FOX Training

Training:Fox:2012

FOX Training Worksheets

Hello World

1. Hello World
2. Librarying LAYOUT1
3. An Introduction to DOMs
4. Basic Actions and Presentation
   1. Editing your own FOX module
   2. Introducing Actions
   3. Alert Popups
   4. fm:expr-out

FOX Reference Guide

The reference guide concisely documents various aspects of FOX functionality, as well as notes which should be helpful to FOX application developers.

It can be found **here**. You may find it useful for some of the later exercises

Planning Application Demo

1. Introduction
2. Initial Data Model
3. Brief Introduction to Packages
4. Planning Application XView
5. Querying into a DOM
6. Removing Data from DOM
7. Schema Data Structures
8. Initialising Data Structures
9. Form set-out
10. Element Presentation
11. Set-out Attributes
12. Querying into a DOM Revisited
13. Query Into a DOM Merging/Replacing Data
14. Query Over
15. Grouping Actions
16. List set-out
17. Assigning and Setting Data
18. DOM Contexts and the Evaluate Context Rule
19. Phantoms
20. Calling Modules
21. Moving and Copying Data
22. Editing Data with an API
23. Entry Themes
24. Phantom Image Buttons

25. Libraries
    1. Contexts
    2. fm:if
    3. What's going on in XX_PLAN_APP_EDIT?
26. fm:validate
27. Storage Locations
28. Updating data with a storage location
29. Using states to break up data entry
30. Introduction to map-sets
31. Storage Locations - Memory Only Map-Set
32. Constructing a map-set
33. Constructing a map-set 2
34. Element Actions and Change Actions
35. Dynamic Map-sets
36. File uploads
37. Security Table

Advanced Training

You will be expected to read the following training sheets as part of your basic training. You will be asked to do these as and when they are required.

1. UREFs
2. Teams

Training:Fox:HelloWorld

Introduction to Working With FOX

Welcome to the FOX training. This is probably the first time you have come in to contact with FOX so we're going to take you through the basics.

What Is FOX?

FOX stands for 'Forms Oriented XML'.

Hello, World!

When you start using a programming language for the first time you usually begin with a basic "Hello, World!" application, in this case, we have created one for you as a FOX module using the code found below. You can view this example module at http://[fox_engine_url]/HELLO_WORLD/new. Please ask your training supervisor what '[fox_engine_url]' should be.

```xml
<?xml version="1.0" encoding="UTF-8"?>

<xs:schema xmlns:fm="http://www.og.dti.gov/fox_module" xmlns:fox="http://www.og.dti.gov/fox" xmlns:order="http://www.og.dti.gov/fox" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
xmlns:xs="http://www.w3.org/2001/XMLSchema"

elementFormDefault="qualified" attributeFormDefault="unqualified"

xsi:schemaLocation="http://www.w3.org/2001/XMLSchema

C:\pvcswork\Fox\source\altova_xmlspy\schema\fox_schema.xsd">

  <xs:import schemaLocation="C:\pvcswork\Fox\source\altova_xmlspy\schema\fox_import.xsd"/>

  <xs:annotation>

    <xs:appinfo>

      <fm:module>

        <fm:header>

          <fm:name>HELLO_WORLD</fm:name>

          <fm:title>Hello World!</fm:title>

          <fm:application-title>FOX Basics</fm:application-title>

          <fm:version-no>$Revision$</fm:version-no>

          <fm:version-desc>$Header$</fm:version-desc>

          <fm:history>

$Log$

          </fm:history>

          <fm:description/>

          <fm:build-notes/>

          <fm:help-text/>

        </fm:header>

        <fm:control>

          <fm:transaction-mode>read-committed</fm:transaction-mode>

          <fm:authentication>not-required</fm:authentication>

        </fm:control>

        <fm:security-list/>

        <fm:library-list/>

        <fm:storage-location-list>

          <fm:storage-location name="main">

            <fm:cache-key string="HELLO_WORLD"/>

            <fm:new-document>

              <fm:root-element>ROOT</fm:root-element>

            </fm:new-document>

          </fm:storage-location>

        </fm:storage-location-list>

        <fm:entry-theme-list>

          <fm:entry-theme name="new">

            <fm:storage-location>main</fm:storage-location>

            <fm:state>state-hello</fm:state>

            <fm:attach>/*</fm:attach>

            <fm:do/>

          </fm:entry-theme>

        </fm:entry-theme-list>

        <fm:action-list>

          <fm:action name="action-dummy">

            <fm:do/>

          </fm:action>
        </fm:action-list>
        <fm:db-interface-list/>
        <fm:presentation>
          <fm:set-page>
            <html>
              <head>
                <title>FOX Basics</title>
              </head>
              <body>
                <br/>
                <br/>
                <br/>
                <br/>
                <br/>
                <fm:include name="buffer-content"/>
              </body>
            </html>
          </fm:set-page>
          <fm:set-buffer name="buffer-content">
            <h2>Hello World!</h2>
          </fm:set-buffer>
        </fm:presentation>
        <fm:state-list>
          <fm:state name="state-hello"/>
        </fm:state-list>
        <fm:template-list/>
        <fm:map-set-list/>
      </fm:module>
    </xs:appinfo>
  </xs:annotation>
  <xs:element name="ROOT">
    <xs:complexType/>
  </xs:element>
</xs:schema>
```

A First Glance

What am I Seeing?

If you haven't yet navigated to the HELLO_WORLD module in a web browser, please do so using the URL above. Once you have done this you will be presented with the webpage that FOX has generated based on the example module source code given above.

If you view the source of the web page you will notice code that the FOX engine has added such as javascript and CSS.

The other thing you will notice is a small link titled 'Testing...' in the top left of the window. Clicking this expands a menu. This is known as the developer toolbar and has several options to help you during your development such as access to the data your page is using, internal information for the engine, information about the current user that is logged in, statistics about the page you have just loaded and much more. This will all be covered as we progress through the training.

What is a FOX module?

FOX uses modules to group logical units of code that result in a web page with varying layouts. For example in a shopping website the 'check-out' might be a module with a summary view and a payment view.

A module is meta-data in an XML structure about how a web page should look and behave. This meta-data is stored in a central database for the FOX engine to consume.

A FOX module is actually an extension of a W3C XML Schema (XSD) document, with most of the FOX-specific markup nested within the *xs:annotation* element. FOX markup is prefixed with the *fm:* namespace; XSD tags are prefixed with *xs:*. By nesting FOX within an XML Schema document FOX has access to information about the structure of the data it is operating on, which it uses for presentation and validation purposes. Furthermore, as XSD is a recognised W3C standard, this makes it easy to edit FOX modules with existing GUI XML editing tools.

Why is so Much Mark-Up Required?

As mentioned in *What am I Seeing?* FOX adds extra code to your page for you - however, much of this additional code is based on the meta-data mark-up in your FOX module.

Some key areas you might want to look at in the code sample above are:

- *fm:header*: The XML elements inside this complex node are generally used to describe the module giving fields such as its name, its title, a description of what it does, and version information that is usually filled in automatically.
- *fm:entry-theme*: This describes an entry point to the module and a single module can have multiple entry points. For example, one entry point might give a read-only view and another can have a fully editable view.
- *fm:action*: An action is similar to a method or function in other programming langauges. Actions manipulate data and perform various other actions.
- *fm:presentation*, *fm:set-page* and *fm:set-buffer*: These elements relate to the layout and presentation of your module and contain standard HTML amongst other FOX commands. i.e. What the user sees.
- *fm:state*: A state is a particular view within the module. In the example of the shopping website you might have one state for the summary and one for the payment.

Training:Fox:LibraryingLayout1

Introduction to fm:presentation

One of FOX's primary goals is to generate HTML pages. For every HTTP request, FOX will decide what to generate based on the current state of the user's activity. FOX is designed so that the generation of HTML is mostly abstracted away from you as a developer. For instance, where in a 3GL language you might write a loop to iterate through a list and output table rows, FOX is able to accomplish this in one command. This is the core of FOX's 4GL approach.

The *fm:presentation* section of a FOX module is where you put the presentational commands you wish FOX to run, plus any additional HTML which may be needed for boilerplate text, page structure, etc. FOX allows most HTML markup, with some notable exceptions (i.e. <script> tags are not permitted). It is the responsiblity of you as a developer to ensure that any custom HTML you put in a module is conformant to any necessary coding standards - for instance avoiding the use of tables to control layout, etc. As you start writing more FOX modules these rules should become more obvious.

In the code fragment below, you can see that the *fm:presentation* element contains 2 other elements; *fm:set-page* and *fm:set-buffer*. This was taken from the HELLO_WORLD module introduced on the previous sheet.

```
<fm:presentation>
  <fm:set-page>
    <html>
      <head>
        <title>FOX Basics</title>
      </head>
      <body>
        <br/>
        <br/>
        <br/>
        <br/>
        <br/>
        <fm:include name="buffer-content"/>
      </body>
    </html>
  </fm:set-page>
  <fm:set-buffer name="buffer-content">
    <h2>Hello World!</h2>
  </fm:set-buffer>
</fm:presentation>
```

fm:set-page

fm:set-page is the top-level command which FOX looks for to decide what to display to the user. It will typically contain the basic HTML of a page - in the example above you can see the <html>, <head> and <body> tags defined. *fm:set-page* is actually a special type of buffer and follows all the rules of a buffer as described below.

fm:set-buffer

A **buffer** is the definition of the presentational markup for a logical area of the screen. Using buffers allows presentation logic to be reused between modules and states, and also helps to keep the fm:presentation section more logically structured.

A buffer is defined using the *fm:set-buffer* command, and named using the *name* attribute. A buffer is a mix of FOX-specific markup tags (prefixed with the *fm:* namespace) and standard HTML. These tags can be nested as required.

fm:include

fm:include is used by a buffer to recursively include the contents of another buffer. This is the main way a page's layout would be built up, i.e. one main buffer would use fm:include to include several other buffers.

A more generic solution

Manually presenting each module to give a consistent look and feel across all modules is obviously time consuming and involves a lot of code duplication. As such, a generic solution is available.

The generic presentation for all modules is defined in a module called LAYOUT1LIB, which is then libraried in to a module called LAYOUT1. LAYOUT1 can be libraried in to your module, giving your module access to the generic presentation template.

LAYOUT1 works by defining an *fm:set-page* command which sets up the layout of a FOX screen and defines a set of buffers for various areas of the screen (left hand side menu, header, footer etc) which you can override if you need, like a template. At a bare minimum, you must define a buffer called *buffer-content* using the *set-buffer* command. This will contain all the content for your module. LAYOUT1 uses the *fm:include* command to effectively 'request' from you a buffer called *buffer-content*. You'll notice that FOX produces an error if it attempts to include a buffer which is not defined.

NOTE Using LAYOUT1 means you no longer need to include the *fm:set-page* element in your module. If you do, you will override the generic presentation defined in LAYOUT1.

Linking modules together

Librarying a module is the FOX equivalent of an import or include statement in other languages. To library a module, an *fm:library* element needs to be added to the *fm:library-list* in your module.

The syntax for *fm:library* is:

```
<fm:library>[module name]</fm:library>
```

Where *[module name]* is the name of the module you want to link in.

Libraries will be discussed in more detail later on in the training.

Example

We've modified the HELLO_WORLD module to use LAYOUT1, which you can view here: http://[fox_engine_url]/HELLO_WORLD_LAYOUT1/default (use the same fox_engine_url as the previous sheet)

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:fm="http://www.og.dti.gov/fox_module" xmlns:fox="http://www.og.dti.gov/fox" xmlns:order="http://www.og.dti.gov/fox" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" attributeFormDefault="unqualified"
xsi:schemaLocation="http://www.w3.org/2001/XMLSchema
C:\pvcswork\Fox\source\altova_xmlspy\schema\fox_schema.xsd">
  <xs:import schemaLocation="C:\pvcswork\Fox\source\altova_xmlspy\schema\fox_import.xsd"/>
  <xs:annotation>
    <xs:appinfo>
      <fm:module>
        <fm:header>
          <fm:name>HELLO_WORLD_LAYOUT1</fm:name>
          <fm:title>Hello World!</fm:title>
          <fm:application-title>FOX Basics</fm:application-title>
          <fm:version-no>$Revision$</fm:version-no>
          <fm:version-desc>$Header$</fm:version-desc>
          <fm:history>
$Log$
          </fm:history>
          <fm:description/>
          <fm:build-notes/>
          <fm:help-text/>
        </fm:header>
        <fm:control>
          <fm:transaction-mode>read-committed</fm:transaction-mode>
          <fm:authentication>not-required</fm:authentication>
        </fm:control>
        <fm:security-list/>
        <fm:library-list>
          <fm:library>LAYOUT1</fm:library>
        </fm:library-list>
        <fm:storage-location-list>
          <fm:storage-location name="main">
            <fm:cache-key string="HELLO_WORLD_LAYOUT1"/>
            <fm:new-document>
              <fm:root-element>ROOT</fm:root-element>
            </fm:new-document>
          </fm:storage-location>
        </fm:storage-location-list>
        <fm:entry-theme-list>
          <fm:entry-theme name="new">
            <fm:storage-location>main</fm:storage-location>
            <fm:state>state-hello</fm:state>
            <fm:attach>/*</fm:attach>

            <fm:do/>

          </fm:entry-theme>

        </fm:entry-theme-list>

        <fm:action-list>

          <fm:action name="action-dummy">

            <fm:do/>

          </fm:action>

        </fm:action-list>

        <fm:db-interface-list/>

        <fm:presentation>

          <fm:set-buffer name="buffer-content">

            <h2>Hello World!</h2>

          </fm:set-buffer>

        </fm:presentation>

        <fm:state-list>

          <fm:state name="state-hello">

            <fm:action-list/>

            <fm:presentation/>

          </fm:state>

        </fm:state-list>

        <fm:template-list/>

        <fm:map-set-list/>

      </fm:module>

    </xs:appinfo>

  </xs:annotation>

  <xs:element name="ROOT">

    <xs:complexType/>

  </xs:element>

</xs:schema>
```

The code for HELLO_WORLD_LAYOUT1 is shown above.

If you compare it to the code for HELLO_WORLD from the previous sheet, you can see that we no longer have a *fm:set-page* element, as this is defined for us in LAYOUT1.

You can also see that we have an *fm:library* linking in the module LAYOUT1, which results in the module being presented very differently.

Training:Fox:IntroductionToDOMs

Introduction

Learn about the DOM (Document Object Model) and how FOX uses the DOM to manipulate XML internally. Also, learn about contexts in FOX and how they reference different points in the DOM.

Syntax

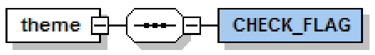
The DOM specified by the W3C describes different ways that a program can update the content, style and structure of a document, in FOX's case an XML document.

FOX has several different DOMs. You will primarily use the "data" DOM (also known as the "root" DOM) which is where the majority of the information for the current screen is stored, but there are other DOMs with different lifespans, ranging from the "temp" DOM which lasts for one page churn (a client -> server -> client request/response cycle), to the "session" DOM which lasts until the browser is closed.

You can reference different DOMs in FOX using a context notation in your XPath expression. This is extremely useful in that you can reference different DOMs concisely in one statement. The syntax of the context notation is:

```
:{context}
```

There are reserved context names for the different DOMs. Contexts are pointers, so they need to be evaluated. For example, the theme DOM is referenced using the :{theme} syntax in an XPath expression. The :{theme} context is evaluated to an absolute XPath internally in FOX, so you don't need to worry about a long, complicated XPath expression. If the theme DOM looked like the following:



You could find out the value of CHECK_FLAG by referencing it with the XPath:

```
:{theme}/CHECK_FLAG
```

Below is a list of some commonly used DOMs, their scope (lifespan), their FOX XPath contexts (always lower case) and a description. Being aware of the other contexts available is useful.

DOM Name	Scope	Context	Description
Data	1 Module Call	:{root}	This is the main area of data storage and can be easily written back to the database in a Storage Location.
Theme	1 Module Call	:{theme}	Used to store data that you don't want stored with the module data and is usually used for presentation fields and storing temporary values or flags. You can mark up the Theme DOM.
Temp	1 Page Request	:{temp}	Used to set values or collate data from multiple sources to be used in one place for a transaction, but its life span is very short and cannot be used for setting out data.
Session	1 Browser Session	:{session}	Used for one Browser session, that is for the life of one Browser window. It is useful for storing copied data to be pasted into different modules.
User	1 Portal Session	:{user}	Stores information about the current user logged in. You should not write information to it.
Sys	1 Module Call	:{sys}	Stores useful system information and is constantly updated by FOX. You should not write information to it.
Params	1 Module Call	:{params}	Contains any parameters passed to the module when it was called. You should not write data to it.

Environment	Module call (copy forward)	:{env}	This acts as a pseudo-combination of the :{params} and :{session} DOMs. Whereas the :{params} DOM only contains data passed forward on the most recent module call, the :{env} DOM can be written to or read from at any time (regardless of module calls), much like the :{session} DOM - but data is only ever passed forwards when calling a module. When you exit a module, any changes made to the :{env} DOM are not visible to the caller.
Error	1 Module Call	:{error}	Contains a list of validation errors, generated when the fm:validate command is run. You don't generally write data to it, and normally only to amend error text content.

Observing DOM contents

You can use the developer toolbar (see Hello World) to observe the contents of various DOMs used by FOX
With the developer toolbar expanded you should see a series of contexts (:{root}, :{theme} etc.). Clicking on one of these will open a new window showing you the contents of the selected DOM.

Running XPaths over DOMs

Often, when developing a FOX module you'll need to test your XPaths on the content of various DOMs. The developer toolbar provides you with an XPath evaluator for just this purpose.
With the developer toolbar expanded, click on the XPath link. This will open a new window which you can use to test XPath expressions using FOX DOM contexts.

Multi-context XPaths

A powerful feature provided by the FOX XPath engine is the ability to combine more than one context in an XPath. This allows you to query elements in one DOM based on values in another. This is useful in cases where you have (for example) a list of values to validate against in one DOM and the data is stored elsewhere. By providing multi-context XPath support, FOX removes the single-document constraint of XPath and allows you greater flexibility for managing your data sets.

Exercises

View the module located here: http://[fox_engine_url]/HELLO_WORLD_DOM_DATA/new. This is a modified version of HELLO_WORLD_LAYOUT1 where some data has been loaded into the :{root} and :{theme} DOMs. Have a look at the :{root} and :{theme} DOMs using the developer toolbar.

Exercise 1

Using the XPath evaluator, write an expression to return the EMPLOYEE element for the employee with ID 98 in the :{theme} DOM.

Exercise 2

Using the XPath evaluator, write an expression to return the ID element for the employee with FORENAME *Jamie* in the :{root} DOM.

Exercise 3

Look at the :{user} and :{sys} DOMs. Many of the elements in the :{user} DOM do not have meaningful values as you are not currently logged in as a user.

Exercise 4

Using the XPath evaluator, extract the module element from the :{sys} DOM.

Exercise 5

Using the XPath evaluator, write a multi-context XPath that selects the first employee in the :{theme} DOM and returns their details from the :{root} DOM, matching on ID.

Training:Fox:Editing your own FOX module

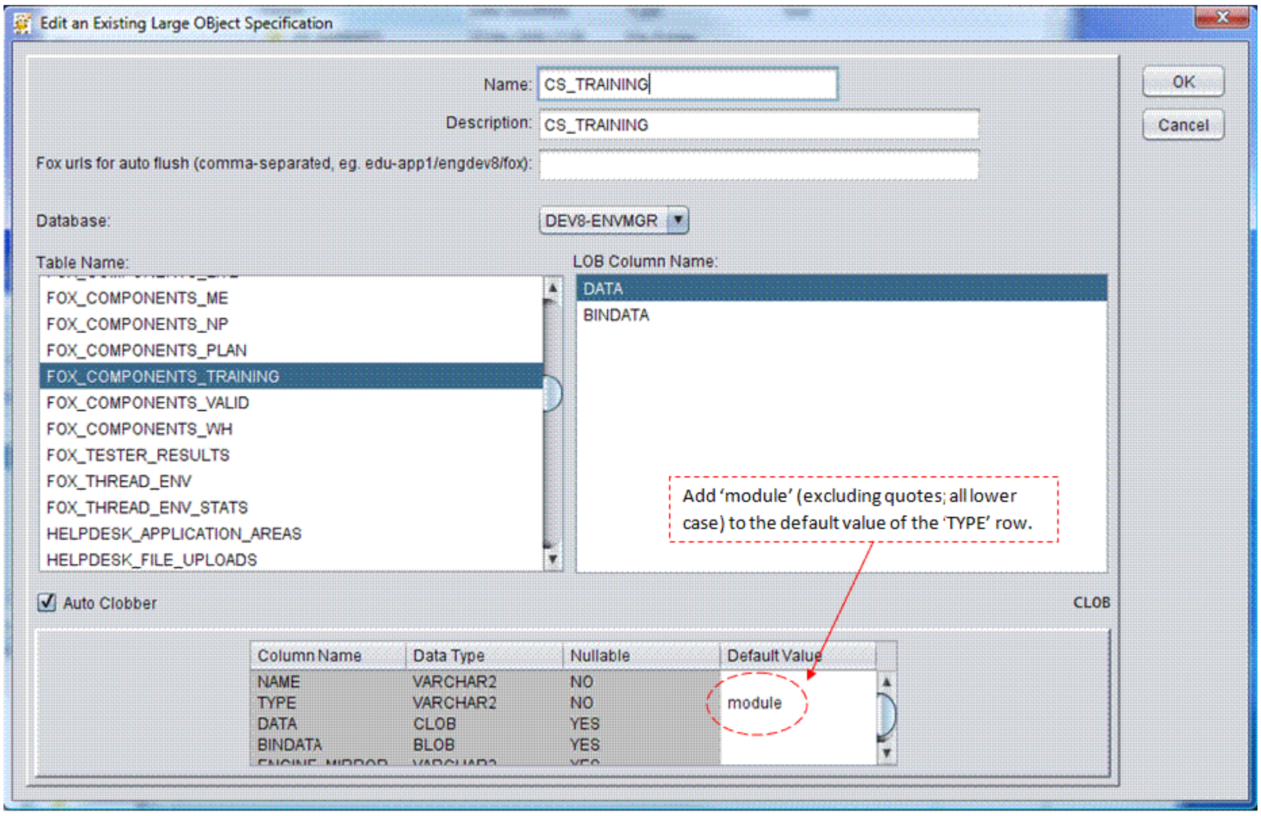
Its time to start editing your own FOX module.

Getting Started

To start, get a copy of HELLO_WORLD_DOM_DATA (ask your training supervisor how to get this) and save it as XX_HELLO_WORLD, where XX are your initials. Make sure this file is editable as well; you may have to manually set this.
Now open your XX_HELLO_WORLD file and do a find and replace changing all instances of the text 'HELLO_WORLD_DOM_DATA' with 'XX_HELLO_WORLD' (where XX are your initals).

Configure Clobber by going to (from the main screen) Edit -> Database LOB Resources..., selecting the XX_TRAINING object and then clicking Edit

Now change the default value of the 'TYPE' row to 'module' (lower case, excluding quotes):


Edit an Existing Large OBject Specification

Name: CS_TRAINING
Description: CS_TRAINING
Fox urls for auto flush (comma-separated, eg. edu-app1/engdev8/fox):
Database: DEV8-ENVMGR

Table Name:
- FOX_COMPONENTS_ME
- FOX_COMPONENTS_NP
- FOX_COMPONENTS_PLAN
- FOX_COMPONENTS_TRAINING
- FOX_COMPONENTS_VALID
- FOX_COMPONENTS_WH
- FOX_TESTER_RESULTS
- FOX_THREAD_ENV
- FOX_THREAD_ENV_STATS
- HELPDESK_APPLICATION_AREAS
- HELPDESK_FILE_UPLOADS

LOB Column Name:
- DATA
- BINDATA

Add 'module' (excluding quotes; all lower case) to the default value of the 'TYPE' row.

Auto Clobber ✓

CLOB

Column Name	Data Type	Nullable	Default Value
NAME	VARCHAR2	NO	
TYPE	VARCHAR2	NO	module
DATA	CLOB	YES	
BINDATA	BLOB	YES	

Next, add your XX_HELLO_WORLD module to Clobber and set up a Clobber for it (this can be done by dragging and dropping the file you wish to Clobber onto the Clobber main screen and clicking 'ok' on all prompts). You should already have configured Clobber previously, but ask your training supervisor if you need help.

Your module is now ready to be edited. Feel free to play around with the content of your module, by changing the HTML in *buffer-content* before moving on to the next training sheets.

You can view your module at: http://[fox_engine_url]/XX_HELLO_WORLD/new

Getting your changes to appear

Every time you save your XX_HELLO_WORLD module, Clobber will copy it to the database, so the database is always kept up to date.
However, this does not mean that your changes will actually appear on the webpage generated from your module when you refresh the page.
In order to make your changes appear, you need to flush the module cache. This is done by clicking the flush link on the developer toolbar.

Why cache modules?

Modules can be large XML files, especially if multiple modules are libraried together (librarying is discussed in detail later, but is effectively an intelligent merge of many individual modules into a single module). Additionally, the FOX engine will parse a module definition when it is loaded to speed up execution time, analogous to a 3GL compiler. For these reasons the FOX engine stores a local cache of module definitions in memory, and will only retrieve a module from the database the first time it is needed.

Because of this, developers will need to flush the module cache to force the engine to refresh its cached copy. There is an overhead in doing this as all the modules the engine knows about (not just the one you have changed) must be re-fetched an re-parsed, so don't flush unless you need to.

Remember:

- You only need to flush ONCE. Do not click flush multiple times as this is pointless and can slow the engine down for others.
- You only need to flush if you have modified a FOX module. Changes to the database, i.e. creating new tables, or updating rows, do not require a flush.

Training:Fox:Introducing Actions

Introduction

Creating actions in a FOX module and setting them out on the screen as a link or button.

Syntax

The fm:action command takes the following syntax:

```
<fm:action name="action-name" namespace:run="."/>
```

The namespace:run attribute tells FOX that the action will be runnable.

The fm:action-out command takes the following syntax:

```
<fm:action-out action="action-name" namespace:mode="."/>
```

The action attribute tells FOX what action is to be called when the link or button is clicked. The namespace:mode attribute tells FOX which of the attributes within an action should be used.

Concepts

An action can be declared in the action list for the module, or within a state action list. If it is declared within a state, the action can only be used in that state. A state can see any action that has been declared in the top level action list.

When the action is declared, it must have at least one namespace attributed to it if it is to be set out on the screen. Namespaces will be discussed in a later chapter. For now, use the namespace *fox*

fm:action-out is a quick and dirty method of adding actions to the screen. A better way would be to use a function called menu-out which lists all the actions for a given namespace. It can be set to flow down, or across. This will be covered in a later chapter.

When using an action-out FOX will put the action on the screen as a hyperlink. A hyperlink is one type of widget. The action widget can be changed to a button using the namespace:widget attribute. This is specified on the action declaration.

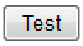
Examples

The following code declares an action that is to be set out on the screen:

