Search</label>
                        <search-field name="@id_operation" label="ID" style="width:20px;" op="eq" />
                        <search-field name="l_operation" label="Name"  style="width:20px;" op="find" />
                        <search-field name="fk_perimeter" label="Perimeter" style="width:20px;" op="eq" >
                                <option>All</option>
                                <xdfl:BUF_GET name="perimeter"/>
                        </search-field>
                        <search-field name="statut_vent_n" label="Statut" style="width:20px;" op="eq" >
                        <option>All</option>
                        <option value="1">filled</option>
                        <option value="0">not filled</option>
                        </search-field>
                        <reload label="reload page" type="link" class="label_italic_small" style="float:right;"/>
                </SEARCH>


                <DATA>
                        <input-request url="../xdfl/xtn.class_wrapper.xdfl" classname="optimmo.search_operation"
action="get_extended" >
                                <request>
                                        <filter field="fk_charge">@@[VAL:auth_user.login]@@</filter>
                                        <o_budget_operation>
                                                <filter
field="year_budget_operation">@@[VAL:ARG.year]@@</filter>
                                        </o_budget_operation>
                                </request>
                        </input-request>
                </DATA>
                </VIEW_DEFINITION>
</xdfl:OUTPUT >
</xdfl:IF>

</XDFL>
```

The first thing that you may notice is the amount of XDFL code before the xdfl:OUTPUT tag. This is often the case as soon as you start to deal with user authorisations.
First lines of code check whether the user has the rights for accessing this application. The application is called "REAL_ESTATE_MNGT", and the "get_user" module loads user rights in a value. If the user rights include "REAL_ESTATE_MNGT" then the script displays the view, else it displays an error page. You can find more on that in the chapter detailing the security model.
Next lines loads entries of a table in a buffer (perimeters) that will be used within the definition as content for a dropdown list (at <search-field name="fk_perimeter"…><xdfl:BUF_GET …>).

This view wouldn't be complete without the XSL. Yet because this XSL is *very* long (mainly because of XSL string formatting, which is not the purpose here), we will only show the new parts (new compared to the example in the "first script" chapter) :

Complex XSL example (extract) :

```
<xsl:apply-templates select="DATA"/>
	</xsl:template>
	<xsl:template match="DATA">
	<table cellpadding="0" cellspacing="0" class="tabTable"> <!-- 3 lignes, 1 colonne -->
		<tr>
		<td style="width:100%;">
		<table cellpadding="0" cellspacing="0" class="tabTable"> <!-- 1 ligne, n colonnes headers -->

			<tr>
				<td class="tabHeader" style="width:20px;"><a href="javascript:view_setOrder(
'@id_operation')">ID</a></td>
				<td class="tabHeader" style="width:100px;"><a href="javascript:view_setOrder(
'l_operation')">Name</a></td>
				<td class="tabHeader" style="width:20px;"><a href="javascript:view_setOrder(
'amount')">Amount </a></td>
				<td class="tabHeader" style="width:40px;"><a href="javascript:view_setOrder(
'd_start')">Start date</a></td>
				<td class="tabHeader" style="width:20px;"><a href="javascript:view_setOrder(
'd_end')">End date</a></td>
				<td class="tabHeader" style="width:20px;"><a href="javascript:view_setOrder(
'perimeter')">Perimeter</a></td>
			</tr>
		</table>
		</td>
		</tr>
		<tr>
		<td style="width:100%;">
		<table cellpadding="0" cellspacing="0"> <!-- p lignes, 1 colonne -->
			<xsl:apply-templates select="o_operation"/>
		</table>
		</td>
		</tr>
	</table>
	</xsl:template >
```

This extract displays the headers for the columns. Columns are "ID", "Name", "Amount", "Start date", "End date", "Perimeter". A new feature with this XSL is the use of the view_setOrder() function *(view.js)*. This function enables the user to sort entries by clicking on the column. Notice that all you have to do to use this function is to provide it with the name of the column you want to sort entries on, upon user click (when used in a href HTML tag).

- ***Use them***

CALLING VIEWS

View are called using either regular html syntax, or by using the framework's nav_openview() function *(common.js)*. Entering the full URL of the view is usually done in default.htm file, and the nav_openview function is used mainly from within other pages (tree views, views and forms).
Here are some examples :

Calling a view in html

```
<frame src="./html/view.html?def=../xdfl/operation.view.xdfl&id=A001" name="view"  marginheight=0 marginwidth=0
framespacing=0  border=0 frameborder=0 >
```

This is pretty straightforward : just like tree views, you call a view by calling view.html file with a "def" argument equals to the path to the view. Additional parameters can be passed to the view's URL and then used in the view's XDFL script using values. Example could be "@@[VAL:ARG.name_of_parameter]@@" for reading the value using parsed expression (see XDFLEngine documentation for more on parsed expressions).

Using nav_openview

```
if ( window.confirm(msg))
        {
        nav_openView( "../xdfl/nav.view.xdfl" , "mode=choose_dir"+g_amp+"doctype_id=-1"+g_amp+"title=Document
Path"+g_amp+"callback=formdoc_move2('RETURN_VAL')", "_blank")
        }
```

This sample is extracted from the custom JavaScript section of a form. Upon user action (probably on a button), it asks confirmation for the user and then loads the view called "nav.view.xdfl". Apparently this view uses at least 4 parameters : "mode", "doctype_id", "title", "callback". Third parameter of the call, "_blank", means we want to open the view as a pop-up in a new window.

VIEWS TAGS/ELEMENTS

View has all the elements common to all templates. Those are : CONFIG, JAVASCRIPT, DEBUG and MENU elements. You will find their description in the tree view template chapter. Those elements are used within VIEW_DEFINITION element.

- **<VIEW_DEFINITION>** : this tag contains the tree definition.
  - **<SEARCH>** : Search panel definition.
    - **label** : title for the Search panel (usually set to "Search").
    - **@label-class** : class for the label
    - **@label-style** : style for the label
    - **@opened** : set to 0 for loading the search panel in "closed" state. Default state for the panel is "opened".
    - **search-field** : element for defining a search field.
      - **@name** : name of the field (this name is the name of the field as defined in the database object, which may not be necessarily the same as the name in the database).
      - **@class** : class for the element containing the html code for the search field.
      - **@style** : style for the element containing the html code for the search field.
      - **@label** : label for the field.

- **@label-class** : class for the label.
- **@label-style** : style for the label.
- **@searchfield-class** : class for the search field (the input itself, not the label). This attribute is also used as class for the dropdown list.
- **@searchfield-style** : style for the search field (the input itself, not the label). This attribute is also used as style for the dropdown list.
- **Option** : options are used when the search field is not an input but should instead be a list of values (options). Every option element/tag under a search-field is a value in the list.
  - **@value** : value for the element of the list.
  - **Other** : input text for the option element is displayed as label for the corresponding value

? **reload** : display a button or a hypertext link for reloading the view with default parameters. Note : this element may not be present in all release.
- **@type** : either “button” or “link”.
- **@class** : class for the button or link.
- **@style** : style for the button or link.
- **@src** : URL for the icon (button only).
- **@label** : text for the link (link only).

Additional sub-elements : those elements are additional sub-elements of common elements added to the view template only.

? **MENU/menu-bar/menu-navpage** : menu navigation panel. This element enables the user to navigate among entries list.
- **@class** : class for the panel
- **@label-style** : style for the panel’s label.
- **@label-class** : class for the panel’s label.
- **@icon-class** : class for the “previous” and “next” icons.
- **@icon-style** : style for the “previous” and “next” icons.
- **@pagenum-class** : class for the input displaying the page number.
- **@pagenum-style** : style for the input displaying the page number.
- **@pagesize-class** : class for the input displaying the page size.
- **@pagesize-style** : style for the input displaying the page size.

? **MENU/menu-bar/menu-display** : display preference dropdown list element. This element displays a dropdown menu containing a list of display preferences. In practice, what this menu does is to allows the user to choose which XSL is going to be applied for displaying entries list *(experimental feature)*.
- **@class** : class for the element div.
- **@style** : style for the element div.
- **@icon-class** : class for the element’s icon.
- **@icon-style** : style for the element’s icon.
- **@label** : label for the element.
- **@label-class** : class for the label.
- **@label-style** : style for the label.
- **@select-class** : class for the dropdown list.
- **@select-style** : style for the dropdown list.

- ✍ **option** : every option sub-element adds a line in the dropdown list. Clicking on the option select the corresponding XSL.
  - ✍ **@url** : URL to the XSL for this option.
  - ✍ **@name** : name for this option.

? **CONFIG/begin-with-data** : tells whether the view loads data at load time or wait for the user to click on "search" before doing so

? **CONFIG/on_data_load** : sets the name of the JavaScript that should be called after initializing the view and every time data has to be loaded.

✍ *Hack them*

BUILDING PROCESS :

This section details the whole building process from the view XDFL definition to its html result.

client-side

1. URL entered by the client : src="./html/view.html?def=../xdfl/myview.xdfl", or nav_openView called.
2. loading view.html.
3. view_init() is called after view.html page has loaded *(view.js)*
4. check client parameters, such as browser version. *(common.js)*

✍ *loading definition*

5. get the name of the definition for the view (in this case "../xdfl/myview.xdfl") *(common.js)*
6. tells the server to execute the definition and send the result back. *(common.js)*

server-side

7. Executes myview.xdfl and send the result back to the client. Basically, result is everything between the two <xdfl:OUTPUT> tags. This part is pure XDFLengine. For anything related to it, please reefer to XDFLEngine documentation.

client-side

8. Parse result as a DOM tree and store the tree in the g_XMLDEF variable. *(common.js)*

✍ *initializating display*

9. Get the name of the xsl that we will use for displaying in the "style" URL parameter or use "view.xsl" if none is provided. In our case none is provided since we want to use view.xsl. *(common.js)* Warning : this xsl is used for displaying view elements. It should not be mixed up with the XSL used to display entries.
10. Get the view.xsl from the server. *(common.js)*

server-side

11. Return asked xsl file

client-side

12. Parse returned xsl file as a DOM tree and store this definition in the g_XMLSTYLE variable. *(common.js)*
13. Apply the XSL to the definition and load the result into the document. *(common.js)*

✍ *getting view parameters and custom xsl*

14. Get the "pagesize" configuration parameter of the definition. *(view.js)*
15. Get the name of the xsl used to display entries and ask it to the server. *(view.js)*

server-side

16. Return XSL

client-side

17. Parse XSL as a DOM tree and stores the result into the gview_xmlCurrentXSL variable.*(view.js)*

    ✍ *listing view's data*

18. Read the input-request node and send request for data by calling module of a class whose name's defined in this node.*(view.js)*

server-side

19. Receive module call. Call the module and send result back to the client

client-side

20. Receive result of module call, parse it as a DOM tree and store this tree in the gview_XMLDATA global variable.*(view.js)*
21. Builds the PAGE DOM tree as a subset of the g_XMLDATA DOM tree and stores it into the gview_XMLPAGE variable. This DOM tree contains only the subset that has to be displayed, based on the number of entries per page configuration parameter.*(view.js)*
22. Apply the gview_xmlCurrentXSL XSL to the Page.*(view.js)*
23. Load result into the web page.*(view.js)*

As you see, views are a lot more complex to build than tree views. This is fairly normal since we want to display data loaded dynamically whereas tree views only had to display a static definition.

CUSTOMIZATION

Now that we are more familiar with views, let's see how I managed to change the look of the view in our "first screen" chapter. Main things that we changed and that may be hard to find on your own is : the navigation panel is on the bottom of the screen, there are no space between elements of the view. Let's start with this latest one :

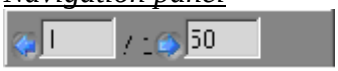
No Spaces between elements

Almost every graphical element in the view is associated with a CSS class. Only thing you have to do when customizing the look of those elements is to find the corresponding class and modify it. Let's have a look at the view.xsl file, since that's the place where you create the HTML for the view and tell which elements uses which class :
First thing you see is the following line :

