

OXF Tutorial

An Introduction to the OXF XML Platform

Table of Contents

1	Introduction	4
1.1	Goal and Structure of Tutorial	4
1.2	Running the Tutorial Examples	4
2	Hello World Classic.....	4
2.1	Introduction	4
2.2	Pages.....	4
2.3	Page Templates.....	5
2.4	Hello World Classic: Combine Content and Presentation	5
3	Hello World and Separation of Concerns: The MVC Architecture	6
3.1	Introducing the Model View Controller (MVC) Architecture	6
3.2	Hello World MVC: Separate Content from Presentation	7
4	Hello World XForms: Managing User Input.....	8
4.1	Introducing XForms	8
4.1.1	XForms Model and Instance.....	8
4.1.2	Creating an XForms Instance	9
4.1.3	XForms Controls	9
4.2	Hello World XForms: Collecting User Input.....	10
5	Hello World Page Flow: Managing Multiple Pages	12
5.1	Introducing Page Flow.....	12
5.2	Declaring and Navigating Multiple Pages	13
5.3	Sharing XForms between Pages	14
6	The BizDoc Application	16
6.1	Introduction	16
6.2	BizDoc Functional Overview	16
6.3	OXF Background: Using Schemas	19
6.3.1	Validating User Input Using XML Schema and XForms.....	19
6.4	OXF Background: XML Pipelines Using XPL	21
6.4.1	A Simple Pipeline: Combine Two XML Documents	22
6.4.2	Lazy Processor Execution Within a Pipeline.....	23
6.4.3	A Simple 2-Stage Pipeline: Apply an XSL Transform	24
6.4.4	Getting Data In and Out of a Pipeline using Parameters	25
6.4.5	href vs. ref in Pipelines	26
6.4.6	Pipeline Inputs and Outputs Within a Page Flow	26
6.5	The BizDoc Summary Page.....	27
6.5.1	Introduction	27
6.5.2	Summary Page: The Page Model.....	28
6.5.3	Simplified Summary Page View.....	30
6.5.4	Complete Summary Page View	31
6.6	The BizDoc Detail Pages.....	33
6.6.1	Introduction	33
6.6.2	Detail Page: The XForms Model	34
6.6.3	Detail Page: The Page View	35
6.7	BizDoc Actions: Navigating from Summary to Detail.....	36
6.7.1	Executing an Action Pipeline	36
6.7.2	Passing Data Out of an Action to Another Page's XForms Instance.....	38
6.8	BizDoc Actions: Saving and Deleting Documents	38
6.8.1	Introduction	38
6.8.2	Conditionals in XPL	39
6.8.3	Updating and Inserting Claim Documents.....	40
6.8.4	Deleting Claim Documents.....	40
6.8.5	Miscellaneous Actions.....	41
7	BizDoc and the Real World: Changing Requirements	41
7.1.1	Introduction	41

7.1.2 Steps	41
8 Conclusion and Further Steps	42

Table of Figures

Figure 1 – Pages in a Web Application	4
Figure 2 - Hello World Classic.....	6
Figure 3 – MVC Components in Classic Hello World	7
Figure 4 - Hello World and User Input.....	10
Figure 5- Hello World Greeting	10
Figure 6 - XForms Hello World and MVC.....	11
Figure 7 – Ask Name Hello World Page	13
Figure 8 – Display Name Hello World Page	13
Figure 9 - XForms Instance Update using XUpdate.....	15
Figure 10 - BizDoc Summary Page.....	16
Figure 11 - BizDoc First Detail Page	17
Figure 12 - BizDoc Second Detail Page	18
Figure 13 - BizDoc Page Flow by Action	19
Figure 14 - XML Processor Inputs and Outputs.....	22
Figure 15 - Simple Pipeline to Aggregate Two Files	23
Figure 16 - Two-Processor Pipeline	25
Figure 17 - Pipeline with Input and Output.....	26
Figure 18 - XForms Instance, Model and View Connections in the Page Flow Controller	27
Figure 19 - Summary Page Model and View Connection	28

1 Introduction

1.1 Goal and Structure of Tutorial

The goal of this tutorial is to familiarize the reader with the technologies and steps required to build applications with the OXF platform. This tutorial assumes you have a basic background creating web applications using Java or PHP or similar web development environment, but no background using OXF. A minimal background in XML is required, which should include the basic element and attribute syntax of an XML document, and exposure to XPath and XSLT. At the end of the tutorial, you should be able to develop web applications using OXF, with more advanced concepts and complete detail being covered in the Reference section of the OXF documentation.

The tutorial is structured as a series of increasingly complex “Hello World” walkthroughs, followed by a complete walkthrough of a simplistic document-centric application called “BizDoc,” which illustrates creating, reading, updating, and deleting (also known as CRUD operations) XML business documents from a database.

1.2 Running the Tutorial Examples

While not required, it is helpful to have access to an OXF installation and example application source code. It might also be helpful to interact with the running application to verify behavior described during this tutorial. Please see the Installing OXF section of the OXF documentation for information on running the BizDoc application mentioned in this tutorial. You can also interact with the running examples on the Orbeon web site at <http://www.orbeon.com/oxf>.

2 Hello World Classic

2.1 Introduction

The Hello World Classic application is a single-page application displaying the text “Hello World”. We later consider variants of Hello World, including one where the user of the application can enter a name in a text field, submit it, and receive a personalized greeting. In the process, we examine the basic concepts underlying OXF applications. The source code for the Hello World application is available in `oxf.war`, under `WEB-INF/resources/examples/tutorial`. You can modify the XML files in that directory, and see the changes immediately reflected in the application as you reload pages.

2.2 Pages

Web applications appear to a user as a collection of pages presenting information and taking user input, which may cause the application to display another page. A page is a piece of user interface sent to a web browser. A page can consist of almost anything that displays in a browser, including a report formatted in a table, a data entry form, or a plain page of narrative text. A very simple application may have only one page, while a complex application can have hundreds of them

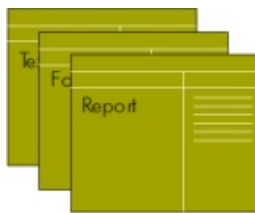


Figure 1 – Pages in a Web Application

Each page can have 3 main components:

- **Page Content:** The raw information the page intends to communicate to the user, as opposed to formatting aspects.
- **Page Presentation:** The formatting made up of the fonts, colors, and other design elements.
- **User Actions:** Any actions that can be taken on the page, such as following a hyper-link to another page or pressing a button on a form that accepts user input.

These components are handled by different parts of OXF, which we will explore later in this tutorial.

NOTE: OXF web applications are accessed through regular web browsers, using familiar technologies such as HTML, CSS and JavaScript. OXF does not require the installation of plugins.

2.3 Page Templates

A page in OXF is built with the following core technologies standardized by the World Wide Web Consortium (W3C):

- XHTML (the XML-compatible version of HTML)
- XSLT as the page template language
- XForms as the forms handling language
- CSS as the page styling language

Learning XHTML is extremely easy for developers who already know HTML. The difference is mainly a matter of closing opening tags and quoting attributes. For more information on XHTML, see <http://www.w3.org/MarkUp/>.

There are many common page template languages, including JSP, PHP, and ASP. The idea behind a page template language is to describe the elements of a page that do not change using a markup language such as HTML or XHTML, and then to interleave commands to fill-in the parts that need to be modified dynamically, when the page is actually displayed. OXF uses XSLT as its page template language, which provides benefits including the following:

- It is an industry standard (W3C)
- It is designed to natively handle XML data
- It does not require learning Java or other procedural languages
- It integrates very well with XForms and XHTML

2.4 Hello World Classic: Combine Content and Presentation

A simple static page displaying the text “Hello World!” might use the following XHTML page template:

view.xhtml:

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Hello World</title>
  </head>
  <body>
    <p>Hello World!</p>
  </body>
</html>
```

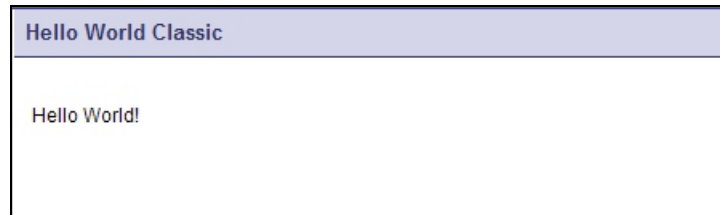


Figure 2 - Hello World Classic

A page template is almost all that is needed to create a simple application. The only additional work that must be done is to associate the page with a URL that the user types in their browser. This is done by declaring the page in an important OXF configuration file called the *page flow*, and usually named `page-flow.xml`. For Hello World, this file contains the following entry:

`page-flow.xml`:

```
<config xmlns="http://www.orbeon.com/oxf/controller">
  <page path-info="/tutorial-hello1" view="view.xhtml"/>
  <epilogue url="oxf:/config/epilogue.xpl"/>
</config>
```

The `page` entry declares URL addressable pages in the application. In this case, when the path `/tutorial-hello1` is requested by the user of the application, the page view defined by `view.xhtml` will be displayed to the user. The `epilogue` is a special entry that is always present in a page flow file. The epilogue takes care of applying the look and feel common to all pages in an application.

As an exercise, you are encouraged to run the OXF examples and modify the `view.xhtml` template, for example by adding some XHTML elements. After each change, reload the example page to see the changes immediately reflected in your web browser.

3 Hello World and Separation of Concerns: The MVC Architecture

3.1 Introducing the Model View Controller (MVC) Architecture

The Classic Hello World application above consists of a simple static page template. It is static because the data displayed is always the same: the application always displays “Hello World”. More often a page needs to generate or access data dynamically from one or more sources, format it, and then display it. To accomplish this in a maintainable manner, an OXF page is built on the Model / View / Controller (MVC) architecture. The main benefit of such a page architecture is the clearly decoupled interaction of the following components:

- **Model:** Responsible for calling business logic and producing data to be displayed by the view.
- **View:** Receives data from the model and formats it for the user of the application.
- **Controller:** Responsible for defining pages by associating a URL with a view that works with each model. The controller can also specify how a user moves from one page to another based on user input.

In OXF, the MVC architecture works on a page by page basis and each page can have a model and a view. Model and view exchange data between each other in XML format. This contrasts with other platforms where the data exchanged is done using programming language-specific structures, such as Java objects.

Using MVC terminology, the Classic Hello World example above can be explained as follows.

- **Model:** There is no separate model in this simple example. The static string “Hello World” is embedded in the view.

- **View:** The view consists of the `view.xhtml` page template, which is XHTML markup that defines the basic presentation of the page (including the model data “Hello World”). In most situations, an XSLT page template is the most appropriate way of defining a page view.
- **Controller:** The controller is handled by the OXF platform, and is configured by the `page-flow.xml` file shown above. In this example, we define a single page and the view for that page.

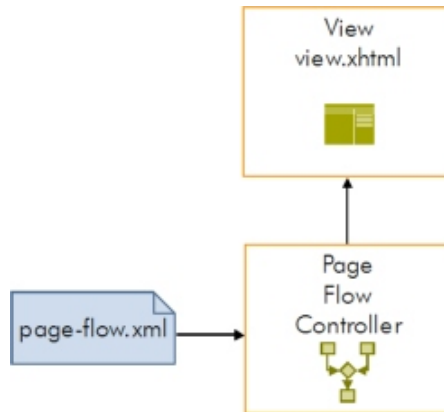


Figure 3 – MVC Components in Classic Hello World

3.2 Hello World MVC: Separate Content from Presentation

Now consider that instead of displaying a static “Hello World!” message by embedding the string directly in the page template, we want to retrieve it from another source, so the “view” can be separated from the “content”. This is done using a page model. A model is a separate component that either has or retrieves data that will be displayed by the view. The controller will orchestrate the interaction between the model and the view, and will send the data in the model to the page view, which then formats the data for user presentation.

This example uses a model containing static data, shown below. A later section will provide the example of a model whose data is created dynamically, for example read from a database or retrieved from a Web service. The model is a very simple static XML document called `static-model.xml` containing one `<name>` element:

`static-model.xml`:

```
<name>World</name>
```

Now examine the `view.xml` page view, which is a modified version of the Hello World Classic’s `view.xhtml` page, implemented as an XSLT 2.0 template. OXF recognizes that it is an XSLT template thanks to the addition of the XSLT namespace and version declarations:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xsl:version="2.0">
  <head>
    <title>Hello World</title>
  </head>
  <body>
    <p>Hello World!</p>
  </body>
</html>
```

Such a template, because it uses XSLT, is typically stored in a file with a “.xsl” extension. We can modify the page flow to reflect this change:

```
<config xmlns="http://www.orbeon.com/oxf/controller">

  <page path-info="/tutorial-hello2" view="view.xml"/>

  <epilogue url="oxf:/config/epilogue.xml"/>

</config>
```

Now we modify the page template to extract data from the model using the XSLT `xsl:value-of` construct. `xsl:value-of` uses an XPath expression in its `select` attribute, like `"/name"` below, to extract data from the source XML document. In this case, we are looking for the value of the `<name>` element:

view.xml

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xsl:version="2.0">
  <head>
    <title>Hello World</title>
  </head>
  <body>
    <p>Hello <xsl:value-of select="/name"/>!</p>
  </body>
</html>
```

So for example the XSLT code `<xsl:value-of select="/sample/message"/>`, when applied to an XML document that looks like:

```
<sample>
  <message>testing 123</message>
</sample>
```

results in the string “testing 123”.

The last step required for a working application is to add the model declaration to this page’s entry in the page flow:

```
<config xmlns="http://www.orbeon.com/oxf/controller">

  <page path-info="/tutorial-hello2" model="static-model.xml" view="view.xml"/>

  <epilogue url="oxf:/config/epilogue.xml"/>

</config>
```

You can experiment by modifying the static content of the model, or by creating a more complex model and reflecting the changes in the view.

4 Hello World XForms: Managing User Input

4.1 Introducing XForms

4.1.1 XForms Model and Instance

Web applications use forms when collecting data from users. OXF uses the W3C XForms standard, where the definition of the form data and associated data validation is defined in what is called an *XForms model*. An XForms model contains an empty, “skeletal” XML instance document that defines the data you wish to collect from the user. Note that in this tutorial we will typically refer to this empty XML document used by the XForms model as the “XForms instance,” or simply the “instance.” When the form is submitted and processed, the empty

instance document is automatically filled in by OXF with the data entered into the form by the user. For example, the instance for the simple XML document defined in `xforms-model.xml` above would look like this:

```
<name></name>
```

Or, using the XML shortcut for empty elements:

```
<name />
```

It is entirely up to the developer to determine the structure and complexity of the XForms instance document. Your business document and any schema that defines it will dictate the format of your XForms instances.

Continuing with our simple `name` example above, we add a `myform` root element to enable the collection of both first and last names.

```
<myform>
  <first-name></first-name>
  <last-name></last-name>
</myform>
```

Again the instance elements, in this case `first-name` and `last-name`, are left empty. They will be filled-out by the OXF XForms engine when the user submits a form. For example after submission, the filled-out instance might contain the data “Mark” and “Twain”. This resulting instance can then be processed by the application, for example be persisted in a database.

```
<myform>
  <first-name>Mark</first-name>
  <last-name>Twain</last-name>
</myform>
```

4.1.2 Creating an XForms Instance

The important thing to remember is that an XForms instance document is simply an XML document where all the data has been removed and only the elements remain. Once the instance is created, as per the XForms standard, it has to be embedded in an XForms model before the XForms model can be used. For small instances, it is easiest to inline the instance within the XForms model’s file, as follows:

`xforms-model.xml`:

```
<xforms:model xmlns:xforms="http://www.w3.org/2002/xforms">
  <xforms:instance>
    <myform>
      <first-name/>
      <last-name/>
    </myform>
  </xforms:instance>
</xforms:model>
```

4.1.3 XForms Controls

The second aspect of XForms consists of XForms *controls*. Controls are visual form elements such as:

- Input fields and text areas
- Submit buttons
- Dropdown lists
- Radio buttons and checkboxes

XForms controls are not defined in the XForms model or instance described in section 4.1.1 above. Instead, they are part of the page's presentation, and are defined in the page template, usually interleaved with XHTML. This allows positioning controls, for example, in XHTML tables, etc.

Most controls are *bound* to the XForms instance with a special attribute called `ref`. Consider for example a simple input text field. The corresponding XForms control is called `xforms:input`. Binding such a control to the `first-name` element of the instance described above is done in a page's view file, as follows:

```
<xforms:input ref="/myform/first-name"/>
```

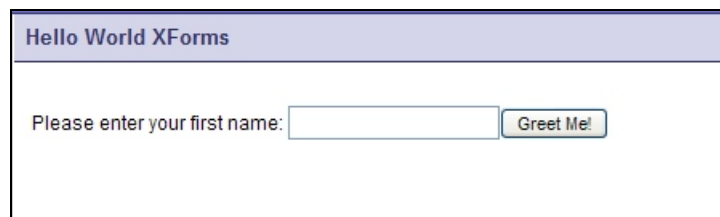
What this achieves is the following:

- When the page is displayed, it contains a graphical text field control,
- When the user fills-out data in the text field and then submit the form, the data entered is stored by the XForms engine in the `first-name` element of the instance
- If the XForms instance contains a value in the `first-name` element, the control displays the value when the page is shown

The following variation on the Hello World example shows a practical use of XForms controls.

4.2 Hello World XForms: Collecting User Input

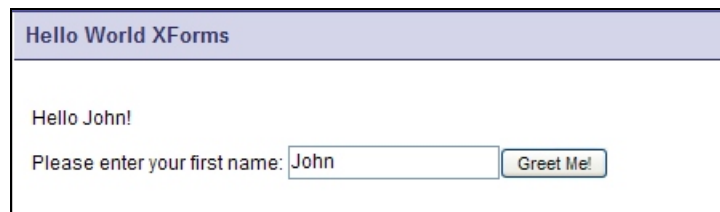
Let's modify the Hello World example to accept user input. This example uses a single page to display a form requesting the user's first name, and then upon submission, the same page is loaded and the greeting "Hello first-name-entered" is displayed at the top of the page. The page must display a text field and a submit button:



The screenshot shows a web browser window with a title bar. The page has a purple header bar with the text "Hello World XForms". Below the header, the text "Please enter your first name:" is followed by a text input field and a button labeled "Greet Me!".

Figure 4 - Hello World and User Input

When the user presses the button, the page is reloaded with a greeting:



The screenshot shows the same web browser window as Figure 4, but the page has been reloaded. The purple header bar still says "Hello World XForms". Below the header, the text "Hello John!" is displayed. Below that, the text "Please enter your first name:" is followed by a text input field containing the text "John" and a button labeled "Greet Me!".

Figure 5- Hello World Greeting

Consider the following XForms instance, which will hold the user's first name upon submission by the user's web browser.

```
<myform>
  <first-name/>
</myform>
```

As seen above, the instance is encapsulated in an XForms model:

`xforms-model.xml`:

```
<xforms:model xmlns:xforms="http://www.w3.org/2002/xforms">
  <xforms:instance>
    <myform>
      <first-name/>
    </myform>
  </xforms:instance>
</xforms:model>
```

The page that will use this XForms model needs to be configured to use the model, so an XForms declaration is added to the page's entry in the page flow as follows:

```
<config xmlns="http://www.orbeon.com/oxf/controller">

  <page path-info="/tutorial-hello3" xforms="xforms-model.xml" view="view.xsl"/>

  <epilogue url="oxf:/config/epilogue.xpl"/>

</config>
```

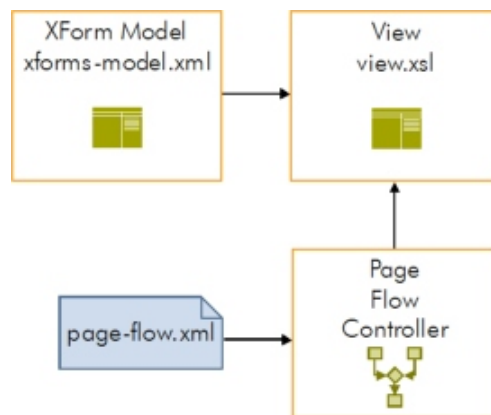


Figure 6 - XForms Hello World and MVC

The Hello World page template is modified with XForms controls to display the input form to the user:

view.xsl:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:xforms="http://www.w3.org/2002/xforms"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xsl:version="2.0">
  <head>
    <title>Hello World</title>
  </head>
  <body>
    <xforms:group>
      <p>Hello <xsl:value-of select="/myform/first-name"/>!</p>
      <p>
        Please enter your first name:
        <xforms:input ref="/myform/first-name"/>
        <xforms:submit>
          <xforms:label>Greet Me!</xforms:label>
        </xforms:submit>
      </p>
    </xforms:group>
  </body>
</html>
```

The following XForms controls are used:

- the mandatory `xforms:group`, which must enclose all other XForms controls

- `xforms:input`, which displays a text field
- `xforms:submit`, which displays a button that will trigger form submission

Note the mandatory XForms namespace declaration (`xmlns:xforms`) at the beginning of the document. The `ref="/myform/first-name"` attribute on the `xforms:input` element instructs the XForms engine to bind the user input entered in the text field to the `first-name` element of the XForms instance document.