```
<fm:action name="action-example" fox:run="." fox:widget="button" fox:prompt="Test">
  <fm:do>
    [ACTION STUFF]
  </fm:do>
</fm:action>
```

The presentation code to set the action on the screen would be:

```
<fm:action-out action="action-example" fox:mode="."/>
```

The result puts this on the screen: 

Exercises

Use your XX_HELLO_WORLD module for the following exercises.

Exercise 1

Create a new action inside state-hello with the name "action-link". This action does not have to do anything.

Set the action out underneath the Hello World text.

Exercise 2

Create a new action inside the module action-list called "action-button". Make it a button and add a prompt that says "My First Button".

Set the action out underneath action-link.

Don't forget to save your module changes, flush the module cache and refresh the webpage before testing your changes

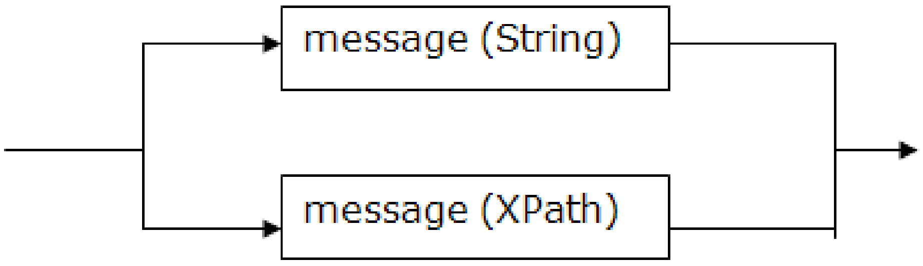
Training:Fox:Alert Popups

Introduction

Learn how to popup a message on the screen from both static text and a dynamically generated string.

Syntax

The fm:alert syntax is as follows:

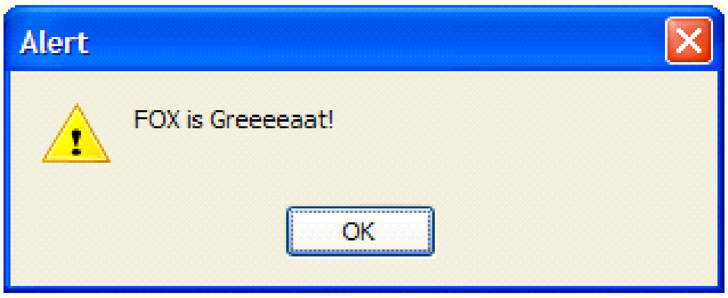


fm:alert

For example, the command:

```
<fm:alert message="FOX is Greeeeaat!"/>
```

Will popup:



FOX is Greeeeaat!

This is all very well and good when you want to alert the users with a static message, but not very useful apart from that!

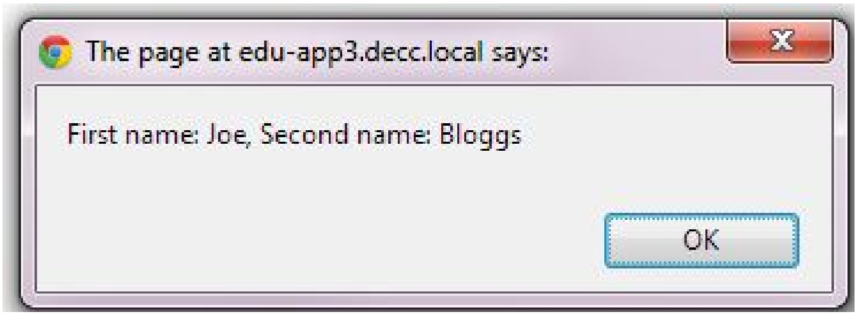
The other value the message attribute can take is an XPath expression.

Examples

For example, if you wanted to alert the user about the first and second name of the first employee in the data DOM, you could use:

```
<fm:alert message="string(concat('First name: ', :{root}/EMPLOYEE_LIST/EMPLOYEE[1]/FORENAME, ', Second name: ', :{root}/EMPLOYEE_LIST/EMPLOYEE[1]/SURNAME))"/>
```

In the <fm:do> part of an action, or entry into a module, to give you:


The page at edu-app3.decc.local says:

First name: Joe, Second name: Bloggs

Try it out!

It uses the XPath functions "string", which is mandatory since alerts must return a string, and the function "concat", which concatenates 2 or more strings together.

Exercises

Use your XX_HELLO_WORLD module for the following exercises.

Exercise 1

Make an alert message popup when the module loads up with the words "Hello World!"

Hint: You can put any command that's valid inside an fm:action's fm:do block (including fm:alert) directly inside the <fm:do> section of the entry theme. These commands will run when the user first enters the module via this entry theme.

Exercise 2

Add an <fm:alert> to the "action-button" action from the previous chapter. The message should read "My First Button pops up this message."

Exercise 3

Alter the message from Exercise 2 to read "Welcome to [module_name]: [module_title]" by using XPath to select the [module_name] and [module_title] from the sys DOM.

Don't forget to save your module changes, flush the module cache and refresh the webpage before testing your changes

Extra XPath commands can be found on this reference sheet: http:/ / www. mulberrytech. com/ quickref/ XSLT_1quickref-v2.pdf

Training:Fox:fm:expr-out

Concepts

As mentioned on Librarying LAYOUT1, the *fm:presentation* element can contain a mix of HTML and FOX mark-up to handle the presentation of a module.

fm:expr-out is a FOX command which writes out the text content of the result of an XPath expression, with an optional format mask applied.

Syntax

```
<fm:expr-out match="XPath expression" [type="XML schema datatype"] [formatMask="Literal String"]/>
```

Parameters

match

The XPath expression whose result is set out. If the expression returns multiple nodes, only the text content of the first node will be set out. If the expression returns a node with children, only the text content of the parent node will be set out.

type

- xs:date
- xs:dateTime
- xs:decimal
- xs:string

Default: "xs:string"

The XML Schema datatype [1] that the match should be treated as. This affects whether the value should be treated as a NUMBER or a DATE when it is passed to Oracle to apply the *formatMask*.

If this is set to "xs:string", *formatMask* will be ignored.

formatMask

The Oracle-style format mask [2] to be applied to the match before it is set out. This is done in the engine or if this fails then on the database, so any valid Oracle format model is permissible.

Example

To display the xs:date '2011-03-29' as 29-Mar-2011:

```
  <fm:expr-out match=":{theme}/SOME_XS_DATE" type="xs:date" formatMask="DD-Mon-YYYY"/>
```

Exercises

Use your XX_HELLO_WORLD module for the following exercises.

Exercise 1

Add a *fm:expr-out* to *buffer-content* which writes out the text "Goodbye World".

Exercise 2

Use *fm:expr-out* to write out the first and second name, separated by a space, of the second EMPLOYEE in the root DOM.

Exercise 3

Use *fm:expr-out* to write out the sysdate element in the sys DOM in the format *DD-MON-YY*.
Don't forget to save your module changes, flush the module cache and refresh the webpage before testing your changes

References

[1] http://www.w3.org/TR/xmlschema-2/#built-in-primitive-datatypes
[2] http://download.oracle.com/docs/cd/B19306_01/server.102/b14200/sql_elements004.htm

Training:Fox:PlanAppIntroduction

Introduction to the Training Planning Application Demonstration

The purpose of this section is to teach various elements of FOX application development by walking through building a sample application. We will be writing a Demonstration Planning Application Handling System which allows users to create and submit mock applications for building works.

What can I expect?

Each page/section will introduce an element of FOX development with instructions and examples on how to use the feature. At the end of some sections there will be exercises and guidance on adding the feature to the demo system.
The pages will include relevant links to the FOX Reference Guide. While it is not essential that these are explored for the purposes of building the demo, it is recommended as it will provide you with a larger view of FOX's capabilities.

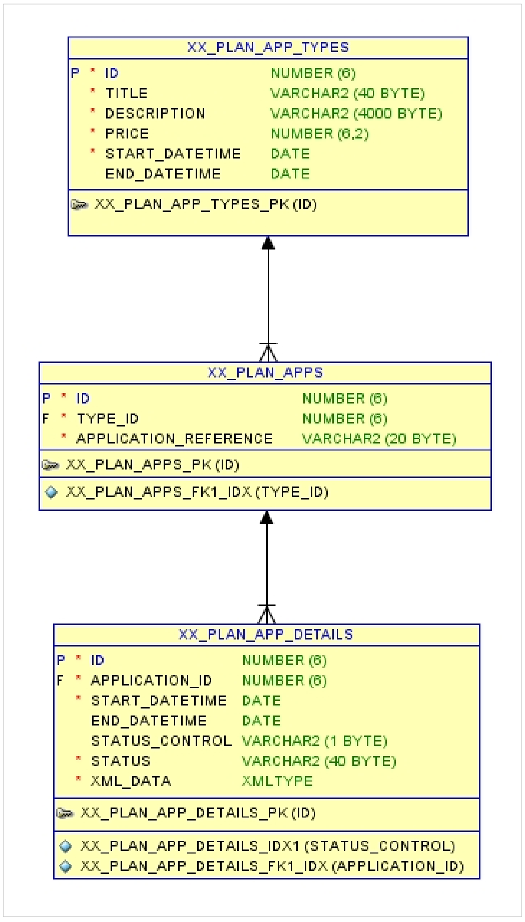
What If I get lost?

Feel free to ask your supervisor/tutor for help with any of the exercises or for further insight into any topic. Also there are sample modules from various stages of the application's progress available in the training environment.

Training:Fox:PlanAppDataModel

Data Model

The image below shows the initial Data Model for the Demo System:


XX_PLAN_APP_TYPES
P * ID NUMBER (6)
* TITLE VARCHAR2 (40 BYTE)
* DESCRIPTION VARCHAR2 (4000 BYTE)
* PRICE NUMBER (6,2)
* START_DATETIME DATE
END_DATETIME DATE
XX_PLAN_APP_TYPES_PK (ID)
XX_PLAN_APPS
P * ID NUMBER (6)
F * TYPE_ID NUMBER (6)
* APPLICATION_REFERENCE VARCHAR2 (20 BYTE)
XX_PLAN_APPS_PK (ID)
XX_PLAN_APPS_FK1_IDX (TYPE_ID)
XX_PLAN_APP_DETAILS
P * ID NUMBER (6)
F * APPLICATION_ID NUMBER (6)
* START_DATETIME DATE
END_DATETIME DATE
STATUS_CONTROL VARCHAR2 (1 BYTE)
* STATUS VARCHAR2 (40 BYTE)
* XML_DATA XMLTYPE
XX_PLAN_APP_DETAILS_PK (ID)
XX_PLAN_APP_DETAILS_IDX1 (STATUS_CONTROL)
XX_PLAN_APP_DETAILS_FK1_IDX (APPLICATION_ID)

NOT NULL columns are marked with an asterix.

Master Detail Pattern

In the model the Planning Application record is split into two tables, XX_PLAN_APPS, the master, and XX_PLAN_APP_DETAILS, the detail.

This pattern is useful in situtations where an audit trail of changes to a document needs to be maintained. Effectively this allows us to maintain a change log of the detail by creating a new current record every time it is updated and archive the previous record.

The key benefits of this model are:

- We have a master id that we can reference consistently;
- The *current* detail record can be edited or viewed easily;
- The entire history of edits and/or key status changes on a record can be queried in the database, qualified by the date created and ended

The 'start_datetime' and 'end_datetime' columns record the time of each update and a 'C' in the 'status_control' column marks the current record as shown in the table below:

START_DATETIME	END_DATETIME	STATUS_CONTROL	Interpretation
20-06-2012	NULL	C	Current record
20-06-2012	21-06-2012	NULL	Past record that has since been updated

Other combinations are not usually permitted.

N.B. The status of the application, whether it is a draft, deleted or submitted, is usually stored in a separate column; in this case 'status'.

Exercises

1. Build the tables shown here in the trainingmgr schema (with XX replaced with your initials) making sure to include the TABLESPACE and XMLTYPE COLUMN .. STORE AS commands. Use ALTER TABLE commands to build the primary keys, indices and foreign keys relationships shown, refering to the Oracle standards if you need.
2. Build a sequence for each table to provide IDs for each record.
3. Perform the following GRANTs from your TABLEs to the APPENV schema to allow the FOX engine to interact with your data model:

TABLE name	GRANT TO appenv
xx_plan_app_types	SELECT, UPDATE, INSERT
xx_plan_apps	SELECT, UPDATE
xx_plan_app_details	SELECT, UPDATE, INSERT

Training:Fox:PlanAppPackages

Database Packages

While PL/SQL is beyond the scope of this training it is necessary to use a minimal amount to create and update records in the data model while using this training application.

A sample package XX_PLANNING_APPLICATION has been created which handles (nearly) all the DML required for the Planning Application Training System. Feel free to spend some time looking over this if you are interested. The package is split into two files, one defining the specification (the public functions, procedures and variables) and one describing the body (the internal code behind the specification and private components).

Interaction with the package will be handled with simple function and procedure calls and code will be provided.

Data Loading script

A PL/SQL script XX_PLANNING_APPLICATION_LOAD has also been created that will load some sample data into the data model. This script uses the XX_PLANNING_APPLICATION package mentioned above to INSERT records into the tables. Again, take time to look this over if you like.

Exercises

The sample files are located in your version control system alongside the HELLO_WORLD samples.

1. Take a copy of the XX_PLANNING_APPLICATION package specification and body and replace all the occurences of XX with your initials, including the package name. Compile the resultant code on the Training Environment and GRANT EXECUTE on the package to the FOX application environment schema APPENV. If any errors occur, it might be worth double checking that your XX_ tables conform to the data model.
2. Take a copy of the XX_PLAN_APP_DATA_LOAD script and replace all the occurences of XX with your initials and run the script on the Training Environment. Check that the data is in your tables and COMMIT the transaction.

Training:Fox:PlanAppXView

Introduction

Building an XView for our Planning Applications will speed searching and ease listing of results later in our system.

See Introducing XViews to learn more about XViews.

Exercises

1. Create an XView from your basetable XX_PLAN_APP_DETAILS, call it XX_XVIEW_PLAN_APP_DETAILS and extract the sample XML data into columns alongside the existing relational columns: Include Contact Names, Address, Postcode, Work Start Date and Detail of Work from the xml and all relational columns.
2. Grant SELECT privileges on your new XX_XVIEW_PLAN_APP_DETAILS XView to the Fox engine "appenv".

Training:Fox:PlanAppQueryIntoDOM

Introduction

Our first task is to build a simple search screen that will query the current Planning Applications in the database.

<fm:query>

fm:query is used to define the query data transformation rules. It is defined in within a **<fm:db-interface>** in the **<fm:db-interface-list>** of a FOX module.

```
  <fm:db-interface name="dbint-department">
    <fm:query name="qry-department">
      <fm:target-path match="./DEPARTMENT_LIST/DEPARTMENT" />
      <fm:select>
SELECT
  d.deptno
, d.name
, d.location
FROM trainingmgr.department d
      </fm:select>
    </fm:query>
  </fm:db-interface>
```

Concepts

<fm:target-path>

The `match` attribute of this element provides a Simple XPath defining an element which will contain the column set for each row returned by this query. It is evaluated relative to the node which was targeted in the `match` attribute of the associated `fm:run-query` command. As it is evaluated relatively, the target-path match can only use a simple XPath and does not support Contexts.

Typically a target-path is defined on a query which will return multiple rows. When a target-path is evaluated, any elements which do not exist are initialised. The rightmost element in a target-path is initialised for every row.

E.g. the target-path `./DEPARTMENT_LIST/DEPARTMENT` will initialise one element called DEPARTMENT_LIST which will contain as many DEPARTMENT elements as there are rows in the result set of the query. Each DEPARTMENT will contain the expected XML structure of a single row.

<fm:select>

The SQL select statement is defined in the **<fm:select>** element.

<fm:run-query>

fm:run-query is used to invoke an SQL statement to retrieve data from the database.

```
<fm:run-query interface="dbint-department" query="qry-department" match=":{root}"/>
```

The Interface and Query attributes are required as they tell FOX which query to use. Match tells FOX what to run the query against.

Exercises

1. Make a copy of the skeleton module XX_PLAN_APP_SEARCH (with XX replaced with your initials) for editing. It would be useful to have clobber set up to clob your changes to the training environment. Once the module is clobbed, run the module to make sure it is working.
2. Add a fm:db-interface and a fm:query to the module to SELECT a list of applications into the :{root} DOM from the data model. Each application should have an id, application_reference, its type's title and its current detail's status together with some of the fields from your XView. Hint: Try the query out in TOAD to make sure that it runs succesfully and returns the data you expect (Do not include closed detail records).
3. Add a fm:action to run the query by placing an fm:run-query in its fm:do element, the match attribute should point to the :{root} DOM.
4. Set out the action on the page, run it and verify that data has appeared in the :{root} DOM. (Don't forget to flush the module cache after making changes).

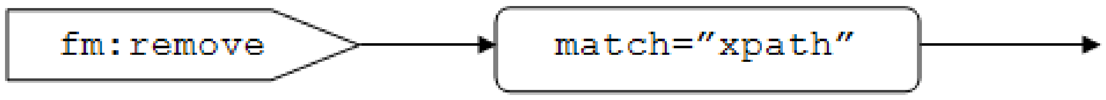
Training:Fox:PlanAppRemoveFromDOM

Introduction

Use fm:remove to remove XML nodes (elements, text, other) from a DOM using XPath.

This is simple but dangerous as entire legs of data can be obliterated in a single invocation.

Syntax



Examples

Remove a whole list

```
<fm:remove match="/*/DEPARTMENT_LIST"/>
```

Remove specific elements under the root node

```
<fm:remove match="/*/DEPARTMENT_LIST/DEPARTMENT/NAME | /*/DEPARTMENT_LIST/DEPARTMENT/LOCATION"/>
```

Remove all departments where DEPTNO is greater than 10 [relative XPath]

```
<fm:remove match="/*/DEPARTMENT_LIST/DEPARTMENT[DEPTNO &gt; 10]"/>
```

Exercises

Make the following changes to your XX_PLAN_APP_SEARCH module:

1. Implement an action to remove the last APPLICATION from the APPLICATION_LIST in the ROOT DOM.
2. Rewrite the action to remove the last APPLICATION's elements but not the APPLICATION element itself.
3. Rewrite the action to remove all the APPLICATIONs of TYPE 'upgrade'.
4. Rewrite the action to clear the APPLICATION_LIST.

Training:Fox:PlanAppSchemaDataStructures

Introduction

This chapter explains the schema data structures that are regularly used in FOX modules in further detail. The schema elements are placed after the **<xs:annotation>** in the FOX module **<xs:schema>** and are based on XSD definitions.
In order to set-out any data from the DOM on the screen the data structure must be defined in this way.

Concepts

Open the module TRAINING_DEPARTMENT_SCHEMA.xml in XMLSPY and look at the schema structure.

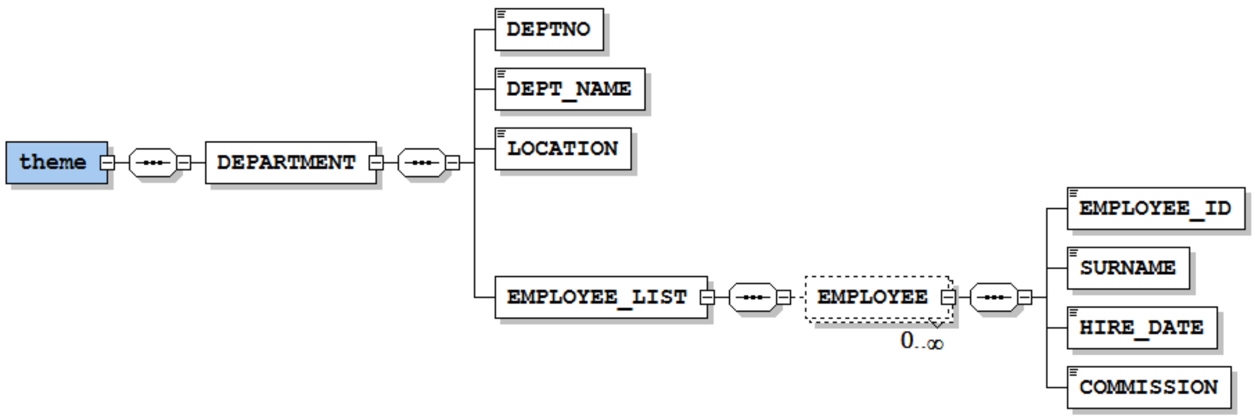
This is a module complete with the schema information.
It consists of two main parts:

1. **<xs:annotation>**
   The module code
2. **<xs:element name="...">**
   The module schema for a specified DOM (eg root)

In text view you will see the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema>
  <xs:annotation>
   ...
  </xs:annotation>
  <xs:element name="theme">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="DEPARTMENT">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="DEPTNO" type="xs:integer"/>
              <xs:element name="DEPT_NAME" type="xs:string"/>
              <xs:element name="LOCATION" type="xs:string"/>
              <xs:element name="EMPLOYEE_LIST">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="EMPLOYEE" minOccurs="0" maxOccurs="unbounded">
                      <xs:complexType>
                        <xs:sequence>
                          <xs:element name="SURNAME" type="xs:string"/>
                          <xs:element name="HIRE_DATE" type="xs:date"/>
                          <xs:element name="COMMISSION" type="xs:integer"/>
                        </xs:sequence>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

In the schema view of the theme element you will see the following:
To access the schema view, click the Schema/Design WDSL view icon on the left of the Text View icon.
Follow this by clicking the diagram icon on the left of the theme element.



What is a simple type?

This refers to a single element with no sub-elements in the schema. These are leaf nodes when viewing the schema in the Schema (tree) View. For example, DEPT_NAME is a simple type.

What is a complex type?

A complex type is an element that has one or more sub-elements contained within it. These are branches when viewing the schema in the Schema (tree) View. For example, theme, EMPLOYEE_LIST and DEPARTMENT are complex types.

What is a repeating element?

This is a complex type that can exist more than once in the DOM while only being defined once. It is typically used to represent a list or a table of data with several rows. They are always defined inside of a list element, usually having the same name but with the suffix "list". For example, EMPLOYEE_LIST is a complex type list which contains the repeating complex type EMPLOYEE.

Exercises

In order to add some search fields to our XX_PLAN_APP_SEARCH module we need to define them in a schema structure.

1. Add a complex type element named "theme" to the schema. Data defined and initialized here will appear in the :{theme} DOM.

2. Define a non-repeating complex type named SEARCH_FIELDS under theme and define the simple xs:string types REFERENCE, TYPE, STATUS, CONTACT_NAME and SITE_POSTCODE as its sub-elements.

N.B. There will be no visible change to the DOMs or the module screen on completing these steps.

Training:Fox:PlanAppInitialisingDataStructures

Introduction

As you can see, defining a schema data structure does not give it an immediate presence in the DOM. Data must be pulled in (by query) to the DOM to match the structure or the structure must be initialised.

Concepts

Consider we are designing a FOX screen that allows us to create a DEPARTMENT in the database.
We wish to have a collection of fields that the user can input data into as shown in the simplified schema structure below:

```
<xs:element name="DEPARTMENT" department:edit=".">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="DEPTNO" type="xs:integer"/>
      <xs:element name="DEPT_NAME" type="xs:string"/>
      <xs:element name="LOCATION" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Since the fields will not be populated by a query, and will be blank on entry to the module, they will need to be initialised explicitly into the DOM in order to be placed on the screen.

<fm:init>

The fm:init command initialises a schema structure into the DOM of a module.
For our DEPARTMENT create screen example this would be required on entry to the module and placed in the **<fm:do>** of a **<fm:entry-theme>** element, although **<fm:init>**s are often seen in **<fm:action>**s.

```
<fm:init target=":{theme}/DEPARTMENT" for-schema="*"/>
```

The target attribute XPath points the FOX engine to a node in the schema structure and the for-schema attribute XPath acts from this node and chooses the schema elements to initialise.
The example above would initialise the DEPARTMENT element and its sub-elements.

XPaths are relative

The XPaths used in these attributes are all relative to the target record.

Consider the following for the for-schema attribute (please revise XPaths if these don't make sense):

for-schema XPath	node-set
"."	The record only, no child elements
"*"	All child elements
"*/*"	All children of all child elements
"//*"	The element and all descendants
"* | INFO/*"	All child elements and all child elements of the INFO element are initialised

Not using several inits

A word about initialising efficiently. If you want to initialise several specific elements that exist in the schema, you can use the pipe, "|". For example, instead of doing <fm:init target="/*/CAT"/> and then <fm:init target="/*/DOG"/> you can do <fm:init target="/*" for-schema="CAT | DOG"/>.
A common mistake that people make is to initialise too much. Consider the following for-schema XPath "/*/SECTION_A//*" that will initialise the whole of section A (with all descendants) when perhaps only a small part of it was needed.

Exercises

1. Add an fm:init command to the entry-theme in your XX_PLAN_APP_SEARCH module to initialise the complex-type SEARCH_FIELDS and its sub-elements. Run the module and check the DOM to make sure these elements appear. Use the EntryPoint button in the Fox testing bar on your Fox module page to see the changes.
2. Read the fm:init reference page carefully. There is a lot of detail, but you do not need to remember it all at the moment; however it is worth knowing that different options are available to you.
3. Backup your working fm:init command (comment it out in the XML). Create a new fm:init call to test out the different variants of the command and see how they behave. Once you have explored the behaviour, restore your backed-up fm:init call and move on to the next section.

Training:Fox:PlanAppFormSetOut

Estimated duration: 2 hours

Introduction

In order to display our newly defined search fields there are several more steps. These involve using namespaces, attributes and the **<fm:set-out>** command to control which elements of a schema data structure are displayed and how.

Concepts

FOX namespace

FOX modules use XML namespaces, in a slightly unorthodox way, to control the content of the screen.
Every fox module has the FOX namespace defined in its **<xs:schema>** element:

```
<xs:schema ... xmlns:fox="http://www.og.dti.gov/fox" ...>
  ...
</xs:schema>
```

This namespace is general to the module and is "always on".
The attributes controlling elements and actions and how they are set out to the screen are of the FOX namespace or a custom namespace equivalent.

Custom namespaces

Custom namespaces within a module allow data to be grouped into sections of data that are displayed differently.
For example a FOX module allowing the user to edit a DEPARTMENTs details may define the department namespace in the **<xs:schema>** element:

```
<xs:schema ... xmlns:fox="http://www.og.dti.gov/fox" xmlns:department="http://www.og.dti.gov/fox" ...>
  ...
</xs:schema>
```

N.B. Currently when using the older XML editor each custom namespace URI should be set to the same as the xmlns:fox URI, in this case *"http://www.og.dti.gov/fox"*.
Using more modern XML editors, these URIs must all be different in order for the module to validate (or fact, be well formed). The current standard is *"xmlns:[namespace-prefix]=http://www.og.dti.gov/fox/[namespace-prefix]"* for each custom namespace. The department namespace URI above would thus become *"http://www.og.dti.gov/fox/department"*.