```
<xsl:import href="../xsl/common.xsl"/>
```

What that means is that the view.xsl file also use common.xsl. We'll have to look into this file as well…
Basically, what we see when looking into those files is that every section in the view's definition is mapped to a HTML . This div has a class attribute (ending with "_ctnr" for container), and this class is defined in either common.css or view.css. Plus, all those sections are embedded into a HTML element with class "whole_ctnr". Thus, removing spaces between elements simply consists in removing all "padding", "margin" or "border" properties in the containers' CSS classes (whole_ctnr, title_bar_ctnr).

Navigation panel



In the view template, this panel is supposed to be right under the title bar, on top of the search bar. In our example, we wanted to put this pane under the result. Problem is, this panel can be

defined in a <menu-bar>, which is in the <MENU> section, and this section is in the "top_ctnr" div, which is on the top of the page. So, the only solution is to create a new section in the definition that will be mapped to a "bottom_ctnr" displayed at the bottom of the screen, and fill it.

That leads us to modifying the view.xsl and common.xsl files :

common.xsl : we want to add the bottom container into the "common" template which will look this way (new lines in red) :

```
<xsl:template name="common">
	<body>
	<xsl:apply-templates select="JAVASCRIPT"/>
	<div class="whole_ctnr" >
		<xsl:if test="MENU">
		<div class="top_ctnr">
			<xsl:apply-templates select="MENU"/>
		</div>
		</xsl:if>
		<div>
			<xsl:call-template name="content"/>
		</div>
		<div class="bottom_ctnr">
			<xsl:apply-templates select="BOTTOM-BAR"/>
		</div>
		<xsl:if test="CONFIG[debug=1]">
			<div>
				<xsl:call-template name="debug"/>
			</div>
		</xsl:if>

	</div>
	</body>
</xsl:template>
```

As you see, we decided to call the section "BOTTOM-BAR". And we also decided to put the corresponding section right after before the DEBUG div.
We now have to create the template for handling "BOTTOM-BAR" tags.
That's the job of the view.xsl. Simply add the following lines to this file (anywhere between two templates definition) :

```
<!-- BOTTOM BAR -->
<xsl:template match="BOTTOM-BAR">
	<xsl:apply-templates select="menu-bar" />
</xsl:template>
```

That's it !
All we wanted was to use the menu-bar element and sub elements.

And now let's use this new section in the view definition (new lines in red) :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<XDFL xmlns:xdfl="xdfl">
	<xdfl:OUTPUT xml-header="1">
		<VIEW_DEFINITION>
			<CONFIG>
				<title>Books</title>
				<xsl>../xsl/book.xsl</xsl>
				<pagesize>50</pagesize>
				<debug>0</debug>
			</CONFIG>
			<MENU>
				<title-bar icon="../images/library.jpg" icon-style="border:outset 2px;" label="Books"
label-style="font-size: 64;margin:0;"/>
			</MENU>
			<SEARCH>
				<label>Search</label>
				<search-field name="@id" label="id" op="eq"/>
				<!-- POSTGRES VERSION -->
				<search-field name="name" label="name" op="~*"/>
```

```
                                        <!-- ORACLE VERSION :
                                        <search-field name="name" op="find" />
                                        -->
                                </SEARCH>
                                <BOTTOM-BAR>
                                        <menu-bar>
                                                <menu-navpage/>
                                        </menu-bar>
                                </BOTTOM-BAR>
                                <DATA>
                                        <input-request url="../xdfl/xtn.class_wrapper.xdfl" classname="book" action="get"
flat="1">
                                                <request>
                                                        <order field="name" op="ASC"/>
                                                </request>
                                        </input-request>
                                </DATA>
                        </VIEW_DEFINITION>
                </xdfl:OUTPUT >
</XDFL>
```

We simply moved the menu-bar from the MENU section which is displayed on the top to the BOTTOM-BAR section which is displayed on the bottom.

Forms

Form is the most versatile template. There is very little chance that a user screen can not be done using forms. Form template is also meant to be extensible through the use of nuggets. Nuggets can be seen as pieces of interface that you can assemble in order to build your page. Thus whenever possible, you shouldn't hack the template but rather add new nuggets. The next chapter will deal with nuggets.

✍ *Sample analyzed*

This form is a form used in the Banjan project. It is a an administration form used for creating users and modifying their rights. Don't expect to understand everything in this form from the first time. What is important is to understand how forms builds the interface dynamically from the XML data it receives.

sec.user.form.xdfl

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


 <!--
 ========================================================================
 //
 // GEDEON security
 //
 //_____________________________________________________________________
 //  sec.user.form.xdfl
 //_____________________________________________________________________
 //! @file       sec.user.form.xdfl
 //! @brief      users management form
 //! @author     Guillaume Baurand
 //! @date       17/05/2004
 //! @version    1.1
 //_____________________________________________________________________
 // //======================================================================
-->


<XDFL xmlns:xdfl="xdfl">
<xdfl:MODULE_EXEC name="xtn.get_user" />
<xdfl:MODULE_EXEC object="USER" name="check_action" action="SECURITY"/>

<xdfl:OUTPUT xml-header="1" >
        <FORM_DEFINITION type="form">

                <CONFIG>
                        <nuggets>../javascript/nuggets.xml</nuggets>
                        <debug>0</debug>
                </CONFIG>

                <MENU >
                        <title-bar icon="../images/sec.user.gif" label="Utilisateur"/>
                        <menu-bar>
                                <menu-button icon="../images/ged.back.16.gif" label="Back"
action="history.back(1)"/>
                                <menu-button icon="../images/ged.save.16.gif" label="Save" action="formuser_save()"
/>
                                <menu-button icon="../images/ged.delete.16.gif" label="Delete"
action="formuser_delete()" />
                        </menu-bar>
                </MENU>

                <JAVASCRIPT>
                        <![CDATA[
                                //-------------------------------------------------------
                                function formuser_save( p_strAction, p_strArgs, p_strCallBack)
                                {

                                        var msg=""
```

```
                                        // checking fields
                                        if( formDATA_get( "user/@login")=="" )              msg+=" 'Login'  field missing
\n"

                                        if (msg!="")
                                        {
                                                alert("Form can't be saved :\n"+msg)
                                                return false
                                        }
                                        else
                                        {
                                                form_save( p_strAction, p_strArgs, p_strCallBack)
                                                alert("Save done.")
                                                return true
                                        }
                                }

                                //-------------------------------------------------------
                                function formuser_delete()
                                {

                                        if( form_getAction()==CT_FORMACTION_INSERT )
                                        {
                                                alert("Form must be saved before doing this action !")
                                                return
                                        }

                                        var msg=""
                                        msg+="User is going to be deleted .\n"
                                        msg+="\nContinue ?"

                                        if ( window.confirm(msg))
                                        {
                                                form_save('delete')
                                                alert("User deleted")
                                                history.back(1)
                                        }
                                }
                        ]]>
                </JAVASCRIPT>

                <LAYOUT>

                        <node select="secuser" >

                                <node class="ged_document_tab">

                                        <label class="ged_document_tab_title" label-
class="ged_document_tab_title_label">User</label>

                                        <node class="ged_document_tab_field">
                                                <label class="ged_document_tab_field_label">ID</label>
                                                <input select="@id" class="ged_document_tab_field_input_ro"
readonly="1" style="width:40px;"/>
                                        </node >

                                        <node class="ged_document_tab_field">
                                                <label class="ged_document_tab_field_label">Login</label>
                                                <input select="@login" class="ged_document_tab_field_input" />
                                        </node >
                                </node >

                                <node class="ged_document_tab" >

                                        <label class="ged_document_tab_title" label-
class="ged_document_tab_title_label">Groups</label>

                                        <node class="ged_document_tab_action">

                                                <action type="choose"
                                                        view="../xdfl/sec.group.tabview.xdfl" target="_blank"
viewargs="mode=choose"
                                                        class="ged_menu_tab_button_small"
                                                        icon="../images/ged.add.16.gif" style="float:left;" >
```

```
                                                    <xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
                                                        <xsl:output method="xml"/>
                                                        <xsl:template match="secgroup">

                                                            <secmembership action="insert" id=""
id_user="" id_group="">
                                                                <xsl:copy-of select="."/>
                                                            </secmembership>
                                                        </xsl:template>
                                                    </xsl:stylesheet>

                                                </action>
                                                <label style="width:200px;font-weight:bolder;" >Add
group</label>
                                            </node>

                                            <node select="secmembership" class="ged_document_tab_field">


                                                <action type="delete" icon="../images/ged.delete.16.gif"
class="ged_menu_tab_button_small" style="float:right;"/>

                                                <label type="icon" icon="../images/sec.group.gif"
                                                    class="ged_document_tab_element"
                                                    icon-class="ged_directory_icon"
                                                    label-class="ged_directory_label_doc" >
                                                    ###secgroup/@name###
                                                </label>

                                            </node>
                                        </node >
                                        <node class="ged_document_tab" >

                                            <label class="ged_document_tab_title" label-
class="ged_document_tab_title_label">Authorizations</label>

                                            <node class="ged_document_tab_action">

                                                <action type="choose"
                                                    view="../xdfl/sec.action.tabview.xdfl" target="_blank"
viewargs="mode=choose"
                                                    class="ged_menu_tab_button_small"
                                                    icon="../images/ged.add.16.gif" style="float:left;" >


                                                    <xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
                                                        <xsl:output method="xml"/>
                                                        <xsl:template match="secaction">
                                                        <action_grant action="insert" id=""
type_actor="2" id_actor="" type_secured="3" id_secured="" grant_level="1">
                                                                <xsl:copy-of select="."/>
                                                            </action_grant>
                                                        </xsl:template>
                                                    </xsl:stylesheet>
                                                </action>
                                                <label style="width:200px;font-weight:bolder;" >Add action
</label>
                                            </node >
                                        <node select="action_grant" class="ged_document_tab_field">


                                                <action type="delete" icon="../images/ged.delete.16.gif"
class="ged_menu_tab_button_small" style="float:right;"/>

                                                <input select="@grant_level" type="check"
class="ged_document_tab_field_input" style="float:right;" />

                                                <label type="icon" icon="../images/sec.grant.gif"
                                                    class="ged_document_tab_element"
                                                    icon-class="ged_directory_icon"
                                                    label-class="ged_directory_label_doc" >
                                                    ###secaction/@tag###
                                                </label>
```