Notice that the first time the page is loaded it displays “Hello” with no name. This is because the user has not input anything, yet we are trying to display her first name! To avoid this, the example can be further improved by making the display of the “Hello” text conditional in the page template. This condition is expressed with the XSLT `xsl:if` element, where the greeting is displayed only if the `first-name` element is not the empty string:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:xforms="http://www.w3.org/2002/xforms"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xsl:version="2.0">
  <head>
    <title>Hello World - Version 3</title>
  </head>
  <body>
    <xforms:group>
      <xsl:if test="/myform/first-name != ''">
        <p>Hello <xsl:value-of select="/myform/first-name"/>!</p>
      </xsl:if>
      <p>
        Please enter your first name:
        <xforms:input ref="/myform/first-name"/>
        <xforms:submit>
          <xforms:label>Greet Me!</xforms:label>
        </xforms:submit>
      </p>
    </xforms:group>
  </body>
</html>
```

5 Hello World Page Flow: Managing Multiple Pages

5.1 Introducing Page Flow

Pages by themselves don't do much without the possibility to navigate from one to another based on user activity. Navigation is usually triggered by user actions such as pressing a button or icon, or following a link. In OXF, the description of the relationship between pages in an application is called page flow, and is defined in the `page-flow.xml` file. This file declaratively defines:

- How each page is assembled, according to a Model / View / Controller (MVC) architecture
- How each page possibly transitions to one or more other pages (the actual page flow)

The `page-flow.xml` file is the configuration for the Controller component of OXF's implementation of the MVC architecture. Let's expand the Hello World example into two pages:

- `ask-name-view`, a page asking for the user's first name
- `display-name-view`, a page displaying the users-name as entered in the first page

The two pages will look as follows:

Hello World Page Flow

Please enter your first name:

Figure 7 – Ask Name Hello World Page

Hello World Page Flow

Hello John!

Figure 8 – Display Name Hello World Page

5.2 Declaring and Navigating Multiple Pages

The page flow must first declare each page individually:

```
<config xmlns="http://www.orbeon.com/oxf/controller">

  <page id="ask-name" path-info="/tutorial-hello4" xforms="xforms-model.xml"
    view="ask-name-view.xhtml">
  </page>

  <page id="display-name" path-info="/tutorial-hello4/display-name"
    xforms="xforms-model.xml" view="display-name-view.xhtml">
  </page>

  <epilogue url="oxf:/config/epilogue.xpl"/>

</config>
```

In this particular example, the two pages use the same XForms model, but have different page views. Also note that the two pages are assigned identifiers with the `id` attribute. Those identifiers are used later to navigate between pages.

Each page now needs to accept a user action, which will be captured and stored in the XForms instance document by a new `action` element. Note that the new element can be called anything – here we choose `action` for simplicity.

`xforms-model.xml`:

```
<xforms:model xmlns:xforms="http://www.w3.org/2002/xforms">
  <xforms:instance>
    <myform>
      <action/>
      <first-name/>
    </myform>
  </xforms:instance>
</xforms:model>
```

In the `ask-name` page view, we specify that we want to set the `action` element of our XForms instance (`/myform/action`) to the string “next” when the button labeled “Greet Me!” is pressed:

```
<xforms:submit>
  <xforms:label>Greet Me!</xforms:label>
  <xforms:setvalue ref="/myform/action">next</xforms:setvalue>
</xforms:submit>
```

In the page flow, where we define the `ask-name` page, we now specify that a certain action must be taken when this “next” condition is detected in the XForms instance. This is done by augmenting the page definition with an action element:

```
<page id="ask-name" path-info="/tutorial-hello4" xforms="xforms-model.xml"
  view="ask-name-view.xhtml">
  <action when="/myform/action = 'next'"/>
</page>
```

Finally, we need to tell the page flow what must happen when the action is taken. In this case, we want to navigate the user to the second page called `display-name`. This is done by adding a `result` element to the action element. The action below causes the browser to load the page with id `display-name` when the “Greet Me!” button is pressed:

```
<page id="ask-name" path-info="/tutorial-hello4" xforms="xforms-model.xml"
  view="ask-name-view.xhtml">
  <action when="/myform/action = 'next' ">
    <result page="display-name"/>
  </action>
</page>
```

5.3 Sharing XForms between Pages

The only issue remaining is that the second page’s XForms instance will not contain the information in the first page’s XForms instance; each page acts independently from the other unless specified otherwise. In other words, OXF does not pass the XForms instance from one page to the next automatically. To pass the XForms instance from one page to another, we explicitly copy the exact data we want shared between instances from the first page to the second. This is done with `XUpdate`, a simple language designed to update XML documents. (For more information on `XUpdate`, please see the `XUpdate` section of the OXF Reference Guide.) The `XUpdate` code is embedded in the `result` element:

```
<page id="ask-name" path-info="/tutorial-hello4" xforms="xforms-model.xml"
  view="ask-name-view.xhtml">
  <action when="/myform/action = 'next' ">
    <result page="display-name">
      <xu:update select="/myform/first-name">
        <xu:value-of select="document('oxf:instance')/myform/first-name"/>
      </xu:update>
    </result>
  </action>
</page>
```

The update inserts XML into the `display-name` page’s empty XForms instance. To access the content of this new XForms instance in the `display-page`, the construct `document('oxf:instance')` must be used. The result of the `xu:update` code above is that the content of the `display-page` instance’s `/myform/first-name` element is updated with the value of the `ask-name` page instance’s `/myform/first-name` element; see the figure below:

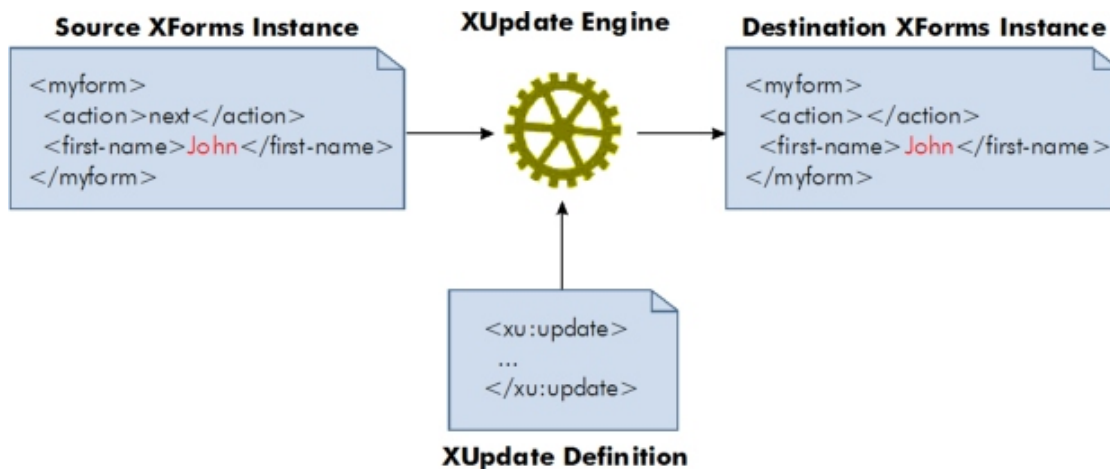


Figure 9 - XForms Instance Update using XUpdate

After the `display-name` page has collected the user's input, the action condition has triggered, and the XForms instance has been copied to the `display-page`'s XForms instance, the `display-name` page must access the first name stored in the instance. This is done with XSLT, as shown previously. The `display-name` page becomes:

```

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:xforms="http://www.w3.org/2002/xforms">
  <head>
    <title>Hello World - Version 4</title>
  </head>
  <body>
    <xforms:group>
      <p>Hello <xsl:value-of select="/myform/first-name"/>!</p>
    </xforms:group>
  </body>
</html>

```

Similarly, the second page can contain a submit button linking back to the first page, and passing the first name back to the first page again. We refer the reader to the full Hello World Page Flow source code for full details.

6 The BizDoc Application

6.1 Introduction

The BizDoc application is a small portion of a fictitious business document processing application. The business documents used by BizDoc are claim documents. BizDoc consists of three pages that allow the user to create, update, and delete claims. The application has the following very simple features:

- Create, update, and delete documents that adhere to a specific XML business document format
- Store and retrieve such documents in an open source XML database (see <http://www.exist.org>)
- Provide a simple user interface to view a list of documents or edit the details of a single document
- Automatically ensure document validity during all stages of input and editing using XML schema

The example application showcases the following capabilities of the OXF platform:

- Use XML Schema to automate form-based business processes, saving time and ensuring standards compliance
- Externalize and re-use input validation rules, easing requirements tracking and facilitating re-use
- Create a user interface that decouples presentation from content, enabling rapid customization

The source code of the BizDoc application is available in `oxf.war` under `WEB-INF/resources/examples/bizdoc`, or, if you are running the standalone BizDoc application, under `oxf-bizdoc.war` under `WEB-INF/resources/bizdoc`. You can modify the XML files in that directory, and see the changes immediately reflected in the application as you reload pages.

6.2 BizDoc Functional Overview

The application is built with the following pages:

- A **summary page** listing all documents by ID that are stored in the database, with a button to create new documents, and buttons to edit or delete existing documents
- **Detail pages** used when creating new documents, or when editing an existing document. The Detail pages use 2 page, wizard-style interface

Summary		
OXF BizDoc		
Available Documents		
Document Identifier	View	Delete
48A52C1F-F567-6A1A-C974-21F9C2A268C5	<input type="button" value="View"/>	<input type="button" value="X"/>
AF61CC78-9E1D-F5C8-F678-4B7E418F720E	<input type="button" value="View"/>	<input type="button" value="X"/>
8790626D-CE76-0964-4659-4F4AEAC8DD7B	<input type="button" value="View"/>	<input type="button" value="X"/>
<input type="button" value="Import Documents"/> <input type="button" value="New Document"/>		

Figure 10 - BizDoc Summary Page

Detail - Step 1

OXF BizDoc

Document Information

Identifier 48A52C1F-F567-6A1A-C974-21F9C2A268C5

Claimant Name

Title

Dr.

?

Last Name

Doe

?

First Name

John

?

Suffix

?

Claimant Address

Street Name

N Columbus Dr.

?

Street Number

511

?

Unit Number

?

City

Chicago

?

State

IL

?

Zip Code

60611

?

Country

USA

?

Email

jdoe@acme.org

?

Save

Back

Next

Figure 11 - BizDoc First Detail Page

Detail - Step 2

OXF BizDoc

Document Information

Identifier 48A52C1F-F567-6A1A-C974-21F9C2A268C5

Additonal Claimant Information

Gender ☒ Male ☐ Female ☐ Unknown ?

Birth Date 1972-10-01 ?

Marital Status Domestic Partner ?

Occupation Manager ?

No comments at this point!

Comments ?

Children

Birth Date 2003-02-02 ?

Name Marco ? X

Add Child

Claim Information

Accident Type Foot Injury ?

Accident Date 2004-07-06 ?

Calculated Rate 10 ?

Save

Back

Figure 12 - BizDoc Second Detail Page

The following page actions are used by BizDoc:

Description	Action Name	Source Page	Destination Page
Show document detail	show-detail	Summary	detail-1
Delete document	delete-documents	summary	summary
Import documents	import-documents	summary	summary
New document	new-document	summary	detail-1
Return to summary	Back	detail-1	summary
Save document	Save	detail-1 or detail-2	detail-1 or detail-2
Go to next detail page	Next	detail-1	detail-2
Go to previous detail page	Back	detail-2	detail-1

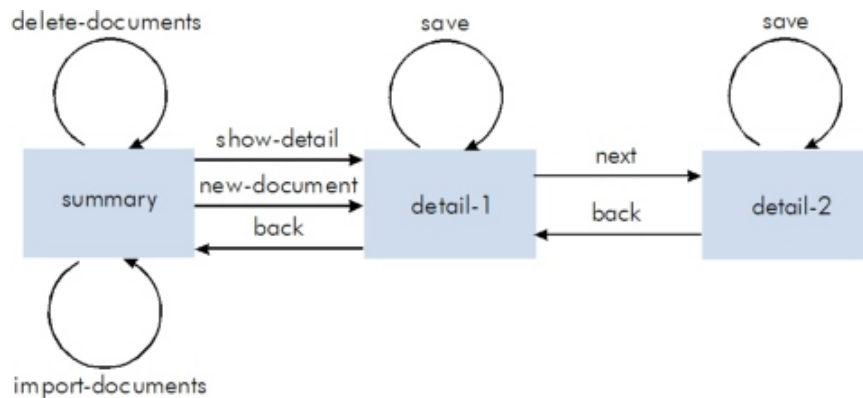


Figure 13 - BizDoc Page Flow by Action

6.3 OXF Background: Using Schemas

Before continuing with the BizDoc walkthrough, we need to review a few additional pieces of OXF background information.

6.3.1 Validating User Input Using XML Schema and XForms

For this tutorial, we create our own simplified XML business document schema using W3C Schema (XSD). Our document, which we will call the claimant Schema, describes a simple insurance claim document. Documents that conform to the claimant Schema contain claimant information and data related business rules, including constraints on data types, data values, data formats, required fields, and simple field level calculations. We describe the Schema in a file called `claim-schema.xsd`. A typical validating XML instance of the claimant Schema might look like the following:

```

<claim xmlns="http://orbeon.org/oxf/examples/bizdoc/claim">
  <insured-info>
    <general-info>
      <name-info>
        <title-prefix>Dr.</title-prefix>
        <last-name>Doe</last-name>
        <first-name>John</first-name>
        <title-suffix></title-suffix>
      </name-info>
      <address>
        <address-type>BillingAddress</address-type>
        <address-detail>
          <street-name>N Columbus Dr.</street-name>
          <street-number>511</street-number>
          <unit-number></unit-number>
        </address-detail>
        <city>Chicago</city>
        <state-province>IL</state-province>
        <postal-code>60611</postal-code>
        <country>USA</country>
        <email>jdoe@acme.org</email>
      </address>
    </general-info>
    <!-- More data follows -->
    <!-- ... -->
  </insured-info>
</claim>

```

Notice that we use an XML namespace declaration at the beginning of the document. When using a declaration of the form `xmlns="..."`, the namespace declaration becomes the *default namespace* for the document. Here, we defined the namespace with the following, arbitrary string:

`http://orbeon.org/oxf/examples/bizdoc/claim`

Note that even though this string looks like an HTTP URL, it is never used by OXF to retrieve a file that could be stored at such a location on the web. This namespace should strictly be seen as a unique string that identifies a type of document, here an insurance claim. Using `orbeon.com` as a host name for example makes it clear that the schema was developed by a company called Orbeon. Another person or organization would be very unlikely to use that same host name. This reduces the possibility of two companies or individuals using the same namespace by mistake.

6.3.1.1 Why use XML Schema?

The BizDoc application relies on the OXF XForms engine to validate user input using the claimant XML Schema. The benefit of using a schema to describe data and related data rules is that this important information can be collected in one separate, standard file, leaving the rule enforcement and format validation to software like OXF. Assume, for example, that we have a simplified version of our claimant Schema, and we have the following sample XML instance document:

```

<insured>
  <first-name>Mark</first-name>
  <last-name>Twain</last-name>
  <age>68</age>
</insured>

```

This document would validate correctly if our simplified claimant Schema was the following:

```

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="insured">
    <xs:complexType>
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
        <xs:element name="first-name" type="xs:string"/>
        <xs:element name="last-name" type="xs:string"/>
        <xs:element name="age" type="xs:positiveInteger"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>

```

```
</xs:complexType>
</xs:element>
</xs:schema>
```

NOTE: If you are not familiar with the syntax of W3C Schema, do not worry: there are visual tools that enable drag and drop XML Schema creation, shielding the user from the syntax.

On the other hand, the following sample XML instance document would not validate correctly, because the age is not a positive integer:

```
<insured>
  <first-name>Mark</first-name>
  <last-name>Twain</last-name>
  <age>thirty-eight</age>
</insured>
```

The XForms engine is able to enforce the rules defined by the Schema, detecting such error conditions and automatically reporting errors in the user interface so the user of the application can correct them.

6.3.1.2 How to use XML Schema in OXF

Once a schema is defined, the application developer must go through the step of creating an empty XForms instance document. Using the simplified `claim-schema.xsd` above, the empty XForms instance would be:

```
<insured>
  <first-name/>
  <last-name/>
  <age/>
</insured>
```

For a general overview on creating an XForms instance from an XML schema or an XML document, please refer above to section 4.1.2.

6.4 OXF Background: XML Pipelines Using XPL

XPL is a powerful declarative language for processing XML using a pipeline metaphor. XML documents enter a pipeline, are efficiently processed by one or more processors as specified by XPL instructions, and are then output for further processing, display, or storage. XPL features advanced capabilities such as document aggregation, conditionals (“if” conditions), loops, schema validation, and sub-pipelines. This section introduces XPL and provides two basic examples. It also discusses one common use case for pipelines in OXF: business logic for building dynamic web pages. For more detailed information about XPL, please see the XPL reference guide.

XPL pipelines are built up from smaller components called XML processors. An XML processor is a software component which consumes and produces XML documents. New XML processors are most often written in Java. But most often developers do not need to write their own processors because OXF comes standard with a comprehensive library. Example OXF processors include an XSTL processor, database processors that interface with both SQL and native XML databases, and a serializer processor that writes XML documents to disk. XPL orchestrates these to create business logic, similar to the way Java code “orchestrates” method calls within a Java object.

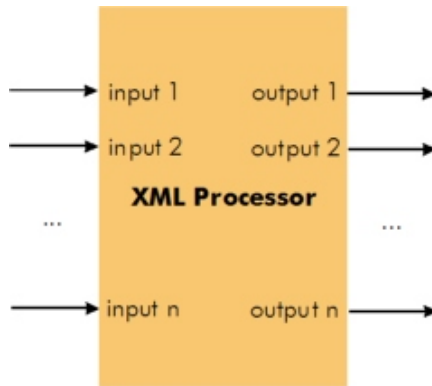


Figure 14 - XML Processor Inputs and Outputs

6.4.1 A Simple Pipeline: Combine Two XML Documents

Consider the following pipeline definition:

```
<p:config xmlns:p="http://www.orbeon.com/oxf/pipeline"
  xmlns:oxf="http://www.orbeon.com/oxf/processors">

  <p:processor name="oxf:file-serializer">
    <p:input name="config">
      <config>
        <file>my-result-file.html</file>
        <directory>c:/temp</directory>
      </config>
    <p:input name="data" href="aggregate('root', file1.xml, file2.xml)"/>
  </p:processor>

</p:config>
```

This pipeline contains one processor, named “oxf:file-serializer”. This processor is a standard part of the OXF processor library. It permits the serialization (or writing) of an XML document to a file on disk. It expects two inputs, specified by `p:input`, which must be named as follows:

- `config`, containing an inline configuration document `<config/>` that has elements that specify the file name and directory where the document should be saved, and
- `data`, which is the data (i.e., XML document) that will ultimately be written to disk.

For comparison, you can imagine how the file-serializer might be written in a traditional procedural language. For example in Java, you might have the following method signature:

```
public void serializeFile(String fileName, String dir, String inputData);
```

Looking at the file-serializer processor’s `data` input, we see the use of `aggregate()`. The `aggregate()` function is a built-in XPL function that takes two or more documents and combines them into a single document under a new root element. For example, assuming the following content for `file1.xml` and `file2.xml`:

`file1.xml`:

```
<list>
  <item>apple</item>
  <item>orange</item>
</list>
```

`file2.xml`:

```
<person>
```

```
<first-name>Bob</first-name>
<last-name>Marley</last-name>
</person>
```

The document resulting from `aggregate('newroot', file1.xml, file2.xml)` is:

```
<newroot>
  <list>
    <item>apple</item>
    <item>orange</item>
  </list>
  <person>
    <first-name>Bob</first-name>
    <last-name>Marley</last-name>
  </person>
</newroot>
```

This pipeline can be represented graphically:

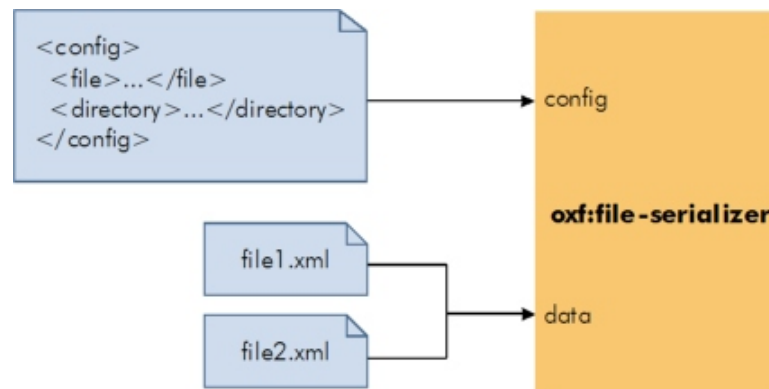


Figure 15 - Simple Pipeline to Aggregate Two Files

This pipeline executes as follows:

1. It calls the `oxf:file-serializer` processor
2. The File serializer processor reads its `config` input
3. The File serializer processor then reads its `data` input
4. Reading the data input causes aggregate function to execute, which causes the two specified files, `file1.xml` and `file2.xml`, to be read and aggregated
5. The File serializer receives that resulting document
6. The File serializer saves the document under the name specified in its configuration document under the `config` element

6.4.2 Lazy Processor Execution Within a Pipeline

It is important to note that the execution ordering of processors within an OXF pipeline follows the principle of *lazy evaluation*. This means that what's important in a pipeline is its final result, and this same algorithm is applied when determining the execution order of the processors within a given pipeline. This means that OXF *always looks at the serializers and output parameters first*. This approach may look non-intuitive at first, but it allows for many processing optimizations. For example, it does not make sense to execute an XSLT transformation if nobody is using its output document!

6.4.3 A Simple 2-Stage Pipeline: Apply an XSL Transform

Most often, pipelines use several XML processors and connect them together to perform more complex business logic. Consider the following pipeline, which performs an XSLT transformation on a document, and then writes the result to a file on disk:

```
<p:config xmlns:p="http://www.orbeon.com/oxf/pipeline"
  xmlns:oxf="http://www.orbeon.com/oxf/processors">

  <p:processor name="oxf:xslt">
    <p:input name="config" href="my-transformation.xml"/>
    <p:input name="data" href="my-input-document.xml"/>
    <p:output name="data" id="transformation-result"/>
  </p:processor>

  <p:processor name="oxf:file-serializer">
    <p:input name="config">
      <config>
        <file>my-result-file.html</file>
        <directory>c:/temp</directory>
      </config>
    <p:input name="data" href="#transformation-result"/>
  </p:processor>

</p:config>
```

This pipeline uses two processors:

- `oxf:xslt`, an XSLT transformer
- `oxf:file-serializer`, which writes an XML document to a file on the file system, as seen above

Each processor mandates a certain number of inputs and outputs, and expects specific names for these inputs and outputs. The first processor declares two inputs and one output:

- A `config` input, pointing to an XSLT style sheet named `my-transformation.xml`
- A `data` input, pointing to an input document under named `my-input-document.xml`
- A `data` output, identified by the identifier `transformation-result`

The second processor declares two inputs:

- A `config` input, containing an inline configuration document specifying where the document must be saved
- A `data` input, which accepts input from the output of the previous processor. NOTE: The connection to the output of the previous processor is specified by “pointing” to the previous processor’s id `transformation-result` using `href="#transformation-result"`.

Again please note that both the XSLT and file-serializer processors, which are both part of the OXF processor library, require inputs and outputs to match the exact number and name given above. If the number or name of inputs or outputs differ, an error will be generated. This pipeline can be represented graphically:

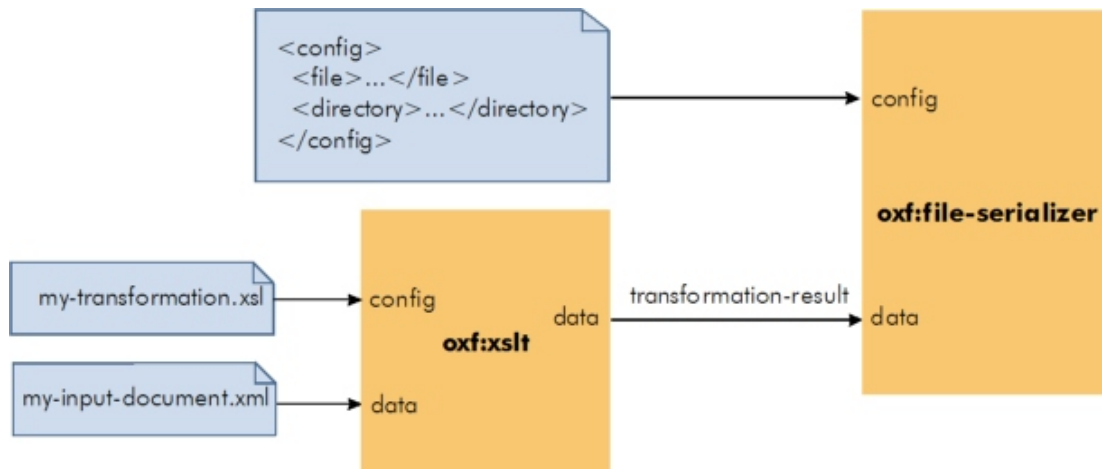


Figure 16 - Two-Processor Pipeline

This pipeline executes as follows (see section 6.4.2 for more info on ordering):

1. It calls the `oxf:file-serializer` processor
2. The File serializer processor reads its `config` input
3. The File serializer processor then reads its `data` input
4. This causes the data output of the `oxf:xslt` processor to be read
5. The `oxf:xslt` processor, in turn, reads its `config` input, which is a file called `my-transformation.xml`
6. It then reads its `data` input, a file called `my-input-document.xml`
7. Having both its style sheet and input document, the XSLT transformer executes the transformation
8. This produces a result sent to the `data` output
9. The File serializer receives that document
10. The File serializer saves the document under the name specified in its configuration

Here again, the lazy evaluation processing described in section 6.4.2 is highlighted, this time with two connected processors. The XSLT transformation is executed only because the serializer is reading the XSLT transformation's output. For most pipelines, however, XPL will follow an intuitive top down, left to right execution order. In this particular case, for example, you can very well consider simply that:

1. The first processor, `oxf:xslt`, executes. It reads its two input documents, and produces the `transformation-result` document.
2. The `oxf:file-serializer` processors executes. It reads its configuration input, and the `transformation-result` document produced previously by the XSLT transformation.

You can imagine how this pipeline might be written in a traditional procedural language. For example in Java-like pseudo-code, you might have the following:

```
String tranformationResult = myXSLTUtil.transform(transformation, myInputDoc);
myFileUtil.write("c:\\temp\\myResultFile.html", tranfomrationResult);
```

Finally, note that our example pipeline has all of it's processor inputs and outputs hard-coded to sepcific values. What if we wanted to re-use the pipleine above by parameterizing it to write the results to a file based on a parameter value, or output the result of the transform to another pipeline instead of writing it to disk? We will explore pipeline parameterization in section next.

6.4.4 Getting Data In and Out of a Pipeline using Parameters

When a model file contains static content, like in the `static-model.xml` example of section 3.2 above, the static content is sent directly to the page's view. However if the model resides in an XPL pipeline (i.e., in a file ending in the `.xpl` extension), we need a way to pass the necessary XML documents *into* the pipeline, and a way to get

the resulting documents *out* of the pipeline. This is achieved with the use of pipeline parameters. Pipelines parameters allow pipelines to read XML documents passed to them, as well as produce XML documents as output.

Pipeline input and output parameters are defined using the `p:param` construct. For example, the following fictitious model pipeline is designed to combine the data input by the user (the XForms instance) with data from a file on disk. The combined document is sent to the view for presentation formatting. This pipeline declares a single input named `instance`, and a single output named `data`. The pipeline uses the identity processor and the `aggregate()` function. The `oxf:identity` processor acts as a data pass-through. Data passed into the processor is passed unchanged out of the processor. It does nothing except send the content of its `data` input to its `data` output.

```
<p:config xmlns:p="http://www.orbeon.com/oxf/pipeline"
  xmlns:oxf="http://www.orbeon.com/oxf/processors">

  <p:param name="instance" type="input"/>
  <p:param name="data" type="output"/>

  <p:processor name="oxf:identity">
    <p:input name="data" href="aggregate('newroot', #instance, my-data.xml)"/>
    <p:output name="data" ref="data"/>
  </p:processor>

</p:config>
```

Our use of the identity processor becomes useful when combined with the `aggregate()` function on the processor's input or output. The output document that results after the identity processor has called `aggregate()` on its `data` input is accessed by its `data` output. In this example, the identity's output document is then "sent" or "hooked up" to the *output of the pipeline*, which is also named `data`, using the `ref` attribute.

6.4.5 href vs. ref in Pipelines

The use of `ref` should not be confused with `href`. `href` is used to read information from a data source, be it a file, web site, pipeline input, or output of a previous processor within a pipeline. In contrast, the use of `ref` in OXF is limited to specifying that the output of a processor should be the ultimate output of the pipeline itself.

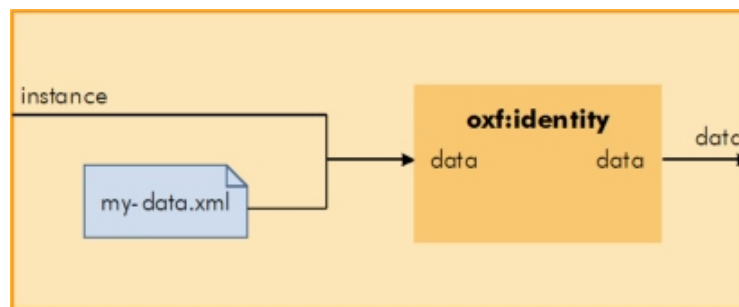


Figure 17 - Pipeline with Input and Output

6.4.6 Pipeline Inputs and Outputs Within a Page Flow

We have seen above in section 3.2 how to create a simple, static model. In many cases, a model cannot be static: it must execute business logic, such as connect to a database, perform manipulations and calculations, and then return data depending on the result of this logic. When a model needs to perform business logic in OXF, the model is written with XPL.

In general, pipelines can have zero or more inputs and zero or more outputs, each arbitrarily named. However, within the context of the page flow, as we are discussing here, a pipeline that wishes to participate as a model within the OXF page flow architecture *must* adhere to specific conventions regarding the number and name of inputs and outputs. In the context of the page flow controller file, pipelines must:

- **Pipeline inputs:** Exactly 0 or 1 input can be declared. The only input available to the pipeline will be the page's XForms instance document. To access this document, you must first declare a single pipeline input called `instance`, which is declared as follows: `<p:param name="instance" type="input"/>`. A processor in the pipeline that reads from this input will receive the filled-in XForms instance created after the user has submitted the form for the page.
- **Pipeline outputs:** Exactly 0, 1, or 2 outputs can be declared.
 - A model with 0 output exists for side-effect reasons only, for example recording an entry in a log. It does not pass data to the page view.
 - If the model does want to pass data to the view, it must declare an output called `data`, which is declared as follows: `<p:param name="data" type="output"/>`. XML that is output by a model pipeline on its `data` output will be automatically connected to the view's `data` input.
 - In some cases, a model needs to modify the XForms instance before the view accesses it. In this case, the model should declare an output called `instance`, which is declared as follows: `<p:param name="instance" type="output"/>`.

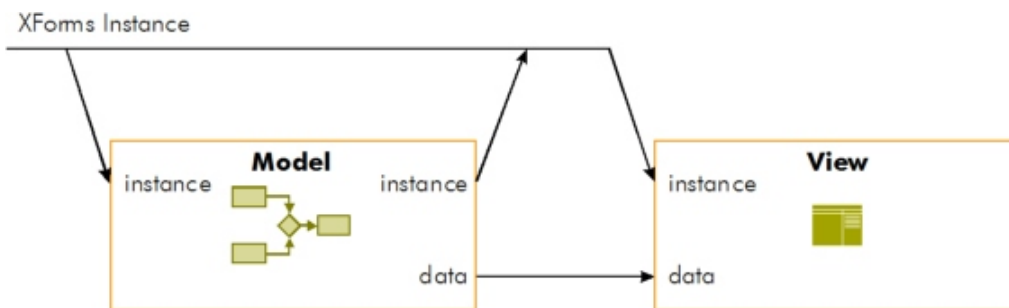


Figure 18 - XForms Instance, Model and View Connections in the Page Flow Controller

6.5 The BizDoc Summary Page

6.5.1 Introduction

The summary page of the BizDoc application must perform the following tasks:

System or User Requirements	Responsible Page Component
Connect to a database	Model
Retrieve a list of documents identified by a key	
Format and display the list of documents	View
Present the user with a method to accomplish the following tasks for a given documents in the list: <ol style="list-style-type: none"> 1. Visualize a particular document's details and allow field-level editing 2. Delete a particular document from the database 	Action (as defined in the page flow)
Add a new document to the database	Action (as defined in the page flow)
Import documents to the database	Action (as defined in the page flow)

The following figure shows how the model and view of the Summary page are interconnected:



Figure 19 - Summary Page Model and View Connection

6.5.2 Summary Page: The Page Model

6.5.2.1 The Model

The page model for the BizDoc application's Summary page is dynamic. It returns, on its data output, an XML document ready to be formatted by the view. The page model makes use of a single processor, `oxf:xml-db-query`, which is used to query an XML database. In the BizDoc application, the eXist open source database is used, and is accessed via this `oxf:xml-db-query` processor using the XML:DB API. Let's examine the complete code for the model:

`summary-model.xpl:`