Marking up schema data for display

The next step is to mark up the schema data elements that we wish to display as part of the namespace's group.
For the DEPARTMENT edit module we might do the following:

```
<xs:element name="DEPARTMENT" department:edit=".">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="DEPTNO" type="xs:integer" department:ro="."/>
      <xs:element name="DEPT_NAME" type="xs:string" department:edit="."/>
      <xs:element name="LOCATION" type="xs:string" department:edit="."/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

- An *namespace:ro* attribute indicates an element is read only. A simple type element will have its data presented as text or read-only input field.
- An *namespace:edit* attribute indicates an element is editable. A simple type element will have its data presented in an input field.

The contents of these attributes is a Complex XPath boolean controlling whether the data is set out.
An expression that returns a populated node-set evaluates as true and one that returns a empty node-set evaluates as false.
The value ".", which is often used as above, matches the current node and as such always evaluates as true.

If, for example, you wished to show the DEPARTMENT's LOCATION field only in the presence of a "flag" element in the :{root} DOM you might do the following:

```
<xs:element name="LOCATION" type="xs:string" department:edit=":{root}/SHOW_DEPT_LOCATION"/>
```

<fm:set-out>

A basic fm:set-out command has the following syntax and is placed within an **<fm:set-page>** or **<fm:set-buffer>** dictating where the data is set out on the page.

```
<fm:set-out match=":{theme}/DEPARTMENT" department:mode="."/>
```

The *match* attribute tells FOX which schema elements to target for set-out (sub-elements included), in this case DEPARTMENT.
The *department:mode* attribute filters these elements to those marked up with the *department:ro/edit* attributes.
The value of this can be used to add another Complex XPath boolean condition to the set out, on top of those defined at the schema level. Both must evaluate true for each element to be set out.

Alternatively the *department:mode* attribute could be replaced with *department:view*. This would force all relevant elements to be set-out in read-only mode:

```
<fm:set-out match=":{theme}/DEPARTMENT" department:view="."/>
```

N.B.

- **Data elements must exist in the DOM; set-out does not create elements.**
- Although the FOX namespace could have been used in the examples, it is considered best-practice to only use it when required.
- Data not setting-out properly is a common hurdle when developing in FOX, look to the fm:set-out checklist for assistance.

Exercises

1. In your XX_PLAN_APP_SEARCH module a search namespace has been provided for you. Append search:edit attributes to the SEARCH_FIELDS complex_type and its simple sub elements that always evaluate true.
2. Add a **<fm:set-out>** to your buffer-content that will display the editable SEARCH_FIELDS to the screen. Run the module and check that these fields are displayed and editable (don't worry about appearance for now).

Training:Fox:PlanAppElementPresentation

Introduction

Having managed to display your SEARCH_FIELDs you may not be happy with the way they have been displayed. This section will introduce some ways to control how an element is displayed by a **<fm:set-out>** command.

Concepts

Without additional information the FOX engine uses its default settings, for example FOX will generate a Field Width of 80 and a Height of 5 for string inputs. There are two ways to override this when implementing a schema structure.

XSD mark-up

Consider the following change to our DEPARTMENT edit screen schema:

```
<xs:element name="DEPARTMENT" department:edit=".">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="DEPTNO" type="xs:integer" department:ro="."/>
      <xs:element name="DEPT_NAME" department:edit=".">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:maxLength value="20"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="LOCATION" type="xs:string" department:edit="."/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

This syntax will inform the engine that the DEPT_NAME element is expected to be no more than 20 chars long and FOX will respond by displaying an input of more suitable size. It also can be used to assist validation of user input; covered later.
Many of the XSD restrictions described in the XML training are supported by FOX and can be very useful.

Namespace attributes on the schema elements

For more aesthetic changes there a number of FOX attributes that can be added, with the fox namespace or a custom one, to schema elements to further control the set-out:

```
<xs:element name="DEPARTMENT" department:edit=".">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="DEPTNO" type="xs:integer" department:ro="." department:prompt="Number"/>
      <xs:element name="DEPT_NAME" department:edit="." department:fieldWidth="10" department:prompt="Name">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:maxLength value="20"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="LOCATION" type="xs:string" department:edit="."/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

The *department:prompt* attributes override the label of the fields they are on. The default label/prompt is the element name init-capped and with underscores replaced with spaces.
If the *prompt* is specified as "" (no prompt text), the prompt will not be displayed at all.

The *department:fieldWidth* attribute will reduce the width of the displayed input further down to 10 characters. It will also increase the height of the input to 2 characters as the undefined dimension takes the value of the data length over the defined dimension.
2 (Height) = 20 (MaxLength) / 10 (Defined Width).

Adding a *department:fieldHeight* of 1 would give a 10 by 1 input, although entry of up to 20 characters would still be permitted:

```
<xs:element name="DEPT_NAME" type="xs:string" department:edit="." department:fieldWidth="10" department:fieldHeight="1"
  department:prompt="Name">
  ...
</xs:element>
```

Other Attributes

The following is a table of other attributes that can be used:

Parameter	Comments	Common Values
widget	Determines the type of widget to enter/display the data for each field **The widgets determine the HTML form widget used. Widgets with a '+' appended will always be rendered using the widget, but it will be greyed out and non-editable in ro mode. Widgets without a '+' will be rendered as text in ro mode.** For example, if you want an element displayed as a read only input box, you will use an 'input+' value on the widget attribute and namespace:ro. NB. There is no difference between input and input+ (same for other widgets) in edit mode.	input input+ date tickbox radio link link+ button button+ selector selector+ file file-new obscure
hint	Help text for the user on a specific data item	String
prompt	The prompt displayed next to the edit box	String
promptAlignX	Aligns prompt on X axis	left right center
promptAlignY	Aligns prompt on Y axis	bottom center top
prompt-short	Static prompt used for column headers on lists etc…	String
promptLayout	Positions the prompt either above or to the left of the element	north west
fieldWidth	Fixed display width for an element's text box	Integer
fieldMinWidth	The minimum width of a data entry text box	Integer
fieldMaxWidth	The maximum width of a data entry text box	Integer
fieldHeight	Fixed display height for an element's text box	Integer
fieldMinHeight	The minimum height of a data entry text box	Integer
fieldMaxHeight	The maximum height of a data entry text box	Integer
fieldxlock	Sets the element's position on the x-axis	Integer
fieldylock	Sets the element's position on the y-axis	Integer

N.B.

The Namespace Attributes will only be used when setting out on the Namespace used; in this case *department*. If you wish the attribute to apply across more than one namespace the *fox* namespace can be used.

Exercises

1. Add XSD Mark-up to the SEARCH_FIELDs of your XX_PLAN_APP_SEARCH module, restrict each of the fields to a sensible length where appropriate.
2. Add some Namespace Attributes to the SEARCH_FIELDs to further affect the display.

Training:Fox:PlanAppSetOutAttributes

Introduction

In addition to the mark-up you can add to the schema data structure to affect the display of a set-out command, there are Namespace Attributes that can be applied to the **<fm:set-out>** itself.

Concepts

These attributes apply to the displayed group as a whole rather than the individual elements. Continuing our DEPARTMENT example:

```
<fm:set-out match=":{theme}/DEPARTMENT" department:mode="." department:promptLayout="north" department:formMaxCols="4" department:formColChars="40"/>
```

The *department:promptLayout* attribute with a value "north" places the labels of each element above the input. The default is "west".
The *department:formMaxCols* and *department:formColChars* attributes define the table dimensions of the group of fields, here 4 columns of 40 characters each.
N.B. When the prompt layout is west both the label/prompt and the field count as separate columns.

Other Attributes

The following is a table of some other attributes that can be used:

Parameter	Comments	Common Values
formMaxCols	Maximum number of columns allowed in the form	Integer
formColChars	Maximum number of characters allowed per column	Integer
fieldClass	Name of a CSS class to apply to this field.	String
fieldStyle	Use this to assign different style sheet attributes	String
debug	Displays a grid in table view and list view	Boolean
promptLayout	The position of the prompt relative to the editable data box	north west

debug is a particularly useful attribute as it will display the grid that the engine is using to set-out the data:

```
  <fm:set-out match=":{theme}/DEPARTMENT" department:mode="." department:promptLayout="north"
    department:formMaxCols="4" department:formColChars="40"
department:debug="true"/>
```

N.B. If you include prompt hints or entry-helper icons, you should set the *formColChars* attributes to be 3 characters longer to accommodate each extra icon. If you do not do this, it may affect the set out you desire.

Exercises

1. Turn on debug mode by adapting the <fm:set-out> in your XX_PLAN_APP_SEARCH module.
2. Use set-out attributes to affect the layout until you are happy with it. Make sure you understand the various effects of each option.
3. Turn off debug mode before continuing.

Training:Fox:PlanAppQueryIntoDOMRevisited

Introduction

Now that we have a group of beautifully set-out search fields we need to link them to our search query so that they affect the data returned.

Concepts

Recall our previous DEPARTMENT example.

```
  <fm:db-interface name="dbint-department">
    <fm:query name="qry-department">
      <fm:target-path match="./DEPARTMENT_LIST/DEPARTMENT">
      <fm:select>
SELECT
  d.deptno
, d.name
, d.location
FROM trainingmgr.department d
      </fm:select>
    </fm:query>
  </fm:db-interface>
```

Imagine that we only want to return DEPARTMENTs where their location contains a character string specified by a field stored in a DOM. We can use a Bind Variable and a **<fm:using>**:

```
  <fm:db-interface name="dbint-department">
    <fm:query name="qry-department">
      <fm:target-path match="./DEPARTMENT_LIST/DEPARTMENT">
      <fm:select>
SELECT
  d.deptno
, d.name
, d.location
FROM trainingmgr.department d
WHERE d.location LIKE '%' || :location || '%'
      </fm:select>
      <fm:using name=":location">:{theme}/DEPT_SEARCH/LOCATION</fm:using>
    </fm:query>
  </fm:db-interface>
```

<fm:select>

The select statement can optionally have Bind Variable placeholders in which case an **<fm:using>** defines where each variable is in a DOM. Bind Variables can be bound by :name or by position number with :1, see :location in the query above.

NB: It is best practise to use named bind variables (i.e. use :location in preference to :1) in order to aid the readability of queries, and to make it easier to run a query in a database IDE.

Best-practise coding standards dictate that named bind variables should be used wherever possible.

<fm:using>

One **<fm:using>** element is required for each Bind Variable and contains an XPath to the data in the DOM.
For named variables the order of the **<fm:using>** elements is not important, numbered Bind Variables will need to be placed in the order their number dictates.
The **<fm:using name=":Bind_Variable_Name">** attribute name is only required when binding by name. e.g. name=":location" above.
The XPath can either be contained in the element as a text node, as above, or within an optional *datadom-location* attribute.
The XPath is evaluated relative to EACH Subject Element (in the run-query match clause, see below).
There is an optional **<fm:using sql-type="">** attribute sql-type that tells Oracle the SQL datatype provided. Default:VARCHAR2
There is an optional **<fm:using datadom-type="">** attribute datadom-type that tells FOX the XMLSchema datatype. The default datadom-type is either obtained from the xs:element type or defaults to xs:string. As xs:date/xs:datetime are often defined on xs:element, there is implicit date conversion.
Entire XML structures can be passed to oracle, e.g.:

```
<fm:using name=":xml_data" sql-type="xmltype" datadom-type="dom" datadom-location="./DETAILS"/>
```

N.B.

- Do not use any of the SQL reserved words as your bind variable names. e.g. :data.
- If data is to be returned back to an existing element, the columns that are queried in either have to have the same names, or use aliases, otherwise new elements will be created in the DOM.
- There is currently a bug in FOX (at time of writing) involving named bind variables if you have an fm:using with a name that is the substring of another. e.g. :p_address and :p_address_id. This should be taken into account when naming bind variables.

Exercises

1. Update your module XX_PLAN_APP_SEARCH to use the values in the search fields to narrow the applications returned. Applications should be returned if their relevant columns contain a superstring of the corresponding search field with data in. Fields left blank should not affect the result. The logic required for this may take some thought.
2. Add a mode attribute to your run-query to ensure old results are cleared from the APPLICATION_LIST before new results are returned.

Training:Fox:PlanAppQueryIntoDOMMergeReplace

Introduction

This section covers the different modes available when running a query.

Concepts

Consider that we want to requery our DEPARTMENT_LIST using various modes.

<fm:query>

```
  <fm:db-interface name="dbint-department">
    <fm:query name="qry-department">
      <fm:target-path match="DEPARTMENT_LIST/DEPARTMENT">
      <fm:primary>
        <fm:key>DEPTNO</fm:key>
      </fm:primary>
      <fm:select>
SELECT
  d.deptno
, d.name
, d.location
FROM trainingmgr.department d
      </fm:select>
    </fm:query>
  </fm:db-interface>
```

<fm:primary>

This element tells FOX the primary key to use when looking for data already in the DOM (usually the primary key of the table).
N.B. The **<fm:key>** element should reference the key as it appears in the DOM (hence the caps in this example).

<fm:run-query>

Modes

```
  <fm:run-query interface="dbint-department" query="qry-department" match=":{theme}" mode="ADD-TO | AUGMENT | PURGE-ALL | PURGE-SELECTED"/>
```

If *mode* is omitted then the FOX engine chooses between ADD-TO and AUGMENT depending on a combination of the run-query *match* and the query *target-path*.

Mode	Description
ADD-TO	Appends new results to existing ones, may result in duplicates.
AUGMENT	Returned results are merged over existing ones where keys match. Returned results element's take precedence except when NULL or non-existent. Unmatched existing results remain unchanged. Unmatched returned results are appended.
PURGE-ALL	Empties the existing results before adding returned results.
PURGE-SELECTED	Empties any existing results where keys match those of returned results. Returned results are appended. Unmatched existing results remain unchanged.

Exercises

Run module TRAINING_PLAN_APP_SEARCH_QMODE and experiment with the buttons available. To see the difference between each of the methods:

- Query using any mode
- Strip to ID
- Init the bonus elements
- Restrict the search with a field entry
- Search with the mode of your choice

Training:Fox:PlanAppQueryOver

Introduction

This section covers querying-over, a useful technique to run a query for every item in a node-set.

Concepts

Consider that we want to append ' (UK)' to the LOCATION sub-element of DEPARTMENT records in the DOM.

<fm:query>

```
  <fm:db-interface name="dbint-department">
    <fm:query name="qry-department-by-no">
      <fm:select>
SELECT
  d.location || ' (UK)' location
FROM trainingmgr.department d
WHERE d.deptno = :deptno
      </fm:select>
      <fm:using name=":deptno">./DEPTNO</fm:using>
    </fm:query>
  </fm:db-interface>
```

<fm:run-query>

Query Over

```
<fm:run-query interface="dbint-department" query="qry-department-by-no" match=":{theme}/DEPARTMENT_LIST/DEPARTMENT"/>
```

The run-query match clause is used to identify subject elements to run the SQL for. The SQL statement will be executed for each subject element identified in the match.

- If there are 50 "DEPARTMENT" elements inside ":{theme}/DEPARTMENT_LIST" above, the SQL statement will be run 50 times.
- If there are no "DEPARTMENT" elements, the SQL statement will NOT RUN at all, but will not error.

SQL bind variables are obtained relative to each subject element ("DEPARTMENT" above) This is a powerful 4GL feature as one fm:run-query can fire off many different queries (one for each match). This technique is referred to as "Query Over". In the example we are querying over "DEPARTMENT" elements.
When match is omitted, default match=".", this SQL runs once with the attach point as a subject.

N.B. This example will run as if it was in AUGMENT mode on a key. As there is a query for each record the key is unnecessary.

Exercises

1. Run the module TRAINING_PLAN_APP_SEARCH_QOVER. Click 'Search' to produce a list of applications and then 'Search Over' which crudely appends '/a' to the REFERENCE of APPLICATIONs of TYPE 'Upgrade'.
2. Think about how this may be done using a Query Over technique and check the module code.

Training:Fox:PlanAppGroupingActions

Introduction

This section covers more advanced techniques for displaying actions on the screen with the use of namespace attributes and **<fm:action-out>**.

Concepts

Namespace Attributes on <fm:action>

Similar to the mark-up on the schema data structure that controls how it is displayed, there are namespace attributes available to control the display of an **<fm:action>** The following are 3 action statements:

```
<fm:action name="action1" ted:run="." fox:widget="button" ted:prompt="Action 1">
  <fm:do>
    ACTION
  </fm:do>
</fm:action>
<fm:action name="action2" tom:run="." tom:widget="button" tom:prompt="Action 2">
  <fm:do>
    ACTION
  </fm:do>
```

```
</fm:action>

<fm:action name="action3" ted:run="." ted:widget="button" tom:run="." tom:widget="link" fox:prompt="Action 3">

  <fm:do>

    ACTION

  </fm:do>

</fm:action>
```

The *run* attributes are similar to the *edit/ro* attributes on a schema element and mark the action for display on a particular namespace.
The *widget* attribute controls how the action is displayed, as a button or a link, and *prompt* controls the wording.
These examples also show how to use more than one namespace on an item, to display it differently under different conditions. The *fox* namespace is used to produce a global effect on display under both the *ted* and *tom* namespaces.

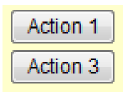
<fm:menu-out>

The **<fm:menu-out>** is used to display groups of actions with common namespaces out on the screen.
It should be used over **<fm:action-out>** where possible.

Using our action examples above, a corresponding menu-out command could look like this:

```
<fm:menu-out ted:mode="." fox:flow="down"/>
```

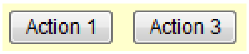
The *mode* attribute works in the same way as it does on an **<fm:set-out>**, stating what namespace should be used with a boolean XPath condition.
The *flow* attribute tells FOX whether to set the actions out "across" the screen, or "down".
This example will set out only the actions with *ted:run* down the screen:



If the menu-out command looked like this:

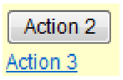
```
<fm:menu-out ted:mode="." fox:flow="across"/>
```

The result will look like this:



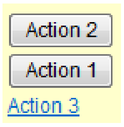
Using the *tom* namespace, action 3 is rendered as a link rather than a button:

```
<fm:menu-out tom:mode="." fox:flow="down"/>
```



By using both namespaces, all the buttons are set out:

```
<fm:menu-out tom:mode="." ted:mode="." fox:flow="down"/>
```



Ordering Actions

You can re-order actions using the *displayOrder* attribute. This forces FOX to layout the actions in the specified order. By convention the *displayOrder* values are usually incremented by 10 to allow additional actions to be added in between at a later date if required.

```
<fm:action name="action1" ted:run="." fox:widget="button" ted:prompt="Action 1" fox:displayOrder="10">

  <fm:do>

    ACTION

  </fm:do>

</fm:action>

<fm:action name="action2" tom:run="." tom:widget="button" tom:prompt="Action 2" fox:displayOrder="20">

  <fm:do>

    ACTION

  </fm:do>

</fm:action>

<fm:action name="action3" ted:run="." ted:widget="button" tom:run="." tom:widget="link" fox:prompt="Action 3" fox:displayOrder="30">

  <fm:do>

    ACTION

  </fm:do>

</fm:action>
```

Exercises

1. Update your XX_PLAN_APP_SEARCH module by setting out the following actions under the search fields using an **<fm:menu-out>**. You can either use the existing custom namespace *search* or define a new one. (You may have some of these actions already defined).
   1. "Search": Runs the search query and populates the :{theme} DOM with the results.
   2. "Clear Fields": Clears the search fields. (This may take two commands).
   3. "Clear Results": Removes the results from the :{theme} DOM.
2. Experiment with the namespace attributes available until you are happy with the menu. Make sure that you define a *displayOrder* on each action to ensure consistent placement.

Training:Fox:PlanAppListSetOut

Introduction

The next step for our search module is to display the results on the screen rather than leave them hidden in the :{theme} DOM.

Concepts

Setting out a list using FOX is very similar to setting out a form.
The first step is to define the list in the schema data structure complete with namespace attributes.

Consider the EMPLOYEE_LIST extracted and updated from our DEPARTMENT schema example.
We wish for the elements to be read-only:

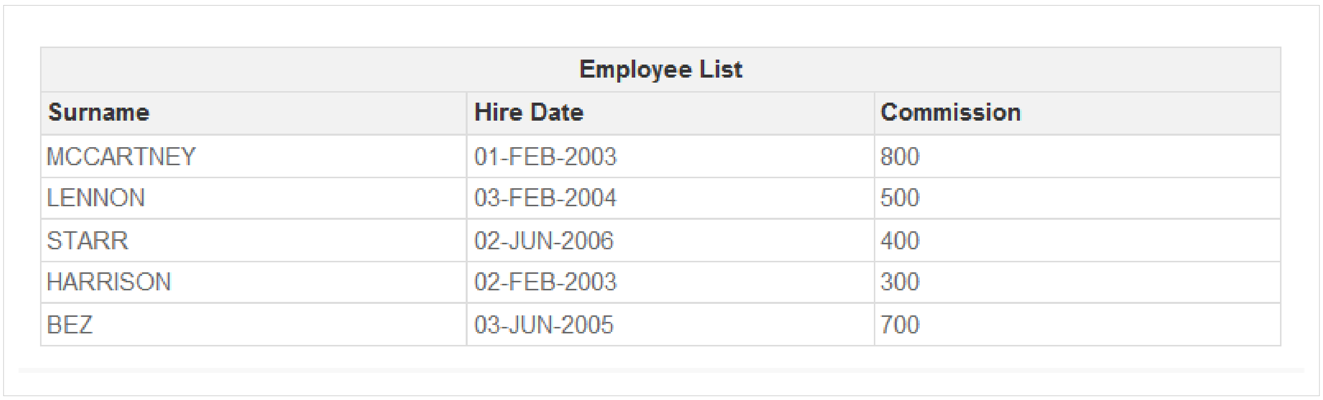
```
<xs:element name="EMPLOYEE_LIST" employee:ro=".">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="EMPLOYEE" minOccurs="0" maxOccurs="unbounded" employee:ro=".">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="EMPLOYEE_ID"/>
            <xs:element name="SURNAME" type="xs:string" employee:ro="."/>
            <xs:element name="HIRE_DATE" type="xs:date" employee:ro="."/>
            <xs:element name="COMMISSION" type="xs:integer" employee:ro="."/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

The read-only namespace attribute *employee:ro* has been appended to fields that are to be set-out.
Note that EMPLOYEE_ID is not marked-up as we do not wish it to be set-out. It is not necessary to define such elements, but it has been included for clarity.

The set-out is performed as before:

```
<fm:set-out match=":{theme}/EMPLOYEE_LIST" employee:mode="."/>
```

FOX will display data as a list view if it finds at the target element, a child element with a maxOccurs="unbounded". Else, it will display your data as a form.
A populated EMPLOYEE_LIST will display as shown:



Employee List		
Surname	**Hire Date**	**Commission**
MCCARTNEY	01-FEB-2003	800
LENNON	03-FEB-2004	500
STARR	02-JUN-2006	400
HARRISON	02-FEB-2003	300
BEZ	03-JUN-2005	700

A lot of the namespace attributes that apply to form set-outs are valid for list set-outs too. For example, to remove the "Employee List" title, we could add a *employee:prompt* with an empty value "" to the EMPLOYEE_LIST schema element.

Exercises

1. Write a schema data structure for your application search results in XX_PLAN_APP_SEARCH, choose which elements to display and mark them up with read-only namespace attributes.
2. Add a set-out below your search action menu to display the returned results.

Training:Fox:PlanAppAssigningAndSettingData

Introduction

Sometimes it is important to be able to assign or initialise elements using hard-coded values, XPath Calculations or data from a DOM **<fm:assign>** is a useful tool to accomplish this.

Concepts

<fm:assign>

There are two ways of using the **<fm:assign>** function. The first one initialises the element you are referring to if it doesn't already exist:

```
<fm:assign initTarget=":{root}/A" textValue="PRIMARY_DATA"/>
```

The other only assigns a value to an existing target element:

```
<fm:assign setTarget=":{root}/A" textValue="PRIMARY_DATA"/>
```

As you can see you must supply a target (that is initialised or just set) and a value. This value can be a text value, in the above example the target will be set to "PRIMARY_DATA". You can also use an XPath expression in the following way:

```
<fm:assign initTarget=":{root}/A" expr=":{root}/B"/>
```

N.B.: <fm:assign initTarget="xPath"/> only works for a single element. To use **<fm:assign>** with a repeating element structure (including list items, check boxes, radio groups, etc.), you would first need to init the elements using **<fm:init>**, followed by **<fm:assign setTarget="XPath"/>**.

Initialising a tree structure

Whenever you are initialising data, using **<fm:init>** or **<fm:assign>**, if you target a node in a tree structure that does not exist, FOX will initialise all its ancestors at the same time. See Example 1 below.

N.B.: You cannot use an fm:assign to initialise repeating elements. In these cases, you must first initialise the elements and then assign values to them.

The :{assignee} context

When you are assigning a value using a setTarget and an XPath, the :{assignee} context refers to the original value of the element. This means you can use it in your assignment. See Example 2 below.

Examples

Example 1

The following assignment is used in a module to create a CONTEXT structure. By using this assignment the CONTEXT element is created, as is the SUBJECT sub element that in turn has a sub element called REF_ID1.

```
<fm:init target="/*/CONTEXT/SUBJECT/REF_ID1"/>
<fm:assign setTarget="/*/CONTEXT/SUBJECT/REF_ID1" expr="sring(concat(:{params}/P_ID), 'OU')"/>
```

The REF_ID1 element will get the value of the XPath which is concatenating the P_ID element in the :{params} DOM with the letters 'OU'.

Example 2

You can use the :{assignee} context to compute the new value when you assign a value to an element:

```
<fm:assign initTarget="/*/ITEM/PRICE" expr="round(:{assignee} div 100 * 105 * 100) div 100"/>
```

Example 3

You can use :{assignee} multiple times within an fm:assign statement:

```
<fm:init target="/*/EMPLOYEE_LIST/EMPLOYEE/FULLNAME"/>

<fm:assign setTarget="/*/EMPLOYEE_LIST/EMPLOYEE/FULLNAME" expr="string(concat(:{assignee}/../FORENAME, ' ' , :{assignee}/../SURNAME))"/>
```

Exercises

1. Add another action within your XX_PLAN_APP_SEARCH module that creates an APPLICATION element with hard coded values for its sub-elements. This should set-out as if it had been the result of your search query. Check this works before removing the action.

Training:Fox:PlanAppListEvaluateContext

Introduction

Now that we have an operational Application search screen it would be nice to be able to do something with the returned results.
The next few sections will cover the techniques required.