```
                                        </node >

                                        <node class="ged_document_tab_action">

                                                <action type="choose"
                                                        view="../xdfl/sec.category.tabview.xdfl" target="_blank"
viewargs="mode=choose"
                                                        class="ged_menu_tab_button_small"
                                                        icon="../images/ged.add.16.gif" style="float:left;" >

                                                        <xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
                                                                <xsl:output method="xml"/>
                                                                <xsl:template match="seccategory">

                                                                        <category_grant action="insert" id=""
type_actor="2" id_actor="" type_secured="1" id_secured="" grant_level="">
                                                                                <xsl:copy-of select="."/>
                                                                        </category_grant>
                                                                </xsl:template>
                                                        </xsl:stylesheet>

                                                </action>
                                                <label style="width:200px;font-weight:bolder;" >Add
category</label>
                                        </node >


                                        <node select="category_grant" class="ged_document_tab_field">


                                                <action type="delete" icon="../images/ged.delete.16.gif"
class="ged_menu_tab_button_small" style="float:right;"/>

                                                <input select="@grant_level" type="select"
class="ged_document_tab_field_input" style="float:right;">
        <option><VALUE>0</VALUE><LABEL>------ Niveau ------</LABEL></option>
        <option><VALUE>1</VALUE><LABEL>Lecture</LABEL></option>
        <option><VALUE>2</VALUE><LABEL>Read / Write</LABEL></option>
        <option><VALUE>3</VALUE><LABEL>Administration</LABEL></option>
                                                </input >

                                                <label type="icon" icon="../images/sec.category.gif"
                                                        class="ged_document_tab_element"
                                                        icon-class="ged_directory_icon"
                                                        label-class="ged_directory_label_doc" >
                                                        ###seccategory/@name###
                                                </label>
                                        </node >

                                        <node class="ged_document_tab_action">

                                                <action type="choose"
                                                        view="../xdfl/sec.object.tabview.xdfl" target="_blank"
viewargs="mode=choose"
                                                        class="ged_menu_tab_button_small"
                                                        icon="../images/ged.add.16.gif" style="float:left;" >

                                                        <xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
                                                                <xsl:output method="xml"/>
                                                                <xsl:template match="secobject">

                                                                        <object_grant action="insert" id=""
type_actor="2" id_actor="" type_secured="2" id_secured="" grant_level="">
                                                                                <xsl:copy-of select="."/>
                                                                        </object_grant>
                                                                </xsl:template>
                                                        </xsl:stylesheet>

                                                </action>
                                                <label style="width:200px;font-weight:bolder;" >Add
object</label>
                                        </node >
```

```
                                        <node select="object_grant" class="ged_document_tab_field">


                                                <action type="delete" icon="../images/ged.delete.16.gif"
class="ged_menu_tab_button_small" style="float:right;"/>

                                                <input select="@grant_level" type="select"
class="ged_document_tab_field_input" style="float:right;">
        <option><VALUE>0</VALUE><LABEL>------ Rights ------</LABEL></option>
        <option><VALUE>1</VALUE><LABEL>Read</LABEL></option>
        <option><VALUE>2</VALUE><LABEL>Read / Write</LABEL></option>
        <option><VALUE>3</VALUE><LABEL>Administration</LABEL></option>
                                                </input >

                                                <label type="icon" icon="../images/sec.object.gif"
                                                        class="ged_document_tab_element"
                                                        icon-class="ged_directory_icon"
                                                        label-class="ged_directory_label_doc" >
                                                        ###secobject/@obj_type###
                                                        ###secobject/@obj_name###
                                                        ###secobject/@obj_id###
                                                </label>
                                        </node >
                                </node >
                        </node >
                        </LAYOUT>
                        <DATA>
                                <input-insert url="../xdfl/xtn.class_wrapper.xdfl" classname="sec.user"
action="get_default"  />
                                <input-update url="../xdfl/xtn.class_wrapper.xdfl" classname="sec.user"
action="get_by_id" id="@@[VAL:ARG.id]@@" />

                                <output-insert url="../xdfl/xtn.class_wrapper.xdfl" classname="sec.user"
action="insert" />
                                <output-update url="../xdfl/xtn.class_wrapper.xdfl" classname="sec.user"
action="update" />
                                <output-delete url="../xdfl/xtn.class_wrapper.xdfl" classname="sec.user"
action="delete" />
                        </DATA>
                        <ERRORS></ERRORS>
                </FORM_DEFINITION>
        </xdfl:OUTPUT>
</XDFL>
```

That seems to be a lot of code, yet if you copy this code into a good XML text editor, you will see that it isn't that big.

MAIN SECTIONS

First let's look at the commentary. Using commentary for describing the form is highly recommended even for the simplest ones. It is always good to know what the form takes as additional parameters, and what type of action it does (it is very common that forms call server-side modules to do a lot of side-actions such as sending a mail, compressing files, etc.)[5]. You may use any type of syntax for your commentaries as long as it is XML-compliant.
Next two important lines check the user rights, in order to know whether user on the client-side has the rights to use this form. Since this form is part of the SECURITY domain, the script only checks for SECURITY among all user's rights.
Then comes the form definition itself. All the form definition is contained between the FORM_DEFINITION opening and closing tags. The definition for this form includes five sections :

- the CONFIG section where you can see that debug is set to 0 (no debug information) and that the file containing all the nuggets for this form is "../javascript/nugget.xml" ;

[5] And as you may have noticed, this example is not enough commented !

- the MENU section sets the menu buttons for the form, in this example three buttons are used for returning to previous page, saving the entry, and deleting it ;
- the JAVASCRIPT section containing all custom JavaScript which basically check fields, save or delete user (depending on the form's action), and display custom messages ;
- the LAYOUT section is where all the look of the form is defined and is also the place where nuggets are used ;
- the DATA section is where you say what server-side script you want to call depending on form's action.

This last DATA section is the first you have to understand in order to be able to load and save information from forms. In our example you can see two "input" nodes and three "output" nodes.
Input nodes are used for loading data. In our example, you have two input nodes : one is for opening the form in "insert" mode and loads a default set of data (this default set of data being defined in the "get_default" module of the "sec.user" class), the other one is for opening the form in "update" mode, and loads the entry having "@@[VAL:ARG.id]@@" for id. This parsed value[6] is the value of the "id" parameter of the form's call. The form knows what input node it has to use depending on the value of another parameter of the call named "action". This parameter can be set to whatever you want, but in our case only the "insert" and "update" value will do something (since we only have "input-insert" and "input-update" nodes).
Output nodes are used for anything (could be saving data, inserting data, sending a mail, or whatever a server-side script of a class can do). In our case we have three output for inserting, updating and deleting entries. When calling form_save() JavaScript, the framework automatically call the module of the class whose name's given in the "output-action" node (e.g. : "output-update" when form's action is "update"). Note that you can override this behavior by specifying the name of the action yourself (as in the JavaScript custom function formuser_delete (), where the form_save() call has one "delete" parameter in order to use the "output-delete" node and delete the user currently in the form).
Note that this section is slightly more complex than with views, since views didn't have "action".

MENU, CONFIG and JAVASCRIPT sections are working the same way than with views.

FORM LAYOUT BUILDING ; NODES ; "SELECT" ATTRIBUTE

Nodes are nuggets and thus will be described in the nugget chapter. Yet they are so important to the way forms work that it is impossible to understand forms without knowing about them.
The LAYOUT section, as said right above, is a description of the form look. Yet forms are dealing with dynamic content. For example, let's suppose you want to display a line for every entry in a SQL table : you don't know in advance how many entries there's going to be. So how can you tell what you want the form to look like ?
Well, first you know one thing : the DATA structure is a XML tree. What you want the form to do is to map the DATA structure into a set of HTML elements for display. The way the framework deals with that is by mapping pre-built graphic elements (called "nuggets") to an XPath path relative to the DATA tree.
Let's see that in our example :

```
<node select="secuser" >
```

That is the first line in the LAYOUT section. What it says is : "for each secuser XML node in the DATA section, I want the form to build…". Everything inside this tag will be repeated for

[6] see the XDFLengine documentation for more about parsed values

every secuser node[7]. Since this tag is the first being mapped, the "secuser" XML nodes full XPath path would be "DATA/secuser".

Then the next time we have a node with a select attribute (that is, actually mapped) is :

```
<node select="secmembership" class="ged_document_tab_field">
```

This node nugget maps to secmembership DATA XML nodes (secmembership are user's membership to groups). Since this node is inside the previous node nugget, the full path for those "secmembership" XML nodes is "DATA/secuser/secmembership".

If you translate this into pseudo-code for traditional programming, it would be two nested "foreach" loops :

```
For  each USER in secuser do :
        For each secmembership in USER do :

        End for each
End for each.
```

Note that almost every nugget can be mapped to DATA tree nodes, using "select" attribute. Mapping a nugget to a DATA node using "select" means you want the nugget to be used if and only if the DATA node is present.

Note (1) : This LAYOUT section is tightly related to the structure of DATA for the form. This structure depends on the <input-…> nodes and the class it says to use. In our example, this structure is the secuser DB object defined in the secuser class. Here is this definition in a shortened version. Look at XDFLEngine documentation for more on Database Objects :

Corresponding DB Object (simplified version)

```
        <xdfl:_CLASS_DECLARE class="sec.user" extends="xtn.dbobject" init_class="init_class">
                <xdfl:PASSIVE >
                        <node name="secuser" DBtable="SEC_USER" pkey="@id" >
                                <field name="@id"   DBfield="user_id" />
                                <field name="@login"          DBfield="user_login"   />

                              <node name="category_grant" DBtable="SEC_GRANT" >
                                    <field name="@id"                       DBfield="grant_id" />
                                    <field name="@type_actor"      DBfield="grant_actortype" />
                                     <field name="@id_actor"               DBfield="grant_actorid"  />
                                     <field name="@type_secured"   DBfield="grant_securedtype" />
                                     <field name="@id_secured"      DBfield="grant_securedid"  />
                            <field name="@grant_level"            DBfield="grant_level"  />
                                      <node name="seccategory" DBtable="SEC_CATEGORY" >
                                           <field name="@id"  DBfield="category_id"  />
                                           <field name="@name" DBfield="category_name" />
                                          <field name="category_description" DBfield="category_description" />
                                      </node >
                                </node >

                                <node name="object_grant" DBtable="SEC_GRANT" >
                                <field name="@id" DBfield="grant_id" />
                                        <field name="@type_actor"     DBfield="grant_actortype" />
                                         <field name="@id_actor"                 DBfield="grant_actorid" />
                                         <field name="@type_secured"DBfield="grant_securedtype" />
                                         <field name="@id_secured"    DBfield="grant_securedid"  "/>
                                <field name="@grant_level"   DBfield="grant_level"  />
                                         <node name="secobject" DBtable="SEC_OBJECT" >
                                                <field name="@id" DBfield="object_id"  />
                                                <field name="@obj_id"         DBfield="object_uid" />
                                                <field name="@obj_name" DBfield="object_name" />
                                                <field name="@obj_type" DBfield="object_type" />
                                         </node >
```

[7] Since this form is only dealing with one user at a time, there's never going to be more than one secuser in the DATA section. But that's not the point here.