```

<p:config xmlns:p="http://www.orbeon.com/oxf/pipeline"
  xmlns:oxf="http://www.orbeon.com/oxf/processors"
  xmlns:xdb="http://orbeon.org/oxf/xml/xmlldb">

  <p:param name="data" type="output" schema-href="summary-model.xsd"/>

  <!-- Return the ids of all the documents -->
  <p:processor name="oxf:xml-db-query">
    <p:input name="datasource" href="../datasource.xml"/>
    <p:input name="query">
      <xdb:query collection="/db/oxf/adaptive-example">
        xquery version "1.0";
        <result>
          { / document-info/document-id }
        </result>
      </xdb:query>
    </p:input>
    <p:output name="data" ref="data"/>
  </p:processor>

</p:config>

```

6.5.2.2 Validating Pipeline Output Using XML schema

First, notice the pipeline has a `data` output. The purpose of the pipeline is to output an XML document that is dynamically generated by a database query, and this is accomplished by connecting the output of the `xml-db-query` processor to the output of the pipeline using `<p:output name="data" ref="data"/>`. When the page view receives the result document from the query, it is important to know the format, or schema, of the XML document. For our example, we have chosen to generate query result documents of the form:

```

<result>
  <document-id>abc-1234</document-id>
  <document-id>def-5678</document-id>
</result>

```

In other words, the query result document has a root element we have decided to name `result`, and a sequence of zero or more `document-id` elements containing the database ids of the documents satisfying the query. The choice of this structure is chosen by the developer, but the following condition must be met to ensure the page displays properly: the person doing the page UI design and the person developing the business logic (model) must agree on the format of the query information they share. It is possible for developers and UI designers to

formalize and enforce these conditions by using an XML schema to define and document the expected format. OXF supports the following two schema languages:

- W3C XML Schema version 1.0, the most widespread XML schema language
- Relax NG, an alternate, easier to learn language, standardized by OASIS

We use W3C Schema (XSD) for this example. For example, one Schema that validates the output of our example model can be expressed as follows:

summary-model.xsd:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="result">
    <xs:complexType>
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
        <xs:element name="document-id" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Using this Schema, the output of the model can be validated at runtime simply by adding a `schema-href` attribute to the pipeline's output:

summary-model.xpl:

```
<p:config xmlns:p="http://www.orbeon.com/oxf/pipeline"
  xmlns:oxf="http://www.orbeon.com/oxf/processors">

  <p:param name="data" type="output" schema-href="summary-model.xsd"/>

  ...
</p:config>
```

Validation using a schema may have performance implications and is certainly not required, but doing so will make your application more robust. For example when used during development and testing, if the query results passed by the model to the view do not follow the expected schema, a runtime error will be generated and can be handled before the data progresses through the application.

In general, the output of any processor *or any pipeline* can use the `schema-href` attribute to enforce validation of schema-defined business rules on the component's output. OXF makes it easy to leverage the power of XML schema languages.

6.5.2.3 Querying a Database for Model Data

The `summary-model.xpl` pipeline calls the `oxf:xml-db-query` processor, which facilitates interaction with XML databases that support the XML:DB protocol. `oxf:xml-db-query` exposes the following inputs and outputs:

- A `datasource` input specifying database connection information
- A `query` input accepting a query in XPath or XQuery format
- A `data` output producing the result of the database query

The `datasource` input refers to a configuration file describing how connections are made to the embedded eXist XML database:

datasource.xml:

```
<datasource>
  <!-- Specify the driver for the eXist database -->
  <driver-class-name>org.exist.xmlldb.DatabaseImpl</driver-class-name>
  <!-- This causes the use of the embedded eXist database -->
```

```
<uri>xmldb:exist:///db</uri>
</datasource>
```

The query input contains the XQuery code, encapsulated in an `xdb:query` element, which also specifies the collection the query must address. XQuery is an XML query language being standardized by the W3C. You can view XQuery as a powerful superset of XPath. The example query is as follows:

```
<xdb:query collection="/db/oxf/adaptive-example">
  xquery version "1.0";
  <result>
    {/document-info/document-id}
  </result>
</xdb:query>
```

The purpose of this simple query is to retrieve the document ids of all documents in the database, and then return this information as a document that conforms to the format described above in section 6.5.2.2. The result document will be rooted with a `result` element, and then contain a list of all document ids in the database collection. In our sample database, a document's id is stored in the document itself; the XQuery code required obviously depends on the format of the documents in the database.

After the query is executed, the data output of the `oxf:xmldb-query` processor is hooked up to the data output of the pipeline itself using the following construct: `<p:output name="data" ref="data"/>.`

6.5.3 Simplified Summary Page View

Let's first examine a simplified version of the Summary page's template, which is a very simple XHTML page with a couple of XSLT constructs:

summary-view.xsl:

```
<html xmlns:xsl="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xmlns="http://www.w3.org/1999/xhtml">

  <body>
    <table>
      <tr>
        <th>Document Id</th>
      </tr>
      <xsl:for-each select="/result/document-id">
        <tr>
          <td><xsl:value-of select="."/></td>
        </tr>
      </xsl:for-each>
    </table>
  </body>
</html>
```

It is important here to remember the format of the query document the view receives. A database that contained two documents might return the following query result document:

```
<result>
  <document-id>abc-1234</document-id>
  <document-id>def-5678</document-id>
</result>
```

The view uses XSLT's `for-each` looping statement to iterate over the sequence of `/result/document-id` elements in this query result document. This looping builds an XHTML table row by row, and will produce one table row per `document-id` element in the model document. The text value of each document is displayed in a table cell. Against the 2-line document above, the dynamically generated XHTML table rows would look like:

```
<tr>
  <td>abc-1234</td>
</tr>
```

```
<tr>
  <td>def-5678</td>
</tr>
```

Notice that the view's `xsl:value-of` elements have been replaced with the value the extracted from the input document.

6.5.4 Complete Summary Page View

The simple page view shown above is complete: it displays the document ids returned from the database. However it does not meet all the application requirements originally specified. For example, it does not offer any functionality such as viewing, deleting, and creating new documents. To add user interaction, we must add XForms controls to the page view.

The first step in creating a form to handle user commands and capture user input is to create the XForms model that contains the information needed to execute the commands. In this case, we want to let the user specify an action (view, delete, etc.) and a possible document id, which we can code as follows:

summary-xform.xml:

```
<xforms:model xmlns:xforms="http://www.w3.org/2002/xforms">
  <xforms:instance>
    <form>
      <action/>
      <document-id/>
    </form>
  </xforms:instance>
  <xforms:submission method="post"/>
</xforms:model>
```

The `action` and `document-id` elements are used by the page view to store the name of the action performed, as well as the id of the document on which to perform that action. Note that we chose to root our XForms instance document with an element called `<form>`. There is nothing special about the “form” name; we could have called it “summary-form” or any other name that makes sense. Here we are collecting form information, so we chose “form”.

The complete page view is:

summary-view.xsl:

```
<html xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xmlns:xforms="http://www.w3.org/2002/xforms"
      xmlns="http://www.w3.org/1999/xhtml">

  <body>
    <xforms:group ref="/form">
      <table>
        <tr>
          <th>Document Id</th>
          <th></th>
        </tr>
        <xsl:for-each select="/result/document-id">
          <tr>
            <td>
              <xsl:value-of select="."/>
            </td>
            <td>
              <xforms:submit>
                <xforms:label>View</xforms:label>
                <xforms:setvalue ref="action">show-detail</xforms:setvalue>
                <xforms:setvalue ref="document-id">
                  <xsl:value-of select="."/>
                </xforms:setvalue>
              </xforms:submit>
              <xforms:submit>
                <xforms:label>Delete</xforms:label>
                <xforms:setvalue ref="action">
                  delete-documents
                </xforms:setvalue>
                <xforms:setvalue ref="document-id">
                  <xsl:value-of select="."/>
                </xforms:setvalue>
              </xforms:submit>
            </td>
          </tr>
        </xsl:for-each>
      </table>
      <p>
        <xforms:submit>
          <xforms:label>Import Documents</xforms:label>
          <xforms:setvalue ref="action">import-documents</xforms:setvalue>
        </xforms:submit>
        <xforms:submit>
          <xforms:label>New Document</xforms:label>
          <xforms:setvalue ref="action">new-document</xforms:setvalue>
        </xforms:submit>
      </p>
    </xforms:group>
  </body>
</html>
```

The additions consist of an `xforms:group` element and several `xforms:submit` controls. The `xforms:group` surrounds all XForms controls to set a base path in the XForms instance, here `/form`. This is a convenience, which then makes it possible to use `ref` attributes relative to that base path, as we do when we referred to `<xforms:setvalue ref="document-id">`, instead of the longer `<xforms:setvalue ref="/form/document-id">`.

Submit controls are used to place buttons on the page that trigger certain actions. All submit controls have the same features:

- A `label` element, specifying the text on the button
- One or more `setvalue` elements, that set values on the XForms instance document when the button is pressed. In our example,

- all of them set the `/form/action` element to the name of the action to perform,
- some of them additionally set the `/form/document-id` element to the document id on which the action must be performed

The XForms controls within the `xsl:for-each` loop will be generated dynamically from this XSLT before being sent to the browser. The value set in the `document-id` element is extracted from the page model and set with `xsl:value-of`:

```
<xforms:submit>
  <xforms:label>View</xforms:label>
  <xforms:setvalue ref="action">show-detail</xforms:setvalue>
  <xforms:setvalue ref="document-id">
    <xsl:value-of select="."/>
  </xforms:setvalue>
</xforms:submit>
```

For example, if the model document contains two `document-id` elements:

```
<result>
  <document-id>abc-1234</document-id>
  <document-id>def-5678</document-id>
</result>
```

The dynamically generated XHTML and XForms controls ultimately sent to the browser is:

```
<tr>
  <td>abc-1234</td>
  <td>
    <xforms:submit>
      <xforms:label>View</xforms:label>
      <xforms:setvalue ref="action">show-detail</xforms:setvalue>
      <xforms:setvalue ref="document-id">abc-1234</xforms:setvalue>
    </xforms:submit>
    <xforms:submit>
      <xforms:label>Delete</xforms:label>
      <xforms:setvalue ref="action">delete-documents</xforms:setvalue>
      <xforms:setvalue ref="document-id">abc-1234</xforms:setvalue>
    </xforms:submit>
  </td>
</tr>
<tr>
  <td>def-5678</td>
  <td>
    <xforms:submit>
      <xforms:label>View</xforms:label>
      <xforms:setvalue ref="action">show-detail</xforms:setvalue>
      <xforms:setvalue ref="document-id">def-5678</xforms:setvalue>
    </xforms:submit>
    <xforms:submit>
      <xforms:label>Delete</xforms:label>
      <xforms:setvalue ref="action">delete-documents</xforms:setvalue>
      <xforms:setvalue ref="document-id">def-5678</xforms:setvalue>
    </xforms:submit>
  </td>
</tr>
```

6.6 The BizDoc Detail Pages

6.6.1 Introduction

The detail pages of the BizDoc application must perform the following tasks:

System or User Requirements	Responsible Page Component
Format and display the XML document available in the XForms instance using XForms controls	View
Save the document to the database	Action (as defined in the page flow)
Navigate back and forward to the Summary page and the following Detail page.	Actions (as defined in the page flow)

The detail pages consist of two nearly identical pages that we call `detail-1` and `detail-2`, organized in a wizard-like fashion, where the user can navigate from the first page to the second page and vice-versa. They differ only by how much of the claim document they are allowed to edit. In addition, `detail-2` illustrates the use of more XForms controls. We focus on the first detail page, `detail-1`. The purpose of the detail pages is to allow the user of the application to either:

- Create a new document
- Edit an existing document retrieved from the database

For the detail pages, these two operations are very similar:

New Document	Existing Document
The document is empty before the user edits it	The document may contain data previously entered
When saving, the document must be created in the database	When saving, the existing document in the database must be overwritten

Each detail page is built in a manner similar to the summary page, but with its own XForms model and page view. It does not require a page model, for the following reasons:

- The XML document to edit is stored entirely in the XForms instance. There is no need, when showing the detail page, to retrieve the document from the database, as done in the Summary page. If this was the case, a model would be required.
- Saving the document is handled by action pipelines.

For example, here is the page flow declaration for the `detail-1` page:

```
<page id="detail-1" path-info="/adaptive/detail-1" xforms="detail-form.xml" view="detail-view-1.xml"/>
```

For more information on how a page view accesses the XForms instance or output from the model, please refer to section 6.4.6.

6.6.2 Detail Page: The XForms Model

To edit an existing claim document, the XForms model must contain an instance holding the claim document to be edited. This allows binding XForms controls in the page view to different elements of the claim document.

The claim document XForms model is as follows:

`detail-xforms-model.xml`

```
<xforms:model xmlns:xforms="http://www.w3.org/2002/xforms"
  xmlns:xi="http://www.w3.org/2003/XInclude"
  schema="sample-form-schema.xsd">
  <xforms:instance>
    <form>
      <action/>
      <document-id/>
      <document>
        <xi:include href="empty-instance.xml"/>
      </document>
    </form>
  </xforms:instance>
</xforms:model>
```

```

        </document>
      </form>
    </xforms:instance>
    <xforms:submission method="post" />
  </xforms:model>

```

The XForms model contains an instance with `action` and `document-id` already present in model so that we can capture these gestures by the user. Here, however, we use XInclude (see <http://www.w3.org/TR/xinclude/>), a standard XML inclusion mechanism, to “read in” the actual empty claimant instance we wish to edit. The included file, called `empty-instance.xml`, contains the skeleton of the claimant XML document. Using XInclude allows you to reuse the empty instance between the `detail-1` and `detail-2` XForms models, and the summary page’s new-document action.

`empty-instance.xml`

```

<claim xmlns="http://orbeon.org/oxf/examples/bizdoc/claim">
  <insured-info>
    <general-info>
      <name-info>
        <title-prefix></title-prefix>
        <last-name></last-name>
        <first-name></first-name>
        <title-suffix></title-suffix>
      </name-info>
    </general-info>
  </insured-info>
  <!-- Etc. -->
</claim>

```

6.6.3 Detail Page: The Page View

The page template for the detail page is built like a regular XHTML page, but with XForms controls included to define the form. Here is a fragment of the detail page view – the XForms controls are highlighted in red:

```

<html xmlns:xsl="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xmlns:xforms="http://www.w3.org/2002/xforms"
      xmlns:claim="http://orbeon.org/oxf/examples/bizdoc/claim"
      xmlns="http://www.w3.org/1999/xhtml">

  <body>
    <xforms:group ref="/form/document">
      <xforms:group ref="claim:claim">
        <table>
          <xforms:group ref="claim:insured-info/claim:general-info">
            <tr>
              <th align="right" valign="top">Party Information</th>
              <td valign="top">
                <xforms:group ref="claim:name-info">
                  <table>
                    <tr>
                      <th>Title</th>
                      <td><xforms:input ref="claim:title-prefix"/></td>
                    </tr>
                    <tr>
                      <th>Last Name</th>
                      <td><xforms:input ref="claim:last-name"/></td>
                    </tr>
                  </table>
                </xforms:group>
              </td>
            </tr>
          </xforms:group>
        </table>
      </xforms:group>
      <hr/>
      <table>
        <tr>
          <td>
            <xforms:submit>
              <xforms:label>Save</xforms:label>
              <xforms:setvalue ref="/form/action">save</xforms:setvalue>
            </xforms:submit>
          </td>
        </tr>
      </table>
    </xforms:group>
  </body>
</html>

```

This view uses the same `xforms:group` elements as the example in section 4 above. Again these are not strictly required, but allow addressing groups of instance elements without having to repeat paths from the root element of the whole document. In particular, the first one, `<xforms:group ref="/form/document">`, allows all the nested `ref` attributes to be relative to the root of the claim document. For example, referring to the root element of the claim document is done with `ref="claim:claim"` instead of `ref="/form/document/claim:claim"`. One XForms submit control is shown here, and sets an action name to the value "save".

6.7 BizDoc Actions: Navigating from Summary to Detail

6.7.1 Executing an Action Pipeline

Let's now consider the Summary page view again. As we have seen with the Hello World example in section 5 above, pressing a button submits the form and sets the specified values in the XForms instance. For example, with the following XForms submit control:

```

<xforms:submit>
  <xforms:label>View</xforms:label>
  <xforms:setvalue ref="action">show-detail</xforms:setvalue>
  <xforms:setvalue ref="document-id">abc-1234</xforms:setvalue>
</xforms:submit>

```

Pressing the button associated with this control causes the creation of the following XForms instance:

```
<form>
  <action>show-detail</action>
  <document-id>abc-1234</document-id>
</form>
```

Let's look at the entry within the page flow that defines the summary page:

```
<page id="summary" path-info="/adaptive" xforms="summary-xform.xml"
      model="summary-model.xpl" view="summary-view.xsl">
  <!-- View a document -->
  <action when="/form/action = 'show-detail'" action="find-document.xpl"/>
</page>
```

We have seen before how an action can be triggered when an XPath expression against the XForms instance evaluates to true. Here, when the instance's `/form/action` element contains the value "show-detail", the `find-document.xpl` action is executed. The `action` element specifies an `action` attribute that indicates the appropriate pipeline to run. In this example, the `find-document` pipeline is run, which is shown below:

`find-document.xpl`:

```
<p:config xmlns:p="http://www.orbeon.com/oxf/pipeline"
          xmlns:oxf="http://www.orbeon.com/oxf/processors"
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
          xmlns:xdb="http://orbeon.org/oxf/xml/xmlldb">

  <p:param name="instance" type="input"/>
  <p:param name="data" type="output"/>

  <!-- Dynamically build query -->
  <p:processor name="oxf:xslt">
    <p:input name="data" href="#instance"/>
    <p:input name="query"> <!-- Here we specify the XSL stylesheet in-line -->
      <xdb:query collection="/db/oxf/adaptive-example" xsl:version="2.0">
        xquery version "1.0";
        <document-info>
          {(//document-info[document-id
            = '<xsl:value-of select="*/document-id"/>'][1]/*}
        </document-info>
      </xdb:query>
    </p:input>
    <p:output name="data" id="dynamically-generated-query"/>
  </p:processor>

  <!-- Run query -->
  <p:processor name="oxf:xmlldb-query">
    <p:input name="datasource" href="datasource.xml"/>
    <p:input name="query" href="#dynamically-generated-query"/>
    <p:output name="data" ref="data"/>
  </p:processor>