Concepts

Recap

Recall the list of some commonly used DOMs:

DOM Name	Scope	Context	Description
Data	1 Module Call	:{root}	This is the main area of data storage and can be easily written back to the database in a Storage Location.
Theme	1 Module Call	:{theme}	Used to store data that you don't want stored with the module data and is usually used for presentation fields and storing temporary values or flags. You can mark up the Theme DOM.
Temp	1 Page Request	:{temp}	Used to set values or collate data from multiple sources to be used in one place for a transaction, but its life span is very short and cannot be used for setting out data.
Session	1 Browser Session	:{session}	Used for one Browser session, that is for the life of one Browser window. It is useful for storing copied data to be pasted into different modules.
User	1 Portal Session	:{user}	Stores information about the current user logged in to the UK Oil Portal. You should not write information to it.
Sys	1 Module Call	:{sys}	Similar to the Session DOM, but is constantly updated by FOX. You should not write information to it.
Params	1 Module Call	:{params}	Contains any parameters passed to the module from when it was called. You should not write data to it.
Environment	Module call (copy forward)	:{env}	This acts as a pseudo-combination of the :{params} and :{session} DOMs. Whereas the :{params} DOM only contains data passed forward on the most recent module call, the :{env} DOM can be written to or read from at any time (regardless of module calls), much like the :{session} DOM - but data is only ever passed forwards when calling a module. When you exit a module, any changes made to the :{env} DOM are not visible to the caller.
Error	1 Module Call	:{error}	Contains a list of validation errors, generated when the fm:validate command is run. You don't generally write data to it, and normally only to amend error text content.

Predefined Contexts

FOX also has predefined contexts for different parts of a DOM:

- :{action} references the context of the current action. You will learn about contextual actions when you read about phantoms in a later chapter.
- :{attach} references the current attach point in the Data or Theme DOM. You can also use the "." notation.
- :{baseself} is equivalent to the 'base' element of the XPath expression (i.e. the node that would be returned by an XPath of "."). This is useful for back-referencing within predicates as XPath does not natively provide a method for doing this.

  For example, within

```
fox:validate="./A/B[. = 1 or . = :{baseself}/C]"
```

  the "." within the predicate refers to element B. The :{baseself} context refers to the XPath's evaluate context (the initial "." in the outer statement).

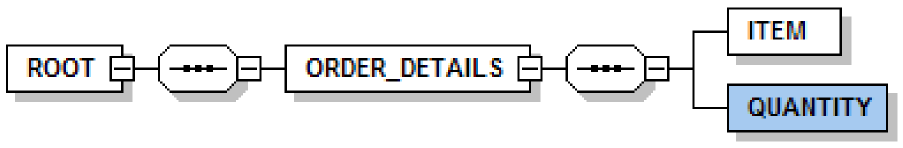
You can also create your own contexts in the **<fm:context-localise>** command and as a pointer in a **<fm:for-each>** loop.

The "Evaluate Context" Rule

The "Evaluate Context" rule dictates where a relative XPath expression evaluates from, using the notation "." (period) to reference the evaluation point.
The rule is that:

- "." evaluates to the current element if the element is complex.
- "." evaluates to the current element's complex parent if the element is simple.

For example, in this diagram:



If you were trying to mark-up the QUANTITY element, "." would evaluate to ORDER_DETAILS which is the last complex element. However, if you were marking up ORDER_DETAILS then "." would evaluate to ORDER_DETAILS.

Training:Fox:PlanAppPhantoms

Introduction

You have previously seen how you can set out buttons/links using the **<fm:action-out>** and **<fm:menu-out>** functionality. These do not allow buttons/links to be displayed in amongst standard **<fm:set-out>** form or list displays. A more severe limitation is that these actions run in a global context, they are not anchored to any particular data; this makes it impossible to add an action to a row in a list that removes or edits that row.

This is where the Phantom Elements come in.

In a list set-out for example; a phantom can be used to add an action to every row of the list. With a list of EMPLOYEEs you may wish to give the option to remove each employee next to each row.

Phantom Elements are specified in the schema part of your FOX module. They are simply placeholders for buttons or links. Phantom Elements are set out using the same size and positioning attributes as for other list/form display elements.

The Phantom Element is the only case where **<fm:set-out>** displays something when no data exists in the DOM (i.e. element node). As no data exists for Phantom Elements, the **<fm:init>** and **<fm:validate>** commands ignore them also - it's as though they are not defined in the Schema. Note that while a phantom does not correspond to an element in the DOM, the schema element that contains the phantom *must* exist.

Concepts

You must include the library 'LAYOUT1' in the module in order to use phantoms.

Schema Definition

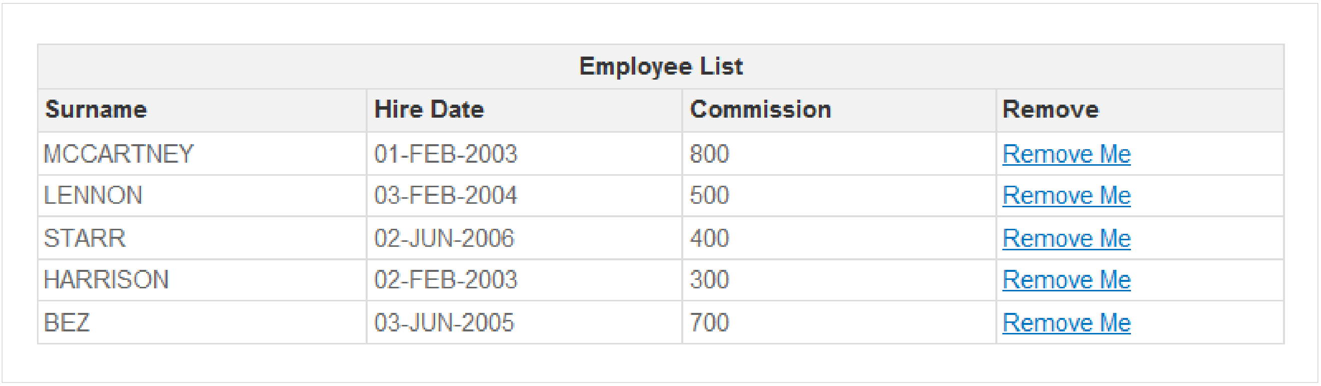
The definition in the schema of a typical phantom element can look as follows:

```
  <xs:element name="EMPLOYEE" minOccurs="0" maxOccurs="unbounded" employee:ro=".">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="EMPLOYEE_ID"/>
        <xs:element name="SURNAME" type="xs:string" employee:ro="."/>
        <xs:element name="HIRE_DATE" type="xs:date" employee:ro="."/>
        <xs:element name="COMMISSION" type="xs:integer" employee:ro="."/>
        <xs:element name="phantom-remove" type="phantom" employee:ro="." employee:run="."
          employee:widget="link" employee:action="action-remove"
          employee:confirm="string(concat('Are you sure you wish to
remove ', ./SURNAME, '?'))"
          employee:prompt="Remove Me" employee:prompt-short="Remove"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
```

This phantom element is called "phantom-remove". Standards dictate that phantoms should be named "phantom-[action]" where [action] is what the phantom action does in lower case. This makes them easy to distinguish from other **<xs:element>**s.
Note the following:

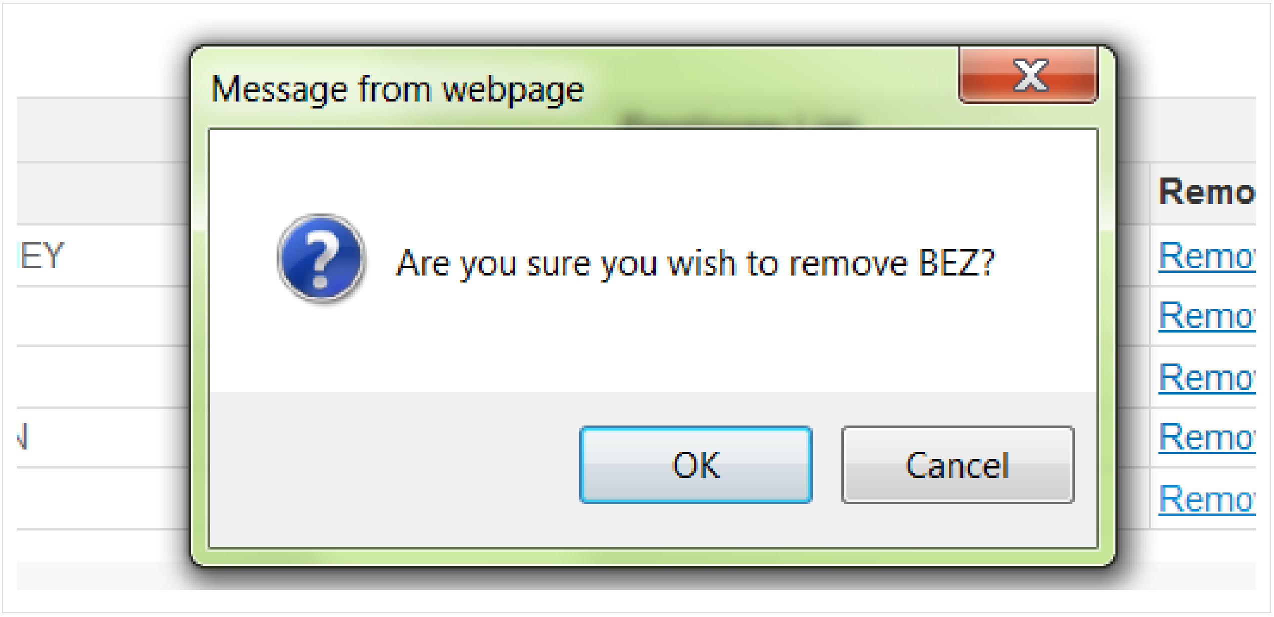
- The *type* attribute has been set to phantom.
- There is both an *edit/ro* and a *run* namespace attribute that evaluates true.
- The namespace attribute *widget* has been set to display the action as a link.
- The namespace attribute *action* tells the FOX engine which action to run when clicked.
- The optional namespace attribute *prompt* controls the text of the link.
- The optional namespace attribute *prompt-short* controls the text of the column header.
- The optional namespace attribute *confirm* asks the user to confirm (using the value as the prompt) with a javascript window before running the action. In this case an XPath has been used to add the employee's name to the confirm prompt.

Here is an example of how this list could be displayed:



Employee List			
Surname	Hire Date	Commission	Remove
MCCARTNEY	01-FEB-2003	800	Remove Me
LENNON	03-FEB-2004	500	Remove Me
STARR	02-JUN-2006	400	Remove Me
HARRISON	02-FEB-2003	300	Remove Me
BEZ	03-JUN-2005	700	Remove Me

And here is an example of the confirmation dialogue:



Message from webpage

Are you sure you wish to remove BEZ?

Action Definition and the :{action} Context

The phantom's related action is defined in the same way as before. However, when an action is run from a schema defined element there is the additional option of using the :{action} context in the action code.
The :{action} context follows the "Evaluate Context" rule from the previous section starting from the element from which the action is referred to.
Thus generally in a list you can use the :{action} context to point to the collection/record containing the phantom element, in the example the EMPLOYEE element.
The :{action} context exists only after the action is fired therefore you won't be able to use it as part of a confirm

alert (Try "." instead, as above).

Following on from the example, here is how "action-remove" could be defined:

```
<fm:action name="action-remove">
  <fm:do>
    <fm:remove match=":{action}"/>
  </fm:do>
</fm:action>
```

N.B. This will only remove the element from the DOM and screen, not the Database record.

Exercises

1. Add a remove link similar to the EMPLOYEE example to each APPLICATION result in your XX_PLAN_APP_SEARCH module.
2. Add a confirm to the phantom that includes the application reference in the dialogue.

Training:Fox:PlanAppCallingModules

Estimated duration: 1 hour

Introduction

Now that we have a functional search module, it is time to introduce another module and link them together.
This section will cover calling other modules and passing parameters between modules.

Concepts

To call another module from a FOX module you can use **<fm:call-module>** . You can pass parameters to the new module and you can receive data back from it when it returns.

<fm:call-module>

An fm:call-module call is quite straight forward. It is done in an action:

```
 <fm:call-module module="TRAINING_EMPLOYEE_EDIT" theme="edit" type="modal"
 callback-action="action-department-edit-callback"
params=":{action}/EMPNO"/>
```

This will call the module TRAINING_EMPLOYEE_EDIT.
FOX maintains a module stack for each user session. Every time a **<fm:call-module>** is executed the called module is pushed on to the stack.
Performing an **<fm:exit-module>** will pop the top module from the stack and return the user to their previous module:

```
 <fm:exit-module/>
```

Call type

You can specify two different types of module calls: modal and modeless. Modal is when the new module replaces the old one in the same window, modeless will open a new window and navigation can continue in both windows.

NB: You should not use modeless as a call type any more. Instead use either modeless-orphan-same-session or modeless-orphan-new-session (see here for a description of the different call types).

Callback-action

You can specify an action in your module that will run when a called module returns control to the caller.

Theme

Theme here is the name of the entry-theme that you wish to enter the called module on. Multiple entry-themes for modules will be covered later. The name of the Theme DOM stems from the fact that it is valid for entering into a module on an entry theme. This means that the theme DOM is valid for each module from entering on an entry theme to exiting the module.

When a new module is called, an empty Theme DOM is created for that entry theme. When an exit-module is called and focus returns to a previous module, the Theme DOM for that module is restored.

Params

The params attribute can be specified and is an XPath that points to values that will be passed to the new module. These parameters are made available to the new module in the :{params} DOM in elements of the same names as in the calling module’s DOM.

N.B. You can pass more than one parameter using a pipe '|' in the XPath or set up a complex element in the Temp DOM and pass the contents of that element (i.e element/*).

Parameters can also be passed into the :{params} DOM using an HTML GET. e.g.

Say this URL accesses the training module on the edit theme.

```
http://edu-address.../TRAINING_EMPLOYEE_EDIT/edit
```

Then **this** URL...

```
http://edu-address.../TRAINING_EMPLOYEE_EDIT/edit?EMPNO=35
```

Would insert **this** data structure into the :{params} DOM

```
<EMPNO>
  35
</EMPNO>
```

This is generally a handy way to call a fox module with parameters without using another module to provide them. In practice this is more often used in emailed links (such as password resetting or questionnaire's)

N.B. be careful not to expose internal database ID's. **?USER_ID=32** is bad practice.

Returning Data

If your module is called by another then during the use of your module data can be written to the :{return} DOM. The data in this DOM will be returned to the calling module when you perform an **<fm:exit-module>**.

Data returned from a module is available to the calling module in the :{result} DOM. There will be an exercise in this on a later sheet.

Exercises

1. Download a copy of XX_PLAN_APP_EDIT and rename it with your initials.
2. Replace the remove phantom in your XX_PLAN_APP_SEARCH module with an edit phantom that calls your XX_PLAN_APP_EDIT and passes in the APPLICATION row's application id as an element called APPLICATION_ID. Note you may need to update your search query to return the APPLICATION_ID when fetching rows.

Training:Fox:PlanAppMovingAndCopyingData

Introduction

This section introduces the **<fm:copy>** and **<fm:move>** commands. It can be necessary to copy or move nodes from one DOM to another; particularly when passing data between modules or building temporary structures to pass to the database.

Concepts

<fm:copy>

A **<fm:copy>** command only has two attributes, a source and a destination:

```
<fm:copy from=":{action}" to=":{return}"/>
```

This would copy the element matched by the :{action} context to the :{return} DOM ready to be passed back to a calling module's :{result} DOM on exit.

<fm:move>

A **<fm:move>** works just the same as copy command except the source element is removed from the relevant DOM:

```
<fm:copy from=":{result}/EMPLOYEE" to=":{theme}/DEPARTMENT/EMPLOYEE_LIST"/>
```

This would extract and remove a returned EMPLOYEE element from the :{result} DOM and append it to a DEPARTMENT's EMPLOYEE_LIST.

N.B. Don't forget it is possible to target just the text data of an element in XPath by appending '/text()'.

Exercises

1. On exit from your XX_PLAN_APP_EDIT copy the id of the APPLICATION being edited question back to the search screen. Change your XX_PLAN_APP_SEARCH screen to re-query this application so that the changes are reflected in relevant result row.

Training:Fox:Editing Data with an API

Introduction

Until now we have only been querying and displaying the data currently in the database. Obviously, we need to be able to edit the data and save it to the database as well as just fetching it.
This worksheet will explain how we do this, by introducing the *fm:api* command.

fm:api allows you to embed anonymous PL/SQL blocks and DML statements and then run them with *fm:run-api*.

They differ from *fm:query* blocks because they do not return result sets, but you can use bind variables both as an input and as an output, so data can be returned in that way.

Syntax

The *fm:api* command specifies the API block within an *fm:db-interface*. When creating a *db-interface*, you can mix and match both *fm:query* and *fm:api* blocks, but the *fm:query* ones must be declared before the *fm:api* ones.

```
<fm:api name="api name">
  <fm:statement>
PL/SQL or DML statements go here
  </fm:statement>
  <fm:using
     [name="name of bind variable" ]
     [direction="in|out|in out"]
     [datadom-type="date|datetime|dom|string|time" ]
     [datadom-location="XPath expression of local value to bind"]
     [sql-type="clob|date|varchar|xmltype"]>
        XPath Expression (If datadom-location doesn’t exist)
  </fm:using>
</fm:api>
```

fm:api blocks can be called in a similar way to *fm:query* blocks, by using the fm:run-api command in an *fm:do* block.

The *fm:using* child element has optional attributes and can contain an XPath. It can be used in the same way as an *fm:query* using clause, but it is possible to specify the direction and type of a bind variable.

When specifying a datadom-type, it is customary to use a datadom-location to match an XPath, rather than include an XPath expression as a text node within the *fm:using* tag elements. The datadom-type specifies the type of data coming from or going to the location targeted by the XPath expression and the sql-type specifies a value similar to that of Oracle’s own datatypes to dictate how Oracle should treat it.

Under most circumstances, FOX should guess the datatypes, but occasionally it is necessary to override this.

fm:run-api attributes:

Attribute	Values
api	Name of API to run
interface	Name of db-interface
match	XPath to run API in context of
mode	ADD-TO AUGMENT PURGE-ALL PURGE-SELECTED

The *fm:run-api* command runs similarly to the *fm:run-query* statement, except the API attribute is used to run a specific *fm:api*.

The match clause makes the API execute over one or more elements, changing the context of each execution based on the XPath expression. For instance, the bind variable XPath will run in context of the match clause, so ./NAME will be evaluated to <match_clause>/NAME.

Examples

The *fm:statement* in the *fm:api* can either contain an anonymous PL/SQL block or a DML statement. As a PL/SQL block with bind variables, it can look as follows:

```
  <fm:api name="api name">
    <fm:statement>
DECLARE
  l_mynum NUMBER(4) := :num;
  l_mystr VARCHAR2(30) := :str;
  l_myxml XMLTYPE;

BEGIN

  SELECT
    XMLELEMENT("MY_DATA"
    , XMLELEMENT("MY_STRING", l_mystr)
    , XMLELEMENT("MY_NUM", l_mynum)
    )
  INTO l_myxml
  FROM dual;

  :xml_out := l_myxml;

END;
    </fm:statement>
    <fm:using name=":num">/*/SOME_NUMBER</fm:using>
    <fm:using name=":str" direction="in" datadom-type="string"
              datadom-location="/*/SOME_STRING" sql-type="varchar" />
    <fm:using name=":xml_out" direction="out" datadom-type="dom"
              datadom-location="/*/SOME_XML" sql-type="xmltype" />
  </fm:api>
```

The above code will take a Number and String from the Data DOM, binding it to the PL/SQL. The PL/SQL will do some processing, in this case it will generate an XML structure based on the two bind variables, and then it will bind the result outwards to a location in the Data DOM as XML.

The first and second fm:using statements show the different syntax that can be used. For the first fm:using statement, only one attribute is set and the XPath is specified as a text node, however for the second statement, everything is specified, overriding FOX's own guess.

A *fm:api* using a DML statement might look like this:

```
  <fm:api name="api name">
    <fm:statement>
UPDATE trainingmgr.department d
SET d.location = 'London'
WHERE d.id = :dept_id
    </fm:statement>
    <fm:using name=":dept_id">ID</fm:using>
  </fm:api>
```

NB: Anonymous blocks and DML statements are not automatically committed by the *fm:api*, you need to manually commit any changes using

```
  <fm:transaction operation="COMMIT"/>
```

It should also be noted that the value of 'operation' is case-sensitive.

Warning

Never issue a manual COMMIT statement inside an fm:api command, and remember to check that any PL/SQL functions or procedures referenced do not contain any COMMIT statements. All of the code run inside an fm:api should be transaction-safe, so that if a subsequent error occurs the entire workload can be rolled back. This way, the database state is preserved; either everything succeeds, or nothing is changed.

Exercises

Use your XX_PLAN_APP_EDIT (where XX are your initials) file for the following exercises.

Exercise 1

Write an API called "api-type" which sets the price of the application type with title 'New', in the trainingmgr.xx_plan_app_types table (where xx are your initials), to 30.
Create an action that runs your api and commits the change.
Set this action out on the screen somewhere.

NB: Use Toad to view your change

Exercise 2

Write an API called "api-save-app" in a db-interface called "dbint-save-data".
This API should update the xml_data column of the current detail for the selected application.
Create an action called "action-save-changes" that runs this api and commits the change. Use the *edit* namespace for this action. Make the action a button with the prompt "Save Changes" and add a confirm that asks the user to confirm that they want to save their changes.
Set this action out on the screen somewhere.

NB: Look at the first example for clues on how to bind in the xml_data from the root DOM - The 'sql-type' and 'datadom-type' attributes of the 'using' element MUST be specified!

Make a change to the one or more of the fields and run your action.
Return to the search screen, and then edit the same application again. You should notice that your changes were loaded, as you updated the data stored in the database.

Exercise 3

Alter your "api-save-app" API to use the following anonymous block, instead of an update statement (replace xx with your initials)

```
DECLARE
  l_app_detail_id NUMBER(10);
BEGIN
  l_app_detail_id := trainingmgr.xx_plan_app.update_application_data(
    p_application_id => :app_id
  , p_xml_data => :xml
  );
END;
```

This piece of PL/SQL calls a function in the trainingmgr.xx_plan_app.update_application_data package which handles updating the xml_data of the detail record for you.

NB: You may need to change the names of the bind variables to match those you used in Exercise 2

Training:Fox:PlanApp Entry Themes

Introduction

On this worksheet, you will learn about code entry points, using FOX's Entry Themes.

Concepts

One Fox Module can be used for a variety of different purposes. E.g. Manage Orders:

- To capture a new order. All information is editable.
- To view an existing order. All information is presented read only.
- To edit an "In Progress" order. All information is shown, only some fields are editable.

The Manage Orders Fox Module could have 3 entry themes: "new", "view", "edit". When a Fox Module is called, you can choose which entry theme to use .
fm:entry-theme is a definition of how the Module can be called, and what should happen.

- One module call has only one theme of entry. This theme is fixed until module ends.
- *fm:storage-location* references the Data DOM Storage Location (see Storage Location chapter for more information).
- *fm:state* specifies the initial module state when entry theme is used.
- *fm:attach* specifies the initial attach point when entry theme is used.
- *fm:do* is the code that should be run when entry theme is used.

The entry theme *fm:do* block is often used to initialise or refresh DOM data on entry. E.G.

- Theme "new", all XML DOM structures are initialised (*fm:init*) before user input starts.

- Theme "view", no actions are taken, data must remain unchanged.
- Theme "edit", ordered product prices in DOM are refreshed from the PRODUCT table.

Any FOX meta-code can be used in the entry theme *fm:do* block, including:

- *fm:run-query* to load data from database into the DOM.
- *fm:attach* to change the attach point - required when attach point data only just created.
- *fm:state* (push or replace) – change module state depending on condition

Example

```
<fm:entry-theme name="create">
  <fm:storage-location>sl-create</fm:storage-location>
  <fm:state>state-create</fm:state>
  <fm:attach>/CREATE_APPLICATION</fm:attach>
  <fm:do>
    <fm:init target=":{root}" for-schema="*"/>
  </fm:do>
</fm:entry-theme>
```

The example entry theme above is called *create*, uses the storage location *sl-create* and state *state-create* and attaches to <*storage-location-root-element*>*/CREATE_APPLICATION*.
The *fm:do* block inits the data DOM using a schema.

Exercises

Use your XX_PLAN_APP_EDIT (where XX are your initials) from the previous training sheets for these exercises.
NB: If you were unable to complete any of the exercises from the previous training sheet, create a fresh copy of TRAINING_PLAN_APP_EDIT and and save it as XX_PLAN_APP_EDIT (where XX are your initials). Make sure to change the *fm:name* element too.

Exercise 1

Your module currently has an entry theme called *edit*.
Add a second entry theme called *view*, which uses:

- *sl-main* storage location
- *state-view* state
- /* as an attach point
- Runs the query *qry-get-app* matching on :{root}

Exercise 2

Add a state to your module called *state-view*.
Override the *action-back* action in this state, to just exit the module without showing a confirm message. You will need to create an <*fm:action-list*> within your new state and create a new action in it called *action-back* (this will override the globaly defined *action-back* action).
Define a new namespace called *view*.
Mark up the XML scheme to be read only with the view namespace.
Set out the schema in the view state; use the same layout as the edit state, only using the view namespace.

Directly call your module to test your changes to the entry theme and state.

Exercise 3

Return to your XX_PLAN_APP_SEARCH module.
Add a second phantom to the APPLICATION element in the schema which calls an action called *action-view-app*.
Define *action-view-app*. This action needs to call XX_PLAN_APP_EDIT using the view entry theme.

Training:Fox:PlanAppPhantomImageButtons

Introduction

This section introduces an alternative way of presenting actions on the screen, by using image buttons instead of links or text buttons.
This can save valuable screen space, especially when setting out phantom actions for each row of a table.

Concepts

Image Buttons

The syntax to use an image as a button is based on that of a text button, e.g.:

```
  <fm:action name="action-go" go:widget="button" go:imageUrl="img/silk_accept" go:prompt="Go" go:hint="Make it so!">

    ...

  </fm:action>
```

The namespace attribute *imageUrl* indicates an image button and provides the image. The namespace attribute *prompt* now forms the title of the on-hover hint overlay, with *hint* providing the text.