```
                                    </node >
                                    <node name="action_grant" DBtable="SEC_GRANT" >
                                              <field name="@id" DBfield="grant_id"  />
                                              <field name="@type_actor"    DBfield="grant_actortype"  />
                                              <field name="@id_actor"                 DBfield="grant_actorid"  />
                                              <field name="@type_secured"DBfield="grant_securedtype" />
                                              <field name="@id_secured"    DBfield="grant_securedid"  />
                          <field name="@grant_level"   DBfield="grant_level" />
                                              <node name="secaction" DBtable="SEC_ACTION" >
                                                        <field name="@id"              DBfield="action_id"  />
                                                        <field name="@tag"             DBfield="action_tag" />
                                              <field name="action_description" DBfield="action_description" />
                                             </node >

                                    </node >

                                    <node name="secmembership" DBtable="SEC_MEMBERSHIP" >
                                              <field name="@id"  DBfield="membership_id" />
                                              <field name="@id_user"        DBfield="user_id" />
                                              <field name="@id_group" DBfield="group_id" />
                                              <node name="secgroup" DBtable="SEC_GROUP" >
                                                        <field name="@id" DBfield="group_id"    />
                                                        <field name="@name" DBfield="group_name"  />
                                                        <field name="group_description" DBfield="group_description"  />

                                              </node >
                                    </node >
                          </node >
                </xdfl:PASSIVE >
        </xdfl:_CLASS_DECLARE >
```

Note (2) : do not confuse "node" nugget which are used in the Form LAYOUT section ; XML "node" of the DATA tree ; and "node" in the Database Object definition.

INCLUDES

You may have noticed some strange "###" characters. They are special escaping characters used to insert dynamic strings.
Let's suppose you want to write the name of every group the user belongs to. Using nodes and select attribute, you will reach the groups' memberships for the user, and then what ?
Then you have to use includes to display the name. Let's look at our sample (few lines after the select="membership") :

```
<label type="icon" icon="../images/sec.group.gif"
        class="ged_document_tab_element"
        icon-class="ged_directory_icon"
        label-class="ged_directory_label_doc" >
                ###secgroup/@name###
</label>
```

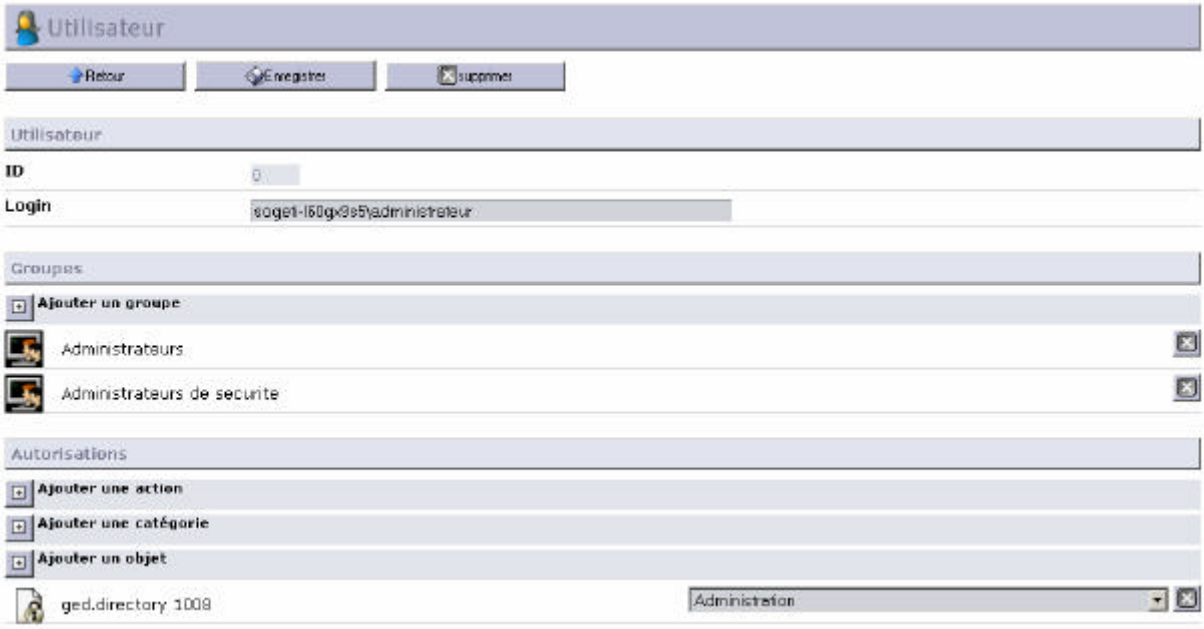
Here, we used both "select" to reach the structure, and includes to display the information. The "label" tag of type "icon" is a nugget used to display an icon and a label side by side. But the text for the label is not an attribute of the nugget , no "select" here ! it has to be provided as input instead. That is why we have to use ###secgroup/@name###[8] which is the XPATH to the name of the group, relative to the current node (full path is DATA/secuser/secmembership/secgroup/@name).

RESULT

The result for this screen is as follow (French version):

[8] "secgroup/@name" means "name" attribute of "secgroup" node. "secgroup/name" means "name" sub-node of the "secgroup" node. Look in the web for more XPath tutorials.

Form example :



✍ *Use them*

CALLING FORMS

Forms are called using either regular html syntax, or by using the framework's nav_openForm() function *(common.js)*. Entering the full URL of the form is usually done in default.htm file, and the nav_openForm function is used mainly from within other pages (tree views, views and forms).
Here are some examples :

Calling a form in html

```
<frame src="./html/form.html?def=../xdfl/operation.form.xdfl&id=A001" name="rightframe"  marginheight=0
marginwidth=0 framespacing=0  border=0 frameborder=0 >
```

This is pretty straightforward : just like views, you call a form by calling form.html file with a "def" argument equals to the path to the form XDFL script. Additional parameters can be passed to the URL and then used in the form script using ARG values. Example could be "@@[VAL:ARG.name_of_parameter]@@" for reading the value using parsed expression (see XDFLEngine documentation for more on parsed expressions).

Using nav_openForm

```
nav_openForm("../xdfl/sec.object2.form.xdfl", formDATA_get( "directory/@id"),
"update","obj_type=ged.directory","_blank")
```

This sample is taken from the Banjan project. First parameter is the name of the script for the form (value for the "def" parameter). Second parameter is the id of the entry you want to edit and is the value for the "id" parameter (look at annex A for details on the formDATA_get function). Third parameter is "update", it sets the action for the form (value for the "action" parameter). Fourth parameter is a list of additional parameter and values. Last parameter of the call, "_blank", means we want to open the form as a pop-up in a new window.

Note that calling this JavaScript function is strictly equivalent to calling the form.html file with "def", "id" and "action" parameter set to "../xdfl/sec.object2.form.xdfl", result of formDATA_get("directory/@id") , and "update" respectively, and appending "&obj_type=ged.directory" to the URL. Thus you can consider that "def", "id" and "action" URL parameters are reserved parameters of the framework.

FORMS TAGS/ELEMENTS

Form has all the elements common to all templates. Those are : CONFIG, JAVASCRIPT, DEBUG and MENU elements. You will find their description in the tree view template chapter. Those elements are used inside the FORM_DEFINITION section.

- **<FORM_DEFINITION>** : this tag contains the form definition.
  - **<SEARCH>** : Search panel definition.
    - **label** : title for the Search panel (usually set to "Search").

✍ *Hack them*

BUILDING PROCESS :

This section details the whole building process from the form XDFL definition to its html result.

client-side

1. URL entered by the client : src="./html/form.html?def=../xdfl/myform.xdfl", or nav_openForm called.
2. loading form.html.
3. form_init() is called after form.html page has loaded *(form.js)*
4. check client parameters, such as browser version. *(common.js)*

✍ *loading definition*

5. get the name of the definition for the view (in this case "../xdfl/myform.xdfl") *(common.js)*
6. tells the server to execute the definition and send the result back. *(common.js)*

server-side

7. Executes myform.xdfl and send the result back to the client. Basically, result is everything between the two <xdfl:OUTPUT> tags. This part is pure XDFLengine. For anything related to it, please reefer to XDFLEngine documentation.

client-side

8. Parse result as a DOM tree and store the tree in the g_XMLDEF variable. *(common.js)*

✍ *initializating display*

9. Get the name of the xsl that we will use for displaying in the "style" URL parameter or use "form.xsl" if none is provided. In our case none is provided since we want to use form.xsl. *(common.js)* Warning : this xsl is used for displaying form elements. It does not display nuggets but simply the main parts of the screen.
10. Get the form.xsl from the server. *(common.js)*

server-side

11. Return asked xsl file

client-side

12. Parse returned xsl file as a DOM tree and store this definition in the g_XMLSTYLE variable. *(common.js)*

13. Apply the XSL to the definition and load the result into the "document" JavaScript object of the page. *(common.js)*

✍ *getting form parameters (form.js)*

14. Get the "action" URL parameter and set the action for the form.
15. Build the DEF DOM tree from the "LAYOUT" section of the definition and stores this tree in the "gform_XMLDEF" variable. Note: this DOM tree is a subset of the g_xmlDEF dom tree, and is specific to the form template.

✍*build nugget structure (form.js)*

16. Get "nuggets" parameter giving the name of the file containing the nuggets.
17. Get the nuggets file from the server.

server-side

18. Return nuggets file.

client-side

19. Create an empty DOM tree. Assign it to gform_XSLForm variable.
20. Parse nugget file. For each <nugget> nodes : add content of <xsl> node to the gform_XSLForm DOM tree ; add content of the <javascript> node to the HTML document.

✍*load XML data (form.js)*

21. Read the <input-*action*> node where *action* is current action for the form. Call the corresponding module of the corresponding class on the server.

server-side

22. Execute request and return result to the client.

client-side

23. Build DATA DOM tree with the result of the request and store it in the gform_XMLDATA variable.

✍*Build the form (form.js)*

24. Build the DESCR DOM tree by merging DEF DOM tree in gform_XMLDEF with DATA DOM tree in gform_XMLDATA. Store the result in the gform_XMLDESCR variable.
25. Build display by applying the nugget DOM tree (stored in the gform_XSLForm variable) to the DESCR DOM tree (stored in the gform_XMLDESCR variable).
26. Insert result of previous operation into the "elem_formCTNR" element of the "document" HTML object of the page using its "innerHTML" property.
27. Fill elements of the "document" HTML object of the page with content from the DATA DOM tree.

Done !

In order to get a graphic view of those steps, you are advised to look at the "architecture" chapter, and specially the "Big Picture" paragraph.

UPDATING DATA

Once the data is loaded and displayed, user can modify values either by filling a field, checking a box, etc. Every time the user modify the *display* of the form, the framework automatically modify the corresponding value in the DATA structure (DATA DOM tree), and mark the node either as to be updated or deleted (by setting an "action" attribute to either "update", "delete" or "insert"). This way, when the user saves the form, the "form_save" function only has to give the new data structure as input for the module corresponding to the form's action (see details of the DATA section for more). If you're familiar with XDFL

database objects, you already know that an XML structure it is all the DB_SET and DB_GET xdfl tags needs for updating Database tables.

CUSTOMIZATION

You should normally never customize forms apart from changing colors in css files or modifying sections either common to all templates, such as menu section, or specific to the form. Since adding section process is already detailed in the View chapter it won't be covered here.
Instead, you should create (or sometimes modify) nuggets. If you ever create a nugget that you think could be useful to someone else, please share it with us by sending it to admin of the sourceforge XDFL engine site.

Extending Forms / using nuggets

Nuggets are graphic elements and their behavior. In other word, they are elements of user interface. They are used in Forms in the “Layout” section. By using nuggets, building an interactive graphical interface for a form is only a matter of what nugget you want to map on what data. This process of mapping nuggets on data is already explained in the Form chapter but we will give additional examples here. Since this process heavily uses XPath, you are encouraged to read examples of this technology first. Good XPath tutorials can easily be found on the web.

✍ *Nugget example*

Let’s suppose our form has to deal with this kind of set of data :