</p:config>
```

Similar to a pipeline that acts as a dynamic page model in the MVC architecture, an action pipeline can:

- Access the page's XForms instance document by reading from their `instance` input. This is shown in the pipeline above in the XSLT processor's code fragment: `<p:input name="data" href="#instance"/>`.
- Output a new XML document on its `data` output. The next stage of processing, defined by the page flow's `result` element, will then be able to read this output. This is shown in the pipeline above in the `oxf:xmlldb-query` code fragment: `<p:output name="data" ref="data"/>`.

The only major difference between the `find-document` action of the Detail page and the page model of the Summary page is that the `find-document` action pipeline uses XSLT to dynamically generate the query for the `oxf:xml-db-query` processor, instead of using a static query. The document returned by the `find-document` pipeline has the following format:

```
<document-info>
  <document-id>abc-1234</document-id>
  <document>
    <!-- The document retrieved from the database is inserted here -->
  </document>
</document-info>
```

This format was arbitrarily chosen by the developer. The constraint is that we need to return the information required by the view from the document retrieved.

6.7.2 Passing Data Out of an Action to Another Page's XForms Instance

What we have achieved so far is, upon user action:

- Running a particular pipeline
- Accessing the Summary page's XForms instance from that pipeline
- Running an XML processor accessing the database
- Sending the XML document retrieved from the database to the pipeline's data output

What remains to be done is to actually doing something with the returned XML document. Namely, what we want to achieve is passing that XML document to the Detail page's XForms instance. As seen in section 5 above, passing data from one page to another is done within the Page Flow using XUpdate code. This is exactly the strategy implemented here.

Unlike the Summary's *page model* pipeline, `summary-model.xml`, whose output is made available to the page view automatically via `summary-model`'s data output, the output of the `find-document` action pipeline is made available to XUpdate code in the Page Flow, so that the XForms instance of the detail page can be updated with the document that has just been retrieved. Accessing the XML document sent by the action pipeline on its data output is accomplished, from XUpdate, using the `document('oxf:action')` function. The XML document is retrieved from the result of the action, and used to update the appropriate parts of the Detail page's XForms instance:

```
<action when="/form/action = 'show-detail'" action="find-document.xml">
  <result page="detail-view-1">
    <xu:update select="/form/document-id">
      <xu:value-of select="document('oxf:action')/document-info/document-id"/>
    </xu:update>
    <xu:update select="/form/document">
      <xu:copy-of select="document('oxf:action')/document-info/document/*"/>
    </xu:update>
  </result>
</action>
```

This is simply going a step further than the Hello World with Page Flow example described in section 5 above. In that example, we passed the user name entered on the first page to the second page. Here, we pass *an entire business document* from the Summary page to the Detail page.

6.8 BizDoc Actions: Saving and Deleting Documents

6.8.1 Introduction

Actions that save and delete pages in the BizDoc application must perform the following tasks:

System or User Requirements	Responsible Page Component
-----------------------------	----------------------------

Connect to a database	Action
Retrieve a document from the database by document id	
Update existing documents, create new documents	

When the user presses the “Save” document of a detail page, the `save-document-action.xml` pipeline is executed, as defined in the Page Flow. `save-document-action` reads the edited XForms instance from its instance input. The overall structure of the `save-document-action` pipeline is as follows:

`save-document-action.xml`

```
<p:config xmlns:p="http://www.orbeon.com/oxf/pipeline"
  xmlns:oxf="http://www.orbeon.com/oxf/processors"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xdb="http://orbeon.org/oxf/xml/xmldb"
  xmlns:xu="http://www.xmldb.org/xupdate">

  <p:param name="instance" type="input"/>

  <p:processor name="oxf:pipeline">
    <p:input name="config" href="../summary/find-document-action.xml"/>
    <p:input name="instance" href="#instance"/>
    <p:output name="data" id="result"/>
  </p:processor>

  <!-- should we update or insert? -->
  <p:choose href="#result">
    <p:when test="/document-info/document-id != ''">
      <!-- Document exists already, UPDATE it -->
      <!-- ... -->
    </p:when>
    <p:otherwise>
      <!-- Document does not exist, INSERT it -->
      <!-- ... -->
    </p:otherwise>
  </p:choose>

</p:config>
```

The first step of `save-document-action` calls `find-document-action` to retrieve a document in the database by id. This is an example of component re-use in OXF. The second step executes some conditional logic:

- If the document exists (i.e., the document-id is the empty string), it is updated
- If the document does not exist, it is created

The claim information entered by the user is simply extracted from the XForms instance using `href="#instance"`.

6.8.2 Conditionals in XPL

The conditional logic above is made using the `p:choose`, `p:when` and `p:otherwise` constructs of XPL. The `href` attribute of `p:choose` refers to an XML document, on which the XPath conditions specified by the `test` attribute of `p:when` are run. For example, if the document returned by the `find-document-action.xml` pipeline is as follows:

```
<document-info>
  <document-id>abc-1234</document-id>
  <document>
    <!-- Document content here -->
    <!-- ... -->
  </document>
</document-info>
```

Then we can conclude that a document was found in the database, because the `document-id` is non-empty. In the case above, the XPath test `/document-info/document-id != ''` returns true and the relevant `p:when` branch is followed, because the content of `/document-info/document-id` is not the empty string. On the other hand, if the `document-id` element contained an empty string, the condition would return false, and the default branch, under `p:otherwise`, is followed.

6.8.3 Updating and Inserting Claim Documents

In the update case, when we find an existing id in the database, we must update the existing document in the database. To accomplish this, we use XUpdate. However we must build the appropriate XUpdate statement dynamically, so we use the XSLT processor to apply a short XSL template that dynamically creates the appropriate XUpdate statement. This code uses the “attribute value template” feature in XSLT, as follows:

```
<p:processor name="oxf:xslt">
  <p:input name="data" href="#instance"/>
  <p:input name="query">
    <!-- put our Xupdate code in-line here, and use XSLT to fill in the dynamic values -->
    <xdb:update collection="/db/oxf/adaptive-example" xsl:version="2.0">
      <xu:modifications version="1.0">
        <xu:update select="/document-info/document[document-id
          = '{/document-info/document-id}']">
          <xsl:copy-of select="/document-info/document/*[1]"/>
        </xu:update>
      </xu:modifications>
    </xdb:update>
  </p:input>
  <p:output name="data" id="query"/>
</p:processor>
```

The `oxf:xml:db:update` processor specifies the format of the update. The dynamically generated query document might look like the following:

```
<xdb:update collection="/db/oxf/adaptive-example" xsl:version="2.0">
  <xu:modifications version="1.0">
    <xu:update select="/document-info/document[document-id = 'abc-1234']">
      <claim:claim>
        <!-- Rest of the claim document -->
        <!-- ... -->
      </claim:claim>
    </xu:update>
  </xu:modifications>
</xdb:update>
```

The embedded XUpdate statement specifies that:

- Any `/document-info/document` element in the database containing a `document-id` element with value “abc-1234” will be updated
- That element is updated with the inline XML document; this will be a claim document with root element `claim:claim`

In other words, we replace the entire content of the existing `document` element with a new document.

The insertion is done similarly, using the `oxf:xml:db-insert` processor. In this case, we simply create an entire `document-info` structure containing `document-id` and `document` elements.

6.8.4 Deleting Claim Documents

The Summary page allows a user to delete a document from the database. The user’s delete action is, as usual, handled with an OXF Action. The file `delete-documents-action.xpl` contains the XPL code to delete documents from the database, based on a list of document ids present in the `/form/document-id`

element of the XForms instance. Deleting a document is done using the `oxf:xml-db-delete` processor, which works similarly to the other three XML:DB processors for querying, updating, and inserting documents.

6.8.5 Miscellaneous Actions

The BizDoc application features other actions not discussed above:

- From the Summary page, a button that initializes the database by importing a static set of documents
- Navigating forward and backward with the Next and Back buttons in the Detail pages

You are referred to the Page Flow to understand how those actions are handled in the Summary and Detail pages. The `import-documents-action.xml` file contains the code for the import action.

7 BizDoc and the Real World: Changing Requirements

7.1.1 Introduction

We have seen how to build a very simple application to create, read, update, and delete XML business documents. An important benefit of using an end-to-end XML approach is that the data model, i.e. the structure of the data entered and stored in the business document, can be easily changed. We detail below the steps required to update the data model.

7.1.2 Steps

Assume that the person-info section of the claimant schema must be augmented with a new piece of information: a general-purpose comment field. For example, instead of being filled-out as follows:

```
<person-info>
  <gender-code>M</gender-code>
  <birth-date>1972-10-01</birth-date>
  <marital-status-code>C</marital-status-code>
  <occupation>Manager</occupation>
</person-info>
```

The new XML fragment should look like this:

```
<person-info>
  <gender-code>M</gender-code>
  <birth-date>1972-10-01</birth-date>
  <marital-status-code>C</marital-status-code>
  <occupation>Manager</occupation>
  <comments>This is a comment.</comments>
</person-info>
```

Several files must be modified to achieve this modification:

1. The XML schema
2. The empty XForms instance (or document skeleton)
3. Any page view that references a claimant document
4. Any existing document in the database that must conform to the new schema, if any

Modifying the XML schema is straightforward: the schema must simply declare the new element, after the `occupation` element:

```
...
<xs:element type="xs:string" name="occupation"/>
<xs:element type="xs:string" name="comments"/>
...
```

Similarly, the empty XForms instance must include the new element:

```
...
<occupation></occupation>
<comments></comments>
...
```

And the page view must include a reference to the new comment information:

```
...
<tr>
  <th align="right" valign="top">Occupation</th>
  <td>
    <xforms:input ref="claim:occupation"/>
  </td>
</tr>
<tr>
  <th align="right" valign="top">Comments</th>
  <td>
    <xforms:textarea ref="claim:comments"/>
  </td>
</tr>
...
```

Since comments can be pretty long, we use here an XForms text area control.

Finally, since our sample BizDoc application only supports one schema at a time, we must convert the existing documents in the database, if needed, to validate against the new XML schema. If the new data fields are allowed to be empty, as is the case with the `comments` field, or if new fields have default values, this task can be automated with a simple “schema upgrade” pipeline that:

1. Fetches a list of all current document id’s from the database
2. For each document
 - a. Reads the document from the database
 - b. Transforms it to validate with the new schema with XSLT
 - c. Saves the document back to the database

8 Conclusion and Further Steps

This tutorial has covered many of the OXF basics. However, it does not cover areas such as:

- J2EE integration: user authentication, session management, SQL data sources, EJB integration, etc.
- Detailed description of XForms controls
- All standard XML processors
- Application Packaging and deployment
- Web Services
- Etc.

You are encouraged to look at the OXF documentation, as well as the OXF examples portal, for further information.