Similarly for a phantom element:

```
  <xs:element name="phantom-remove" type="phantom" employee:ro="." employee:run="." employee:widget="button" employee:imageUrl="img/silk_delete"

    employee:action="action-remove"

employee:confirm="string(concat('Are you sure you wish to remove ',

./SURNAME, '?'))"

    employee:prompt="Remove" employee:hint="Remove this employee"/>
```

Multiple Phantoms

Sometimes it is useful to be able to group several row action phantoms together in one ACTIONS column rather than define a separate column for each. This can be done by wrapping the phantoms in an 'ACTIONS' complex element in the schema data structure:

```
...

  <xs:element name="EMPLOYEE" minOccurs="0" maxOccurs="unbounded" employee:ro=".">

    <xs:complexType>

      <xs:sequence>

        <xs:element name="EMPLOYEE_ID"/>

        <xs:element name="SURNAME" type="xs:string" employee:ro="."/>

        <xs:element name="HIRE_DATE" type="xs:date" employee:ro="."/>

        <xs:element name="COMMISSION" type="xs:integer" employee:ro="."/>

        <xs:element name="ACTIONS" employee:ro=".">
```

```
          <xs:complexType>

            <xs:sequence>

              <xs:element name="phantom-view" type="phantom" employee:ro="." employee:run="." employee:widget="button" employee:imageUrl="img/silk_magnifier"

                employee:action="action-view" employee:prompt="View"

employee:hint="View this employee" employee:fieldWidth="1"/>

              <xs:element name="phantom-edit" type="phantom" employee:ro="." employee:run="." employee:widget="button" employee:imageUrl="img/silk_page_edit"

                employee:action="action-edit" employee:prompt="Edit"

employee:hint="Edit this employee" employee:fieldWidth="1"/>

            </xs:sequence>

          </xs:complexType>

        </xs:element>

      </xs:sequence>

    </xs:complexType>

  </xs:element>

...
```

Setting the namespace attribute *fieldWidth* reduces the nested columns within the ACTIONS column to size 1 and allows the image buttons to appear next to each other.

Initialisation

Although phantoms do not require initialisation in order to be set-out, ACTIONS, the surrounding non-phantom does.
This can either be done by initialising ACTIONS for each row of the list, or with a nifty query based trick.

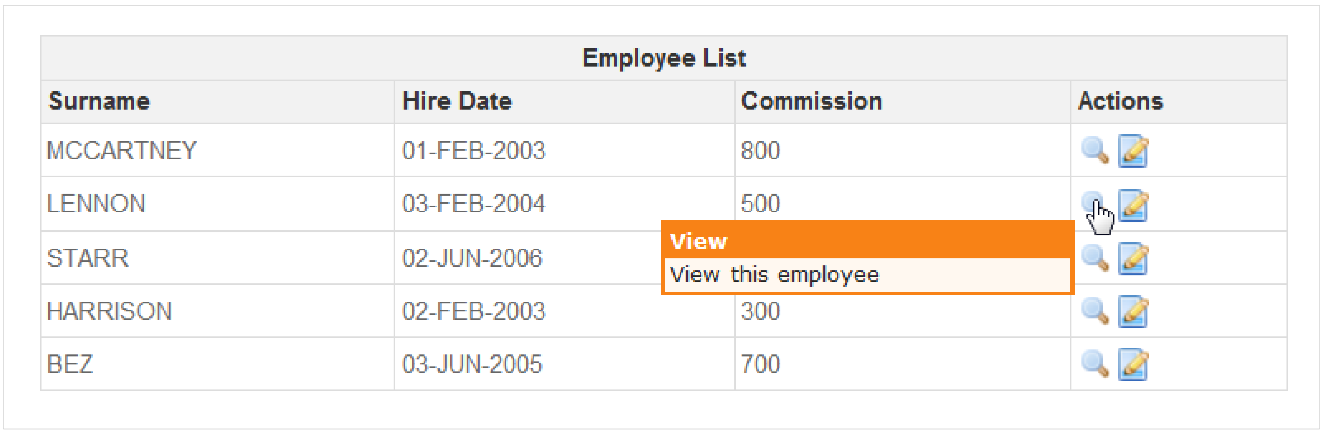
Imagine this is our previous query that populates our EMPLOYEE rows with information from the database:

```
SELECT
  xex.employee_id
, xex.surname
, xex.hire_date
, xex.commission
FROM trainingmgr.xview_employee_xml xex
```

We can add a 'dummy' column to the query, identified as "ACTIONS", which stubs the ACTIONS schema element with a NULL value allowing the nested phantoms to be set-out with no extra work:

```
SELECT
  xex.employee_id
, xex.surname
, xex.hire_date
, xex.commission
, NULL actions
FROM trainingmgr.xview_employee_xml xex
```

Here is how the list may now appear:



Employee List			
Surname	Hire Date	Commission	Actions
MCCARTNEY	01-FEB-2003	800	🔍 📝
LENNON	03-FEB-2004	500	🔍 📝
STARR	02-JUN-2006		🔍 📝
HARRISON	02-FEB-2003	300	🔍 📝
BEZ	03-JUN-2005	700	🔍 📝

Exercises

1. Convert the phantoms in your XX_PLAN_APP_SEARCH module to phantom image buttons in one column.

Use *img/silk_magnifier* for the view action, and *img/silk_page_edit* for the edit action

Training:Fox:PlanApp Libraries

Introduction

The *fm:library-list* section is used to include FOX libraries/modules.

Concepts

FOX processes the *fm:library-list* to incorporate any other FOX modules specified into the current module. The result can be seen in the "ModMerger" DOM available from the developers menu.

The Library In Process

FOX takes the current module and looks at *fm:library-list*. All sections from each *fm:library* are included, so long as they are not already defined in the current module. If definitions [e.g. actions, set-buffers etc] are already defined [i.e. same name] in the current module, they are not libraried in, unless they are marked up with *stub-overload="."*, in this case they are called stubs, (not the same as previously mentioned stub data) the content is over-written by the libraried in version. Entry themes are not libraried in. Any namespace FOX finds in the definitions is inspected to see if it already exists in the current module. If it exists, FOX renames the namespace for the libraried in module to make it unique. There are two exceptions to this rule: *fox* is a namespace and is always global. other namespaces can by defined as global, and these are not renamed: *xmlns:orders="*http://www.og.dti.gov/fox_global*"*. At the end of the process FOX looks to the next *fm:library* for more modules to include. The version of the processed module becomes the ‘current module’ and the algorithm is recursive.

Generic Layout Modules

LAYOUT1 is the most common module libraried in. It contains an extensive set of buffers and actions to provide a generic layout for most FOX applications.

Using actions from other modules

As mentioned above, actions from other modules will be merged into your module if you library the module in. This allows you to call actions from other modules using the *fm:call* command.

fm:call

```
<fm:call action="action name"/>
```

Exercises

Use your XX_PLAN_APP_EDIT (where XX are your initials) module from the previous training sheets for these exercises.
NB: If you were unable to complete any of the exercises from the previous training sheet, take a copy of TRAINING_PLAN_APP_EDIT_V2, and save it as XX_PLAN_APP_EDIT (where XX are your initials). Make sure to change the *fm:name* element too.

If you look at the *fm:library-list* in your module, you'll see that it libraries in 2 modules; LAYOUT1, which provides the generic FOX layout template, and PLANNING_DEC046X_WRAP, which is a wrapper for a module called DEC046X, which is used to pick addresses.
The wrapper limits the functionality of DEC046X to only select a small number of cached addresses, rather than performing a full address search (which costs money!).

Exercise 1

You'll notice that in the schema there are 2 ADDRESS complex elements. Mark up both of these with the edit and view namespaces (edit should be editable and view read only).
Mark up the FULL_ADDRESS and POSTCODE child elements in the same way.
Add a *h2*, underneath the CORRESPONDENCE set-out, with the text "Site Address".
Set-out the ADDRESS element underneath SITE_DETAILS under this header.
Do this in both the edit and view states.

Exercise 2

Add a phantom as a child element to both ADDRESS elements.
The phantoms should be a button with the prompt "Select Address" and they should only be set out on the edit namespace.
Both phantoms should call *action-choose-address*.

Exercise 3

Add the following code to the edit entry theme:

```
<fm:assign initTarget=":{theme}/DEC046X/PARAMS/CALLBACK_ACTION_SUCCESS" textValue="action-search-address-callback-success"/>

<fm:assign initTarget=":{theme}/DEC046X/PARAMS/CALLBACK_ACTION_CANCEL" textValue="action-search-address-callback-cancel"/>

<fm:assign initTarget=":{theme}/DEC046X/PARAMS/CALLBACK_ACTION_FAILURE" textValue="action-search-address-callback-failure"/>
```

Add the following code to the module level action-list. (Don't worry, the next couple of sheets will talk through the new syntax)

```
<fm:action name="action-choose-address">
  <fm:do>
    <fm:context-set scope="state" name="address" xpath=":{action}"/>
    <fm:call action="action-DEC046X-search-div"/>
    <fm:init target=":{theme}/SHOW_POPUP"/>
    <fm:call action="action-OVERLAY_POPUP_LIBRARY-set-no-page-scroll"/>
    <fm:call action="action-OVERLAY_POPUP_LIBRARY-set-popup-size-small"/>
  </fm:do>
</fm:action>
<fm:action name="action-search-address-callback-success">
  <fm:do>
    <fm:remove match=":{theme}/SHOW_POPUP"/>
    <fm:assign initTarget=":{address}/FULL_ADDRESS" expr=":{theme}/DEC046X/RESULT/ADDRESS"/>
    <fm:assign initTarget=":{address}/POSTCODE" expr=":{theme}/DEC046X/RESULT/POST_CODE"/>
    <fm:assign initTarget=":{address}/ABS_REC_NUM" expr=":{theme}/DEC046X/RESULT/ABS_REC_NUM"/>
    <fm:assign initTarget=":{address}/GRID_EASTING" expr=":{theme}/DEC046X/RESULT/GRID_EASTING"/>
    <fm:assign initTarget=":{address}/GRID_NORTHING" expr=":{theme}/DEC046X/RESULT/GRID_NORTHING"/>
  </fm:do>
</fm:action>
<fm:action name="action-search-address-callback-cancel">
  <fm:do>
    <fm:remove match=":{theme}/SHOW_POPUP"/>
  </fm:do>
</fm:action>
<fm:action name="action-search-address-callback-failure">
  <fm:do>
    <fm:alert message="string(:{theme}/DEC046X/RESULT/STATUS_MESSAGE)"/>
    <fm:remove match=":{theme}/SHOW_POPUP"/>
  </fm:do>
</fm:action>
```

Exercise 4

Add the following code beneath the menu-out in the edit state. (Again, the new syntax will be discussed shortly)

```
<fm:if test=":{theme}/SHOW_POPUP">
  <fm:then>
    <fm:include name="buffer-DEC046X-search-postcode"/>
  </fm:then>
</fm:if>
```

You should now have 2 buttons set out on screen that say "Select Address". If you click on one, a popup window should appear listing a series of addresses. If you click on an address, then its value will be copied back into the address and postcode elements on your page.

Training:Fox:PlanApp Contexts

Contexts

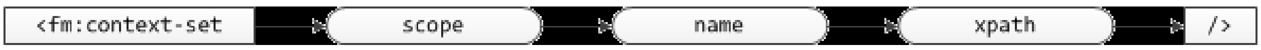
Contexts were mentioned on the An Introduction to DOMs training sheet.
It is possible to define your own contexts that you can use in your FOX module.
This is done using the *fm:context-set* command.

fm:context-set

Description

Sets a context that can be referenced within future xpaths.

Syntax



Attribute Summary

Attribute	Data Type	Description	Required
scope	String Literal	• localised - Used only when inside fm:context-localise and fm:for-each blocks. • state - Sets the context for the current state only, used everywhere else.	Yes
name	String Literal	The name to use to reference the context, eg if you use the name "test" you can refer to it with ":{test}". Can also be used to override the default contexts such as :{action}.	Yes
xpath	Single Node Complex XPath	The node for the context to reference.	Yes

Example

```
<fm:context-set scope="state" name="address" xpath="/*/ADDRESS"/>
```

fm:context-clear

Used to clear a context so that it can no longer be used (see context-clear).

action-choose-address

In our *action-choose-address* action we added to XX_PLAN_APP_EDIT, we use a custom context to extend the lifespan of the :{action} DOM.

```
<fm:action name="action-choose-address">
  <fm:do>
    <fm:context-set scope="state" name="address" xpath=":{action}"/>
    ...
    ...
  </fm:do>
</fm:action>
```

By assigning the action DOM to the address context, we can use it to assign the address text to the correct ADDRESS element.

```
<fm:action name="action-search-address-callback-success">
  <fm:do>
    ...
    <fm:assign initTarget=":{address}/FULL_ADDRESS" expr=":{theme}/DEC046X/RESULT/ADDRESS"/>
    <fm:assign initTarget=":{address}/POSTCODE" expr=":{theme}/DEC046X/RESULT/POST_CODE"/>
    ...
  </fm:do>
</fm:action>
```

Without using a custom context, the action DOM would have expired, and we would not be able to use one generic action to assign the address to the correct ADDRESS element.

Training:Fox:PlanApp fm:if

Introduction

Traditional 3GL branching logic. Try to keep usage to a minimum.

Syntax

```
  <fm:if test="xpath">
    <fm:then>
      [command-list]
    </fm:then>
    <fm:else-if test="xpath">
      [command-list]
    </fm:else-if>
    <fm:else-if test="xpath">
      [command-list]
    </fm:else-if>
    <fm:else>
      [command-list]
    </fm:else>
  </fm:if>
```

Concepts

- fm:else-if and fm:else are optional
- fm:else-if is a repeating item
- test is an XPath expression which is implicitly cast to a boolean using standard xpath cast rules e.g.
  - test="/*//fox-error" returns node list
  - 0 nodes = false
  - 1+ nodes = true
- Can be used in actions and presentation

Examples

Presentation logic

Use in presentation section at state level.

```
<fm:if test="/*/ORDER/ORDER_ITEM_LIST/ORDER_ITEM">
  <fm:then>
    <strong>There is at least one outstanding order item to be processed.</strong>
  </fm:then>
  <fm:else>
    There are no outstanding orders.
  </fm:else>
</fm:if>
```

Action logic

Use in action section.

```
<fm:if test=":{action}/QTY &gt; 5">
  <fm:then>
    <fm:alert message="string(concat('Item quantity is: ', :{action}/QTY))"/>
  </fm:then>
  <fm:else>
    <fm:alert message="Quantity is less than 5"/>
  </fm:else>
</fm:if>
```

Exercises

1. On your XX_PLAN_APP_SEARCH module, use an **<fm:if>** to set-out the results list only if there is at least one APPLICATION in your APPLICATION_LIST.
2. Add a count of the results returned under the table that appears after any search, even if there are no results.

Training:Fox:PlanApp What's going on

entry-theme

```
<fm:entry-theme name="edit">

  <fm:storage-location>sl-main</fm:storage-location>

  <fm:state>state-edit</fm:state>

  <fm:attach>/*</fm:attach>

  <fm:do>

    <fm:run-query interface="dbint-get-data" query="qry-get-app" match=":{root}"/>

    <fm:assign initTarget=":{theme}/DEC046X/PARAMS/CALLBACK_ACTION_SUCCESS" textValue="action-search-address-callback-success"/>

    <fm:assign initTarget=":{theme}/DEC046X/PARAMS/CALLBACK_ACTION_CANCEL" textValue="action-search-address-callback-cancel"/>

    <fm:assign initTarget=":{theme}/DEC046X/PARAMS/CALLBACK_ACTION_FAILURE" textValue="action-search-address-callback-failure"/>

  </fm:do>

</fm:entry-theme>
```

We use *fm:assign* (see Assigning data) to assign various elements needed by DEC046X; these are actually the actions that will provide the address picker functionality.

action-choose-address

```
<fm:action name="action-choose-address">
  <fm:do>
    <fm:context-set scope="state" name="address" xpath=":{action}"/>
    <fm:call action="action-DEC046X-search-div"/>
    <fm:init target=":{theme}/SHOW_POPUP"/>
    <fm:call action="action-OVERLAY_POPUP_LIBRARY-set-no-page-scroll"/>
    <fm:call action="action-OVERLAY_POPUP_LIBRARY-set-popup-size-small"/>
  </fm:do>
</fm:action>
```

1. As explained in the Contexts sheet we start off by defining a context called address that matches the action DOM.
2. We then call an action defined in DEC046X, which we have access to because we've recursively merged DEC046X in through PLANNING_DEC046X_WRAP, which performs our address search.
3. Finally, we init an element in the theme DOM and call a couple of actions which are used to configure the popup that appears with our search results; these are libraried in through PLANNING_DEC046X_WRAP as well.

Presentation

```
<fm:if test=":{theme}/SHOW_POPUP">
  <fm:then>
    <fm:include name="buffer-DEC046X-search-postcode"/>
  </fm:then>
</fm:if>
```

This *fm:if* (see fm:if) tests to see if the SHOW_POPUP element exists in the theme DOM; and if it does, includes a buffer from DEC046X, which displays a popup with the addresses in.

action-search-address-callback-success

```
<fm:action name="action-search-address-callback-success">
  <fm:do>
    <fm:remove match=":{theme}/SHOW_POPUP"/>
    <fm:assign initTarget=":{address}/FULL_ADDRESS" expr=":{theme}/DEC046X/RESULT/ADDRESS"/>
    <fm:assign initTarget=":{address}/POSTCODE" expr=":{theme}/DEC046X/RESULT/POST_CODE"/>
    <fm:assign initTarget=":{address}/ABS_REC_NUM" expr=":{theme}/DEC046X/RESULT/ABS_REC_NUM"/>
    <fm:assign initTarget=":{address}/GRID_EASTING" expr=":{theme}/DEC046X/RESULT/GRID_EASTING"/>
    <fm:assign initTarget=":{address}/GRID_NORTHING" expr=":{theme}/DEC046X/RESULT/GRID_NORTHING"/>
  </fm:do>
</fm:action>
```

This is the action that will be called when we select an address from the popup window.
Firstly, we remove the SHOW_POPUP flag, so that *buffer-DEC046X-search-postcode* is no longer included in our edit state; hiding the address popup.
Then we assign the results of selecting our address to the various child elements of the ADDRESS element. We use our address context to match onto the correct ADDRESS element.

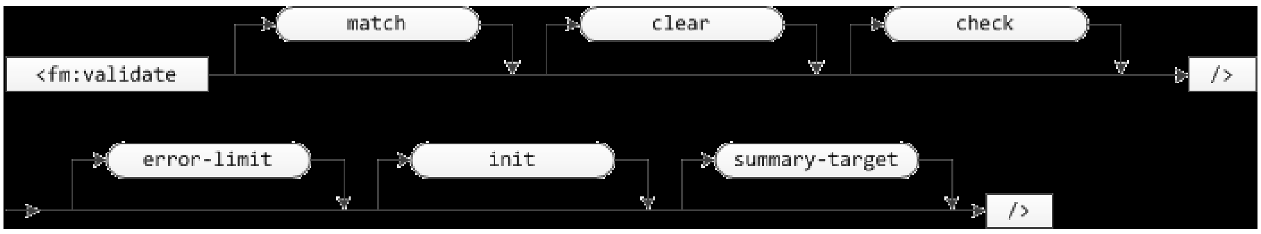
Training:Fox:PlanApp fm:validate

Introduction

Simply adds "fox-error" elements inside elements, usually alongside the text node, in the current DOM where validation fails. The presentation widget showing the item element detects "fox-error" and displays a red "X", next to the element that has caused the error. Also adds summary information to the Error DOM. Validates the matched nodes against the schema definition.

The command is 4GL, one fm:validate (with complex XPath) does a lot of iterative checking.

Syntax



Attribute Summary

Attribute	Data Type	Description	Required
match	Complex XPath	The list of nodes to validate. **Default:** "."	No
clear	String Literal	• `NONE` - Do not clear any errors. • `CLEAR-NODE` - Clear fox-error nodes. • `CLEAR-SUMMARY` - Clear the :{error} DOM. • `BOTH` - Clears both fox-error nodes and the :{error} DOM. **Default:** "`NONE`"	No
check	String Literal	• `NONE` - Do not validate at all, used to just clear errors. • `CONTENT` - Check the content of the nodes against the schema rules. • `CARDINALITY` - Check the cardinality of the nodes against the schema rules. • `ALL` - Check both the cardinality and content of the nodes against the schema rules. **Default:** "`ALL`"	No
error-limit	Integer	Validation ends after the error-limit is reached. **Default:** `100`	No
init	String Literal	• `Y` - Inits a new blank node to put a fox-error in if the node should exist but doesn't. Doesn't use any initialisation rules specified in the schema - always just creates a blank node to put the `fox-error` into. • `N` - Ignores missing nodes. **Default:** "`Y`"	No
summary-target	Simple XPath	Where to place the summary of errors. This is in the context of the current attach point. The default is special in that it points to the :{error} DOM, and is the only way to do so. **Default:** "`:{error}/error-list`"	No

Concepts

Validation levels

1. In field - XMLSchema simple type facets, e.g. xs:maxLength, xs:maxInclusive.
2. Across field - use fox:validate-xpath and fox:validate-xpath-msg on each item.
3. Business logic - Application code to manually fm:assign fox-error/msg elements.

Clear

The clear attribute specifies if previous "fox-error" elements are tidied, during validate pass.

Default "NONE" – gets rid of nothing and can only add "fox-error" elements - repeating fm:validate causes duplicates. "CLEAR-NODE" - clears "fox-error" in item before adding new "fox-error". "CLEAR-SUMMARY" - clears "fox-error" from Error DOM. "BOTH" - clears "fox-error" in item and in summary DOM.<fm:validate clear> should be before manually adding fox-error or subsequent fm:validate.

NB: Clearing is done before any checking is done.

Check

The check attribute specifies what XMLSchema facets to check: "CONTENT" - The value of content text nodes is examined. e.g. xs:maxLength. "CARDINALITY" - XMLSchema minOccurs and maxOccurs is examined. Default "ALL" - both context and cardinality is examined. NONE - no fox-errors are added - useful with clear="BOTH" to simply remove errors.

Error-limit

Setting the error-limit attribute is a safeguard. A very large number of errors would slow down performance. If set, validation stops when this number of errors has been found.

Summary-target

Specify an alternative error summary DOM location.

Supported XSD markup

FOX validation supports a limited subset of XSD markup:

XSD Datatypes

- xs:string
- xs:decimal
- xs:date
- xs:long
- xs:int
- xs:dateTime
- xs:time
- xs:boolean
- xs:positiveInteger
- xs:negativeInteger
- xs:integer
- xs:anyType

XSD Facets (where appropriate)

- xs:totalDigits
- xs:fractionDigits
- xs:length
- xs:maxLength
- xs:minLength
- xs:minInclusive
- xs:maxInclusive
- xs:minExclusive
- xs:maxExclusive

fox:mand

```
fox:mand="XPath Boolean"
```

Specifies that an element's text contents is mandatory. If the element itself does not exist in the DOM, the mand attribute is not enforced. A fox-error element will be created if the element is validated and has no text node.

There is an inconsistency with how mand's XPath is evaluated. When the XPath is checked for validation, the evaluate context rule is followed. When the XPath is checked for whether to add a '*' to the prompt, the XPath is run from the simple node. This can cause problems with either the presentation(problem) or validation(big problem). To avoid this, you must use 'ancestor-or-self::<<parent-complex-node>>/XPath_expression' for any mand with a condition other than '.'

fox:validate-xpath

```
fox:validate-xpath="XPath Boolean"
```

Used to enforce high level validation rules on the associated schema element. If the XPath returns 'true' then the element is considered valid. 'false' is invalid. A custom message can be displayed with fox:validate-xpath-msg.

NB: a validate-xpath is only run if the associated element has text content. If the element does not exist or is empty, validate-xpath is ignored.

Examples

This clears all errors:

```
<fm:validate match="/*//*" clear="BOTH" check="NONE"/>
```

The following checks all errors in the data DOM but stops when 100 errors have been found:

```
<fm:validate match="/*//*" check="ALL" error-limit="100"/>
```

The following is standard practice for validating a whole schema:

```
<fm:validate match="/*//*" check="ALL" clear="BOTH"/>
```

We clear all the fox-error elements before checking everything to avoid creating duplicate fox-errors on schema elements.

NB: You cannot use fm:validate and match on a root element (such as //* or /* or :{root}). This will throw a null error.

Exercises

Use your XX_PLAN_APP_EDIT (where XX are your initials) module from the previous training sheets for these exercises.

Exercise 1

Mark up the APPLICATION_DETAIL schema with various validation attributes (fox:mand, fox:validate-xpath, XSD markup).
Add an action named "action-show-errors" with the prompt "Show Errors". This should validate everything. Set it out next to "Save Changes".

Exercise 2

Add an action named "action-clear-errors" with the prompt "Remove Errors". This should validate, removing all errors. Set it out next to "action-show-errors".

Exercise 3

Call (fm:call) "action-show-errors" from "action-save-changes".
Wrap the *fm:run-api* and *fm:transaction* elements in an *fm:if*, which checks there are no fox-error elements across the whole DOM.
If there are fox-error elements, then show a message saying "The application has errors, please correct these before saving.".

Training:Fox:PlanApp Storage Locations

Introduction

Storage Locations provide FOX with a standard mechanism for accessing/updating units of data. They are a definition of how and where data is stored and accessed. They differ from fm:query definitions as Fox decides where, when, and how to use the SQL/DML defined within the Storage Location.

Syntax

The following code shows what might be included within a storage location:

```
<fm:storage-location name="name">
  <fm:cache-key string="module name :bind">
    <fm:using>bind xpath</fm:using>
  </fm:cache-key>
  <fm:new-document>
    <fm:root-element>root</fm:root-element>
  </fm:new-document>
  <fm:database>
    <fm:query>
      <fm:sql>
SQL statement with :bind
      </fm:sql>
      <fm:using>bind xpath</fm:using>
    </fm:query>
    <fm:insert>
      <fm:sql>
DML statement with :bind
      </fm:sql>
      <fm:using>bind xpath</fm:using>
    </fm:insert>
    <fm:update>
      <fm:sql>
DML statement with :bind
      </fm:sql>
    <fm:using>bind xpath</fm:using>
    </fm:update>
  </fm:database>
</fm:storage-location>
```

Concepts

A global caching mechanism is used across the FOX System. Data is loaded into the memory cache bucket, which is identified using a cache key (I.E. the name of the bucket). All operations are performed on the data held in the memory cache as this provides best performance.
Because the caching mechanism is global across FOX, two different user sessions, running different modules can access a common unit of data - simply by addressing the same bucket (I.E. using the same cache key). This means both users can benefit from seeing the common data consistently and quickly, whilst system memory is optimised as only one version of the data is held.

You can have multiple storage location definitions within a module, but each one should have a unique cache key and name.

Across modules, the cache key must be unique for discrete data. Common data can however be reused by engineering a repeatable cache key.