```
<DATA>
        <catalog year="2004">
                <item id="1" >
                        <name> Beautiful table</name>
                        <price>200 </price>
                        <color ><name>red</name><value>FF0000</value></color>
                        <color><name>white</name><value>FFFFFF</value></color>
                </item>
                <item id="2" >
                        <name> Beautiful chair</name>
                        <price>20</price>
                        <color ><name>purple</name><value>FF00FF</value></color>
                        <color ><name>blue</name><value>0000FF</value></color>
                </item>
        </catalog>
        </DATA>
```

The form has to do this : you want to display the items, and let the user modify some information such as the item’s name and value, and add or remove a color.
Input / Output of the form has already been explained in the form’s chapter, so we won’t give details here.
Now remember, the thing is you don’t know in advance how many items there’s going to be in the catalog. So what you’ll say in the layout is this :

a. I want a panel for the catalog (you know there’s only one catalog) with the year of the catalog for header.
b. For every item I want a sub-panel with its id for header of the sub-panel
c. In this sub-panel I want to display an input box with the name of the item and another one with the price. User has to be able to enter new value in those input box.
d. For every color I want a line to be displayed with a button saying whether the color is still available or not (in other word I want to be able to remove the color from the list).
e. I also want a button for adding colors. When the user clicks on the button, a prompt ask for the name of the new color and adds it to the list. *Note : the best way would have been to pop-up a window displaying a list of colors and let the user choose the new color by clicking on it. This is perfectly doable with the framework using the action nugget with type=choose, but requires the use of another screen : a View displaying colors from an hypothetical COLOR DB table and using*

nav_returnXML() function at user clicking on a color to return the id of the chosen color.

You can now choose which nugget suits your needs by browsing the list of nuggets in the annex of this document. Result after browsing is this :

a. Obviously, a “node” nugget. We’ll use parsed expression for printing the year.
b. Also a node, maybe with different style. Parsed expression as well for the id.
c. Input box is the purpose of the “input” nugget. We’ll map one on the name and one of the price.
d. We’ll use the input nugget with type=”readonly” for the label, and the “action” nugget for the button with type = “delete” (type delete means that clicking on the button will remove the entry from the DATA structure).
e. The “action” nugget can be used to call a custom JavaScript function.

The resulting code is :

```
<node select="catalog" class="catalog_panel" style="width:620px;">
        <label class="catalog_year">YEAR : ###@year###</label>
        <node select="item" class="item_panel" style="width:100%;">

                <node style="width:400px;float:left;">
                        <label class="item_header" >item : ###@id###</label>
                        <label class="field_label"> Name : </label><input select="name" class="item_name"/>
                        <label class="field_label"> Price : </label><input select="price" class="item_price"/>
                </node>

                <node class="color_panel" style="margin-left:400px;width:200px;">
                        <label class="color_header" > List of colors : </label>
                        <node  select="color" class="color_line" style="width:100%;" >
                                <node style="width:200px;">
                                        <label style="float:left;">###name###</label>
                                        <action style="float:right;" class="delete_button" type="delete" msg="Are you
sure you want to remove the color ?" icon="../images/delete.16.gif" />
                                </node>
                        </node>

                        <action label="new color" class="color_button" icon="../images/light1.gif"
>add_color(###@xtn_dataid###)</action>

                </node>
        </node>
</node>
```

Explanation :

-The action with type “delete” means that when you click on the button, it automatically removes the “current” node from the DATA tree (see note below).
-The action with no type calls the javascript function given in input. Since this javascript function should add a color XML structure to the item data structure, we have to provide the DATA id of the item to the function. This id is created by the framework when building the DESC trees. Simply use this attribute.

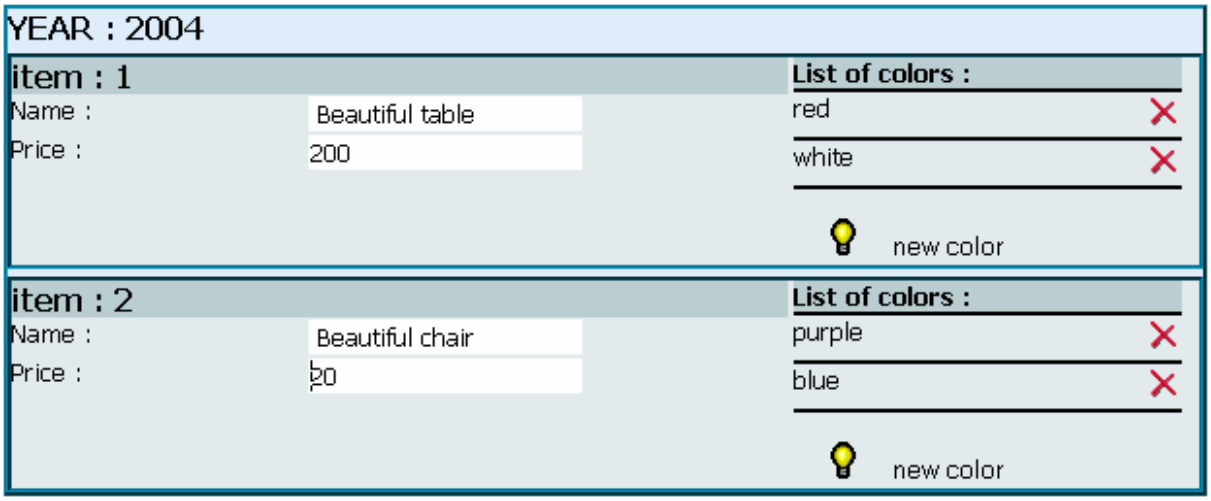
Here is the custom javascript function :

```
<JAVASCRIPT>
                <![CDATA[
                        function add_color(dataid)
                        {
//Prompt user for name and value for the new color
                        var name = prompt("Enter name for color")
                        var value= prompt ("Enter value for color")
```

```
// Then we build the string for the color's XML structure we want to create
                        var l_strNewNode = "<color><name>"+name+"</name><value>"+value+"</value></color>"
//parse this string in order to build a DOM node
                        var l_xmlNewNode = xml_buildDocumentFromString(l_strNewNode)

//get the father's item for the new color
                        l_xmlData = formDATA_selectByID(dataid,true)
//append  the node to the data structure
                        formDATA_appendNode(l_xmlData,l_xmlNewNode)

//we don't need to refresh the display or rebuild the form manually with this nugget, but sometimes you have to.
                        }
                ]]>
                </JAVASCRIPT>
```

We didn't provide any CSS with this form, but here's an example of what you can get (sorry if it hurts your taste, I'm not good at graphics).

Result :



Note : in order to understand a little bit better the "select" thing and its relation to the way nuggets are mapped to the form's DATA :
This code would be wrong :

```
<node select="color">
        <node select="name">
        ###.###
        <action type="delete" icon="delete.gif" msg="are you sure ?"/>
        </node>
</node>
```

You can see why this is wrong by looking at the "DESCR" tree in debug mode : the xtn_dataid for the "action type=delete" node is not the same as the xtn_dataid for the "node select='color'" node. This is because the "action" node is mapped to the same data as it's father's data. In this example, the last select was "name", thus "action" node links to "name" XML tag, instead of "color". Clicking on delete would delete the name and not the color.

✍ -- POWER TIPS -- ✍
It is a good idea to test your layout with hand-written custom XML structure. For doing this, you can create a "get_TestStructure" method in your class, the same way you built the "get_default" method.

You can use includes in nugget's attributes ! Try to customize this screen by setting "background-color" in the style attribute for the color's name to the color value.

✍ *Adding Nuggets*

Sometimes, you need a user interaction or a GUI element that is not in the framework. This chapter will show you how to build your own nugget and use it.

For this chapter we will create a "mailto" nugget. This nugget is used the same way as action, by entering the mail in the nugget's input.

Nuggets are stored in the "nugget.xml" file. As you may notice, nuggets are basically XSL and JavaScript. XSL is meant for transforming XML tags in the Layout section into HTML, and JavaScript is meant to deal with user's interaction with the nugget.

There is one interesting line in this file :

```
<xsl:apply-templates select="node | input | label | action "/>
```

This is where It all begins. You can see right that for the moment, the framework has 4 types of nuggets : nodes, input, label and actions.
The mailto template will also be an action.

A good thing when building your own nugget is to see what nugget looks the closest to what you want to do :
In our case, the "action type=link" is pretty close. The only difference is that it calls a javascript instead of a "mailto".

After toying a little with this nugget, you can see that simply by changing the

```
<xsl:attribute name="href">javascript:<xsl:value-of select="."/></xsl:attribute>
```

line, and replacing "javascript:" with "mailto:", you can use this nugget to call the users' email software.

Now all we have to do is to create our own nugget. This step requires to be familiar with XSL, but that's almost anything. Since our nugget is also an "action", you don't need to modify the interesting line.

Here's the code :

```
<!-- ACTION MAILTO -->
 <nugget>
  <xsl>
   <xsl:template match="action[@type='mailto']">
    <a>
     <xsl:attribute name="id">action_<xsl:value-of select="@xtn_descrid"/></xsl:attribute >
     <xsl:attribute name="style"><xsl:value-of select="@style"/></xsl:attribute>
     <xsl:attribute name="class"><xsl:value-of select="@class"/></xsl:attribute>
     <xsl:attribute name="href">mailto:<xsl:value-of select="."/></xsl:attribute>
     <xsl:if test="@icon">
      <img>
       <xsl:attribute name="id">actionimg_<xsl:value-of select="@xtn_descrid"/></xsl:attribute >
       <xsl:attribute name="src"><xsl:value-of select="@icon"/></xsl:attribute>
       <xsl:attribute name="style"><xsl:value-of select="@icon-style"/></xsl:attribute>
       <xsl:attribute name="class"><xsl:value-of select="@icon-class"/></xsl:attribute>
      </img>
     </xsl:if>
     <xsl:if test="@label">
      <font>
       <xsl:attribute name="id">actionlabel_<xsl:value-of select="@xtn_descrid"/></xsl:attribute >
       <xsl:attribute name="src"><xsl:value-of select="@icon"/></xsl:attribute>
       <xsl:attribute name="style"><xsl:value-of select="@label-style"/></xsl:attribute>
```

```
      <xsl:attribute name="class"><xsl:value-of select="@label-class"/></xsl:attribute>
      <xsl:value-of select="@label"/>
     </font>
    </xsl:if>
   </a>
  </xsl:template>
 </xsl>
</nugget>
```

And here’s how you would use it in a form’s layout :

```
<action type="mailto" class="label" label="email : ">###mail###</action>
```

Annex A : JavaScript core functions and variables

General conventions :

- "DESCR node", "DATA node", "PAGE node", "DEF node", "DISPL node" means a node belonging to the DESCRIPTION/DATA/PAGE/DEFINITION/DISPLAY DOM tree.
- Variables starting with "g" are global.
- Variables starting with "p" are parameters.
- Variables starting with "l" are local to the function.
- Variables having "_xml" in their names reefers to a node (or a set of nodes) in a DOM tree.
- Variables having "_str" in their names are strings of characters.
- Variables having "_int" in their names are numbers.

Examples :
"p_strXML" would be a parameter of type string.
"g_xmlDEF" would be a global variable referring to a DOM node (presumably in the DEFINITION DOM tree).

xml.js : XML manipulation function

Variables :

- **g_ParserCode** : determine the type of browser's XML parser (MSXML or other).
- **g_NavCode** : determine the type of browser (IE or NS).

Functions : all functions have two implementations depending on the browser. MSXML versions start with "msxml", Mozilla versions start with "expat". Both have the same parameters and result.

- **xml_testXMLParser()** : check if the XML Parser is supported.
- **xml_buildEmptyDocument()** : build an empty DOM document
- **xml_buildDocumentFromString(p_strXML)** : build a DOM document from a string given in parameter (the string is the text version of the XML document)
- **xml_sendServer(p_strUrl, p_xmlNode)** : send a POST request to the "p_strUrl" URL with "p_xmlNode" as data for the request. If p_xmlNode is null then simply get the URL. In case of an error, returns an XML document containing the error code and message. Returns server's response as a DOM document.
- **xml_sendServerText(p_strUrl, p_strText, p_boolAsMultiPart)** : same as sendServer only the second parameter is text and not XML data. Returns server's response as text.

- **xml_XSLtransform(p_xmlToTranform, p_xmlXSL)** : transform an XML DOM document using an XSL. Second parameter is the DOM document containing the XSL. Returns result of transformation as a string.
- **xml_selectSingleNode(p_xmlNode,p_strExpr)** : select a single DOM node from the DOM "p_xmlNode" sub-tree, using "p_strExpr" as Xpath expression. Returns the single selected DOM node.
- **xml_selectNodes(p_xmlNode,p_strExpr)** : select a collection of DOM nodes from the DOM p_xmlNode sub-tree using "p_strExpr" as Xpath expression.
- **xml_getXML(p_xmlNode)** : returns content of the p_xmlNode XML node as a string.
- **xml_getFirstChildElem(p_xmlNode)** : returns first child of the "p_xmlNode" DOM node.
- **xml_getNextElem(p_xmlNode)** : returns element next to "p_xmlNode".
- **xml_getAttrParent(p_xmlAttr)** : get the parent node for the Attribute DOM node "p_xmlAttr".
- **xml_getAttribute(p_xmlNode, p_strName, p_strDefault)** : returns a string containing value for the attribute "p_strName" of the node "p_xmlNode". If no value is found then returns "p_strDefault".
- **xml_getValue(p_xmlNode,p_strDefault)** : returns value for the node "p_xmlNode" as a text string. If no value is found then returns "p_strDefault".
- **xml_setValue(p_xmlNode,p_strValue)** : sets value "p_strValue" to the DOM node "p_xmlNode". Node can be either an Attribute or an Element.

common.js : functions and variables used in every template

Variables :

- **g_XMLDEF** : contains the definition for the page.
- **g_XMLSTYLE** : contains the style for the page (used for XSL transformation of the definition)
- **g_arURLParams** : Array for arguments passed in the current URL.

Functions :

- **xtn_initClientEnv()** : get the URL parameters, check the browser version, and initialize corresponding environment variables.
- **xtn_recognizeBrowser()** : check the browser version.

- **xtn_getConfigParam(p_strName, p_strDefault)** : get the value for the parameter “p_strName” of the CONFIG section of the current definition or use “p_strDefault” if none found.
- **xtn_sendRequest(p_xmlNodeRequest, p_strArgs)** : build request according to request root node “p_xmlNodeRequest”, and concatenate “p_strArgs” arguments to the request.
- **xtn_loadDefinition(p_strDefinition)** : get page definition “p_strDefinition” from the server, and fill “g_XMLDEF” global variable with it.
- **xtn_getDefinition()** : return “g_XMLDEF” (which supposedly contains the page definition DOM tree)
- **xtn_getSection(p_strSectionName)** : returns node of the definition corresponding to the section whose name’s given in parameter.
- **xtn_getRequest(p_strSectionName)** : return node of the definition corresponding to the sub-section of the “DATA” section whose name is given in parameter.
- **xtn_initDisplay(p_strStyle)** : initialize display by applying xsl “p_strStyle” on the definition.
- **xtn_processIncludes(p_strTxt, p_xmlNode)** : parse “p_strTxt” for ###xpath###-like special values and replace this value by applying the xpath request on the “p_xmlNode” XML node.
- **nav_openForm(p_strForm, p_strObjId, p_strAction, p_strOptions, p_strTarget)** : open form “p_strForm” with “id” URL argument set to “p_strObjId”, action URL argument set to “p_strAction” and “p_strOptions” being concatenated to URL. "p_strTarget” is either the name of the new frame (pop-up) or empty for opening the form in the calling frame.
- **nav_openView(p_strView , p_strOptions, p_strTarget)** : same as nav_openForm only without “id” or “action” URL argument.
- **nav_showMessage(p_intType, p_strMessage, p_strArgs, p_strCallBack)** : never used.
- **nav_parseURLParams()** : builds g_arURLParams.
- **nav_returnXML(p_XMLNode)** : calls the *opener* page function passed in “callback” URL argument of the *current* page, with “p_XMLNode” as its parameter. This method is used when you want the user to choose an element in a view to fill in the current form (the “opener” is the caller form, the “p_XMLNode” is the chosen element, and “nav_returnXML” is called from the view).
- **nav_returnWindowValue(p_strValue)** : same as nav_returnXML only the returned value is not an XML node but a strValue instead.
- **nav_receiveWindowValue(p_strCallBack)** : call “p_strCallBack” with a small delay.
- **nav_getURLParam(p_strName, p_strDefault)** : returns value for the “p_strName” URL argument or “p_strDefault” if none found.

- **nav_addToURL(p_strUrl, p_strAdd)** : add “p_strAdd” to “p_strUrl” URL.
- **nav_appendUrlArgs(p_strURL)** : appends arguments of the current URL to the “p_strURL” URL.
- **nav_appendContinuedArgs(p_strURL)** : appends arguments of the current URL and defined in the CT_CONTINUED_ARGS global variable (defined in “config.js”) to the “p_strURL” URL.
- **nav_forceFocus( )** : force focus for the current window.
- **nav_setFocusTarget(p_objElem)** : sets target for the next nav_forceFocus() to “p_objElem” element of the HTML page.
- **nav_getCookie(name)** : get value for cookie “name”.
- **nav_setCookie(name, value)** : sets “name” cookie to “value”.
- **nav_deleteCookie (name)** : sets “name” cookie’s expiration date to now minus 1 (means the cookie expired).
- **nav_getCookieVal (offset)** : get value for cookie number “offset” (where “offset” is the index of the cookie in the document.cookie property).
- **html_createElements(strHTML)** : creates a “div” element in the current document, filled with “strHTML”. Returns node for the newly created element.
- **html_toggleDisplay(p_strElemId)** : toggle display for element in the DISPLAY (meaning, the content actually being displayed).
- **html_parseScript(p_strScript)** : adds “p_strScript” to the current document in a “<script language=’javascript’>” tag.
- **xtn_log(p_strLog, p_intLevel)** : displays “p_strLog” in the window’s status field.
- **dispXML(xmlNode)** : Displays XML sub-tree under “xmlNode” node in a javascript message box (works only if CT_MODE_DEBUG is set to true in config.js).
- **debug_showXML(p_XMLdoc)** : displays content of “p_XMLdoc” in the debug section of the page (works only if CT_MODE_DEBUG is set to true in config.js).

config.js : parameters for the JavaScript part of the framework

Variables :

- **CT_MODE_DEBUG** : set to true for enabling debugging or false for disabling it.
- **CT_CONTINUED_ARGS** : names of parameters automatically transmitted between every URL.

Functions :

none

✍ *treeview.js : functions and variables for the Tree View template*

Variables :

none

Functions :

- ✍ **tree_init()** : loads definition and init display.
- ✍ **treeview_toggleNode(strId)** : toggle node in the tree (if node is opened then closes it, if node is closed then opens it). “StrId” is the id of the node. Id is generated by the “treeview.xsl” file at transformation time.

✍ *view.js : functions and variables for the View template*

Variables :

- ✍ **CT_EVENT_ONDATALOAD** : name of the tag in the CONFIG section of the view containing the JavaScript function called right before DATA has been loaded.
- ✍ **CT_EVENT_ONDATALOADED** : name of the tag in the CONFIG section of the view containing the JavaScript function called right after DATA has been loaded.
- ✍ **CT_EVENT_ONCHANGEXSL** : name of the tag in the CONFIG section of the view containing the name of the JavaScript function to call right after a new XSL has been applied to the view.
- ✍ **CT_EVENT_ONDATADISPLAYED** : name of the tag in the CONFIG section of the view containing the name of the JavaScript function to call right after the view’s display has been refreshed.
- ✍ **gview_XMLDATA** : DOM tree containing the DATA for the view.
- ✍ **gview_XMLPAGE** : DOM tree containing data to be displayed for the current page.
- ✍ **gview_strCurrentXSL** : name of the XSL for displaying view’s data.
- ✍ **gview_xmlCurrentXSL** : DOM tree for the XSL used to display view’s data.
- ✍ **gview_intPageSize** : number of elements per page.
- ✍ **gview_intCurrentPage** : index of the page currently being displayed.
- ✍ **gview_intMaxPage** : total number of page (equals number of nodes in gview_XMLDATA / gview_intPageSize).

- **gview_elemDisplayContainer** : root node in which HTML code should be inserted in order to display the view's data. This is where the result of the XSL applied to DATA is put in order for the page to be displayed.
- **gview_boolDATALoaded** : tell whether view's DATA has been loaded or not (used in view_refresh() and view_setPage()).

Functions :

- **view_init()** : initialize client settings, load view definition and display view's data. Note : setting parameter "begin-with-data" to 0 in the CONFIG section, one can prevent the view from loading any data at initialization.
- **view_initGlobals()** : sets global variables.
- **view_ChangeXSL(p_strXSLPath)** : get "p_strXSLPath" file and set it as new XSL for the view.
- **view_getCurrentXSL()** : returns DOM Tree of XSL currently used for displaying view's data.
- **view_setCurrentXSL(p_xmlCurrentXSL)** : set "p_xmlCurrentXSL" DOM tree as the current XSL for displaying the data and refresh the view.
- **view_getData(p_strAction)** : loads view's data using the "DATA/input-p_strAction" node of the view's definition. Note that this node is the node being modified when entering new values for search fields. Note that "p_strAction" should always be "request" for a view, and that it is the default value.
- **view_refreshData()** : refresh view data (that is, loads data and refresh the view).
- **view_refresh()** : refresh the view by applying the XSL to the PAGE (PAGE being the subset of DATA that should be displayed, based on page number).
- **view_setPageSize(p_intPageSize)** : set p_intPageSize as the new page size. A page size is the number of result nodes per page.
- **view_setPage()** : calculates which DATA nodes are to be displayed and copy them from the "gview_XMLDATA" DOM tree to the "gview_XMLPAGE" DOM tree. Note : does not loads new set of data from the server !
- **view_jumpNextPage()** : displays next page. Note : does not loads new set of data from the server !
- **view_jumpPrevPage()** : displays previous page. Note : does not loads new set of data from the server !
- **view_jumpPage(p_intPage)** : displays page number "p_intPage". Note : does not loads new set of data from the server !
- **view_getPage( )** : returns page number of the page currently being displayed (returns gview_intCurrentPage global variable).

- **view_toggleSearch()** : toggle search pane on and off.
- **view_setSearch(p_strField, p_strValue, p_strOp)** : sets a filter for the field “p_strField” with value “p_strValue” and operation “p_strOp”. Note : this function always look for the “input-request” node in the DATA section. “input-request” is the only “input-“ like node tolerated in a view (that is obviously not the case with forms).
- **view_setOrder(p_strField, p_boolNoRefresh)** : order current result in ascending or descending order for the “p_strField” field by adding or modifying filter on the field. If p_boolNoRefresh is false, then the view doesn’t query new data.

form.js : functions and variables for the Form template

Variables :

- **CT_FORMACTION_NONE** : defines value for none action. Usually set to “(none)”.
- **CT_FORMACTION_INSERT** : defines value for insert action. Usually set to “insert”.
- **CT_FORMACTION_UPDATE** : defines value for update action. Usually set to “update”.
- **CT_FORMACTION_DELETE** : defines value for delete action. Usually set to “delete”.
- **CT_EVENT_ONDATALOAD** : name of the tag containing JavaScript function to call just after loading the form’s data.
- **CT_EVENT_ONDATASAVE** : name of the tag containing JavaScript function to call just before saving form’s data.
- **CT_EVENT_ONCHANGE** : name of element’s attribute containing name of the JavaScript function to call when element’s data changed. Default is “on_change”. Example : for an input field, when the user type something new in the field, the content of the “on_change” attribute for the corresponding input element is read and evaluated.
- **CT_NULL** : set to “(NULL)”. Currently unused.
- **gform_boolDATALoaded** : set to true when form’s data is loaded and false when it’s not.
- **gform_XMLDEF** : DOM tree containing the form’s definition.
- **gform_XMLDESCR** : DOM tree containing the form’s description.
- **gform_XMLDATA** : DOM tree containing form’s DATA.
- **gform_XSLForm** : DOM tree containing the form’s XSL. This XSL is created by reading the nuggets file.

- **gform_strAction** : action for the form. Values for actions are defined in variables.
- **gform_intDATAID** : variable used for generating Id attributes for nodes of the DATA tree.
- **gform_intDEFID** : variable used for generating Id attributes for nodes of the DEFINITION tree.
- **gform_intDESCRID** : variable used for generating Id attributes for nodes of the DESCRIPTION tree.

Functions :

- **form_init()** : initialize form by setting client environment, loading form's definition, loading nuggets, loading XML data and creating the form's display.
- **form_initGlobals()** : get/set action for the form, load form.xsl and load form's definition.
- **form_build()** : build description and display for the form.
- **form_parseNuggets()** : read file containing nuggets, create corresponding DOM tree, and load JavaScript scripts contained in the nugget into the page.
- **form_save(p_strAction, p_strArgs, p_strCallBack, p_boolNoRefresh)** : saves form's data by sending "output-"+form's action (e.g. : "output-update" when action parameter is set to "update") node content to the server using the node's attributes for finding which module of which class to call. "p_strAction" parameter is used for overriding current's form action (thus calling "output-"+p_strAction instead). "p_strArgs" contains parameters appended to the request. "p_strCallBack" contains the name of the function (if any) that should be called after saving the form. If p_boolNoRefresh, then the form will not be rebuilt.
- **form_getDocToSave()** : return DATA nodes of the form without any attributes added by the framework (such as Ids).
- **form_setDocSaved(p_XMLDATA)** : clean the p_XMLDATA DOM TREE from any node that has been deleted when the form was saved.
- **form_getAction()** : returns current action for the form.
- **form_setAction(p_strAction)** : set "p_strAction" as current action for the form.
- **formDATA()** : return DATA DOM tree.
- **formDATA_get(p_strPath)** : get node in the DATA DOM Tree matching "p_strPath" path.
- **formDATA_set(p_strPath,p_strValue, updateDisplay)** : Sets data node matching "p_strPath" to "p_strValue" value and updates the display. If "updateDisplay" is set to false then no display update occurs.

- **formDATA_loadXML()** : loads form's DATA from the server by using the "input-"+form's action node (such as "input-insert" if action is "insert").
- **formDATA_getID(p_xmlDATANode)** : return's id of the "p_xmlDATANode" node in the DATA DOM tree. If parameter is an attribute or a text node, then ID of the parent element will be returned.
- **formDATA_setID(p_xmlDATANode,p_intDATAID)** : same as above, but for setting the id of the node.
- **formDATA_genID(p_xmlDATANode)** : generates and ID for the p_xmlDATANode node of the DATA DOM tree.
- **formDATA_selectByID(p_intDATAID,p_boolNode)** : select the node in the DATA DOM tree having p_intDATAID for id.
- **formDATA_selectDESCR(p_XMLNode)** : select nodes in the DESCR DOM tree on which the "p_XMLNode" DATA node is based.
- **formDATA_getAction(p_xmlDATA)** : get "action" attribute value of the "p_xmlDATA" DATA node.
- **formDATA_setAction(p_xmlDATA,p_strAction)** : set "action" attribute to "p_strAction" for the "p_xmlDATA" DATA node.
- **formDATA_deleteNode(p_XMLNode)** : set "p_XMLNode" DATA node "action" attribute to "delete" and delete corresponding nodes in the DESCR DOM tree.
- **formDATA_appendNode(p_XMLNodeParent, p_xmlNode, p_xmlNodeRef)** : appends "p_xmlNode" node as a child of "p_XMLNodeParent" node, and before "p_xmlNodeRef" if not null. Also updates the DESCR DOM tree and display.
- **formDATA_appendNode2(p_XMLNodeParent, p_xmlNode, p_xmlNodeRef)** : same as above, using a different method. Use previous function if this one doesn't work.
- **formDATA_refreshNode(p_XMLNodeParent)** : refresh DESCR nodes based on "p_XMLNodeParent" DATA node. Also updates display.
- **formDATA_update(p_xmlDATANode, p_strValue)** : update "p_xmlDATANode" DATA node with "p_strValue" value. Node can be an element or an attribute.
- **formDEF()** : returns DEFINITION DOM tree
- **formDEF_getID(p_xmlDEFNode)** : returns ID for the "p_xmlDEFNode" DEF node.
- **formDEF_setID(p_xmlDEFNode,p_intDEFID)** : sets ID to "p_intDEFID" for the "p_xmlDEFNode" DEF node.
- **formDEF_genID(p_xmlDEFNode)** : sets a unique ID for the "p_xmlDEFNode" DEF node.

- **formDEF_selectByID(p_intDEFID)** : returns DEF node having ID attribute equals to “p_intDEFID”.
- **formDESCR()** : returns DESCRIPTION DOM tree.
- **formDESCR_getID(p_xmlDESCRNode)** : returns ID for the “p_xmlDESCRNode” DESCR node.
- **formDESCR_setID(p_xmlDESCRNode,p_intDESCRID)** : sets ID to “p_intDESCRID” for the “p_xmlDESCRNode” DESCR node.
- **formDESCR_genID(p_xmlDESCRNode)** : generates and sets a unique ID to the “p_xmlDESCRNode” DESCR node.
- **formDESCR_selectByID(p_intDESCRID)** : select node in the DESCR DOM tree having id equals to “p_intDESCRID”.
- **formDESCR_setDEFID(p_xmlDESCRNode,p_intDEFID)** : set attributes “xtn_defid” of the “p_xmlDESCRNode” DESCR node to “p_intDEFID”. This attribute is supposed to represent the id of the DEF node on which the “p_xmlDESCRNode” DESCR node is based.
- **formDESCR_getDEFID(p_xmlDESCRNode)** : returns the “xtn_defid” attribute of “p_xmlDESCRNode” DESCR node. This attribute is supposed to represent the id of the DEF node on which the “p_xmlDESCRNode” DESCR node is based.
- **formDESCR_selectByDEFID(p_intDEFID)** : select the DESCR nodes based on the DEF node having id equal to “p_intDEFID”. This function uses the “xtn_defid” attributes of DESCR nodes.
- **formDESCR_setDATAID(p_xmlDESCRNode,p_intDATAID)** : set attributes “xtn_dataid” of the “p_xmlDESCRNode” DESCR node to “p_intDATAID”. This attribute is supposed to represent the id of the DATA node on which the “p_xmlDESCRNode” DESCR node is based.
- **formDESCR_getDATAID(p_xmlDESCRNode)** : returns the “xtn_dataid” attribute of “p_xmlDESCRNode” DESCR node. This attribute is supposed to represent the id of the DATA node on which the “p_xmlDESCRNode” DESCR node is based.
- **formDESCR_selectByDATAID(p_intDATAID)** : select the DESCR nodes based on the DATA node having id equal to “p_intDATAID”. This function uses the “xtn_dataid” attributes of DESCR nodes.
- **formDESCR_build()** : build the DESCR DOM tree. Do not update the display.
- **formDESCR_buildRec(p_xmlDEF, p_xmlDATA, p_xmlDESCR)** : builds the DESCR DOM tree starting from p_xmlDEF XML node. Works together with formDESCR_buildElem to build DESCR element.
- **formDESCR_buildElem(p_xmlDEFNode, p_xmlDATANode, p_xmlDESCRparent)** : builds DESCR element corresponding to

p_xmlDEFNode DEF node, and appends this element under p_xmlDESCRparent. Works together with formDESC_buildRec to build DESCR elements.

- **formDESCR_refreshDisplay(p_xmlDESCR)** : refresh display starting with the “p_xmlDESCR” element of the DESCR DOM tree.
- **formDESCR_selectDEF(p_XMLDESCRNode)** : returns node from the DEF DOM tree that has been used to create the “p_XMLDESCRNode” node of the DESCR DOM tree.
- **formDESCR_selectDATA(p_XMLDESCRNode)** : returns node from the DATA DOM tree that has been used to create the “p_XMLDESCRNode” node of the DESCR DOM tree.
- **formDESCR_selectDISP(p_strType, p_XMLDESCRNode)** : returns node from the DISP DOM tree having type “p_strType” and which was created from the “p_XMLDESCRNode” DESCR node. Example of “p_strType” is “input” or “node”.
- **formDESCR_callNuggetFunc(p_strFuncName, p_xmlDESCR, p_xmlDATA, p_strArgs)** : call function “p_strFuncName” of the nugget at node “p_xmlDESCR” with parameters “p_xmlDESCR”,”p_xmlDATA”, “p_strArgs”. Complete name of the function is deduced from the name of the nugget and its type (e.g. input_select_fill(p_xmlDESCR,p_xmlDATA) for a call to the “fill” function of an input node with type “select”).
- **formDESCR_inputChanged(p_intDESCRID)** : function called when the “p_intDESCRID” input node of the form changed. It updates the DATA node related to this input and every other DESCR nodes using it. Finally, it refresh the display.
  Note : if the node has an “on_change” attribute, it calls the corresponding JavaScript prior to any update. This function uses the “input_extract” function of “input” nuggets.
- **formDISP_build()** : builds the display. Display Tree is not saved in a global variable but is inserted directly into the “document” JavaScript object of the page.
- **formDISP_selectById(p_strType, p_intDESCRID)** : returns node from the “document” JavaScript object having id = “p_strType”_”p_inDESCRID”. E.g. formDISP_selectById(“input”,”9”) returns node with id = “input_9”.
- **formDISP_fill()** : fill all nodes of the display having “rendered” attribute equals to “1” with data from the DATA tree. Typically, “input” nodes will be filled with value of the nodes they are mapped to.
- **formDISP_fillNode(p_xmlDATA, p_xmlDESCR)** : fill “p_xmlDESCR” child nodes with data then calls itself recursively on child nodes to explore the tree. First argument is currently not used.

- **formDISP_fillInput(p_xmlDATA, p_xmlDESCR)** : fill “p_xmlDESCR” DESCR node with value of “p_xmlDATA” DATA node. Calls the “fill” function of “input” nugget (“p_xmlDESCR” node has to be an “input” node).

Annex B : Nuggets

Here is a list of all nuggets given by default in the framework along with their parameters and a short description.
Remember all nuggets can be mapped to a DATA XPath, using a “select” attribute.

General remarks :

- “class” means reference to a CSS class.
- All nuggets can be mapped to a DATA node (ie : node in the DATA structure of a form) using “select” attribute.
  Select attribute has to be set to the Xpath pointing to the DATA node. XPath can be relative. Examples of values for “select” attribute are : “item” ; “../item” ; “../.” ; “@id[../disabled = 0]”. What XPath request is accepted highly depends on the version of the XML parser included in the browser (e.g. : MSXML for Internet Explorer).
  When a nugget is mapped to a data node, it is only visible if the structure exists.

Note : make sure to get the latest version of the framework, as nuggets are updated and fixed frequently.

- node :
  - description : creates a DIV section. Input of the node is content for the div.
  - attributes :
    - “style” : node style
    - “class” : node class
  - Tag’s Input : content for the node.

- input (no “type” attribute):
  - Description : creates an input field for enabling the user to enter data. Default value for the input field depends on “select” attribute. Note : input nuggets have to be mapped to a DATA Xpath pointing to an existing node for being displayed.
  - Attributes :
    - “style” : input style
    - “class” : input class
    - “onchange” : name of custom javascript function to be called when input changed.
    - “readonly” : if set to 1 then input is read only.
  - Tag’s input : none
- input (type=”readonly”)
  - Description : creates a DIV and displays a label in it. Note : this nugget isn’t really an “input” : the user can’t change the value for the label.
  - Attributes :
    - “style” : style for the DIV
    - “class” : class for the DIV
    - “label-style” : style for the label
    - “label-class” : class for the label
  - Tag’s input : none.

- input (type=”date”)
  - Description : creates a set of fields specially designed for entering dates. Also checks for input’s validity. Note : the date format for the XML DATA is in MM/DD/YYYY format, but nugget displays it in the DD/MM/YYYY format. Customize the nugget for other formats by changing order of the input in the XSL part of the nugget.
  - Attributes :
    - “style” : style for the DIV containing the input fields.
    - “class” : class for the DIV containing the input fields.
    - “style-day“ : style for the day’s field.
    - “class-day” : class for the days’ field.
    - “style-month” : style for the month’s field.
    - “class-month” : class for the month’s field.
    - “style-year” : style for the year’s field.
    - “class-year” : class for the year’s field.
    - “readonly” : set to 1 if date is read-only.
    - “onchange” : JavaScript function called when user change date.
  - Tag’s input : none
- input (type=”check”)
  - Description : checkbox. This input has to be mapped on a Boolean field.
  - Attributes :
    - “style” : style for the checkbox.
    - “class” : class for the checkbox.
    - “readonly” : if set to 1 then checkbox is read only.
  - Tag’s input : none
- input (type=”select”)
  - Description : a dropdown list of items. User can change the selected element.
  - Attributes :
    - “style” : style for the list.
    - “class” : class for the list.
    - “readonly” : if set to 1 then selected item can’t be changed.
  - Tag’s input : input for the tag gives the list of items in the list. Default item depends on the “value” field.
    Example (warning : case sensitive) :