- Storage Location is a definition of how and where something is stored
  - Defines all database SQL and DML to access a single row of information
  - Complex internal caching aids performance and data sharing
- When a module is loaded, a cache key is generated. This is used as a name of a memory location to store retrieved information for that module
- It is important to get this correct, as two modules with the same storage location cache key will use the same memory location (bucket). If different data structures are involved, this is very bad as the two data structures may be corrupted
- A cache key should include some relevant text as well as either a unique bind variable or a bind variable that has been passed into the module
- The cache key for most modules will be the module name, and some bind variable
- Storage locations are used for accessing:
  - Short term computed lists of values (map-sets, covered later)
  - Static lists of values (map-sets)
  - Module Data DOM – XML stored in database
  - Files stored on the database (e.g. Uploaded Word files)

Examples

An example of a storage location for a module called MOD012X that had no variables passed into it and would use a unique bind variable is:

```
<fm:storage-location name="main">
  <fm:cache-key string="MOD012X :1"> '''(This ':1' is replaced by a unique value populated by FOX)'''
    <fm:using using-type="UNIQUE"/>
  </fm:cache-key>
  <fm:new-document>
    <fm:root-element>ROOT</fm:root-element>
  </fm:new-document>
</fm:storage-location>
```

For the same module, but with a variable called ID passed into the module, you would use the following code to declare the storage location:

```
<fm:storage-location name="main">
  <fm:cache-key string="MOD012X :1">
    <fm:using>:{params}/ID</fm:using>
  </fm:cache-key>
  <fm:new-document>
    <fm:root-element>ROOT</fm:root-element>
  </fm:new-document>
</fm:storage-location>
```

Warning

The storage location cache key is evaluated **before** the entry-theme's **<fm:do/>** block. This means that you should only base a storage location cache key on DOM values that are available prior to entry-theme processing. In practise, this limits you to using :{params}, :{env}, :{session} and :{user} - as no other DOMs have useful data for cache-keys at that particular point of code execution.

Sometimes people make the mistake of augmenting data in the entry-theme and using the resultant value in the cache key. Here is a simplified example.

```
<fm:storage-location name="main">
  <fm:cache-key string="MOD012X :1">
    <fm:using>:{theme}/DETAIL_ID</fm:using>
  </fm:cache-key>
  <fm:new-document>
    <fm:root-element>ROOT</fm:root-element>
  </fm:new-document>
</fm:storage-location>
```

1. When the module is called, the cache-key evaluates *immediately* and resolves to "MOD012X " (the XPath substituted as :1 has returned no data). Note that if multiple callers attempt to access the module at the same time, there will be locking contention, as they are all sharing the exact same cache key.
2. The entry-theme **<fm:do/>** block executes, and populates :{theme}/DETAIL_ID. This results in a different id for each entry.
3. On the next FOX action click, each user's cache key will evaluate to a unique string, for example "MOD012X 13" and "MOD012X 14", as the :{theme}/DETAIL_ID element is populated.

While this seems innocuous enough, the locking contention can cause problems, especially if the entry-theme processing takes a long time to complete. Please be aware of the general principle that cache keys should be based on values available at the point of entry to a module.

Training:Fox:PlanApp Updating data with a storage location

Syntax

Within the storage location element, you can add an fm:database node, to tell FOX that the data from the Data DOM has a direct relationship with a database table. The Data DOM would be stored as a Clob or XML column on this table.

The fm:database syntax has 3 child elements:

- fm:query in which you can write SELECT statements
- fm:update in which you can write UPDATE statements
- fm:insert in which you can write INSERT statements (This is rarely used, most modules will create new records separately)

Aside from having the SQL code inside an fm:sql tag, each of the 3 above elements are syntactically the same as an fm:query in a db-interface.

Here is an example of the syntax:

```
<fm:storage-location name="main">
  <fm:cache-key string="MODULE_NAME:1">
    <fm:using>:{params}/P_PF_ID</fm:using>
  </fm:cache-key>
  <fm:new-document>
    <fm:root-element>ROOT</fm:root-element>
  </fm:new-document>
  <fm:database>
    <fm:query>
      <fm:sql>
SELECT xml_data FROM portal_folders WHERE id = :1
FOR UPDATE OF xml_data NOWAIT
      </fm:sql>
      <fm:using>:{params}/P_PF_ID</fm:using>
    </fm:query>
    <fm:update>
      <fm:sql>
UPDATE portal_folders SET id = id WHERE id = :1
      </fm:sql>
      <fm:using>:{params}/P_PF_ID</fm:using>
    </fm:update>
  </fm:database>
</fm:storage-location>
```

NB: Using the *<fm:using using-type="DATA-XMLTYPE"/>* bind will cause FOX to write the root DOM's XML to the storage location twice, as it is already streamed straight into the LOB locator by default. Therefore it is recommended you do not use this bind, and only use the update statement to set other columns, such as last update times, etc.

If your storage location XML has XViews on it, an update statement still needs to be executed in order to

cause the XView triggers to fire. A self-update (i.e. *SET id = id*) will be sufficient for this.

If you encounter problems with your storage location, that could include errors like this:

```
Storage Location: Update/Query pair do not access the same row/column
(change number inconsistant)
```

then you may have come across a bug in FOX that exists in version *r4.05.02*. In order for your code to work, you will have to change the code in the update statement.

Here is an example of the modified syntax:

```
<fm:update>
  <fm:sql>
UPDATE trainingmgr.xx_plan_app_details
SET xml_data = :1
WHERE application_id = :2
AND status_control = 'C'
  </fm:sql>
  <fm:using using-type="DATA-XMLTYPE"/>
  <fm:using>:{params}/APPLICATION_ID</fm:using>
</fm:update>
```

Concepts

FOX will automatically read and write data back to the database from the Data DOM on each transaction cycle, through the storage location, dependant on what clauses you specify. For instance:

When the module is first entered, data is queried into memory from the database using the fm:query 'SELECT' statement. A select statement is mandatory and should always exist.
If no rows are returned from the SELECT statement, then FOX looks for an fm:insert statement and tries to execute this, to insert stub data into the table.
If the SELECT statement in the fm:query block successfully finds data, and the data changes during a Transaction Process, FOX looks for an fm:update statement to UPDATE the database table. When the request has been processed, the Data DOM and the row in the database table will be the same.

If there are no fm:database statements, then FOX will create a blank XML document with a root element of whatever is specified in the fm:new-document/fm:root-element text.

Storage locations can only operate on a single record; usually derived from a parameter passed to the module. To operate on a record that is part of a list, you can use phantoms.

Exercises

Use your XX_PLAN_APP_EDIT (where XX are your initials) module from the previous training sheets for these exercises.
NB: If you were unable to complete any of the exercises from the previous training sheet, take a copy of TRAINING_PLAN_APP_EDIT_V3, and save it as XX_PLAN_APP_EDIT (where XX are your initials). Make sure to change the *fm:name* element too

Exercise 1

Remove the call to *api-save-app* and the commit from *action-save-changes*. *action-save-changes* should now do nothing, but should still be set out. This is standard practice, as having a "Save" action on screen gives users confidence that their changes have been applied.
Remove the call to *qry-get-app* from the edit and view entry themes.
If you re-enter the module, you'll notice that no fields are displayed, as the data is no longer retrieved from the database.

Exercise 2

Add a SELECT statement to *sl-main*.
This needs to select the xml_data of the latest detail, from trainingmgr.xx_plan_app_details (where xx are your initials), for the selected application, and lock the row for update.
Look at *qry-get-app* if you need help.
If you re-enter the module now, you'll see that all the fields you had previously are displayed again, as the storage location SELECT statement is querying the xml_data into the root DOM.

Exercise 3

Add an update statement to *sl-main* to update the latest detail for the selected application.
As mentioned above, you don't need to set the xml_data in an update statement, so just set id = id. We need to do this so that the XVIEW triggers will fire.
Now, if you make changes to any of the fields, your changes will be streamed back into the database; avoiding the need to run apis.

Training:Fox:PlanApp State transitions

Introduction

This sheet will teach you how to move between states in a FOX screen; which can be used to break up data entry forms.

Concepts

FOX uses a stack structure to store previous modules and states that the user has been to within the current session. This stack can be viewed by clicking the CallStack link in the dev bar.

Each time a user enters a new module or state, it is pushed onto the top of the stack, and when they leave, it can be popped from the stack, or simply replaced.

Thus states can be replaced, which is the normal usage, or pushed onto the stack and then popped off later.

When you push a state, the previous attach point will be stored so you can pop back to the previous location later. You can also specify a new attach point when you push or replace a state. This can be done using the attach attribute on the fm:state command.

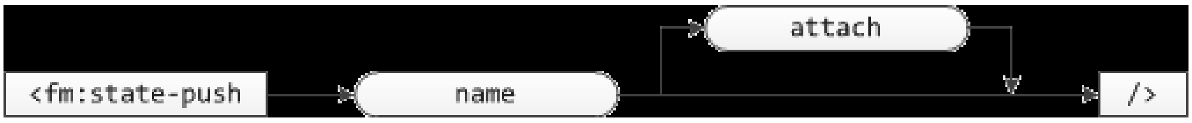
We can use phantoms to call an action within a list that will push a new state and display information relevant to that branch of the list.

fm:state-push

Description

Pushes the new state onto the top of the callstack.

Syntax



Attribute Summary

Attribute	Data Type	Description	Required
name	String Literal	The name of the state to use.	Yes
attach	Single Node Complex XPath	The attach point for the new state.	No

Examples

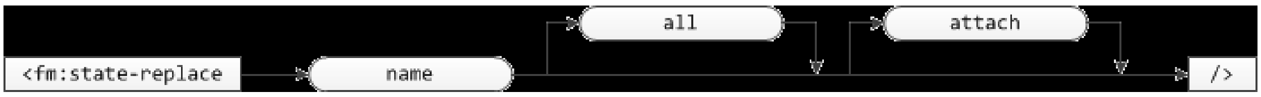
```
<fm:state-push name="string" [attach="XPath"]/>
```

fm:state-replace

Description

Replaces the current state on the callstack with the new state.

Syntax



Attribute Summary

Attribute	Data Type	Description	Required
name	String Literal	The name of the state to use.	Yes
all	String Literal	• `true` • `false` **Default:** `false` **Note:** When true it replaces all states from the callstack with just the new state.	No
attach	Single Node Complex XPath	The attach point for the new state.	No

Examples

```
<fm:state-replace name="string" [all="string" attach="XPath"]/>
```

fm:state-pop

Description

Pops the current state from the callstack, with an implicit exit-module if it is the last state for the current module.

Syntax

```
<fm:state-pop/>
```

fm:state-strict-pop

Description

Pops the current state from the callstack, and does not allow an implicit exit-module, instead it will throw an error when there it is the last state of the current module.

Syntax

```
<fm:state-strict-pop/>
```

Exercises

Use your XX_PLAN_APP_EDIT (where XX are your initials) module from the previous training sheets for these exercises.
NB: If you were unable to complete any of the exercises from the previous training sheet, take a copy of TRAINING_PLAN_APP_EDIT_V4, and save it as XX_PLAN_APP_EDIT (where XX are your initials). Make sure to change the *fm:name* element too.

Exercise 1

Rename *state-edit* to *state-edit-correspondence*. Change the edit entry theme accordingly.
Add two new states called *state-edit-site* and *state-upload-documents* respectively.
Define a *buffer-content* in your 2 new states.
Move the presentation code for the SITE_DETAILS and WORK_PLAN from *state-edit-correspondence* to *state-edit-site*.

Exercise 2

Add an action to *state-edit-correspondence* which pushes *state-edit-site* onto the stack. Use a *fm:menu-out* to set this action out beneath the other actions on the page (you may need to use another namespace).
Add two actions to *state-edit-site*, one which pushes *state-upload-documents* onto the stack, and one which pops off the top of the stack. Use a *fm:menu-out* to set these action out beneath the other actions on the page (you may need to use another namespace).
Add an action to *state-upload-documents* which pops off the top of the stack. Use a *fm:menu-out* to set this action out beneath the other actions on the page (you may need to use another namespace).

Use these actions to move between the various states. have a look at the CallStack (on the dev toolbar) after each state change to see the state of the stack.

Exercise 3

Change the action in *state-edit-site* which pushes the state to replace the state instead.

Move from *state-edit-site* to *state-upload-documents*. Notice that *state-edit-site* does not appear in the CallStack. Pop out of *state-upload-documents*. You should go back to *state-edit-correspondence*.

Change the action back to a state push instead of a replace when you are done.

Exercise 4

Override *action-show-errors* and *action-clear-errors* in states *state-edit-correspondence* and *state-edit-site*. The overridden actions should only validate the part of the schema set out in the particular state.
Override *action-save-changes* in states *state-edit-correspondence* and *state-edit-site*. The overridden action should only check the part of the schema set out in the particular state for fox-error elements.

Training:Fox:PlanApp Introduction to map-sets

Introduction

A map-set, as the name suggests, is a set of mappings. Each map is a Display Key to Data. E.g. "United Kingdom" (display key) to "UK" (data), "Germany" (key) to "D" (data).

Syntax

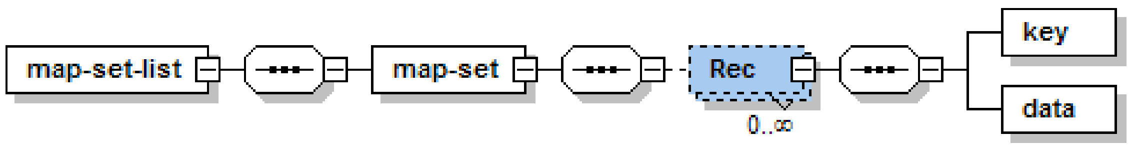
FOX uses XML Schema Enumerations for Drop Downs / Lists Of Values (LOV) on screens. They are satisfactory for simple uses, but static and inflexible for sophisticated applications.

A map-set behaves like an Enumeration LOV, but is highly dynamic in nature. When the example above is associated with a xs:element definition and set-out on screen: User sees a Drop Down LOV with choices: "United Kingdom", "Germany".
However FOX translates user input when stored in XML as follows: "UK" or "D" Here the data value is a simple type (xs:string), but it can also be a complex collection.

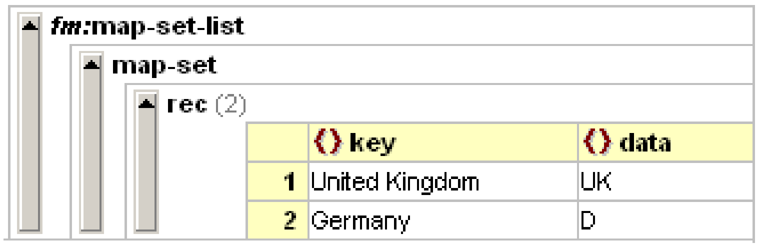
Map-set data is stored in the map-set-list element, and has the following structure:

```
<map-set>
  <fm:storage-location/>
  <fm:do/>
  <fm:refresh-timeout-mins/>
  <fm:refresh-in-background/>
</map-set>
```

For the countries example mentioned above the data would have the following structure:



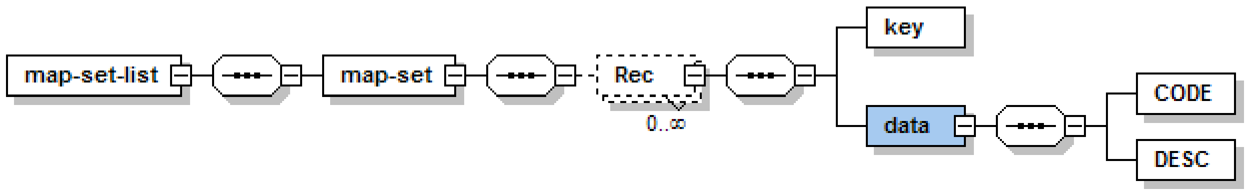
The key in a map set is always an xs:string but the data element can be of any type, including a complex type. The data for the countries example looks like this:



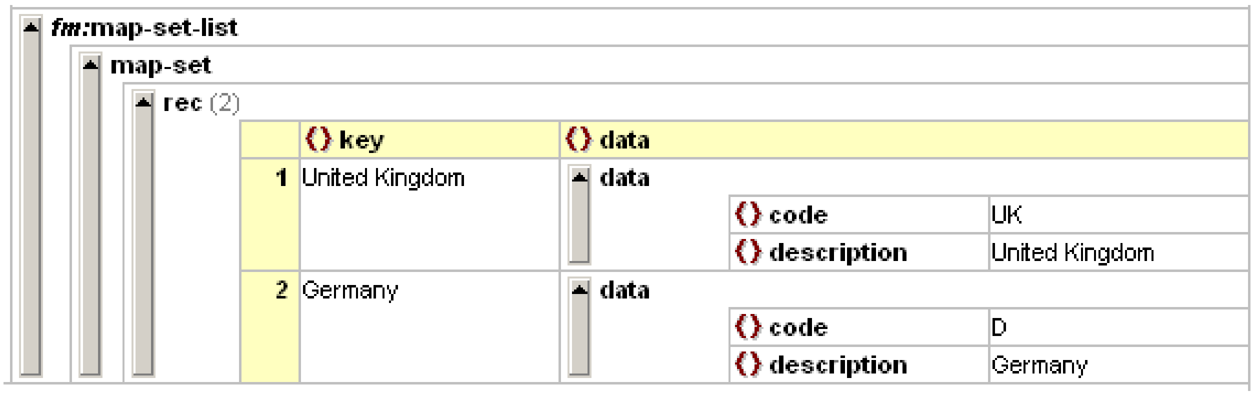
fm:map-set-list
map-set
rec (2)

	key	data
1	United Kingdom	UK
2	Germany	D

An example of where the data is a complex data structure is the following:



And the data:



fm:map-set-list
map-set
rec (2)

	key	data		
1	United Kingdom	data		
			code	UK
			description	United Kingdom
2	Germany	data		
			code	D
			description	Germany

Fox manages the map-set data structures automatically using a set of rules (details later):

- Where the XML data is stored - a Storage Location
- How the XML data is constructed - FOX meta-code
- How long the data is good for - I.E. when it needs refreshing (constructing again)

Map-sets are utilised by display widgets, fm:validate and fm:context-localise (covered later).

Historical Records

`rec` also has an optional boolean child element, `historical`. If `historical` is set to true, the record is only included in the map-set when the element the map-set is applied to already contains the value in the record's `data`. For example, if a user selects UK from a drop-down on an application form, and UK is later marked as historical, the user will continue to see UK on this form but not on any new forms.

Training:Fox:Storage Locations - Memory Only Map-Set

Estimated duration: 30 minutes

Introduction

Storage Locations are a valuable utility in FOX. They are a definition of how and where something is stored.

Syntax

The following code shows what has to be included within a storage location for a memory only map-set:

```
<fm:storage-location name="name">
  <fm:cache-key string="string :bind">
    <fm:using using-type="UNIQUE"/>
  </fm:cache-key>
  <fm:new-document>
    <fm:root-element>map-set-list</fm:root-element>
  </fm:new-document>
</fm:storage-location>
```

Note, the fm:root-element will always be map-set-list when you are creating a storage location for a map-set.

Concepts

Before a map-set can be used we need to define where it is held, a storage location (SL). SLs can be defined in a number of ways, but here we look at "Memory Only" SL. The map-set values created will be cached in memory only. The cache key defines in what bucket. Note the cache key comprises a static name part, and a unique part using bind variable.

Without the unique bind variable, the map-set data will never change, because the storage location will not be changed each time it is loaded.

Standard SLs are XML based. When a SL is used for the first time no data exists. The fm:root-element specifies how to initialise the XML in this first instance. For a map-set it is always map-set-list.

Training:Fox:Constructing a map-set 1

Introduction

This worksheet tells you how to create a map-set without using a database query in the storage location.

Syntax

The main lesson of this worksheet is what you do in the map-set fm:do-block. In this example a query is in fact run in the map-set fm:do-block:

```
<fm:map-set name="ms-mapset-example">
  <fm:storage-location>sl-mapset-example</fm:storage-location>
  <fm:do>
    <fm:run-query interface="dbint-mapsets" query="qry-mapset"/>
  </fm:do>
  <fm:refresh-timeout-mins>0</fm:refresh-timeout-mins>
  <fm:refresh-in-background>true</fm:refresh-in-background>
</fm:map-set>
```

Concepts

A map-set definition always requires a storage location to be defined and referenced first. The storage location will ensure that the XML root element is initialised and no more.

The fm:do block meta-code defines how the remainder of the XML is constructed.

Important: When using do block runs:

- Context :{attach} (attach point) is temporarily moved to the storage location XML root
- The Data DOM is still accessible using :{root}, although the previous attach is not
- Thus any "." based meta-code XPath is operating on the map-set DOM

Any FOX meta-code can be used to construct the map-set DOM:

- A Database Interface Query can be used
- Templates using conditional logic (example 1 below)
- Combinations of the above
- Other, such as fm:for-each, fm:copy, fm:assign (covered later)

The refresh-timeout-mins specifies how long the constructed data is good for:

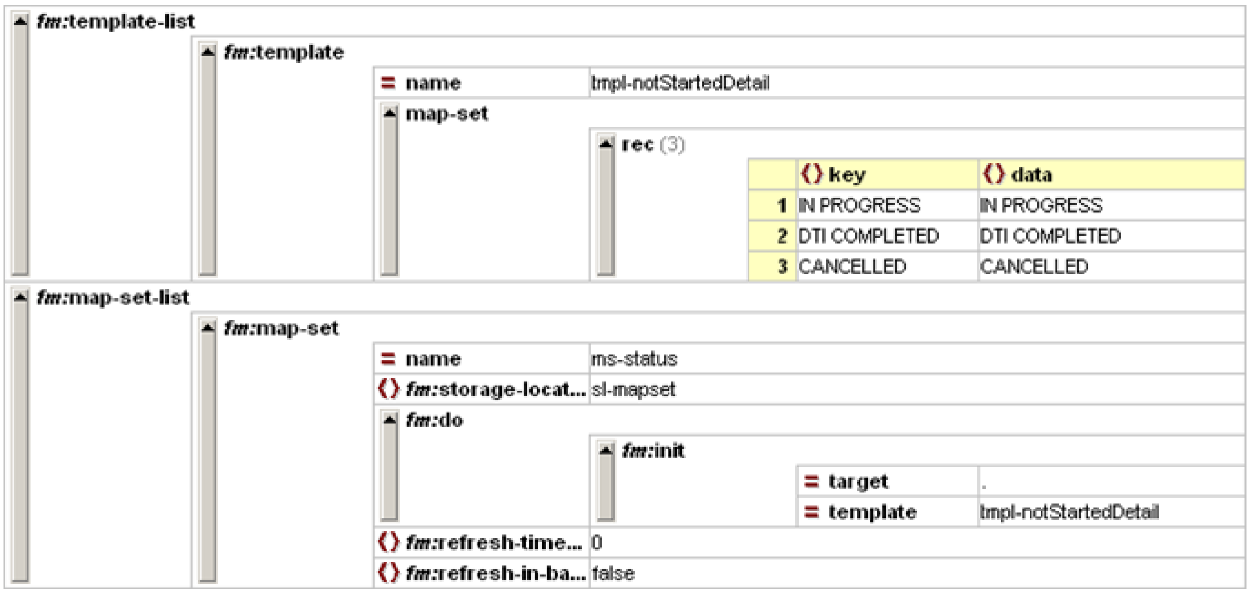
- If underlying data changes frequently, consider 5, 3 or even 0 mins. Otherwise 30 mins+
- Special value of 999999999 mins, actually means never refresh - not even the first time
- Refresh may be earlier than that specified, if XML was flushed from memory to free space.

Important: fm:do is responsible for removing old XML as well as constructing new XML. Consider using fm:run-query mode="PURGE-ALL", or fm:remove at start of fm:do block.

Examples

Using a template

It is possible to use a template to initialise the data in a map set. The following is an edited extract from the module REV002L:



```
fm:template-list
    fm:template
        name                    tmpl-notStartedDetail
        map-set
            rec (3)
```

	key	data
1	IN PROGRESS	IN PROGRESS
2	DTI COMPLETED	DTI COMPLETED
3	CANCELLED	CANCELLED

```
fm:map-set-list
    fm:map-set
        name                    ms-status
        fm:storage-locat...     sl-mapset
        fm:do
            fm:init
                target          .
                template        tmpl-notStartedDetail
        fm:refresh-time...      0
        fm:refresh-in-ba...     false
```

As you can see the map-set ms-status is initialised from the template called tmpl-notStartedDetail when the template is used.

Using a query

You can construct a map-set using a query (as shown in the code example near the top of this page).

The query that you run needs a *fm:target-path* elements matching "map-set/rec", and needs to return 2 columns; aliased as "key" and "data" respectively (lower case). Make sure you use double quotes for the aliases, otherwise when the query is run through Oracle, it will convert the aliases to UPPERCASE and your mapset will return an error.
The "key" column will be the value shown to the user and the "data" column will be the internal value stored in the DOM.

E.G.

```
<fm:query name="qry-mapset">
  <fm:target-path match="map-set/rec"/>
  <fm:select>
SELECT
  INITCAP(name) "key"
, deptno "data"
FROM trainingmgr.department
  </fm:select>
</fm:query>
```

Running this query in the do block of a map-set would init a map-set with 3 elements:

- Programming (mapping to 10)
- Testing (mapping to 20)
- Business (mapping to 30)

This is equivalent to constructing the map-set using the following template:

```
<map-set>
  <rec>
    <key>Programming</key>
    <data>10</data>
  </rec>
  <rec>
    <key>Testing</key>
    <data>20</data>
  </rec>
  <rec>
    <key>Business</key>
    <data>30</data>
  </rec>
</map-set>
```

NB: You should use *mode="PURGE-ALL"* when calling the map-set query, to ensure duplicate rows will not appear in your map-set

Assigning a map-set to an element

To assign a map-set to an element, add the *fox:map-set="name"* attribute to the element (where name is the name of the map-set).

Exercises

For this exercise, take a copy of TRAINING_PLAN_APP_EDIT_CREATE and save it as XX_PLAN_APP_EDIT (where XX are your initials). Make sure to change the *fm:name* element too.
This is the similar to your previous copy of XX_PLAN_APP_EDIT, except that it has a new entry theme *create*, which creates a new planning application.
This entry theme is unusable at the minute, but we're about to fix it.

Exercise 1

Create a new storage location called *sl-app-types-ms*. This will be a map-set storage location, so make sure to define it correctly.
Create a map-set called *ms-app-types*. It should use the storage location you just defined and call a query called *qry-get-app-types*.
Define the *qry-get-app-types* query. This should return:

- *description* column as the key
- *id* column as the data

from trainingmgr.xx_plan_app_types (where xx are your initials).
Assign the *ms-app-types* map-set to the APPLICATION_TYPE element.

The *create* entry theme will now be usable.
Call the module on the *create* entry theme. Use the map-set we just created to select an application type, then click "Create Application".
This will take you to our *edit* screen we developed previously, but you will notice that all the fields are empty; as this is a new application.

Training:Fox:Constructing a map-set 2

Introduction

Map-Sets do not have to be memory only. They can have a map-set structure stored in a database table. This can be queried in using some SQL within the storage location.