```
<input type="select" select="id">
        <option><VALUE>1</VALUE><LABEL>Table</LABEL></option>
        <option><VALUE>10</VALUE><LABEL>Chair</LABEL></option>
</input>
```

- input (type=”radio”)
  - Description : list of radio buttons.
  - Attributes :
    - “style” : style for the DIV section containing the buttons.
    - “class” : class for the DIV section containing the buttons.
    - “style-radio” : style for the radio buttons.
    - “class-radio” : class for the radio buttons.
    - “label-style” : style for radio button labels.
    - “label-class” : class for radio button labels.
  - Tag’s input : same as type=“select”.
- input (type=”xsl”)
  - Description : apply xsl given as input to selected node.

- Attributes :
  - “style” : style for the div containing XSL’s result.
  - “class” : class for the div containing XSL’s result.
- Tag’s input : a full XSL document.

- action
  - Description : actions base HTML element is button. It calls a JavaScript upon user click.
  - Attributes :
    - “style”: button’s style.
    - “class” : button’s class
    - “icon” : url to the button’s image.
    - “icon-style” : style for the button’s image.
    - “icon-class” : class for the button’s image.
    - “label-style” : style for the label.
    - “label-class” : class for the label.
  - Tag’s input : none.
- action (type=’link’)
  - Description : not a button, but an HTML link (either text or icon or both) that does something when the user clicks on it.
  - Attributes :
    - “style”: style for the link.
    - “class” : class for the link.
    - “icon” : URL to the icon for the link.
    - “icon-style” : style for the icon.
    - “icon-class” : class for the icon.
    - “label” : text for the link.
    - “label-style” : style for the text.
    - “label-class” : class for the text.
  - Tag’s input : name of the JavaScript function to be called when the user clicks on the nugget.
- action (type=’delete’)
  - Description : this nugget deletes the DATA node it is mapped on when the user clicks on it. Remember a nugget is mapped to a DATA node even if it has no explicit “select” attribute (in that case, it is mapped to the same node as its parent’s node).
  - Attributes :
    - “style” : style for the button.
    - “class” : class for the button.
    - “disabled” : XPath path. If the path exists then the button is in “disable” state.
    - “icon” : URL to the icon.
    - “icon-style” : icon’s style.
    - “icon-class” : icon’s class.
    - “label” : text to be displayed on the button.
    - “label-style” : style for the text.
    - “label-class” : class for the text.
    - “msg” : warning message.
  - Tag’s input : none.
- action (type=”insert”)

- Description : insert a XML structure into the DATA structure upon user click. The XML structure has to be defined in the root of the <DATA> form's section.
- Attributes :
  - "data" : node to be inserted. This node has to be defined at the root of the <DATA> section of the form.
  - "insert-at" : path to the destination for the XML structure. Default is the current node.
  - "style" : style for the button.
  - "class" : class for the button.
  - "icon" : URL to the icon for the button.
  - "icon-style" : icon's style.
  - "icon-class" : icon's class.
  - "label" : text for the action.
  - "label-style" : text's style.
  - "label-class" : text's class.
- Tag's input : none
- Example :