Syntax

The following code shows the syntax required to create a map-set from a table in the database:

```
<fm:storage-location name="name">
  <fm:cache-key string="string"/>
  <fm:new-document>
    <fm:root-element>map-set-list</fm:root-element>
  </fm:new-document>
  <fm:database>
    <fm:query>
      <fm:sql>
SQL STATEMENT
      </fm:sql>
    </fm:query>
  </fm:database>
</fm:storage-location>
```

Note that the SQL statement is contained within the fm:database part of the Storage Location.

Concepts

Some Lists of Values are quite lengthy and may be reused in a number of modules. In this case, the map-set XML data is preloaded in a table. This is often a manual task.

To access this data a Storage Location is defined within database access clauses (See Example).

A map-set can then be defined using this Storage Location.

Things to note:

- No constructor (fm:do block) is required within the map-set list as data is already created in the table
- Special value of 999999999 on refresh-timeout-mins used as you never need to refresh
- It is possible to use a constructor block that augments (adds to) XML in the table. This will be covered later

Example

The following code will create a Storage Location that contains a map-set:

```
<fm:storage-location name="chems-sl">
  <fm:cache-key string="chems lov"/>
  <fm:new-document>
    <fm:root-element>map-set-list</fm:root-element>
  </fm:new-document>
  <fm:database>
    <fm:query>
```

```
      <fm:sql>
SELECT data
FROM env_mapsets
WHERE domain = 'CHEMICAL_LABEL_CODES'
      </fm:sql>
    </fm:query>
  </fm:database>
</fm:storage-location>
```

The map-set declaration will look similar to the following code:

```
<fm:map-set name="chems">
  <fm:storage-location>chems-sl</fm:storage-location>
  <fm:do/>
  <fm:refresh-timeout-mins>999999999</fm:refresh-timeout-mins>
  <fm:refresh-in-background>false</fm:refresh-in-background>
</fm:map-set>
```

The fm:do block is empty because all the work has been done in the Storage Location.

NB: if you want your map-set to update instantly, you must use the <fm:do> <fm:run-query/> method, or call <fm:refresh-map-set/>.

Exercises

Use your XX_PLAN_APP_EDIT (where XX are your initials) module from the previous training sheets for this exercise.

Exercise 1

Assign a map-set called "ms-title" to the TITLE element in the schema. The map-set needs to select its data from envmgr.env_mapsets where the DOMAIN is "PERSON_TITLE".

Training:Fox:PlanAppElementActionAndChangeAction

Introduction

This section covers two further namespace attributes that can be placed on elements in a schema structure that prompt an action when they undergo changes.

Concepts

namespace:action

namespace:action points to an action to run and prompts a page-churn whenever the element's field changes:

```
<xs:element name="COMMISSION" employee:edit="." employee:run="." employee:action="action-query-department-commission" ... />
```

Changing this COMMISSION element would trigger an immediate page-churn and perform the action specified. In this case a change updates a display of the employee's department's commission elsewhere on the screen. *'N.B* that a *namespace:run* attribute is required that controls, via XPath, when the action is run.

namespace:change-action

namespace:change-action is similar to *namespace:action* but will not prompt a page-churn. When another user interaction causes a page-churn, FOX will check to see if the element's field has changed, and if so will run the action.

When executing a change-action, the :{item} and :{itemrec} contexts are used to refer to the data item which has changed (:{item}) and its parent according to the evaluate context rule (:{itemrec}).

Change-actions ONLY fire when the user has made a change to the data. Changing data programatically (i.e. using fm:assign) will NOT fire the change-action.

```
  <xs:element name="FORENAME" employee:edit="." employee:change-action="action-set-name-change-flag" ... />
  <xs:element name="SURNAME" employee:edit="." employee:change-action="action-set-name-change-flag" ... />
```

Changing either of the FORENAME or SURNAME elements sets a flag element in a DOM. This could prompt a different logic path on saving changes for example.

N.B.' namespace:change-action*s are run before any other action and do not require a* namespace:run*:*

Exercises

1. Add a GENDER element to your XX_PLAN_APP_EDIT module that uses a template map-set for its possible values.

2. Add an action that will immediately change the GENDER to an appropriate value when a gender-specific TITLE is chosen (e.g. choosing 'Ms' will change the GENDER to female).

3. Use *namespace:change-action*s and flags to ensure that your XX_PLAN_APP_SEARCH module only refreshes an edited application's row (on return from XX_PLAN_APP_EDIT) if changes have been made to the fields displayed in the list.

Training:Fox:PlanAppDynamicMapsets

Introduction

It can be useful to have a map-set that adapts for different data in your DOM. You may, for example, wish to restrict the number of options available in a drop-down menu dependent on the value of another field.

Concepts

Imagine that you are querying the following map-set to represent an EMPLOYEE's salary grade:

```
<map-set>
 <rec>
    <key>A</key>
    <data>75000</data>
  </rec>
  <rec>
    <key>B</key>
    <data>40000</data>
  </rec>
  <rec>
    <key>C</key>
    <data>20000</data>
  </rec>
</map-set>
```

You have the following storage location, query, map-set definition and schema structure:

```
<fm:storage-location name="sl-ms-salary-grade">
  <fm:cache-key string="sl-TRAINING_EMPLOYEE_EDIT-ms-salary-grade">
  <fm:new-document>
    <fm:root-element>map-set-list</fm:root-element>
  </fm:new-document>
</fm:storage-location>
...
<fm:query name="qry-ms-salary-grade">
  <fm:target-path match="map-set/rec"/>
  <fm:select>
SELECT
  grade "key"
, base_salary "data"
FROM trainingmgr.salary_grades
  </fm:select>
</fm:query>
...
<fm:map-set name="ms-salary-grade">
  <fm:storage-location>sl-ms-salary-grade</fm:storage-location>
  <fm:do>
```

```
    <fm:run-query interface="dbint-ms" query="qry-ms-salary-grade"/>
  </fm:do>
  <fm:refresh-timeout-mins>0</fm:refresh-timeout-mins>
  <fm:refresh-in-background>true</fm:refresh-in-background>
</fm:map-set>
...
<xs:element name="EMPLOYEE" employee:edit=".">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="EMPLOYEE_ID"/>
      <xs:element name="RANK" employee:edit="." employee:map-set="ms-rank"/>
      <xs:element name="SALARY" employee:edit="." employee:map-set="ms-salary-grade"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

The company has the following rule; *no employee with a rank of 'Manager' can be below grade B.*.

As we wish to make *ms-salary-grade* dependent on salary we must first change its storage location cache key to reflect this:

```
  <fm:cache-key string="sl-TRAINING_EMPLOYEE_EDIT-ms-salary-grade :1">
    <fm:using>:{itemrec}/RANK</fm:using>
  </fm:cache-key>
```

This will cache one version of the map-set for each value of RANK found.

The :{itemrec} context works similarly to the :{action} context. It follows the evaluate context rule and matches the closes complex ancestor-or-self of the element using the map-set, in this case the EMPLOYEE record. There is also an :{item} context that would match the EMPLOYEE/SALARY element.

Note: it is important to think carefully about the cache key you use. If it is too unique, this is inefficient as FOX wastes memory caching duplicate versions of the same data. If it is not unique enough, some rows may end up with incorrect map-set values. A good rule of thumb is that any values which are used to decide how to construct the map-set (i.e. query bind variables, typically :{item} or :{itemrec}, or parts of an fm:if statement, etc) should form part of the cache key.

Mapset <fm:do>

To remove the SALARY grade C for RANKs of manager one option is within the **<fm:do>** of the map-set:

```
<fm:map-set name="ms-salary-grade">
  <fm:storage-location>sl-ms-salary-grade</fm:storage-location>
  <fm:do>
    <fm:run-query interface="dbint-ms" query="qry-ms-salary-grade"/>
    <fm:if test=":{itemrec}/RANK[text() = 'Manager']">
      <fm:then>
        <fm:remove match="./map-set/rec[key = 'C']"/>
      </fm:then>
    </fm:if>
  </fm:do>
  <fm:refresh-timeout-mins>0</fm:refresh-timeout-mins>
  <fm:refresh-in-background>true</fm:refresh-in-background>
</fm:map-set>
```

Within the **<fm:do>** the :{itemrec} and :{item} contexts are available and an XPath '.' matches the map-set-list element.
It would also be able to run completely different queries to populate the DOM based on conditional logic.
N.B. This is the method to use if the map-set is based on a template.

<fm:query>

The :{itemrec} and :{item} contexts are also available when running the query to populate the map-set:

```
<fm:query name="qry-ms-salary-grade">
  <fm:target-path match="map-set/rec"/>
  <fm:select>
SELECT
  grade "key"
, base_salary "data"
FROM trainingmgr.salary_grades
WHERE :rank IS NULL OR (:rank = 'Manager' AND grade != 'C')
  </fm:select>
  <fm:using name=":rank">:{itemrec}/RANK</fm:using>
</fm:query>
```

This gives us greater control and allows more complex logic via the database.

Notes

- It does not make sense to define a storage location select statement for a dynamic map-set. Therefore, you need to query the map-set data in using an fm:run-query command within the fm:do block. If you attempt to create a dynamic map-set which is associated with a database row via a storage location, FOX will error.
  - If you need to use a static map-set as a base for your dynamic map-set, you should consider querying off the *ENVMGR.XVIEW_ENV_MAPSETS* view and filtering down the results based on any required logic.
- In other words, dynamic map-sets must be set-up and altered in the **<fm:do>** of the **<fm:map-set>**.
- Due to a bug in FOX, dynamic map-sets must be set out as ***selector+*** widgets, as opposed to *selector* widgets. This will be fixed in a future release.
- Be careful with your storage location cache key when working with dynamic map-sets. In particular, note that a cache key with a UNIQUE bind is only UNIQUE for the scope of a page churn, NOT for the scope of a map-set field evaluation. UNIQUE is therefore misleading - it is not as unique as you might think it is. You should always make sure you bind the data which drives the dynamic map-set into the storage location cache key.

Exercises

1. Remove the action from CORRESPONDENCE/TITLE that updates the GENDER in your XX_PLAN_APP_EDIT module. Create an action that clears the TITLE on changing GENDER.
2. Convert the CORRESPONDENCE/TITLE map-set to a Dynamic Map-set based on the correspondent's GENDER. Only allow titles that make sense for the gender by removing other ones.

Training:Fox:File Uploads

Introduction

The Upload Widget is used to upload files to the database. It is linked to a file-storage-location which allows FOX to control adding, removing and updating files in the database. Each uploaded file is assigned a unique file-id which can be used to reference it. The following training sheet gives some examples on how the widget can implemented in your module.

Syntax

Use the following markup on an element in your schema to define it as an upload widget:

- **type="file-type"** will mark this element as being used for file uploads.
- **ns:widget="file"** will allow you to upload new files or view and download previously uploaded files.
- **ns:file-storage-location="storage-location-name"** is required to specify the file storage location that is to be used by your upload widget.
- **ns:edit="XPath"** attribute must evaluate to true if you want to be able to upload new files. By not including this attribute or providing an XPath that evaulates to false you can restrict your upload widget to be used for viewing or downloading previously uploaded files.
- **ns:upload-mode** can be one of three values:
  - **interrupt-on-module-push-pop-upload-target-any** - Causes the upload to be interrupted on a module push/pop. If your upload is to a transient DOM (i.e. theme) then a push or pop will cause the upload target DOM element to be inaccessible, so the upload must be cancelled.
  - **no-interrupt-modal-upload-window** - Specify this when the window type is modal - in this case, it is not possible to transform the callstack during an upload.
  - **no-interrupt-upload-target-storage-location** - Specify this when the upload is to the :{root} DOM, and is modeless. The root DOM is still accessible to FOX even after a callstack transformation. Uploads in this mode will not be interrupted, allowing the user to continue to work unimpeded in the parent window, without having to wait for their upload to finish.
- **ns:upload-window-type** is used to define the upload window as modal or modeless
- **ns:upload-widget-style** can be used to control the display of the widget

Examples

```
<!-- Example of storage location -->
<fm:file-storage-location name="name">
  <fm:cache-key string="module name + other text"/>
  <fm:database>
    <fm:query>
```

```
        <fm:sql>
SELECT file_content
FROM mytable
WHERE file_id = :1
FOR UPDATE OF file_content
        </fm:sql>
        <fm:using>UPLOAD_WIDGET_NAME/file-id</fm:using>
      </fm:query>
      <fm:insert>
        <fm:sql>
INSERT INTO mytable(
  file_id
, file_description
, file_content
, created_date
, created_by
)
VALUES(
  :1
, :2
, empty_blob()
, SYSDATE
, :3
)
        </fm:sql>
        <fm:using>UPLOAD_WIDGET_NAME/file-id</fm:using>
        <fm:using>UPLOAD_WIDGET_NAME/captured-fields/description</fm:using>
        <fm:using>:{user}/WUA_ID</fm:using>
      </fm:insert>
    </fm:database>
  </fm:file-storage-location>

  <!-- Example of widget mark up -->
<xs:element name="string" type="file-type" ns:edit="." fox:file-storage-location="string" fox:widget="file"
             fox:upload-mode="no-interrupt-modal-upload-window"
fox:upload-window-type="modal"/>
```

Exercises

Exercise 1

- Create a table on the TRAININGMGR schema to store your file uploads using the following statement:

```
  --Replace XX with your initials.
  CREATE TABLE trainingmgr.xx_plan_app_docs (
    file_id             VARCHAR2(30)     NOT NULL
  , file_description    VARCHAR2(4000)
  , file_content        BLOB             NOT NULL
  , created_date        DATE
  , created_by          NUMBER(20)
  )
  TABLESPACE tbsdata;

  GRANT INSERT, UPDATE, SELECT ON trainingmgr.xx_plan_app_docs TO
appenv;
```

- Create a file storage location connected to your table that will allow you to insert and query uploaded files.

Note: For the created_by column you can retrieve the logged in user from :{user}/WUA_ID.

Exercise 2

The file upload element is already defined in the schema, under **APPLICATION_DETAIL/DOCUMENTS/DOCUMENT_LIST/DOCUMENT**.
Set out the DOCUMENT_LIST element in *state-upload-documents*.
Define an action called *action-add-document* which inits a new DOCUMENT element under DOCUMENT_LIST. Define it on a new namespace, and set it out under the DOCUMENT_LIST.
You should be able to upload as many documents as you want now.

Exercise 3

Define a phantom under DOCUMENT which removes the selected DOCUMENT element from the DOM. Use *img/silk_delete* as the phantom's image.

Training:Fox:PlanAppSecurityTable

Introduction

The security table is a set of rules for overriding active namespaces for set-out/menu-out.
It is a further final way of filtering the content of the screen under different conditions.

Concepts

There are 2 types of security rule, mode and view:

```
<fm:security-list>

 <fm:mode-rule namespace="namespace" operation="ENABLE | DISABLE" [privilege="user privilege" state="state" theme="entry-theme" xpath="XPath"]/>

 <fm:view-rule namespace="namespace" operation="ENABLE" [privilege="user privilege" state="state" theme="entry-theme" xpath="XPath"]/>

</fm:security-list>
```

If you're using a small screen, get a bigger screen.

<fm:mode-rules> can ENABLE a namespace for edit and read-only set-out or DISABLE it.
<fm:view-rules> can ENABLE a namespace in read-only mode.
The namespace, privilege, state and theme attributes can all contain comma separated conditions.

The FOX engine checks the following and optimises processing internally to aid performance:

- privilege – User is logged on, and has one of these named system privileges.
- theme – The module was initially entered under one of these named entry-themes.

- state – The current module state is one of these named states.
- xpath – The XPath test (run relative to root element) returns true.

Security rules work according to the following principles:

1. Every namespace is turned on until mentioned in a security rule.
2. Once a namespace is mentioned in a rule it must be explicitly ENABLE-d (in that rule or another) if required.
3. A rule with operation ENABLE will whitelist the namespace under the specified conditions. Different ENABLE-ing rules are OR-ed together.
4. A mode-rule with operation DISABLE will blacklist the namespace under specified conditions. DISABLE-ing rules overwrite ENABLE-ing rules.
5. Attribute conditions are AND-ed together. e.g. theme="view" privilege="VIEW_OBJECT" means 'on entry theme "view" AND with privilege "VIEW_OBJECT")'
6. Attribute comma-separated-values are OR-ed together. e.g. theme="view, edit" privilege="VIEW_OBJECT" means '(on entry theme "view" OR "edit") AND with privilege "VIEW_OBJECT"'

This rule processing approach is similar to Fire Wall Rule Table processing.

Examples

Example 1

Consider that we have a *department-edit* namespace that we wish to disable in the state "state-view" we may try the security list:

```
<fm:security-list>
 <fm:mode-rule namespace="department-edit" operation="DISABLE" state="state-view"/>
</fm:security-list>
```

This will have the effect of turning off *department-edit* for state "state-view", but as *department-edit* has now been mentioned in a rule, it will be DISABLEd for all conditions (principle 2 above).

Thus we have to enable the namespace for other conditions:

```
<fm:security-list>
 <fm:mode-rule namespace="department-edit" operation="DISABLE" state="state-view"/>
 <fm:mode-rule namespace="department-edit" operation="ENABLE"/>
</fm:security-list>
```

Now consider that we want to only allow *department-edit* where the user has the privilege "EDIT_DEPARTMENT"; we can do this:

```
<fm:security-list>
 <fm:mode-rule namespace="department-edit" operation="DISABLE" state="state-view"/>
 <fm:mode-rule namespace="department-edit" operation="ENABLE" privilege="EDIT_DEPARTMENT"/>
</fm:security-list>
```

N.B. If a user enters the state "state-view" with privilege "EDIT_DEPARTMENT" then *department-edit* will be DISABLEd. Due to principle 4 processing for the namespace stops after a successful DISABLE.

Example 2

Now consider we are scrapping the namespace *department-edit* and wish to use a namespace *department* instead. We want to control whether fields are read-only or editable via a security table and try:

```
<fm:security-list>
 <fm:mode-rule namespace="department" operation="ENABLE"/>
 <fm:view-rule namespace="department" operation="ENABLE" theme="view"/>
</fm:security-list>
```

The mode-rule ENABLE-s *department* as editable for all conditions initially.
The developer has then attempted to force *department* to be read-only under the entry-theme "view" using a view-rule.
Due to principal 3 above, however, the ENABLE-ing rules are OR-ed together and the mode-rule makes the view-rule redundant.
The developer may solve this by doing the following:

```
<fm:security-list>
 <fm:mode-rule namespace="department" operation="ENABLE" theme="edit, create"/>
 <fm:view-rule namespace="department" operation="ENABLE" theme="view"/>
</fm:security-list>
```

Thus allowing *department* fields to be editable on only the "edit" and "create" entry themes, and read-only on "view".

We next want to add a condition so that nobody can view/edit data without the "DEPARTMENT" privilege:

```
<fm:security-list>
 <fm:mode-rule namespace="department" operation="ENABLE" theme="edit, create" privilege="DEPARTMENT"/>
 <fm:view-rule namespace="department" operation="ENABLE" theme="view" privilege="DEPARTMENT"/>
</fm:security-list>
```

The privilege "DEPARTMENT" condition must be added to both rules. Both rules will be evaluated and can ENABLE *department* even if the first fails.

<fm:security-rule>

A set of conditions (state, theme, privilege and XPath) may be defined using an **<fm:security-rule>**.
These can be referenced using the *rule-ref* attribute on another mode or view-rule.
Thus the final example above could be written:

```
<fm:security-list>
 <fm:security-rule name="department-privs" privilege="DEPARTMENT"/>
 <fm:mode-rule namespace="department" operation="ENABLE" rule-ref="department-privs"/>
 <fm:view-rule namespace="department" operation="ENABLE" theme="view" rule-ref="department-privs"/>
</fm:security-list>
```

Why not just use XPaths on the set-out?

These security rules could be applied using XPaths, but these tend to be less readable and would have to be repeated on every set-out.
Use security tables where you can.

Exercises

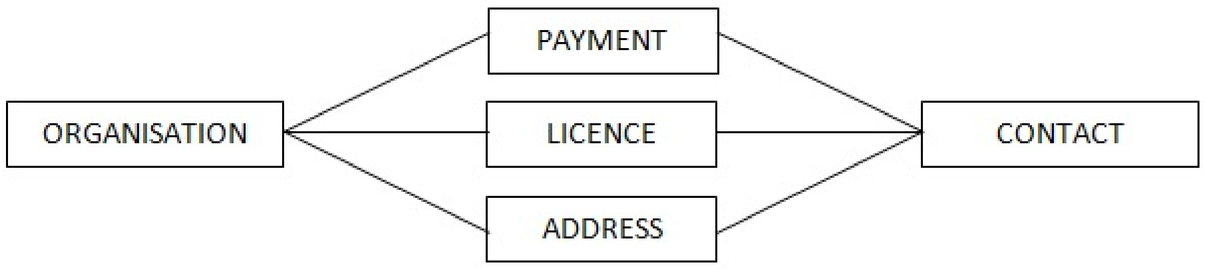
1. Take your XX_PLAN_APP_EDIT module and completely remove the *view* namespace and all of its attributes.
2. Remove the state view, and point the entry-theme view to the same state as edit.
3. Write a security table to make the namespaces edit and upload read-only on the entry-theme view. Make sure the namespaces retain their functionality on the other entry-themes.
4. Access the module under the different entry-themes to test your work.

Training:Architecture:UREFs

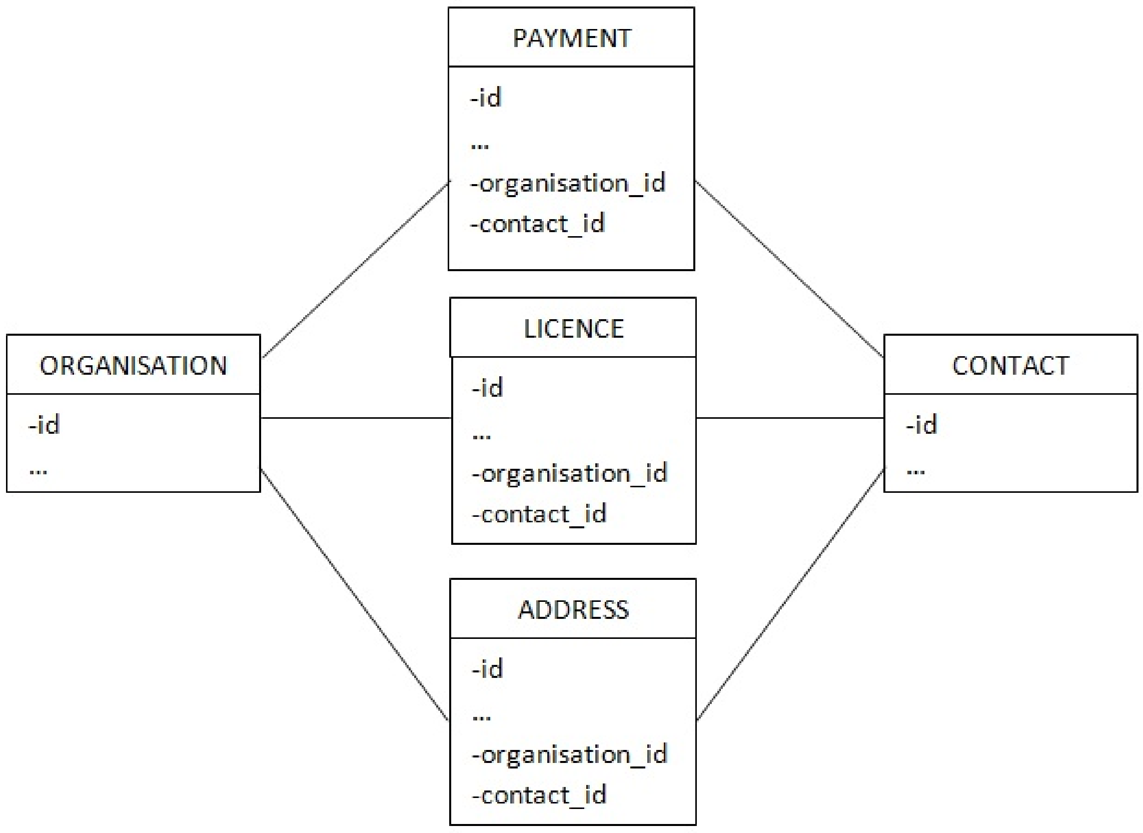
UREFS

UREF Concepts

Consider the situation where you have an entity that can be related to many other entities. The following logical model shows that a PAYMENT record, a LICENCE, and a list of ADDRESSes belong to either an ORGANISATIONs or CONTACTs



This type of model would usually be implemented in the following way with each entity having a foreign key to the appropriate CONTACT or ORGANISATION:



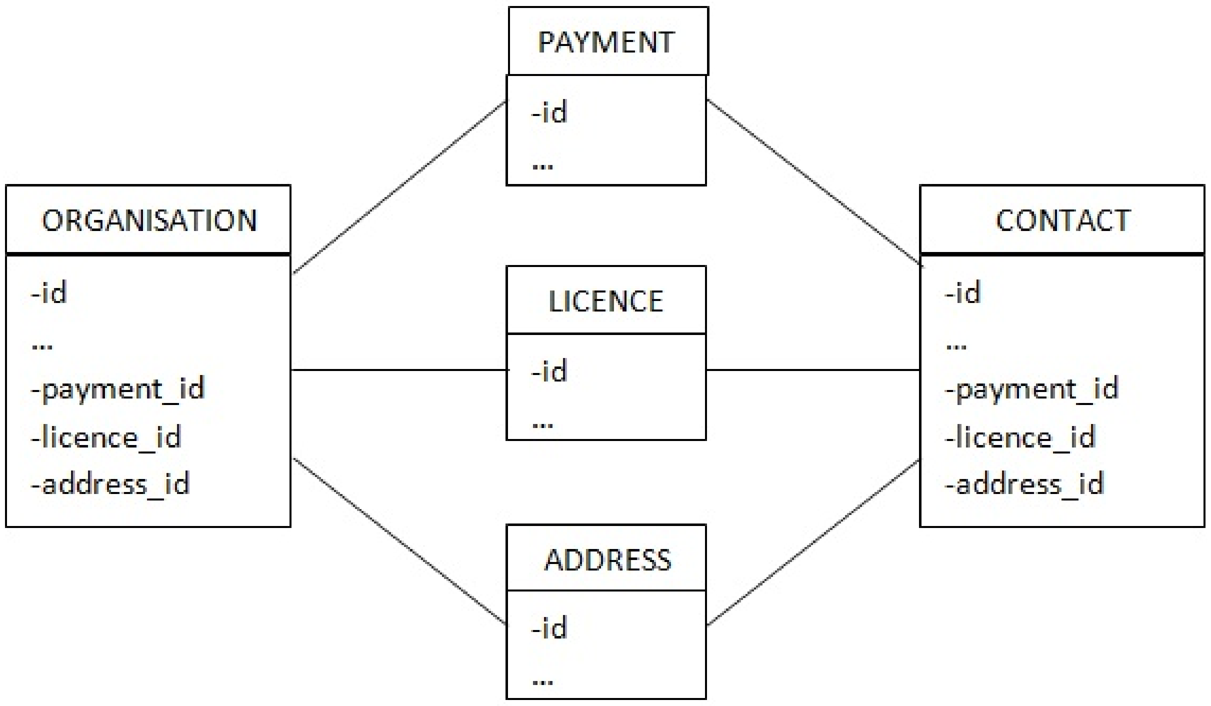
But what if we wanted to find all usages of any given ORGANISATION or CONTACT? Or to determine if the given ORGANISATION or CONTACT has permissions access to a particular PAYMENT, LICENCE or list of ADDRESSes? We would have to write two SQL statements similar to the following:

```
SELECT ‘PAYMENT’, id
FROM payment
WHERE organisation_id = ?
UNION ALL
SELECT ‘LICENCE’, id
FROM licence
WHERE organisation_id = ?
UNION ALL
SELECT ‘ADDRESS’, id
FROM address
WHERE organisation_id = ?
```

```
SELECT ‘PAYMENT’, id
FROM payment
WHERE contact_id = ?
UNION ALL
SELECT ‘LICENCE’, id
FROM licence
WHERE contact_id = ?
UNION ALL
SELECT ‘ADDRESS’, id
FROM address
WHERE contact_id = ?
```

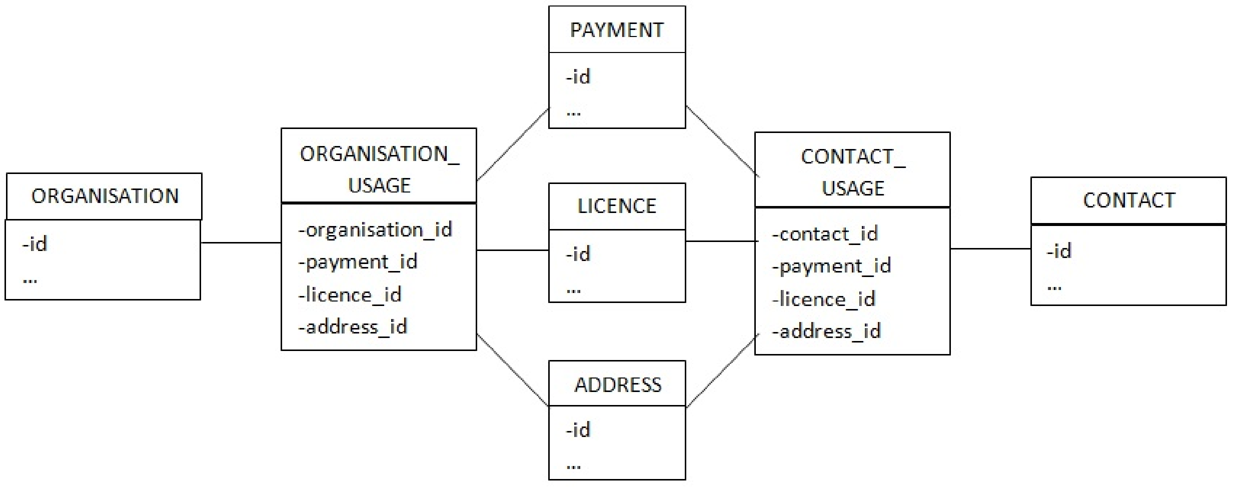
The above query works, but does it scale? What happens when we add or remove an entity? We have to remember to go back and edit both queries to reflect this change. What happens when we have 100 entities that use ORGANISATIONs or CONTACTs? To determine where a particular ORGANISATION or CONTACT is used 100 tables will need to be accessed.

To help with this, the relations in the data model could be reversed in the following fashion:



This revised model now helps with some of the issues, you can easily determine permissions and usages of an ORGANISATION or CONTACT by querying off of either table. However, this has many of the same issues, how does this scale? Every time an entity is added or removed the ORGANISATION and CONTACT tables need to have a column added or removed. What if there are multiple PAYMENT records for a given ORGANISATION or CONTACT? This would require multiple rows in the ORGANISATION and CONTACT tables, one for each PAYMENT leading to an un-normalised model. With multiple rows per ORGANISATION or CONTACT the payment_id, licence_id and address_id foreign keys should be mutually exclusive, thus requiring a constraint that will also need to be maintained when entities are added or removed. It quickly becomes a maintenance nightmare.

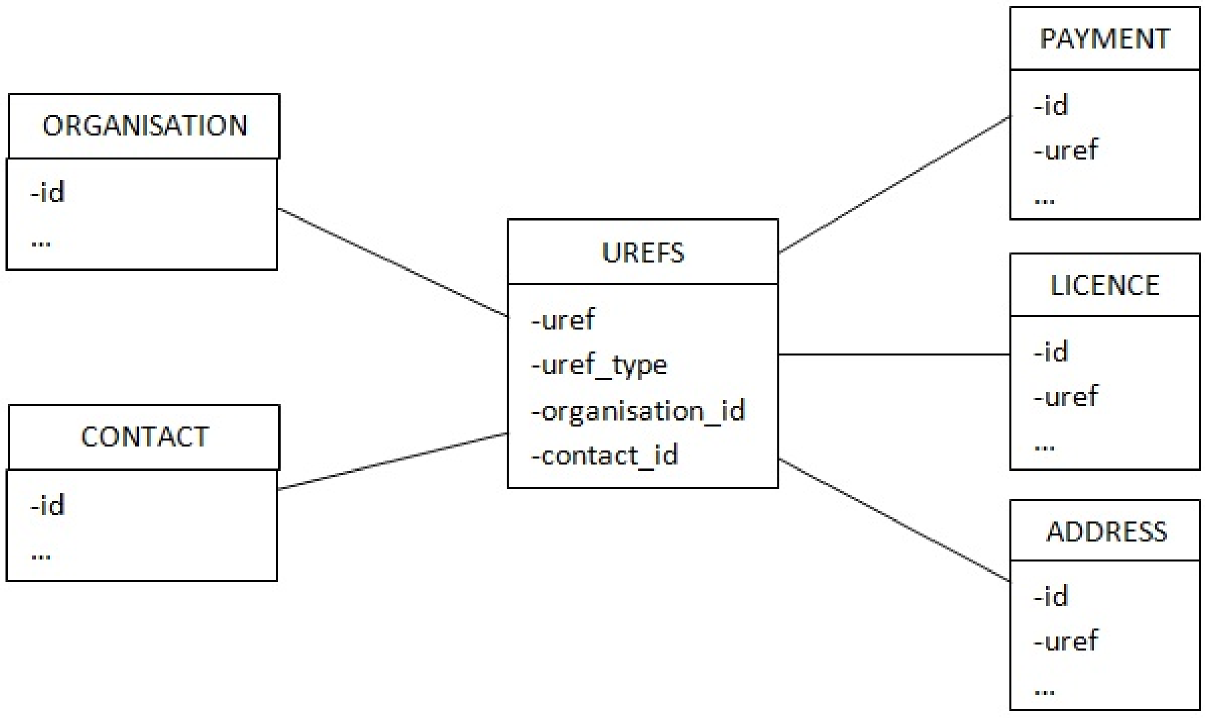
To normalise the model we can easily do the following:



So this now helps with the normalisation but we still have the issue of the maintenance, every time a new entity is added that uses ORGANISATIONs and CONTACTs the ORGANISATION_USAGE and CONTACT_USAGE tables and constraints needs to be amended. This is where UREFs come in.

A UREF is a ‘Universal Reference’ with the idea being that it is a globally unique id (across the entire database) and is centrally managed. UREFs work by initially creating some metadata and then letting the UREF database packages handle the rest. Now we will apply UREFs to the model above with the aim of making maintenance of the model easier.

In the above model you will note that the ORGANISATION_USAGE and CONTACT_USAGE tables achieve the same goal, both relate a single entity to many other entities and in this case all the entities are the same. It is these two tables that can be replaced by a UREF table so that the model becomes the following:



This model now moves all the maintenance into a single location. Much easier – only one constraint to maintain and only one table to add additional columns to.

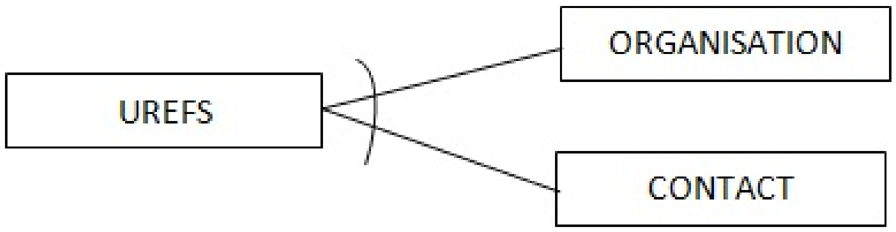
So how does this work? As mentioned earlier a UREF is a globally unique id and is maintained by a UREF package on the database. In this example you would choose to create UREFS for the ORGANISATION and CONTACT tables, this means they can now be universally referenced via the UREFS table. Once the UREF for the ORGANISATION table is created, there will be a one-to-one relationship between the UREFS table and the ORGANISATION table, the UREFS table will have the id populated in the organisation_id column and the uref column will be the globally unique id, such as ‘67ORG’. Likewise a similar approach will happen for the CONTACT table but this time using the contact_id column and an uref such as ‘45CON’.

The uref is simply a concatenation of the original primary key of the base table and the uref_type and for convenience they are human readable. E.g. you can infer that UREF 67ORG is the record from the ORGANISATION table with the id 67. To use the uref you can easily concat an uref type to an id or replace the uref type in an uref to get its original id.

Any existing table on the database whether a new or legacy application can be made universally reference-able with zero impact to the existing application.

When creating the UREF the UREF package on the database generates triggers so that when new records are inserted into the base table (i.e. ORGANISATION or CONTACT) a corresponding record is inserted into the UREFS table.

Likewise, when a new UREF type is added, the column is created on the table automatically and the constraint to maintain the mutually exclusive foreign keys is automatically maintained. This mutually exclusive relationship is known as an 'arc-relationship' and can be shown on a diagram as follows:

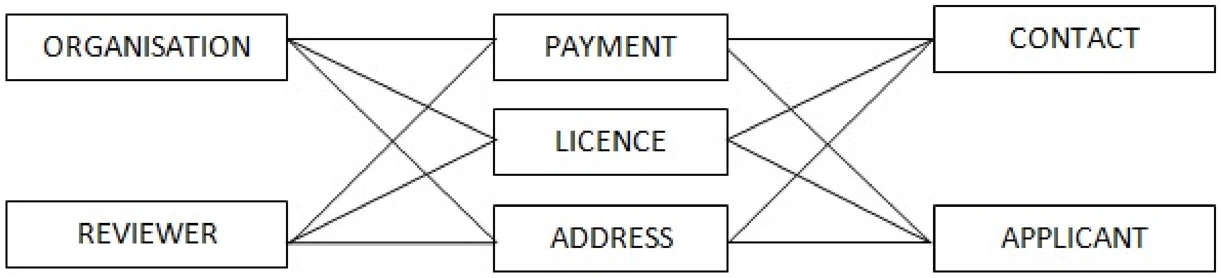


The PAYMENT, LICENCE and ADDRESS table can now have a foreign key to the UREFS table rather than directly to the desired entity. So you can now infer that a given PAYMENT is related to a particular UREF which in turn is related to either a CONTACT or ORGANISATION. The one-to-many relations still exist, multiple PAYMENTs, LICENCEs or ADDRESSes can relate to the same UREF.

Using the UREF model above the task of determining permissions or usages for a given CONTACT or ORGANISATION is straight forward, simply query off of the UREFS table only where the uref_type is either that of a CONTACT or ORGANISATION and the uref is equal to the ORGANISATION id or CONTACT id you are interested in.

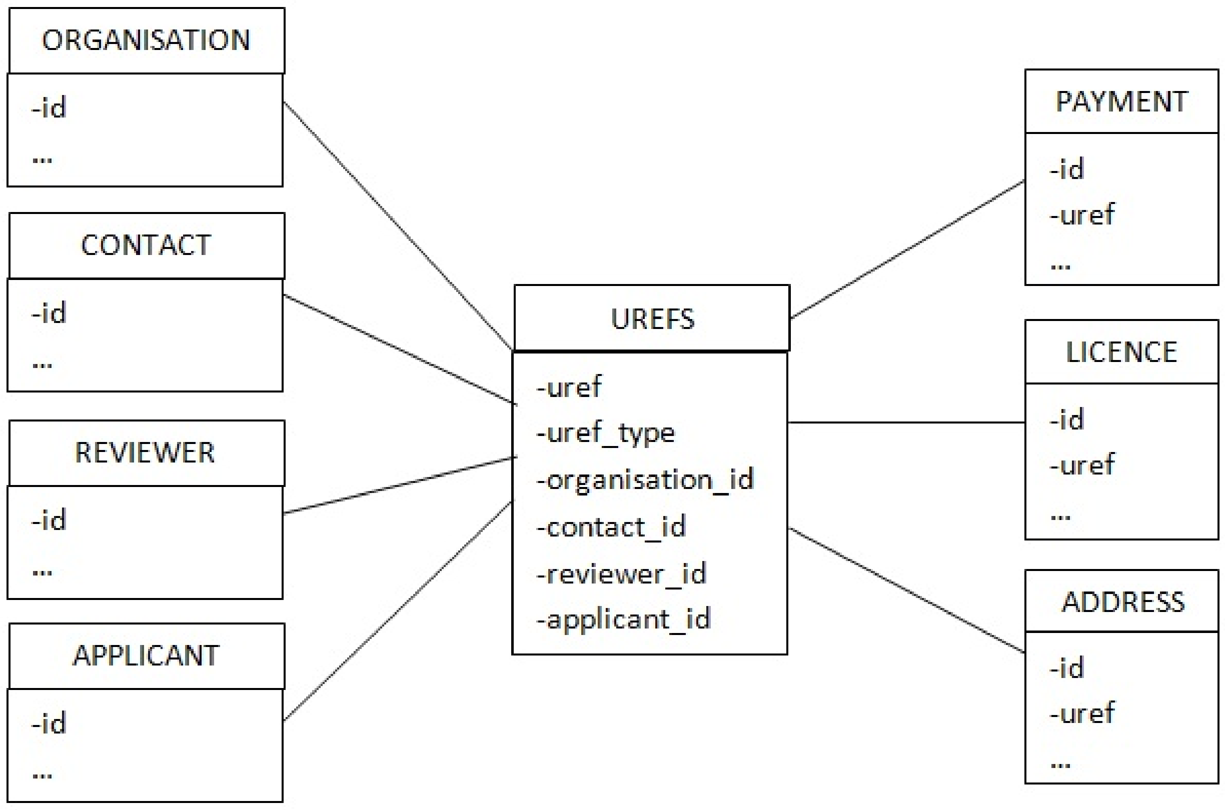
Adding a new entity that also uses CONTACTs or ORGANISATIONs require no maintenance at all - simply include a UREF foreign key in the new entity and the rest is already done for you.

Now consider the effort of adding a more entities that can be referenced by PAYMENTs, LICENCEs and ADDRESSes so that the model looks like the following:



The standard method of modelling this would require two new tables and two new joining table to represent the relationship between REVIEWERs and PAYMENTs, LICENCEs and ADDRESSes and the relationship between APPLICANTs and PAYMENTs, LICENCEs and ADDRESSes. What then happens if you introduce another entity using ORGANISATIONs, CONTACTs, REVIEWERs and APPLICANTs? You would need to update the 4 joining table to add the additional column and modify their mutually exclusive (arc) check constraint. You can really see now how this very easily becomes a model that no one will want to maintain.

Alternatively, using UREFS, you simply create the two new entities (REVIEWER and APPLICANT) and create new UREF types for them. The adding of the column and the check constraint are handled for you in the UREFS table. Nothing else needs to be done. Any additional entities you create that use ORGANISATIONs, CONTACTs, REVIEWERs and APPLICANTs don't require any changes to the rest of the model, they simply need to relate to the UREFS table. An example of how the model will look once the REVIEWER and APPLICANT tables have been added can be seen below.



Whilst the examples given throughout may seem a little fabricated, with a lot of systems we design, these kinds of scenarios occur often. This is largely due to the generic nature in which we design components for reusability. A classic real-life example is the concept of a team. A team is a group of people and each person is given certain roles. This model is generic so that many teams can exist with many different people and role combinations. These teams are very often used for managing access to a particular item, for example, a team will be created for an Oil and Gas Licence, this gives power to the applicant to choose who is in the team and who can work on the application. This means that the team must have a relation to the particular licence application record to infer security. But as this is a generic component, the generic team model will also be used for other system such as managing HR records, or creating planning applications. You can see via this example how a single entity is related to many others and the benefit of doing this is a 'universally reference-able' way helps reduce maintenance.

UREF Creation

Creating UREFs is simple you simply add meta data about the UREF type to the bpmmgr.UREF_TYPES table and the UREFS package takes care of the rest.

1. Grant select, references, index on your base table to bpmmgr.
2. Insert a record in bpmmgr.uref_types referring to the table below for the description of each columns content.
3. Enter a 'c' (Create) or a 'd' (Drop) in the Enter Command Here column and commit to begin the job that will create all the objects and make the base table universally reference-able – Alternatively you can run an anonymous block calling uref_types_auto.command. This will add a corresponding record into the UREFS table and create triggers on the base table to maintain future inserts into this table. When the UREF has been successfully created you will see a success message in the last_system_message column, likewise any error will also be shown in this column. Please bear in mind that if the base table already has a large number of records in it then the UREF may take a while to create.
4. Your UREF is now ready to use.

Column Name	Description	Example
UREF_TYPE	Short mnemonic describing the application. This will be concatenated with the id of each row in the base table.	P15D
OBJECT_OWNER	Schema that the base table resides in.	ENVMGR
OBJECT_NAME	Name of the base table.	PON15D_DETAILS
OBJECT_PK_1_COLUMN_NAME	Primary key of the base table.	ID
OBJECT_PK_2_COLUMN_NAME	Second column name if the base table primary key is composite. (optional)	
UREF_FK1_COLUMN_NAME	Name of the column to use in the UREFS table for the first primary key column.	PON15D_DETAILS_ID
UREF_FK2_COLUMN_NAME	Name of the column to use in the UREFS table for the second primary key column. (optional)	
DESCRIPTION	Description of what this UREF is for	EIA Direction / Chemical Permit Application (Installations)
EXPANDED_TYPE	Application mnemonic	PON15D
CONTROL_XML	Application Generic XML. (optional)	
ENTER_COMMAND_HERE	See point 2 above.	
LAST_SYSTEM_MESSAGE	See point 2 above.	
INTERNAL_NAME	Up to 20chars internal object name.	PON15D_DETAILS
STRATEGY	Defaults to 'STANDARD' but can also be 'JIT'. Please see the note below.	STANDARD

NOTE: UREFs can also be created with a strategy of 'JIT', this creates a just-in-time style UREF. The 'STANDARD' strategy creates one row in the UREFS table for every row in the base table. 'JIT' UREFs only create the record in the UREFS table when they are needed. This needs to be done explicitly by calling the bpmmgr.urefs.create_jit_uref() procedure. A 'JIT' UREF can also be removed via the bpmmgr.urefs.delete_jit_uref() procedure.

'JIT' UREFS are not commonly used and there are two main scenarios where they are useful.

The first scenario is when there are lots of records in a base table but only some of them need a UREF. The second is when some of the data in a base table is historical and will never need a UREF.

Training:Architecture:Teams

Teams

Resources, Contacts Lists and Teams are all the same thing. They are a number of people and/or addresses grouped together for a specific purpose. They may be in context of an object or associated with one or standalone.

The Tables

Resource Types

One resource type is required to define the structure of the resource.

```
decmgr@DB1LV> desc RESOURCE_TYPES
Name                            Null?    Type
------------------------------ -------- ------------
RES_TYPE                        NOT NULL VARCHAR2(30)
XML_DATA                                 SYS.XMLTYPE
```

The XML is metadata that conforms to pvcs:\\CodeSource\XMLSchema\RESOURCE_TYPE.xsd; a description of the elements in this XML is shown below:

RES_TYPE	The resource type used internally, never displayed to users
RES_TYPE_TITLE	Title of the resource as displayed in search screens
DESCRIPTION	Description of the resource displayed in the team edit screen
RESOURCE_NAME_EDITABLE	If true, allows the resource name to be editable, if not, defaults to upper(title)
RESOURCE_NAME_HINT	Only useful if name editable is true, allows user to see the style of team name required. E.g. ERDU ADMIN 1
SCOPED_WITHIN	Either UNIVERSAL_SET or PARENT, indicating if initially, the team is standalone or contextual respectively.
SAVE_EVENT_LABEL	An event to raise when an instance of this team is saved.
UREF_RESTRICT_LIST	Only useful if scoped within is PARENT, contains a list of uref types that this resource can be in context of.
ROLES_LIST	Contains a list of roles found within the team.

Roles

ROLE_NAME	The role name used internally, never displayed to users
ROLE_TITLE	Title of the role as displayed in the team edit screen
ROLE_DESCRIPTION	Description of the role as displayed in list view in the team edit screen
MIN_MEMS	Minimum number of members in this role in this team, enforced by validation
MAX_MEMS	Maximum number of members in this role in this team, enforced by validation
DISPLAY_SEQ	The order the roles are displayed in the team both in list view and grid view
PERSON_REQUIRED	True = a person must be in this role False = a person cannot be in this role Optional = a person may be in this role
PERSON_RESTRICT_LIST	
PERSON_REQ_ACTIVE_WUA	t/f/o

ADDRESS_REQUIRED	True = an address must be in this role False = an address cannot be in this role Optional = an address may be in this role
TELEPHONE_REQUIRED	t/f/o
TELEPHONE_RESTRICT_LIST	
EMAIL_REQUIRED	t/f/o
EMAIL_RESTRICT_LIST	
QUALIFYING_COMMENT_REQ	t/f/o
BRANCH_REQUIRED	t/f/o
AUTHORISATION_REQUIRED	t/f/o
NOTIFICATION_METHOD_REQ	t/f/o
DEFAULT_SYSTEM_PRIV_LIST	Can contain a list of system privileges that membership of this role gives by default to the person in it.

t/f/o = true/false/optional. Changing these values changes what data is required to be entered by a user saving a team with this role. E.g. changing telephone required from optional to true will mean a telephone number is mandatory.

Resources (Team)

One or more resources are required for each instance of the resource type. If the resource is of standalone type (UNIVERSAL_SET) then only one resource is ever required and it can be created using the New Contact List/Team generic functionality. If the resource is contextual (PARENT) then application code must insert a resource and a context (resource_usages – next …)

```
decmgr@DB1LV> desc RESOURCES
 Name                              Null?    Type
 ------------------------------ -------- ------------
 ID                                NOT NULL NUMBER(12)
 RES_TYPE                          NOT NULL VARCHAR2(30)
 XML_DATA                                   SYS.XMLTYPE
```

The XML data conforms to pvcs:\\CodeSource\XMLSchema\RESOURCES.xsd.

Resource Usages (Team Usage [Context])

Resource usages can be inserted for contextual resources. It is possible to associate one resource with one or more universally reference-able objects using this table. Currently the purpose is only PRIMARY_DATA.

```
decmgr@DB1LV> desc RESOURCE_USAGES
 Name                              Null?    Type
 ------------------------------ -------- ------------
 RES_ID                            NOT NULL NUMBER(12)
 UREF                              NOT NULL VARCHAR2(20)
 START_DATETIME                    NOT NULL DATE
 END_DATETIME                               DATE
 PURPOSE                           NOT NULL VARCHAR2(30)
```

Resource Details (Team History Details)

Resource details are managed by the team edit screen only and contain a historical record of each team.

```
decmgr@DB1LV> desc RESOURCE_DETAILS
 Name                              Null?    Type
 ------------------------------ -------- ------------
 ID                                NOT NULL NUMBER(12)
 RES_ID                            NOT NULL NUMBER(12)
 START_DATE                        NOT NULL DATE
 END_DATE                                   DATE
 XML_DATA                                   SYS.XMLTYPE
 STATUS_CONTROL                             VARCHAR2(1)
 STATUS                            NOT NULL VARCHAR2(12)
```

Resource Roles (Team Roles)

Resource roles is a trigger-populated table. The trigger fires every time a new resource detail is inserted, and creates one row for every role in this resource. This enables a role in an instance of a team to be referenced without needing to know what people or addresses are in the team.

```
decmgr@DB1LV> desc RESOURCE_ROLES
 Name                               Null?    Type
 ------------------------------- -------- ------------
 ID                                 NOT NULL NUMBER(12)
 RES_ID                             NOT NULL NUMBER(12)
 ROLE_NAME                          NOT NULL VARCHAR2(30)
```

Associated Views

- Xview Resource Members History is an xview of resource details. It is a relational record of the xml data. This view is for information only and should not be used in application code.
- Resource Members Current is a view of the historical view above and contains the current view of all team memberships. This should be used in application code.

Fox Modules

- DEC024X
  - Allows searching/editing of standalone teams
  - Allows creation of standalone teams
- DEC043L
  - Allows viewing/editing of a team
  - Parameters are required to indicate which resource
    - You can also specify a default display view, grid or list
    - You can also specify whether you want to view or edit

Object Security Rules

There are two object security rule that apply to all teams

- If you are a member of the DTI_SUPER_USERS team type and in the role EDIT_ANY_TEAM, you can edit any resource on the system
- If you are a member of any team, in the RESOURCE_COORDINATOR role in that team, you can edit that team

For anyone else to edit a team object security rules must be written on object uref type RES.

Validation of Teams

Teams are validated when saved. There are three levels of validation

1. Basic validation
   - Ensure that the every role with members is still valid
   - Ensure that minimum and maximum number of people per role is not obeyed
2. More advanced validation
   - Ensure that PERSON_REQUIRED, ADDRESS_REQUIRED and other resource type definition flags set per role are correct
3. Specific validation
   - Resource type specific validation (Not yet implemented)

Example 1 – A standalone team

The resource type is PON15D_DTI_ADMIN

Example 2 – A contextual team

The resource type is COMPANY_ADDRESSES

Example 3 – A multiple contextual team

The resource type is PON15D_COMPANY_CONTACTS

Questions

1. List all resource types with multiple current usages
2. List all web user accounts which can edit team type PON15D_DTI_ADMIN
3. List all web user accounts which can edit team type PON15D_COMPANY_CONTACTS for PON15D/6
4. For a selected address find all teams where this address is used in a contact role. Show the team name, context, role name and address details
5. For the selected address in 4) find all teams where this address is used in a contact role where the role type optionally requires a person
6. As for 5) but where the role type must not have a person
7. Produce a report showing for each team type the number of teams which exist and the number of people allocated to each team and the number of address only roles