```
<action type="insert" select="@id[not(../comment)]" insert-at="../." data="append-comment" />
.
.
.
<DATA>
        …
        <append-comment>
                <comment>
                        <number/>
                        <value/>
                </comment>
        </append-comment>
</DATA>
```

- action (type="check_activate")
  - Description : a checkbox that is checked when a structure exists and unchecked when it doesn't. Big thing is that checking the box creates the structure !! This nugget can be seen as a mix between the input type="checkbox" and the action="insert" nuggets.
    Note : do not map this node to the structure you wish to create using "select" attribute : It won't work !
  - Attributes :
    - "style" : style for the checkbox.
    - "class" : class for the checkbox.
    - "checked" : any not null and defined value for checking the box, 0 or "undefined" for leaving it blank. Note : most of time, this value is a ###include.
    - "data" : data node added when user check the box. Works the same way as action type="insert" nugget.
    - "node" : name of the node once added. This value is usually the same as the child of the node given in "data" attribute (see example). This attribute plays a little bit the role of the "select" attribute.
  - Tag's input : none
  - Example : this example displays the list of all catalog and all items in them. Associations between items and catalog is in a third table called "item_catalog". Whenever an item is in present in the catalog, the box is

checked. Checking the box associates the item and the catalog, by creating the corresponding “item_catalog” node.

```
…
<node select=”catalog”>
        <node select="item" >
                <label >###year ###</label>

                <!-- the id has to be always different from 0 for the trick to work -->
                <action type="check_activate" node="item_catalog" data="append-item_catalog"
checked="###item_catalog[@action='update' or @action='insert' or not(@action)]/@id###"/>

                <!-- reference is the reference of the item in the catalog -->
                <node select="item_catalog" >
                        <input select="reference"/>
                </node>
        </node>
</node>
…
<DATA>
…
<append-item_catalog>
        <item_catalog id="-1" >
                <reference></reference>
        </item_catalog>
</append-item>
</DATA>
```

- action (type=”choose”)
  - Description : displays a pop-up to choose an item from. This action works together with a view template and the **nav_returnXML** JavaScript function.
  - Attributes :
    - “view “ : URL to the view used to display items.
    - “viewargs” : additional parameters to the call.
    - “style” : button’s style.
    - “class” : button’s class.
    - “label” : label for the button.
    - “label-style” : label’s style.
    - “label-class” : label’s class.
    - “icon” : URL to the icon.
    - “icon-style” : icon’s style.
    - “icon-class” : icon’s class.
  - Tag’s input : An XSL document for displaying new elements. This XSL will be used to convert XML sent by the view to XML suited for the form’s structure.
- action (type=”mailto”)
  - Description : launch a “mailto” upon user click.
  - Attribute : same as action type=”link”.
  - Tag’s input : email address.
- label
  - Description : a label in a DIV.
  - Attributes :
    - “style” : div’s style.
    - “class” : div’s class.
    - “label-style” : label’s style.
    - “label-class” : label’s class.
  - Tag’s input : text to be displayed.
- label (type=”icon”)
  - Description : an icon whose text is given as input.
  - Attributes :

        - “style” : style for the div.
        - “class” : div’s class.
        - “icon” : URL to the icon.
        - “label-style” : label’s style.
        - “label-class” : label’s class.
    - Tag’s input : text for the label.
- label (type=”help”)
    - Description : an image with text appearing on rollovers.
    - Attributes :
        - “style” : icon’s style.
        - “class” : icon’s class.
        - “icon” : URL to the image.
    - Tag’s input : text to be displayed while image is loading (“alt” attribute of the “img” HTML tag).

Note : for examples on nuggets, you can download the “banjan” project. It is made with the XML Forms Framework and is open source.